Grundlagen der Programmierung

Skript zur Vorlesung

Ralf Hinze
Technische Universität Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Wintersemester 2019/2020
22. Januar 2020

Vorwort

Eine paar Worte vorneweg.

Worte des Willkommens Herzlich willkommen an der Technischen Universität Kaiserslautern (TUK) im Fachbereich Informatik (FBI).

> welkom, mirë se vini, welkomma, bel bonjou, مرحبا, أهلا وسهلا, bari galoust, xos gelmissiniz, i bisimila, akwaba, ongi etorri, Шчыра запрашаем, swagata, amrehba sisswène, ani kié, dobro došli, degemer mad, добре дошъл, kyo tzo pa eit, benvinguts, marsha vog'iyla, ulihelisdi, bonavinuta, dobrodošli, vítejte, velkommen, pô la bwam/bepôyédi ba bwam, welkom, welcome, bonvenon, tere tulemast, woezon, vælkomin, tervetuloa, welkom, bienvenue, wolkom, binvignut, awaa waa atuu, benvido, bin la v'nu, mobrdzandit/ketili ikhos tkveni mobrdzaneba, herzlich willkommen, Καλώς ήλθατε, tashi delek, malo e lelei, difika dilenga, hoş geldiniz, gazhasa oetiśkom, Ласкаво просимо, khush amdeed, hush kelibsiz chào mừng ông/bà/cô mới đến, bénvnou/wilicome, croeso, dalal ak diam, ékouabô/ékabô

Wir hoffen, dass Sie sich an Ihrer neuen Wirkungsstätte wohlfühlen und mit viel Spass, großem Interesse und Begeisterung studieren.

Einführende Worte Das vorliegende Skript ist parallel zur Vorlesung im Wintersemester 2018/2019 entstanden und wird im laufenden Semester peu à peu aktualisiert. Das Skript versteht sich in erster Linie als Hilfsmittel beim Nacharbeiten der Vorlesung, ist aber mit Einschränkungen auch für das Selbststudium geeignet.

Die Vorlesung „Grundlagen der Programmierung“ beschäftigt sich wie im Titel angedeutet mit grundlegenden Konzepten von Programmiersprachen, Programmierparadigmen und Programmiertechniken. Um gleich ein häufiges Missverständnis auszuräumen: Es handelt sich *nicht* um einen Programmierkurs („Einführung in die Programmiersprache XYZ“). Zwar werden die eingeführten Konzepte in einer konkreten Programmiersprache eingeübt, aber es stehen die Konzepte im Vordergrund, die unabhängig und von allgemeiner Natur sind, und nicht die Features und Feinheiten einer speziellen Programmiersprache. Die 14 Wochen des Wintersemesters sind lang, schränken aber dennoch die Stoffauswahl ein: Wir beschränken uns auf sequentielle Programmierung mit kurzen Exkursionen in die Algorithmik und in die formalen Sprachen; Konzepte paralleler, nebenläufiger oder verteilter Systeme werden später in höheren Semestern vermittelt.

Worte der Orientierung In der Schule wurde Ihnen das Wissen in kleinen wohldosierten Dosen verabreicht. Es war in der Regel klar, was gerade gelernt wurde und jedes Konzept wurde hinreichend lange eingeübt. Das ist an der Universität anders. Zum einen ist hier das Tempo der Wissenvermittlung höher. Zum anderen steht das selbstständige Lernen im Vordergrund. Sie müssen selbst das für Sie passende Lehrmaterial auswählen. Sie müssen selbst überprüfen, was Sie bereits gut und was noch nicht so gut verstanden haben, um dann gezielt an den Problemstellen zu arbeiten. Sie müssen lernen, kritisch mit sich selbst zu sein, eigene Fragen formulieren zu Dingen, die Sie nicht richtig verstehen und dann gezielt nach Antworten auf diese Fragen suchen, durch Selbstdenken, auf den Folien, im Skript, in Büchern, bei Ihren Mitstudierenden, Tutor*innen oder Dozent*innen — aber bitte *nicht*, zumindest nicht ohne einen sehr kritischen Blick im Internet. Wir machen vielfältige Angebote, um Sie bei diesem Prozess zu unterstützen. Nehmen Sie diese wahr!

Das benötigte Vorwissen insbesondere im Hinblick auf mathematische Sachverhalte ist minimal. Die mathematischen Grundlagen legen wir selbst in Kapitel 2. Anhang A.1 fasst mathematisches Basiswissen zusammen — sollten Ihnen zum Beispiel die Notation $\{1, 2\}$ oder $A \cup B$ nichts sagen, werden Sie dort fündig.

Neben den Folien und dem Skript stellen wir Ihnen Programme zum Ausprobieren und Experimentieren zur Verfügung — Hinweise der Form ***module*** *Values.Area* im Seitenrand verweisen auf die entsprechenden Dateien.

> *In den zehn bis 14 Tagen vor dem Rennen auf Kona kann man die Form fast nur noch kaputtmachen. Das ist so, als ob ich mir vor einer Klausur in den letzten Minuten den Lernstoff reinprügele — da bleibt nicht viel hängen.*
>
> — Patrick Lange (1986), *SZ (11. Oktober 2019)*

> *Ich möchte hier die im Laufe der Vorlesung mehrfach — nach meinem Eindruck größtenteils vergeblich — in verschiedener Form getroffene Feststellung wiederholen, daß tieferes Verständnis nicht ohne intensiven Arbeitseinsatz zu erreichen ist, schon gar nicht in einem Gewaltakt in wenigen Tagen oder Wochen. Es ist besser hier dem Rat des Apelles (von Plinius überliefert) zu folgen: Nulla dies sine linea.*
>
> — Gerd Wegner (1938), *Lineare Algebra für Informatiker*

Mahnende Worte „Grundlagen der Programmierung“ (INF-02-01-V-2) ist ein umfangreiches Modul, für das Sie 10 Leistungspunkte oder ECTS Credits erhalten (European Credit Transfer System). Im Vollzeitstudium wird davon ausgegangen, dass 60 Leistungspunkte pro akademischem Jahr gesammelt werden, was einem Aufwand von 1500 bis 1800 Stunden entspricht, also circa 45 bis 46 Wochen à 35 bis 40 Lernstunden. Für 10 Leistungspunkte müssen Sie entsprechend 11 bis 14 Stunden pro Woche für das Modul investieren. Dies ist der Natur nach nur ein Richtwert — Lernfortschritt misst sich nicht in Minuten oder Stunden, sondern in der Anzahl der „Aha-Erlebnisse“. Aber ich bin mir sicher: Wenn sie intensiv und kontinuierlich arbeiten, wird sich dieser Fortschritt auch einstellen. (Fleiß, nicht Intelligenz ist der entscheidende Faktor — der Einfluss der

Intelligenz auf den Lernerfolg wird gemeinhin überschätzt. Vergessen Sie nicht: Mit dem Abitur oder einem vergleichbaren Abschluss haben Sie bereits die Studierfähigkeit nachgewiesen.) Lassen Sie sich nicht entmutigen, wenn sich nicht sofort Erfolge einstellen — bleiben Sie am Ball. Im Studium ist eine solide Frustrationstolerenz nicht von Schaden.

Worte des Dankes Ein besonderer Dank gilt Andres Löh für seine Mitwirkung an der Konzeption der Vorlesung. Eine Vorlesung steht und fällt mit der Organisation und Durchführung des Übungsbetriebes. Ich kann mich glücklich schätzen, immer durch ein tolles Team unterstützt worden zu sein bzw. unterstützt zu werden: in Bonn durch Melanie Gnasa und Julia Kuck und in Kaiserslautern durch Sebastian Schweizer und Peter Zeller. Viele Korrekturleser*innen haben mitgeholfen, Fehler im Skript und auf den Vorlesungsfolien auszumerzen: Judith Stengel, Xenia Mechler, Minh Duc Duong (Minh Đức Dương), Timo Höcker (danke für die Farbschemata für Farbenfehlsichtige), Maurice Kohl und Tobias Zimmermann.

Eine Bitte zum Schluss Auf die Erstellung der Unterlagen wurde viel Zeit und Sorgfalt verwendet. Falls Ihnen dennoch Fehler im Skript oder auf den Vorlesungsfolien auffallen — Fehler inhaltlicher, grammatikalischer, typographischer oder historischer Natur oder Fehler in Programmen — berichten Sie bitte diese in einer kurzen Email an

support@harry-hacker.org

Ich werde mich bemühen, alle Fehler so schnell wie möglich zu korrigieren.

Ansonsten verbleibt mir nur, Ihnen viel Spass beim Lesen zu wünschen!

Ralf Hinze

Das Glück stellt sich ein, wenn dein Werk und deine Worte für dich und für andere von Nutzen sind.

— Buddha

1. Einführung⧹Rechnen und rechnen lassen

There are two things which I am confident I can do very well: one is an introduction to any literary work, stating what it is to contain, and how it should be executed in the most perfect manner; the other is a conclusion, shewing from various causes why the execution has not been equal to what the author promised to himself and to the public.

— Samuel Johnson (1709–1784), *Boswell Life vol. 1, p. 291 (1755)*

Man kann vorzüglich Rechnen lernen, ohne sich jemals zu fragen, was denn das Rechnen vom sonstigen Gebrauch des Verstandes unterscheidet. Wir stellen diese Frage jetzt und betrachten dazu das Rechnen so, wie es uns im Leben zuerst begegnet — als Umgang mit den Zahlen. Wir werden also die Natur des Rechnens an der Arithmetik studieren und dabei am Ende feststellen, dass die Zahlen bei weitem nicht das Einzige sind, womit wir rechnen können.

Zweifellos ist Rechnen ein besonderer Gebrauch des Verstandes. Eine gewisse Ahnung vom Unterschied zwischen Denken im Allgemeinen und Rechnen im Besonderen hat jeder, der einmal mit seiner Mathematiklehrer*in darüber diskutiert hat, ob der Fehler in seiner Klassenarbeit „bloß“ als Rechenfehler oder aber als „logischer“ Fehler einzuordnen sei.

Richtig Rechnen heißt Anwendung der Rechenregeln für Addition, Multiplikation usw. Dies allein garantiert das richtige Ergebnis — und neben ihrer Beachtung und Anwendung ist als einzige weitere Verstandesleistung die Konzentration auf diese eintönige Tätigkeit gefragt. Kein Wunder, dass nur wenige Menschen zum Zeitvertreib siebenstellige Zahlen multiplizieren oder die Zahl π auf hundert Stellen bestimmen!

Damit soll nicht etwa das Rechnen diffamiert werden — es entspricht so gerade der Natur dessen, worum es dabei geht, nämlich den Zahlen. Diese sind Größen, abstrakte Quantitäten ohne sonstige Eigenschaft. Bringe ich sie in Verbindung, ergeben sich neue, andere Größen, aber keine andere Qualität. Der Unterschied von 15 und 42 ist 27, und daraus folgt — nichts. Die mathematische Differenz zweier Größen ist, so könnte man sagen, der unwesentliche Unterschied.

Nur zum Vergleich: Stelle ich meinen Beitrag zum Sportverein und meine Einkommensteuer einander gegenüber, so ergibt sich *auch* ein Unterschied im Betrag. Daneben aber auch einer in der Natur der Empfänger, dem Grad der Freiwilligkeit, meiner Einflussmöglichkeiten auf die Verwendung dieser Gelder und vieles andere mehr. Dies sind wesentliche Unterschiede; aus ihnen folgt allerhand. Unter anderem erklären sie auch die Differenz der Beträge und rechnen sie nicht bloß aus.

Für das Rechnen jedoch spielt es keine Rolle, *wovon* eine Zahl die Größe angibt. Jede sonstige Beschaffenheit der betrachteten Objekte ist für das Rechnen mit ihrer Größe ohne Belang. Deshalb kann das Rechnen in starre Regeln gegossen werden und nimmt darin seinen überraschungsfreien Verlauf. Die kreative gedankliche Leistung liegt woanders: Sie manifestiert sich in der Art und Weise, wie man die abstrakten Objekte der Zahlen konkret aufschreibt. Wir sind daran gewöhnt, das arabische Stellenwertsystem zu verwenden. Dieses hat sich im Laufe der Geschichte durchgesetzt, da sich darauf relativ einfach Rechenregeln definieren und auch anwenden lassen. Das römische Zahlensystem mit seinen komplizierten Wertigkeiten und der Vor- und Nachstellung blieb auf der Strecke: Den Zahlen XV und XLII sieht man den Unterschied XXVII nicht an. Insbesondere das Fehlen eines Symbols für die Null macht systematisches Rechnen unmöglich — gerade wegen der Null war das Rechnen mit den arabischen Zahlen im Mittelalter von der Kirche verboten [Kap00].

Einstweiliges Fazit: Das Rechnen ist seiner Natur nach eine äußerliche, gedankenlose, in diesem Sinne mechanische Denktätigkeit. Kaum hatte die Menschheit Rechnen gelernt, schon tauchte die naheliegende Idee auf, das Rechnen auf eine Maschine zu übertragen. Die Geschichte dieser Idee — von einfachen Additionsmaschinen bis hin zu den ersten programmgesteuerten Rechenautomaten — ist an vielen Stellen beschrieben worden und soll hier nicht ausgeführt werden.[1] An ihrem Endpunkt steht der Computer, der bei entsprechender Programmierung beliebig komplexe arithmetische Aufgaben lösen kann — schneller als wir, zuverlässiger als wir und für uns bequemer.

Mit der Existenz dieses Gerätes eröffnen sich plötzlich ganz neue Möglichkeiten: Jetzt, wo wir rechnen *lassen* können, wird es interessant, Aufgaben in Rechenaufgaben zu verwandeln, die es von Natur aus nicht sind und die wir mit unserem Verstand auch nie so behandeln würden. Es gilt, zu einer Problemstellung die richtigen Abstraktionen zu finden, sie durch Formeln und Rechengesetze, genannt Datenstrukturen und Algorithmen, so weitgehend zu erfassen, dass wir dem Ergebnis der Rechnung eine Lösung des Problems entnehmen können (siehe Abbildung 1.1). Wir bilden abstrakte Modelle der konkreten Wirklichkeit, die diese im Falle der Arithmetik perfekt, in den sonstigen Fällen meist nur annähernd wiedergeben. Die Wirklichkeitstreue dieser Modelle macht den Reiz und die Schwierigkeit dieser Aufgabe und die Verantwortung der Informatiker*in bei der Software-Entwicklung aus.

Die eigentümliche Leistung der Informatik ist es also, Dinge in Rechenaufgaben zu verwandeln, die es von Natur aus nicht sind. Die Fragestellung „Was können wir rechnen lassen?“ ist neu in der Welt der Wissenschaften — deshalb hat sich die Informatik nach der Konstruktion der ersten Rechner schnell aus dem Schoße der Mathematik heraus zu einer eigenständigen Disziplin entwickelt. Die Erfolge dieser Bemühung sind faszinierend, und nicht selten, wenn auch nicht ganz richtig, liest man darüber in der Zeitung: „Computer werden immer schlauer“.

[1] Eine beeindruckende Sammlung mechanischer und elektronischer Rechenmaschinen findet man im Bonner Arithmeum (`www.arithmeum.de`). Konrad Zuses Rechenmaschine Z23 ist im Fraunhofer-Institut für Experimentelles Software Engineering ausgestellt (`www.iese.fraunhofer.de`).

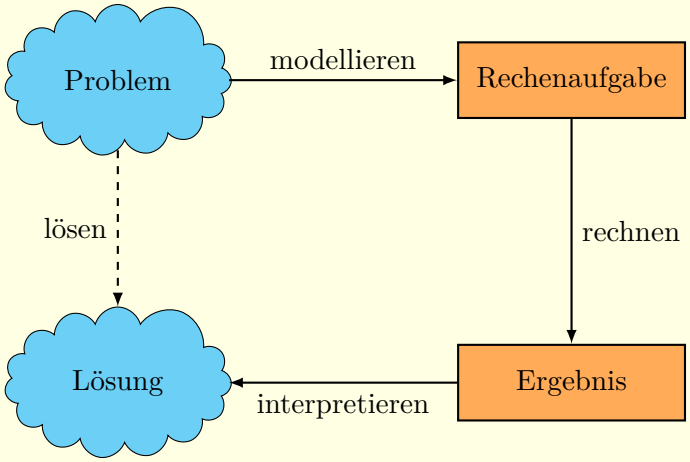


Abbildung 1.1.: Modellbildung in der Informatik.

1.1. Die Aufgabengebiete der Informatik

Informatik ist die Wissenschaft vom maschinellen Rechnen. Daraus ergeben sich verschiedene Fragestellungen, in die sich die Informatik aufgliedert (siehe Abbildung 1.2).

Zunächst muss sie sich um die Rechenmaschine selbst kümmern. In der *Technischen Informatik* geht es um Rechnerarchitektur und Rechnerentwurf, wozu auch die Entwicklung von Speichermedien, Übertragungskanälen, Sensoren usw. gehört. Der Fortschritt dieser Disziplin besteht in immer schnelleren und kleineren Rechnern bei fallenden Preisen. In ihren elementaren Operationen dagegen sind die Rechner heute noch so primitiv wie zur Zeit ihrer Erfindung. Diese für den Laien überraschende Tatsache erfährt ihre Erklärung in der theoretischen Abteilung der Informatik.

Mit der Verfügbarkeit der Computer erfolgt eine Erweiterung des Begriffs „Rechnen“. Das Rechnen mit den guten alten Zahlen ist jetzt nur noch ein hausbackener Sonderfall. Jetzt werden allgemeinere Rechenverfahren entwickelt, genannt Algorithmen, die aus Eingabe-Daten die Ausgabe-Daten bestimmen. Aber — was genau lässt sich rechnen, und was nicht? Mit welcher Maschine? Die *Theoretische Informatik* hat diese Frage beantwortet und gezeigt, dass es nicht berechenbare Probleme[2] gibt, Aufgaben, zu deren Lösung es keinen Algorithmus gibt. Zugleich hat sie gezeigt, dass alle Rechner prinzipiell gleichmächtig sind, sofern sie über einige wenige primitive Operationen verfügen. Weiterhin erweisen sich die berechenbaren Aufgaben als unterschiedlich aufwändig, so dass die Komplexitätstheorie heute ein wichtiges Teilgebiet der Theoretischen Informatik darstellt.

[2] Man kann solche Aufgaben auch als Ja/Nein-Fragestellungen formulieren und nennt sie dann „formal unentscheidbare“ Probleme. Lässt man darin das entscheidende Wörtchen „formal“ weg, eröffnen sich tiefsinnige, aber falsche Betrachtungen über Grenzen der Erkenntnis.

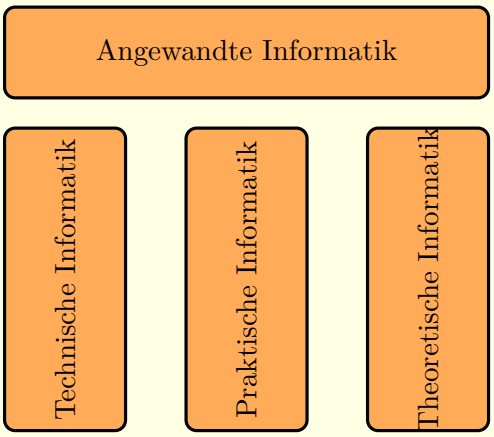


Abbildung 1.2.: Die Aufgabengebiete der Informatik.

Zwischen den prinzipiellen Möglichkeiten des Rechnens und dem Rechner mit seinen primitiven Operationen klafft eine riesige Lücke. Die Aufgabe der *Praktischen Informatik* ist es, den Rechner für Menschen effektiv nutzbar zu machen. Die sogenannte „semantische Lücke“ wird Schicht um Schicht durch Programmiersprachen auf immer höherer Abstraktionsstufe geschlossen. Heute können wir Algorithmen auf der Abstraktionsstufe der Aufgabenstellung entwickeln und programmieren, ohne die ausführende Maschine überhaupt zu kennen. Dies ist nicht nur bequem, sondern auch wesentliche Voraussetzung für die Übertragbarkeit von Programmen zwischen verschiedenen Rechnern. Ähnliches gilt für die Benutzung der Rechner (Betriebssysteme, Benutzungsoberflächen), für die Organisation großer Datenmengen (Datenbanken) und sogar für die Kommunikation der Rechner untereinander (Rechnernetze, Verteilte Systeme). Das World Wide Web (WWW) ist das bekannteste Beispiel für das Schließen der semantischen Lücke: Es verbindet einfachste Benutzbarkeit mit globalem Zugriff auf Informationen und andere Ressourcen in aller Welt. Heute bedarf es nur eines gekrümmten Zeigefingers, um tausend Computer in aller Welt für uns arbeiten zu lassen.

In der *Angewandten Informatik* kommt endlich der Zweck der ganzen Veranstaltung zum Zuge. Sie wendet die schnellen Rechner, effizienten Algorithmen, die höheren Programmiersprachen und ihre Übersetzer, die Datenbanken, Rechnernetze usw. an, um Aufgaben jeglicher Art sukzessive immer weiter und vollkommener zu lösen. Je weiter die Informatik in die verschiedensten Anwendungsgebiete vordringt, desto spezifischer werden die dort untersuchten Fragen. Zur Zeit erleben wir, dass sich interdisziplinäre Anwendungen als eigenständige Disziplinen von der „Kerninformatik“ abspalten. Wirtschaftsinformatik und Bioinformatik sind zwei Protagonisten dieser Entwicklung. Während es früher Jahrhunderte gedauert hat, bis aus den Erfolgen der Physik zunächst die anderen Natur-, dann die Ingenieurwissenschaften entstanden, so laufen heute solche

Entwicklungen in wenigen Jahrzehnten ab.

Der Vormarsch der Informatik geht in allen Lebensbereichen voran: Aus Textverarbeitung wird DeskTop Publishing, aus Computerspielen wird Virtual Reality, und aus gewöhnlichen Bomben werden „intelligent warheads“. Wie gesagt: die Rechner werden immer schlauer. Und wir?

1.2. Einordnung der Informatik in die Wissenschaftsfamilie

Wir haben gesehen, dass die Grundlagen der Informatik aus der Mathematik stammen, während der reale Ausgangspunkt ihrer Entwicklung die Konstruktion der ersten Rechner war. Mutter die Mathematik, Vater der Computer — wissenschaftsmoralisch gesehen ist die Informatik als ein Bastard aus einer Liebschaft des reinen Geistes mit einem technischen Gerät entstanden.

Die Naturwissenschaft untersucht Phänomene, die ihr ohne eigenes Zutun gegeben sind. Die Gesetze der Natur müssen entdeckt und erklärt werden. Die Ingenieurwissenschaften wenden dieses Wissen an und konstruieren damit neue Gegenstände, die selbst wieder auf ihre Eigenschaften untersucht werden müssen. Daraus ergeben sich neue Verbesserungen, neue Eigenschaften und so weiter.

Im Vergleich dieser beiden Abteilungen der Wissenschaft steht die Informatik eher den Ingenieurwissenschaften nahe: Auch sie konstruiert Systeme, die — abgesehen von denen der Technischen Informatik — allerdings immateriell sind. Wie die Ingenieurwissenschaften untersucht sie die Eigenschaften ihrer eigenen Konstruktionen, um sie weiter zu verbessern. Wie bei den Ingenieurwissenschaften spielen bei der Informatik die Aspekte der Zuverlässigkeit und Lebensdauer ihrer Produkte eine wesentliche Rolle.

Eine interessante Parallele besteht auch zwischen Informatik und Rechtswissenschaft. Die Informatik konstruiert abstrakte Modelle der Wirklichkeit, die dieser möglichst nahe kommen sollen. Das Recht schafft eine Vielzahl von Abstraktionen konkreter Individuen. Rechtlich gesehen sind wir Mieter*innen, Student*innen, Erziehungsberechtigte, Verkehrsteilnehmer*innen usw. Als solche verhalten wir uns entsprechend der gesetzlichen Regeln. Die Programmierer*in bildet reale Vorgänge in formalen Modellen nach, die Jurist*in muss konkrete Vorgänge unter die relevanten Kategorien des Rechts subsumieren. Die Analogie endet allerdings, wenn eine Diskrepanz zwischen Modell und Wirklichkeit auftritt: Bei einem Programmfehler behält die Wirklichkeit recht, bei einem Rechtsbruch das Recht.

Eine wissenschaftshistorische Betrachtung der Informatik und ihre Gegenüberstellung mit der hier gegebenen deduktiven Darstellung ist interessant und sollte zu einem späteren Zeitpunkt, etwa zum Abschluss des Informatikstudiums, nachgeholt werden. Interessante Kapitel dieser Geschichte sind das Hilbert'sche Programm, die Entwicklung der Idee von Programmen als Daten, der Prozess der Loslösung der Informatik von der Mathematik, die Geschichte der Programmiersprachen, die Revolution der Anwendungssphäre durch das Erscheinen der Mikroprozessoren und vieles andere mehr.

1.3. Überblick über die Vorlesung

In der Vorlesung beschäftigt uns das „Rechnen lassen“: Wie können wir Aufgaben in Rechenaufgaben verwandeln? Wie können wir einen Rechner programmieren, so dass er diese Aufgaben ohne weiteres Zutun erledigt? Thematisch gehört die Vorlesung somit zur „Praktischen Informatik“. Hier und da werden wir uns allerdings in die anderen Gebiete der Informatik vorwagen, insbesondere in die „Theoretische Informatik“.

Paradoxerweise wird der Rechenknecht selbst, der Computer, bei unserem Streifzug kaum in Erscheinung treten. Dies liegt an der schon angesprochenen semantischen Lücke: zu primitiv sind die Operationen, die ein Computer werksseitig mitbringt. Seine Stärke liegt in der extrem schnellen Ausführung von wenigen einfachen Operationen, nicht in der Bereitstellung ausdrucksstarker Operationen. Aus diesem Grund werden wir im Laufe der Vorlesung unsere eigene Programmiersprache entwickeln: Mini–F#. Eine Sprache, in der sich Rechenregeln bequem und vor allem problemnah formulieren lassen. Dabei geben wir uns der Illusion hin, dass ein entsprechend moderner Rechner diese Programme auch ausführen kann. Tatsächlich ist dies auch möglich: nur nicht nativ (in Hardware), sondern mit Hilfe vieler anderer Programme, die die semantische Lücke schließen (in Software). Der Unterschied ist für die Benutzer*in nicht auszumachen, allenfalls wird die phänomenale Rechengeschwindigkeit moderner Rechner etwas gebremst.

Die Einführung einer eigenen Programmiersprache verfolgt natürlich einen Hintergedanken: Sie erlaubt es uns, viele typische Fragestellung der Praktischen Informatik zu motivieren und zu untersuchen: Wie lässt sich die äußere Form von Programmen festlegen? Wie kann man die Bedeutung eines Programms präzise definieren? Wie entwickelt man systematisch ein Programm? Wann ist ein Programm korrekt? usw. Auf diese Weise wird die Programmiersprache selbst zum Objekt des Studiums. Wie gesagt: die Informatik steht den Ingenieurwissenschaften nahe.

Natürlich ist Mini–F#— wie der Name bereits andeutet — keine völlig neue Programmiersprache; sie steht in der Tradition der Algol-Sprachfamilie und ist eng angelehnt an die Sprache F#, die ihrerseits auf Sprachen wie Standard ML oder OCaml aufbaut. F# ist Mitglied der .NET Sprachfamilie, die weitere Programmiersprachen wie Visual Basic und C# umfasst. F# ist eine sogenannte *Multiparadigmensprache*: Sie unterstützt funktionale, imperative und objektorientierte Programmierung — auf diese drei Programmierparadigmen werden wir im Laufe der Vorlesung intensiv eingehen. Mini–F# ist ein sorgfältig ausgewählte Teilmenge von F# — den gesamten Sprachumfang von F# vorzustellen, ist weder zeitlich möglich, noch wünschenswert. Bei der Entwicklung von Mini–F# standen zwei miteinander konkurrierende Ziele im Vordergrund: Mini–F# sollte einfach zu lernen sein und trotzdem einen guten Überblick über bestehende Sprachfamilien und -kulturen vermitteln. Wir hoffen, dass der Balanceakt geglückt ist und Mini–F# sowohl für Anfänger als auch für Fortgeschrittene hinreichend viel Denkstoff bietet. Idealerweise ist jedes Mini–F# Programm ein gültiges F# Programm. Diesem Ideal kommen wir nahe, ganz erreicht wird es nicht: gelegentlich vereinfachen wir etwas, ganz selten mogeln wir.

Im Einzelnen ist das vorliegende Skript zur Vorlesung wie folgt gegliedert.

Bevor wir mit dem Programmieren und der wissenschaftlichen Betrachtung des Pro-

grammierens loslegen können, benötigen wir noch etwas Rüstzeug. Dies wird kurz und knapp in Kapitel 2 vermittelt.

Wir nehmen die Metapher des Rechnens zunächst sehr wörtlich: Ein Programm ist einfach ein Ausdruck; rechnen wir den Ausdruck aus, erhalten wir das Ergebnis des Programms. Die elementaren Konstrukte für das Rechnen führen wir in Kapitel 3 ein. Am Ende des Kapitels steht eine Programmiersprache, mit der man die prinzipiellen Möglichkeiten eines Rechners ausschöpfen kann. Das Wörtchen „prinzipiell“ ist dabei bedeutsam: Für praktische Belange benötigen wir erheblich mehr Komfort; für diesen wird in den darauffolgenden Kapiteln gesorgt.

Rechner verarbeiten Daten: Personaldaten, Börsendaten, Wetterdaten usw. Was früher in Karteikästen und Aktenordnern verwahrt und aufbewahrt wurde, findet sich heute rechnergerecht aufbereitet auf Speichermedien wieder. Daten machen einen wesentlichen Teil der Modelle aus, die Informatiker*innen von der Wirklichkeit bilden. Wie man Daten repräsentiert und strukturiert, davon handelt Kapitel 4.

Kenntnis des Vokabulars und der Grammatik einer Sprache macht noch keinen Dichter. Dazu gehört mehr: Kenntnis von Rhythmus und Reim, von Takt und Technik und natürlich Kreativität. Beim Programmieren ist es ähnlich: die kreative, gedankliche Leistung geht dort in das Aufstellen von Rechenregeln. Programmiertechniken, ein Thema von Kapitel 5, helfen Rechenregeln systematisch zu entwickeln. Aufgaben wie Rechenaufgaben lassen sich auf vielfältige Art und Weise lösen. Die resultierenden Rechenregeln können sehr unterschiedlich sein und die Ressourcen eines Rechners, Zeit und Platz, unterschiedlich stark beanspruchen. Auch hier sind Programmiertechniken von Nutzen: Sie helfen Programme zu verbessern, sie schneller und platzsparender zu machen. Werden viele Daten verwaltet, muss man sich überlegen, wie man sie geeignet verknüpft, um auf sie zugreifen und manipulieren zu können: Aus Daten werden Datenstrukturen. Einige einfache Datenstrukturen werden wir ebenfalls in Kapitel 5 kennenlernen.

Bevor wir den Computer rechnen lassen können, müssen Formeln und Rechenregeln, sprich Programme, dem Computer kommuniziert werden. Dabei ist bedeutsam, dass sich der Computer in seinen Aktionen allein von der äußeren Form der Programme leiten lässt. Für einen menschlichen Leser mag es keine große Rolle spielen, ob etwa in einem Roman an einer bestimmten Stelle ein Komma oder ein Semikolon steht. Wird hingegen in einem Programm ein Komma durch ein Semikolon ersetzt, kann sich die Bedeutung des Programms wesentlich, vielleicht sogar dramatisch verändern. Aus diesem Grund ist es wichtig, die äußere Form von Programmen präzise und eindeutig festzulegen. Kapitel 6 zeigt, wie dies gemacht werden kann und erzählt dabei eine der großen Erfolgsgeschichten der Informatik.

Kapitel 7 erweitert die Idee des Rechnens. Wurde bis dato ein Programm um des Ergebnisses willen ausgerechnet, kann ein Programm jetzt zusätzlich Effekte haben: Eingaben werden getätigt, Ausgaben erfolgen, Motoren werden gesteuert, Raketen werden abgeschossen usw. Neben diesen, zum Teil spektakulären, externen Effekten betrachten wir auch interne Effekte: Rechnungen werden abgebrochen und wieder aufgenommen, Rechnungen hängen von einem Gedächtnis ab und Rechnungen verändern das Gedächtnis.

Computer dringen immer weiter in unser tägliches Leben vor; immer mehr Aufgaben

werden in Rechenaufgaben verwandelt. Die Aufgaben werden zudem anspruchsvoller und umfangreicher. Komplexe Aufgaben resultieren in komplexen Rechenregeln: Aus Programmen werden Softwaresysteme. Um die Entwicklung großer Systeme meistern zu können, muss man sich etwas einfallen lassen. Kapitel 8 zeigt auf, wie sich Programme organisieren lassen: Wie man Rechenregeln konzeptionell zusammenfasst, sie zu größeren Paketen verschnürt und wie man mit diesen Paketen rechnet.

2. Grundlagen ⧹ Vor dem Rechnen

Angling may be said to be so like the mathematics,
that it can never be fully learnt.
— Izaak Walton (1593–1683), *The Compleat Angler (1653)*

Wir haben in der Einleitung angesprochen, dass wir unsere Programmiersprache Mini–F# selbst zum Objekt des Studiums machen. Dazu brauchen wir einige wenige Grundlagen, die wir in diesem Kapitel legen.

Im Einzelnen: Wir setzen elementare Mathematikkenntnisse insbesondere Kenntnisse der naiven Mengenlehre voraus. (Wenn Sie merken, dass Ihnen Vorwissen fehlt, werfen Sie doch einen Blick in den Anhang A.1.) Alle weiteren mathematischen Konzepte führt Abschnitt 2.1 ein. Abschnitt 2.2 klärt, was das Studium von Programmiersprachen involviert und motiviert die restlichen Abschnitte.

2.1. Endliche Abbildungen und Sequenzen

Endliche Abbildungen Bei der Formalisierung von Mini–F# werden wir häufig Gebrauch von sogenannten *endlichen Abbildungen* machen (engl. finite maps). Wie der Name andeutet, bildet eine endliche Abbildung nur endlich viele Elemente aus ihrem Ursprungsbereich auf Elemente des Wertebereichs ab. Der Wertebereich selbst kann unendlich groß sein.

Sind X und Y Mengen, dann bezeichnet $X \to_{\text{fin}} Y$ die Menge aller endlichen Abbildungen von X nach Y. Ist $\varphi \in X \to_{\text{fin}} Y$, dann bezeichnet $\mathit{dom}\ \varphi \subseteq X$ die Menge aller Elemente aus X, auf denen φ definiert ist, den sogenannten *Definitionsbereich* von φ. Der Definitionsbereich $\mathit{dom}\ \varphi$ einer endlichen Abbildung muss endlich sein. Die Anwendung einer endlichen Abbildung φ auf ein Element x notieren wir mit $\varphi(x)$.

Die endliche Abbildung

$$\{\mathit{Anja} \mapsto \mathit{Spagetti}, \mathit{Lisa} \mapsto \mathit{Tortellini}, \mathit{Florian} \mapsto \mathit{Spagetti}, \mathit{Ralf} \mapsto \mathit{Saltimbocca}\}$$

ordnet zum Beispiel Personen ihre Lieblingsgerichte zu. Ähnlich wie für Abbildungen gilt: jedem Element x aus $\mathit{dom}\ \varphi$ wird genau ein y aus Y zugeordnet, aber Elemente aus Y können durchaus mehrfach zugeordnet werden.

Um endliche Abbildungen zu konstruieren bzw. zu manipulieren, verwenden wir die folgenden Operationen:

- die *leere Abbildung* $\varnothing$ mit
  - $\mathit{dom}\ \varnothing = \emptyset$;

- die *einelementige Abbildung* (auch: *Bindung*) $\{x \mapsto y\}$ mit
  - $dom\,\{x \mapsto y\} = \{x\}$ und
  - $\{x \mapsto y\}(x) = y$;
- die *Erweiterung* von φ_1 um φ_2 notiert φ_1, φ_2 (*Kommaoperator*) mit
  - $dom\,(\varphi_1, \varphi_2) = dom\,\varphi_1 \cup dom\,\varphi_2$ und
  - $(\varphi_1, \varphi_2)(x) = \begin{cases} \varphi_2(x) & \text{falls } x \in dom\,\varphi_2 \\ \varphi_1(x) & \text{sonst} \end{cases}$
- die *Einschränkung* von φ auf $dom\,\varphi \setminus A$ notiert $\varphi \setminus A$ mit
  - $dom\,(\varphi \setminus A) = dom\,\varphi \setminus A$ und
  - $(\varphi \setminus A)(x) = \varphi(x)$.

Man sieht, dass eine endliche Abbildung jeweils durch zwei Angaben definiert wird: ihrem Definitionsbereich und der eigentlichen Zuordnung von Elementen aus dem Definitionsbereich zu Elementen aus dem Wertebereich. (Nur bei der leeren Abbildung erübrigt sich die Angabe der Zuordnung.)

Der Kommaoperator vereinigt im Prinzip zwei endliche Abbildungen; enthalten beide Abbildungen Bindungen für das gleiche Element, so wird der „rechten“ Bindung der Vorzug gegeben:

$$\{Anja \mapsto Spagetti, Ralf \mapsto Pizza\}, \{Ralf \mapsto Eis\} = \{Anja \mapsto Spagetti, Ralf \mapsto Eis\}$$
$$\{Ralf \mapsto Eis\}, \{Anja \mapsto Spagetti, Ralf \mapsto Pizza\} = \{Anja \mapsto Spagetti, Ralf \mapsto Pizza\}$$

Somit ist der Kommaoperator nicht kommutativ: Die endliche Abbildung φ_1, φ_2 ist in der Regel verschieden von φ_2, φ_1.

Aber der Kommaoperator ist assoziativ: $(\varphi_1, \varphi_2), \varphi_3 = \varphi_1, (\varphi_2, \varphi_3)$, siehe Aufgabe 2.2. Deshalb können wir bei mehreren geschachtelten Anwendungen des Kommaoperators die Klammern auslassen und schreiben kurz $\varphi_1, \varphi_2, \varphi_3$. Zusätzlich verwenden wir analog zur Notation von Mengen $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ als Abkürzung für $\{x_1 \mapsto y_1\}, \ldots, \{x_n \mapsto y_n\}$ (wiederholte Anwendung des Kommaoperators).

Sequenzen Mit Hilfe endlicher Abbildungen lassen sich *Sequenzen*, endliche Folgen von Elementen, modellieren: wir nummerieren die Elemente von links nach rechts beginnend mit 0 durch; die Sequenz wird dann durch eine endliche Abbildung repräsentiert, die die „Hausnummer“ auf das jeweilige Element abbildet. Die Sequenz *Anja Florian Lisa Ralf* wird zum Beispiel durch $\{0 \mapsto Anja, 1 \mapsto Florian, 2 \mapsto Lisa, 3 \mapsto Ralf\}$ repräsentiert.

Eine Sequenz s von Elementen aus einer gegebenen Menge A ist somit eine endliche Abbildung des Typs $\mathbb{N} \to_{\text{fin}} A$ mit $dom\,s = \mathbb{N}_n$. Dabei ist $\mathbb{N}_n = \{0, \ldots, n-1\}$ ein Anfangsabschnitt der natürlichen Zahlen. Die Grundmenge A wird auch manchmal *Alphabet* genannt. Die Menge aller Sequenzen über A notieren wir mit A^*. Ist $dom\,s = \mathbb{N}_n$, dann heißt n die Länge von s, notiert $len\,s = n$.

Um Sequenzen zu konstruieren, verwenden wir die folgenden Operationen:

- die *leere Sequenz* ϵ mit
  - $dom\ \epsilon = \mathbb{N}_0$;
- ist $a \in A$, dann verwenden wir oft das Element a selbst als Abkürzung für die *einelementige Sequenz* $\{0 \mapsto a\}$;
- die *Konkatenation* von s_1 und s_2 notiert $s_1 \cdot s_2$ mit
  - $dom\ (s_1 \cdot s_2) = dom\ s_1 \cup \{\, i + len\ s_1 \mid i \in dom\ s_2 \,\}$ und
  - $(s_1 \cdot s_2)(i) = \begin{cases} s_1(i) & \text{falls } i \in dom\ s_1 \\ s_2(i - len\ s_1) & \text{sonst} \end{cases}$
- die *n-fache Wiederholung* von s notiert s^n mit $s^0 = \epsilon$ und $s^{n+1} = s \cdot s^n$.

Zum Beispiel ist $\{0 \mapsto A, 1 \mapsto n\} \cdot \{0 \mapsto j, 1 \mapsto a\} = \{0 \mapsto A, 1 \mapsto n, 2 \mapsto j, 3 \mapsto a\}$. Kürzen wir die resultierende Sequenz mit s ab, dann ist $s(0) = A$ und $s(3) = a$. Da die Konkatenation assoziativ ist, können wir bei geschachtelten Anwendungen die Klammern auslassen: $L \cdot i \cdot s \cdot a$ steht zum Beispiel für die Sequenz $\{0 \mapsto L, 1 \mapsto i, 2 \mapsto s, 3 \mapsto a\}$. Das Symbol für die Konkatenation wird auch oft ausgelassen: $L \cdot i \cdot s \cdot a$ verkürzt sich dann zu $L\ i\ s\ a$.

2.2. Syntax und Semantik

> *Semantics is a strange kind of applied mathematics; it seeks profound definitions rather than difficult theorems. The mathematical concepts which are relevant are immediately relevant. Without any long chains of reasoning, the application of such concepts directly reveals regularity in linguistic behaviour, and strengthens and objectifies our intuitions of simplicity and uniformity.*
>
> — J.C. Reynolds (1935), *(1980)*

Wenn wir eine Programmiersprache präzise beschreiben wollen, müssen wir zwei Dinge festlegen:

- die äußere Form von Programmen, die *Syntax*, und
- deren Bedeutung, die *Semantik*.

Betrachten wir ein Beispielprogramm (ein Teil eines Mini–F# Programms):

```
4711*  (a11 (* speed *) + 815 )
```

Mikroskopisch gesehen besteht das Programm aus einer Folge von Zeichen: der Ziffer `4`, gefolgt von der Ziffer `7`, gefolgt von der Ziffer `1`, gefolgt von der Ziffer `1`, gefolgt von einem Asteriskus `*`, gefolgt von einem Leerzeichen usw. Die Zeichenfolge entspricht im Wesentlichen der Folge von Tasten, die wir betätigen, um das Programm einzugeben.

Als menschlicher Leser sind wir gewohnt — bzw. durch jahrelanges Training geschult — mehrere Zeichen, etwa eine Folge von Ziffern, zu einer Einheit zusammenzufassen.

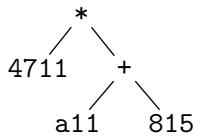
Die ersten vier Zeichen werden die meisten unwillkürlich als Zahl viertausendsiebenhundertundelf lesen. Wie Zeichen zu größeren Einheiten, sogenannten *Lexemen* (engl. tokens), zusammengefasst werden, wird in der *lexikalischen Syntax* einer Programmiersprache festgelegt. Ohne auf Details einzugehen, wird das obige Beispiel in Mini–F# wie folgt gruppiert:

4711	*	(	a11	+	815	)

Nicht alle Zeichen sind für den Rechner gedacht: `(* speed *)` ist ein Kommentar, der sich an den menschlichen Leser richtet. Ebenso sind Leerzeichen für das Rechnen irrelevant. Lassen wir beides weg, besteht das Programm aus sieben Lexemen.

Nun stellen nicht alle Folgen von Lexemen ein gültiges Programm dar: die Folge `) * 4711 815 + ( a11` umfasst die gleichen Lexeme, ist aber kein Mini–F# Programm. In der sogenannten *kontextfreien Syntax* einer Programmiersprache wird festgelegt, welche Folgen von Lexemen gültige Programme sind und welche nicht. Die lexikalische und die kontextfreie Syntax bilden zusammen die *konkrete Syntax* einer Programmiersprache. Konkret, weil sie genau ausbuchstabiert, wie Programme konkret auszusehen haben. *Wie* die konkrete Syntax definiert wird (und wann sie „kontextfrei“ ist), damit beschäftigen wir uns zu einem späteren Zeitpunkt, in Kapitel 6.

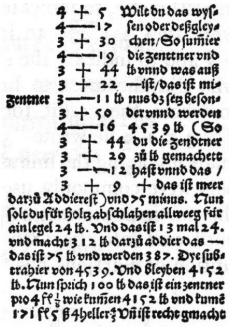
Wir haben schon angekündigt, dass wir nicht nur *mit* Mini–F# arbeiten wollen, sondern auch *über* die Sprache reden wollen. Will man eine Programmiersprache selbst zum Gegenstand des Diskurses machen, dann ist die konkrete Syntax als Ausgangspunkt ungeeignet: Sie ist technischen Einschränkungen unterworfen, sie ist das Produkt vieler Kompromisse und spiegelt nicht zuletzt auch den persönlichen Geschmack der Sprachdesigner*innen wider. Für solche Betrachtungen ist die hierarchische Struktur eines Programms relevant. Die hierarchische Struktur unseres Beispielprogramms sieht in etwa so aus:

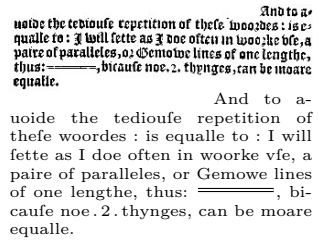


Das Diagramm verdeutlicht, dass sich der Ausdruck aus mehreren Teilausdrücken zusammensetzt. Man spricht auch von einem *Rechenbaum* oder allgemein von einem *abstrakten Syntaxbaum*. Abstrakt deshalb, weil von den technischen Details der konkreten Syntax abstrahiert wird: In der linearen Folge von Lexemen werden zum Beispiel Klammern verwendet, um Teilausdrücke zu gruppieren. Klammern sind hingegen in der *abstrakten Syntax* nicht notwendig; die Baumstruktur legt genau fest, welcher Operand zu welchem Operator gehört.

Der Unterschied zwischen konkreter und abstrakter Syntax wird noch deutlicher, wenn man sich die konkrete Syntax des Rechenbaums in anderen Programmiersprachen anschaut. In der Sprache *Scheme* zum Beispiel werden die Operatoren *vor* die Operanden geschrieben: `(* 4711 (+ a11 815))`. In der Sprache *PostScript* ist es genau anders herum; die Operatoren wandern *hinter* die Operanden: `4711 a11 815 + *`. Zudem müssen

Der deutsche Mathematiker Johannes Widmann verfasste in jungen Jahren ein Buch über kaufmännisches Rechnen („Mercantile Arithmetic“ oder „Behende und hüpsche Rechenung auff allen Kauffmanschafft“). Das 1489 in Leibzig erschienene Werk gilt als das erste gedruckte Buch, in dem die Symbole „+“ und „−“ verwendet werden — allerdings in der Funktion als Vorzeichen, nicht als binäre Operatoren. In dieser Bedeutung wurden sie erst 1518 von dem deutschen Mathematiker und Astronomen Heinrich Schreiber (1492–1526) eingeführt.


4 + 5 Wilebn das wyſ-
4——17 ſen oder deßgley-
3 + 30 chen/So ſumier
4——19 die zenetner vnd
3 + 44 lb vnnd was auß
3 + 22 —iſt/das iſt mi-
Zentner 3——11 lb nus dz ſetz beſon-
3 + 50 der vnnd werden
4——16 4539 lb (So
3 + 44 du die zendtner
3 + 29 zů lb gemachett
3 —v—12 haſt vnnd das /
3 + 9 + das iſt meer
darzů Addiereſt)vnd 75 minus. Nun
ſolt du für Holtz abſchlahen allweeg für
ain legel 24 lb. Vnd das iſt 13 mal 24.
vnd macht 312 lb darzů addier das —
das iſt 75 lb vnd werden 387. Dye ſuß-
trahier von 4539. Vnd bleyben 4152
lb. Nun ſprich 100 lb das iſt ein zentner
p104 ff[e] 1/2 wie kumen 4152 lb vnd kume
171 ff[e] 5 ß 4 heller 1/2 Vñ iſt rechegmache


And to auoide the tedioufe repetition of thefe woordes : is equalle to : I will fette as I doe often in woorke vfe, a paire of paralleles, or Gemowe lines of one lengthe, thus: ═══, bicaufe noe. 2. thynges, can be moare equalle.

Im englischsprachigen Raum wurden die Symbole von dem walisischen Arzt und Mathematiker Robert Recorde bekannt gemacht. In seinem 1557 erschienenen Buch mit dem beindruckenden Titel „The whetstone of witte, whiche is the seconde parte of Arithmetike: containyng thextraction of Rootes: The Coßike practise, with the rule of Equation: and the woorkes of Surde Nombers“ erschien auch das erste Mal das moderne Gleichheitszeichen.

Abbildung 2.1.: Johannes Widmann (1460–1498) und Robert Recorde (1510–1558).

keine Klammern verwendet werden, da sich der Rechenbaum eindeutig rekonstruieren lässt. Apropos eindeutig, die oben erwähnten technischen Einschränkungen der konkreten Syntax haben gerade damit zu tun: man möchte aus der linearen Folge von Lexemen die hierarchische Struktur *automatisch* und auf *eindeutige* Weise herauslesen.

In den folgenden zwei Kapiteln werden wir uns auf die abstrakte Syntax von Mini–F# konzentrieren und die konkrete nur am Rande behandeln. Mit anderen Worten, wir sind zunächst nicht an Äußerlichkeiten, sondern an den inneren Werten interessiert.

Ist die Struktur von Programmen festgelegt, kann man sich der Bedeutung zuwenden, der *Semantik* einer Programmiersprache. Die Semantik legt zum Beispiel fest, dass der Asteriskus `*` die Multiplikation zweier natürlicher Zahlen meint und das Pluszeichen `+` die Addition — das muss nicht notwendigerweise so sein, sondern bedarf in der Tat einer Festlegung. Und was ist die Bedeutung von `a11`? Auch darum muss sich die Semantik kümmern. Wir drücken uns an dieser Stelle vor der Antwort, um nicht zu weit vorzugreifen. Ganz allgemein lässt sich die Semantik mit Hilfe von Auswertungsregeln beschreiben. Zum Beispiel: Wenn `a11` zu 1 auswertet, dann wertet `a11 + 815` zu 816 aus; wenn `a11 + 815` zu 816 auswertet, dann wertet `4711 * (a11 + 815)` zu 3844176 aus. *Die Auswertungsregeln orientieren sich dabei eng an der Struktur eines Programms —*

die Bedeutung eines Ausdrucks wird in der Regel auf die Bedeutung der Teilausdrücke zurückgeführt. Deswegen ist es so wichtig, die Struktur vorher präzise festzulegen.

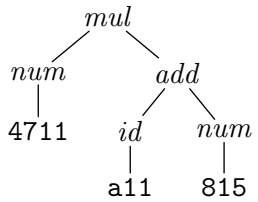
Die beiden folgenden Abschnitte führen das nötige Rüstzeug ein, um die Struktur eines Programms, sprich die abstrakte Syntax, und die Bedeutung eines Programms, sprich die Semantik, zu beschreiben.

2.3. Abstrakte Syntax⧹Baumsprachen

Den hierarchischen Aufbau von Programmen, die abstrakte Syntax, beschreiben wir mit sogenannten *Baumsprachen*. Die folgende Baumsprache namens *Expr* definiert eine einfache Form von arithmetischen Ausdrücken.

$$
\begin{array}{lll}
e \in \mathsf{Expr} ::= & \mathit{num}\ (\mathbb{N}) \\
\mid & \mathit{id}\ \ (\mathsf{Id}) \\
\mid & \mathit{add}\ \ (\mathsf{Expr}, \mathsf{Expr}) \\
\mid & \mathit{mul}\ \ (\mathsf{Expr}, \mathsf{Expr})
\end{array}
$$

Gemäß dieser Definition ist zum Beispiel $\mathit{mul}\ (\mathit{num}\ (\texttt{4711}), \mathit{add}\ (\mathit{id}\ (\texttt{a11}), \mathit{num}\ (\texttt{815})))$ ein Element von Expr. Dargestellt als Baum:



Bäume werden in der Informatik in der Regel mit der *Wurzel* nach oben und mit den *Blättern* nach unten gezeichnet. In unserem Beispiel ist *mul* die Wurzel und `4711`, `a11` und `815` sind Blätter. Insgesamt hat der Baum acht sogenannte *Knoten*. Jeder Knoten verzweigt dabei in eine feste Anzahl von Teilbäumen. In der Definition der Baumsprache wird festgelegt, wieviele Teilbäume jede Knotenart hat und von welcher Art die Teilbäume sind: *add* hat zum Beispiel zwei Teilbäume, beide sind wiederum Elemente von Expr; *id* besitzt einen Teilbaum, ein Element aus einer nicht weiter spezifizierten Menge Id, der Menge von zulässigen Bezeichnern (engl. identifiers).

Die obige Definition führt insgesamt drei verschiedene Dinge ein: einen Namen für die Baumsprache (Expr), Namen für die unterschiedlichen Knotenarten (*num*, *id*, *add* und *mul*) und eine sogenannte *Metavariable* (e). Das Symbol '::=' trennt den Namen der Baumsprache von den Namen der Knoten; die verschiedenen Knotenarten werden durch das Symbol '|' getrennt. (Diese Vereinbarungen legen die konkrete Syntax von Baumsprachen fest. Verwirrend?)

Umgangssprachlich lässt sich die Definition wie folgt lesen: Ein Element $e \in \mathsf{Expr}$ ist entweder von der Form $\mathit{num}\ (n)$ mit $n \in \mathbb{N}$, oder von der Form $\mathit{id}\ (x)$ mit $x \in \mathsf{Id}$, oder von der Form $\mathit{add}\ (e_1, e_2)$ mit $e_1, e_2 \in \mathsf{Expr}$ oder von der Form $\mathit{mul}\ (e_1, e_2)$ mit $e_1, e_2 \in \mathsf{Expr}$. Man sieht, um arithmetische Ausdrücke zu benennen, verwenden wir die Metavariable e

(gegebenenfalls mit einem Index versehen). Man spricht von einer *Meta*variablen, da wir mit ihr *über* Elemente der Programmiersprache sprechen; e selbst ist *kein* Bestandteil arithmetischer Ausdrücke.

Formal ist die Menge Expr durch eine sogenannte *induktive Definition* gegeben. Induktive Definitionen haben stets den gleichen Aufbau: Es gibt Regeln, die bestimmte Teilmengeneigenschaften der definierten Menge festlegen; Regeln, die Abschlusseigenschaften fordern; und schließlich eine Klausel, die fordert, dass die definierte Menge die kleinste Menge mit den genannten Eigenschaften ist. Die formale Lesart unseres Beispiels ist somit die folgende. Die Menge Expr ist die *kleinste* Menge mit den folgenden Eigenschaften:

1. ist $n \in \mathbb{N}$, dann ist $num\ (n) \in \mathsf{Expr}$;
2. ist $x \in \mathsf{Id}$, dann ist $id\ (x) \in \mathsf{Expr}$;
3. sind $e_1 \in \mathsf{Expr}$ und $e_2 \in \mathsf{Expr}$, dann ist auch $add\ (e_1, e_2) \in \mathsf{Expr}$;
4. sind $e_1 \in \mathsf{Expr}$ und $e_2 \in \mathsf{Expr}$, dann ist auch $mul\ (e_1, e_2) \in \mathsf{Expr}$.

Wichtig ist, dass Expr die *kleinste* Menge mit diesen Eigenschaften ist; nur Elemente, die sich auf eine der vier Arten bilden lassen, sind in Expr enthalten.

Wir werden im Laufe der Vorlesung viele Baumsprachen definieren. Dabei nehmen wir uns einige notationelle Freiheiten. So erlauben wir bei der Definition der Knotenarten, Metavariablen stellvertretend für die Baumsprachen selbst zu verwenden. Weiterhin erlauben wir die Knotennamen in konkreter Syntax oder an die konkrete Syntax angelehnt zu notieren. Mit diesen Konventionen verkürzt sich die Definition von Expr zu:

$$
\begin{array}{l}
n \in \mathbb{N} \\
x \in \mathsf{Id} \\
e \in \mathsf{Expr} ::= n \\
\qquad\qquad \mid\ x \\
\qquad\qquad \mid\ e_1 \texttt{ + } e_2 \\
\qquad\qquad \mid\ e_1 \texttt{ * } e_2
\end{array}
$$

Aus dem Knoten $add\ (e_1, e_2)$ ist e_1 `+` e_2 geworden. Es ist wichtig festzuhalten, dass der Unterschied zur ursprünglichen Definition nur ein äußerlicher ist: statt der Formel $mul\ (num\ (\texttt{4711}), add\ (id\ (\texttt{a11}), num\ (\texttt{815})))$ schreiben wir kurz `4711 * (a11 + 815)`, gemeint ist aber stets der gleiche abstrakte Syntax*baum*. Auch wenn wir Ausdrücke linear aufschreiben, zumeist aus Gründen der Bequemlichkeit und der Lesbarkeit, sollte man sich im Geiste stets den jeweiligen Baum vorstellen.

2.4. Beweissysteme

Bei der umgangssprachlichen Beschreibung der Bedeutung arithmetischer Ausdrücke haben wir wenn-dann Aussagen formuliert: *wenn* der Teilausdruck $id\ (\texttt{a11})$ zu 1 auswertet, *dann* wertet der Ausdruck $add\ (id\ (\texttt{a11}), num\ (\texttt{815}))$ zu 816 aus; *wenn* der Ausdruck $add\ (id\ (\texttt{a11}), num\ (\texttt{815}))$ zu 816 auswertet, *dann* wertet weiterhin der Ausdruck

$mul\ (num\ (\texttt{4711}), add\ (id\ (\texttt{a11}), num\ (\texttt{815})))$ zu 3844176 aus. Wenn-dann Aussagen lassen sich mit Hilfe sogenannter *Beweisregeln* formalisieren. Die wenn-dann Aussagen unseres laufenden Beispiels fangen wir mit den beiden folgenden Beweisregeln ein.

$$\frac{id\ (\texttt{a11}) \Downarrow 1}{add\ (id\ (\texttt{a11}), num\ (\texttt{815})) \Downarrow 816}$$

$$\frac{add\ (id\ (\texttt{a11}), num\ (\texttt{815})) \Downarrow 816}{mul\ (num\ (\texttt{4711}), add\ (id\ (\texttt{a11}), num\ (\texttt{815}))) \Downarrow 3844176}$$

Die umgangssprachliche Formulierung „wertet aus zu“ symbolisieren wir durch den Pfeil '$\Downarrow$'. Links von dem Pfeil steht ein arithmetischer Ausdruck, rechts davon eine natürliche Zahl. Über dem Strich der Regeln steht die Voraussetzung oder die Voraussetzungen (der „wenn“ Teil); unter dem Strich steht die Schlussfolgerung (der „dann“ Teil). Beweisregeln können sowohl von oben nach unten als auch von unten nach oben gelesen werden. Von oben nach unten: Wenn ich $id\ (\texttt{a11}) \Downarrow 1$ bereits weiß, dann kann ich $add\ (id\ (\texttt{a11}), num\ (\texttt{815})) \Downarrow 816$ folgern. Von unten nach oben: Wenn ich die Formel $add\ (id\ (\texttt{a11}), num\ (\texttt{815})) \Downarrow 816$ zeigen möchte, muss ich zuerst $id\ (\texttt{a11}) \Downarrow 1$ zeigen.

Beweisregeln können zu *Beweisbäumen* zusammengesetzt werden. Dabei werden die Voraussetzungen einer Regel durch Beweisbäume ersetzt, die die jeweiligen Formeln als finale Schlussfolgerung enthalten. Für unser Beispiel erhalten wir

$$\dfrac{\dfrac{\dfrac{\vdots}{id\ (\texttt{a11}) \Downarrow 1}}{add\ (id\ (\texttt{a11}), num\ (\texttt{815})) \Downarrow 816}}{mul\ (num\ (\texttt{4711}), add\ (id\ (\texttt{a11}), num\ (\texttt{815}))) \Downarrow 3844176}$$

Dass dieser Beweisbaum mehr wie ein Strunk und weniger wie ein Baum aussieht, liegt daran, dass jede Regel genau eine Voraussetzung hat. Allerdings haben wir auch etwas gemogelt: Addition wie Multiplikation haben zwei Teilausdrücke und beide Teilausdrücke müssen zunächst ausgewertet werden, wobei Konstanten wie $num\ (\texttt{4711})$ und $num\ (\texttt{815})$ zu sich selbst auswerten.

$$\dfrac{\dfrac{num\ (\texttt{4711}) \Downarrow 4711}{} \qquad \dfrac{\dfrac{\dfrac{\vdots}{id\ (\texttt{a11}) \Downarrow 1}}{} \qquad \dfrac{}{num\ (\texttt{815}) \Downarrow 815}}{add\ (id\ (\texttt{a11}), num\ (\texttt{815})) \Downarrow 816}}{mul\ (num\ (\texttt{4711}), add\ (id\ (\texttt{a11}), num\ (\texttt{815}))) \Downarrow 3844176}$$

Jetzt sieht man sehr schön die hierarchische Struktur eines Beweisbaumes — der anders als sonst in der Informatik von unten nach oben „wächst“.

Beweisregeln sind ein allgemeines Konzept; ebensowenig wie das Zusammenstellen von Regeln zu Beweisbäumen sind sie auf Auswertungen beschränkt. Ist eine beliebige Menge von Formeln gegeben, etwa durch eine Baumsprache $\phi \in \Phi ::= \cdots$, dann hat eine *Beweisregel* die allgemeine Form

$$\frac{\phi_1 \qquad \cdots \qquad \phi_n}{\phi}$$

Über dem Strich der Regeln stehen wie gesagt die Voraussetzungen (n Formeln); unter dem Strich steht die Schlussfolgerung (eine einzige Formel). Ist $n = 0$, so spricht man auch von einem *Axiom*. Ein *Beweissystem* ist eine Menge von Beweisregeln.

Die Menge aller *Beweisbäume* ist induktiv definiert: Sind $\mathcal{P}_1$, ..., $\mathcal{P}_n$ Beweisbäume mit den Wurzeln ϕ_1, ..., ϕ_n und ist

$$\frac{\phi_1 \qquad \cdots \qquad \phi_n}{\phi}$$

eine Beweisregel, dann ist

$$\frac{\mathcal{P}_1 \qquad \cdots \qquad \mathcal{P}_n}{\phi}$$

ein Beweisbaum mit der Wurzel ϕ. Die Regeln werden zusammengesteckt wie die Klötzchen eines bekannten Spielwarenherstellers. Ist $\mathcal{P}$ ein Beweisbaum mit der Wurzel ϕ, dann sagt man auch $\mathcal{P}$ zeigt oder beweist ϕ.

Kommen wir zurück zu den Auswertungsregeln: Die obigen Beweisregeln sind sehr speziell; allgemeinere Regeln lassen sich mit Hilfe von Metavariablen formulieren.

$$\frac{}{num\ (n) \Downarrow n} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{add\ (e_1, e_2) \Downarrow n_1 + n_2} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{mul\ (e_1, e_2) \Downarrow n_1 n_2}$$

Die erste Regel — eine Regel ohne Voraussetzungen, ein Axiom — legt fest, dass Konstanten zu sich selbst auswerten. Die beiden anderen Regeln formalisieren, dass zunächst beide Teilausdrücke ausgerechnet werden und dass das Ergebnis die Summe bzw. das Produkt der Teilergebnisse ist. Regeln, die Metavariablen enthalten, heißen auch *Regelschemata*.

Bevor Regelschemata zu Beweisbäumen zusammengesetzt werden können, müssen die Metavariablen erst durch konkrete Konstanten bzw. Ausdrücke ersetzt werden. Das Ergebnis einer solchen Ersetzung nennt man auch *Regelinstanz* oder kurz *Instanz*. Ein Regelschema steht somit stellvertretend für die Menge aller seiner Instanzen. Für unser laufendes Beispiel benötigen wir zwei Instanzen des Axioms

$$\frac{}{num\ (4711) \Downarrow 4711} \qquad \frac{}{num\ (815) \Downarrow 815}$$

und jeweils eine Instanz der Regeln für die Addition und die Multiplikation:

$$\frac{id\ (\texttt{a11}) \Downarrow 1 \qquad num\ (\texttt{815}) \Downarrow 815}{add\ (id\ (\texttt{a11}), num\ (\texttt{815})) \Downarrow 816}$$

$$\frac{num\ (\texttt{4711}) \Downarrow 4711 \qquad add\ (id\ (\texttt{a11}), num\ (\texttt{815})) \Downarrow 816}{mul\ (num\ (\texttt{4711}), add\ (id\ (\texttt{a11}), num\ (\texttt{815}))) \Downarrow 3844176}$$

Aus diesen Regelinstanzen kann wiederum der obige Beweisbaum zusammengesetzt werden. Man sieht, Instanzen von Axiomen werden zu Blättern eines Beweisbaumes.

Wir haben im letzten Abschnitt angekündigt, dass wir uns bei der Definition der abstrakten Syntax einige Freiheiten herausnehmen, insbesondere, dass wir die Notation

der Baumknoten an die konkrete Syntax anlehnen. Das hat den Vorteil, dass wir optisch enger an unserer Programmiersprache arbeiten — die Addition wird in Mini–F# durch e_1 `+` e_2 notiert und nicht durch *add* (e_1, e_2) — und den Nachteil, dass wir optisch sehr eng an die Semantik heranrücken — auch in der Mathematik wird die Addition mit '+' notiert. Betrachten wir als Beispiel die entsprechend adaptierte Regel für die Addition.

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \texttt{ + } e_2 \Downarrow n_1 + n_2}$$

Das Pluszeichen tritt zweimal auf, meint aber zwei grundsätzlich verschiedene Dinge: links ist '`+`' ein syntaktischer Bestandteil unserer Programmiersprache, rechts bezeichnet '+' das mathematische Konzept der Addition zweier natürlicher Zahlen. Kurz, '`+`' ist Syntax und '+' ist Semantik. Beides sollte nicht verwechselt werden.

Ganz allgemein haben wir es mit dem Unterschied zwischen *Objektsprache* und *Metasprache* zu tun. Die Objektsprache, in unserem Fall Mini–F#, ist Objekt des Studiums. Um Eigenschaften der Objektsprache zu formulieren, müssen wir uns einer weiteren Sprache bedienen. Da wir *über* die Objektsprache reden, heißt die andere Sprache *Meta*sprache. Metavariablen sind zum Beispiel ein Element der Metasprache. Als Metasprache verwenden wir die Sprache der Mathematik, angereichert mit deutscher Prosa. Wenn die Objektsprache an die Metasprache angelehnt ist — um zum Beispiel Additionen wie in der Mathematik üblich notieren zu können — besteht die Gefahr der Verwechslung oder zumindest der Verwirrung. Wir sind in diesem Abschnitt der Verwechslungsgefahr etwas entgegengetreten, indem wir unterschiedliche Zeichensätze für Objekt- und Metasprache verwendet haben: '`+`' ist ein Element der Objektsprache und '+' ein Element der Metasprache. Jetzt, da wir uns den Unterschied klar gemacht haben, werden wir nicht mehr so fein unterscheiden und '+' auch für Mini–F# Addition verwenden.

Übungen.

1. Die folgenden endlichen Abbildungen modellieren das Fressverhalten von Nagetieren:

$\varphi_1 = \{\mathit{Flecki} \mapsto \mathit{Heu}, \mathit{Fred} \mapsto \mathit{Heu}, \mathit{Hoppel} \mapsto \mathit{Drops}\}$
$\varphi_2 = \{\mathit{Bläcki} \mapsto \mathit{Salat}, \mathit{Flecki} \mapsto \mathit{Möhren}, \mathit{Max} \mapsto \mathit{Gras}, \mathit{Moritz} \mapsto \mathit{Heu}\}$

Welche endliche Abbildungen ergeben die folgenden Ausdrücke?

- φ_1, φ_1 und φ_2, φ_2
- φ_1, φ_2 und φ_2, φ_1
- $\varphi_1, \{\mathit{Hoppel} \mapsto \mathit{Sticks}\}, \varphi_2$
- $\varphi_1 \setminus \{\mathit{Max}, \mathit{Moritz}\}$ und $\varphi_2 \setminus \{\mathit{Max}, \mathit{Moritz}\}$
- $(\varphi_1, \varphi_2) \setminus \{\mathit{Max}, \mathit{Moritz}\}$ und $\varphi_1, (\varphi_2 \setminus \{\mathit{Max}, \mathit{Moritz}\})$
- $(\varphi_1, \varphi_2) \setminus \{\mathit{Fred}, \mathit{Moritz}\}$ und $\varphi_1, (\varphi_2 \setminus \{\mathit{Fred}, \mathit{Moritz}\})$

2. Zeigen Sie, dass der Kommaoperator assoziativ ist.

$$(\varphi_1, \varphi_2), \varphi_3 \quad = \quad \varphi_1, (\varphi_2, \varphi_3)$$

3. Zeigen Sie, dass Sequenzen A^* die algebraischen Eigenschaften eines Monoids erfüllen: die leere Sequenz ist das neutrale Element der Konkatenation und die Konkatenation ist assoziativ.

$\epsilon \cdot s = s = s \cdot \epsilon$

$s_1 \cdot (s_2 \cdot s_3) = (s_1 \cdot s_2) \cdot s_3$

4. Die folgenden Baumsprachen modellieren Dezimalziffern und Dezimalzahlen.

$$\begin{aligned} d \in \mathit{Digit} \quad & ::= \mathit{zero} \mid \mathit{one} \mid \mathit{two} \mid \mathit{three} \mid \mathit{four} \mid \mathit{five} \mid \mathit{six} \mid \mathit{seven} \mid \mathit{eight} \mid \mathit{nine} \\ ds \in \mathit{Numeral} & ::= \mathit{nil} \\ & \quad \mid \mathit{cons}\,(\mathit{Digit}, \mathit{Numeral}) \end{aligned}$$

Die Dezimalzahl 4711 wird zum Beispiel durch den Baum

$$\mathit{cons}\,(\mathit{one}, \mathit{cons}\,(\mathit{one}, \mathit{cons}\,(\mathit{seven}, \mathit{cons}\,(\mathit{four}, \mathit{nil}))))$$

repräsentiert; die niederwertigste Ziffer steht links. Stellen Sie Regelschemata auf, die einem Numeral seinen Wert zuordnen ($ds \Downarrow n$). Konstruieren Sie einen Beweisbaum für die Formel:

$$\mathit{cons}\,(\mathit{one}, \mathit{cons}\,(\mathit{one}, \mathit{cons}\,(\mathit{seven}, \mathit{cons}\,(\mathit{four}, \mathit{nil})))) \Downarrow 4711$$

Zum Knobeln: Wie müssen die Regeln geändert werden, wenn man vereinbart, dass die Wertigkeit der Ziffern von links nach rechts abnimmt, 4711 also durch

$$\mathit{cons}\,(\mathit{four}, \mathit{cons}\,(\mathit{seven}, \mathit{cons}\,(\mathit{one}, \mathit{cons}\,(\mathit{one}, \mathit{nil}))))$$

repräsentiert wird? *Hinweis:* Stellen Sie Beweisregeln auf, die einem Numeral seine Länge (Anzahl der Ziffern) zuordnen.

5. Harry Hacker hat ein neues Spielzeug erworben: einen programmierbaren Roboter. Um den Roboter zu steuern, kann man ihn mit beliebigen Sequenzen, bestehend aus den Buchstaben `F`, `L` und `R`, füttern. Dabei veranlasst `F` den Roboter, 1 cm in die aktuelle Richtung zu gehen; `L` bzw. `R` drehen den Roboter um 1° nach links bzw. rechts. An dem Roboter lässt sich ein Zeichenstift anbringen, so dass man die Spur des Roboters auf dem Fußboden nachvollziehen kann.

- Die Sequenz `FFFLRF` instruiert den Roboter zum Beispiel 4 cm nach vorne zu gehen.
- Mittels `FFLLLF` geht der Roboter 2 cm nach vorne, dreht sich um 3° nach links und geht von dort aus 1 cm in die neue Richtung.

Nach einigen Tagen des Experimentierens ist Harrys Tastatur — insbesondere die Tasten `F`, `L` und `R` — arg in Mitleidenschaft gezogen. Um der endgültigen Zerstörung des Equipments zuvorzukommen, schlägt Lisa vor, eine Robotersteuerungssprache zu entwickeln, die die gleichen Effekte mit weniger Aufwand erzielt. Sie beschließen keine Zeit auf die konkrete Syntax zu verschwenden und legen gleich die abstrakte Syntax fest:

$n \in \mathbb{N}$

$d \in \mathit{Draw} ::= \mathit{forward}$		1 cm nach vorne
$\mid \mathit{left}$		1° nach links
$\mid \mathit{right}$		1° nach rechts
$\mid d_1; d_2$		Konkatenation
$\mid \mathit{repeat}\,(n, d)$		Wiederholung

1. Probieren Sie die Robotersteuerungssprache aus. Schreiben Sie Programme, um
   a) ein Rechteck von 3 cm Breite und 2 cm Länge,
   b) ein gleichseitiges Dreieck mit einer Seitenlänge von 5 cm und
   c) das Haus vom Nikolaus

   zu zeichnen.
2. Helfen Sie Lisa und Harry, die Semantik der Robotersteuerungssprache festzulegen. Geben Sie zu diesem Zweck ein Beweissystem an, das einem Programm eine Folge primitiver Instruktionen zuordnet ($\mathit{Draw} \Downarrow \{\mathtt{F}, \mathtt{L}, \mathtt{R}\}^*$).
3. Wenden Sie das Beweissystem an. Bestimmen Sie mit seiner Hilfe die Bedeutung der Programme aus Teil 1: Geben Sie zu jedem Programm d einen Beweisbaum für $d \Downarrow s$ an, wobei s die Folge der primitiven Instruktionen ist, die Bedeutung von d.

6. Erklären Sie die Begriffe

- Beweisregel,
- Axiom,
- Regelschema und
- Regelinstanz

anhand Ihres Beweissystems aus Aufgabe 2.4.

3. Werte ⟍ Elementares Rechnen

You can never understand one language
until you understand at least two.
— Ronald Searle (1920)

If we spoke a different language,
we would perceive a somewhat different world.
— Ludwig Wittgenstein (1889–1951)

Semantics is a strange kind of applied mathematics;
it seeks profound definitions rather than difficult theorems.
— J.C. Reynolds (1980)

Sprachen, natürliche wie künstliche, sind komplexe Gebilde. Eine Sprache wie Deutsch oder Englisch lernt man über viele Jahre hinweg; vollständig beherrscht man sie wohl nie. Für die Vorstellung unserer Programmiersprache steht uns naturgemäß nicht so viel Zeit zur Verfügung. Wir benötigen sie allerdings auch nicht, da Mini–F# bezüglich Struktur, Bedeutung und Verwendung wesentlich einfacher gestrickt ist. Dies liegt zum einen daran, dass eine Programmiersprache nur einem Zweck dient, nämlich einen oder mehrere Rechner zum Rechnen zu bringen, und zum anderen daran, dass sich die Sprache Mini–F# nicht über Jahrhunderte, sondern nur über wenige Wochen hin entwickelt hat.

Einige Wochen werden wir auch benötigen, um die Sprache vorzustellen und präzise zu definieren. Dabei werden wir die Bestandteile, die sogenannten *Sprachkonstrukte*, in kleinen, wohlportionierten Dosen verabreichen. Die ersten fünf Dosen erhalten Sie in diesem Kapitel. Am Ende des Kapitels kennen und können wir eine Programmiersprache, die *berechnungsuniversell* ist, das heißt, mit der man die prinzipiellen Möglichkeiten eines Rechners ausschöpfen kann. Aus Sicht der Theorie sind die folgenden Kapitel nur Zugabe; aus Sicht der Praxis wird es dann erst interessant.

Jeder Abschnitt in diesem Kapitel und in den meisten der folgenden Kapitel folgt einer einheitlichen Struktur, die in Abbildung 3.1 schematisch dargestellt ist. Prinzipiell wird in jedem Abschnitt *ein* bestimmtes Sprachmerkmal oder neudeutsch Sprachfeature eingeführt.

Zunächst werden jeweils die neu einzuführenden Sprachkonstrukte motiviert. Oft anhand von Beispielen, die zeigen, dass bestimmte Aufgaben sich mit den bis dato eingeführten Sprachmitteln nicht oder nur schlecht bewerkstelligen lassen. Anschließend wird die Syntax, die äußere Form, und die Semantik, die Bedeutung der Konstrukte, definiert.

Ein Programm in Mini–F# ist technisch gesehen ein *Ausdruck*; im einfachsten Fall ein arithmetischer Ausdruck wie $4711 + 815$ oder schlicht eine Zeichenkette (engl. string) wie `"Hello, world!"`. Neue Features führen neue Ausdrücke ein: Die syntaktische Kategorie

Motivation

Abstrakte Syntax

$e \in \mathsf{Expr} ::= \cdots$ *Ausdrücke*

Statische Semantik

$t \in \mathsf{Type} ::= \cdots$ *Typen*

Dynamische Semantik

$\nu \in \mathsf{Val} ::= \cdots$ *Werte*

Vertiefung

Blick über den Tellerrand

Abbildung 3.1.: Struktur eines Abschnitts.

Expr der Ausdrücke wird im Laufe der Vorlesung schrittweise um neue Formen erweitert. Um uns nicht in syntaktischen Details zu verlieren, beschreiben wir nur die *abstrakte Syntax*, nicht die konkrete. Erstere weicht aber wie bereits angekündigt in der Regel nur unwesentlich von der letzteren ab. Wenn es Abweichungen gibt, benennen wir sie kurz. (Anhang A.2 fasst die Unterschiede zusammen.)

Ausdrücke sind zunächst einmal beliebig kombinierbar, ein bemerkenswertes Feature unserer Sprache. Nicht alle Kombination ergeben jedoch Sinn. Zum Beispiel lässt das Programm `"Hello, world!"` $*$ 4711 einen Programmierfehler vermuten. Die *statische Semantik* fängt diese sinnlosen Ausdrücke ab (technisch würden wir auch von wertlosen Ausdrücken sprechen). Zu diesem Zweck werden Ausdrücke mit Hilfe sogenannter *Typen* in Schubladen eingeteilt: Ein arithmetischer Ausdruck erhält zum Beispiel den Typ Nat. Die statische Semantik legt dann fest, dass die Multiplikation zwei natürliche Zahlen verarbeitet und eine natürliche Zahl zum Ergebnis hat.

$$\frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1 * e_2 : \mathit{Nat}}$$

Für jedes neu eingeführte Konstrukt wird eine derartige *Typregel* angegeben. Diese Beweisregeln spezifizieren die zweistellige Relation

$$e : t$$

zwischen Ausdrücken und Typen. *Lies:* „e hat den Typ t“.

Die statische Semantik hat eine wichtige ordnende Funktion: Die meisten Abschnitte führen *einen* neuen Typ oder *ein* neues Typkonstrukt ein, zusammen mit Sprachkonstrukten, die auf diesem Typ arbeiten.

Ein Ausdruck e heißt *wohlgetypt*, wenn es einen Typ t gibt, so dass sich $e : t$ mit den Regeln der statischen Semantik ableiten lässt. Der Ausdruck $4711 * 2 + 815$ ist wohlgetypt, der Ausdruck `"Hello, world!"` $* 4711$ ist es nicht.

Wohlgetypte Ausdrücke können ausgerechnet werden und nur solche. Wie dies geschieht, spezifiziert die *dynamische Semantik*. Die dynamische Semantik legt zum Beispiel für die Multiplikation fest, dass beide Argumente ausgerechnet werden und dass das Ergebnis das Produkt der Teilergebnisse ist.

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 * e_2 \Downarrow n_1 n_2}$$

Für jedes neu eingeführte Konstrukt wird mindestens eine derartige *Auswertungsregel* angegeben. Auswertungsregeln spezifizieren die zweistellige Relation

$$e \Downarrow \nu$$

zwischen Ausdrücken und Werten. *Lies:* „e wertet zu ν aus“. Ein Wert ist das Ergebnis eines Programms; zum Beispiel ist der Wert eines arithmetischen Ausdrucks eine natürliche Zahl. Mit jedem neu eingeführten Typ werden wir auch den Bereich der Werte um entsprechende Elemente erweitern. Ein Typ ist im Prinzip die Menge aller zugehörigen Werte. Da Werte nicht ausgewertet werden müssen, haben die Auswertungsregeln für diese Ausdrücke stets die triviale Form

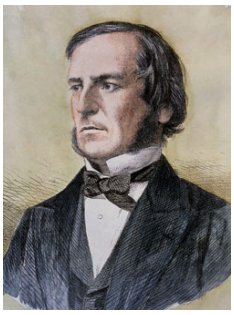
$$\frac{}{\nu \Downarrow \nu}$$

Ein Programm, sprich ein Ausdruck, wird ausgerechnet, indem die Auswertungsregeln für die einzelnen Bestandteile zu einem Beweisbaum kombiniert werden, siehe Abschnitt 2.4. Zur Erinnerung: Für den arithmetischen Ausdruck $4711 * 2 + 815$ erhalten wir zum Beispiel den folgenden Beweisbaum.

$$\dfrac{\dfrac{\dfrac{}{4711 \Downarrow 4711} \qquad \dfrac{}{2 \Downarrow 2}}{4711 * 2 \Downarrow 9422} \qquad \dfrac{}{815 \Downarrow 815}}{4711 * 2 + 815 \Downarrow 10237}$$

Konstanten wie 4711 oder 815 werten zu sich selbst aus — Konstanten *sind* Werte. Die baumartige Darstellung zeigt sehr schön, wie die einzelnen Auswertungsregeln kombiniert werden. Am Anfang der Vorlesung werden wir Beispielrechnungen in aller Ausführlichkeit ohne jedwede Auslassung notieren. Sobald wir etwas Erfahrung gewonnen haben, kürzen wir Teilrechnungen ab oder lassen sie ganz aus.

$$\dfrac{\dfrac{}{4711 * 2 \Downarrow 9422}}{4711 * 2 + 815 \Downarrow 10237}$$



Der englische Mathematiker George Boole entwickelte in seiner Schrift „The Mathematical Analysis of Logic“ von 1847 den ersten algebraischen Logikkalkül und begründete damit die moderne mathematische Logik.

Boole stellte die Wahrheitswerte durch die Zahlen 0 und 1 dar und drückte die logischen Operationen entsprechend durch arithmetische Operationen aus.

56 OF HYPOTHETICALS.

1st. Disjunctive Syllogism.

Either X is true, or Y is true (exclusive),	$x + y - 2xy = 1$
But X is true,	$x = 1$
Therefore Y is not true,	$\therefore y = 0$
Either X is true, or Y is true (not exclusive),	$x + y - xy = 1$
But X is not true,	$x = 0$
Therefore Y is true,	$\therefore y = 1$

Abbildung 3.2.: George Boole (1815–1864).

Hier sind die trivialen Teilrechnungen für die Konstanten unter den Tisch gefallen.

Jeder Abschnitt schließt mit Anwendungen der neu eingeführten Konstrukte. Da wir in diesem Kapitel nur das Grundvokabular der Sprache vermitteln, sind die Beispielprogramme ebenfalls recht elementar, insbesondere in den ersten Abschnitten. Gelegentlich blicken wir auch über den Tellerrand hinaus und schauen uns an, was andere Programmiersprachen (insbesondere F# selbst) zu dem jeweiligen Thema anbieten.

3.1. Boolesche Werte

Nicht-triviale Programme treffen viele Entscheidungen. Im einfachsten Fall wird überprüft, ob ein bestimmter Sachverhalt wahr oder falsch ist: Ist das Konto überzogen? Ist Florian größer als Lisa? usw. Das Ergebnis einer solchen Überprüfung repräsentieren wir durch einen Wahrheitswert: *true* oder *false*. In Abhängigkeit von einem Wahrheitswert kann die Rechnung dann einen bestimmten Verlauf nehmen. Das linguistische Konstrukt, das eine abhängige Berechnung realisiert, ist die *Alternative*:

if e_1 ***then*** e_2 ***else*** e_3

Wertet der Ausdruck e_1, die sogenannte *Bedingung*, zu *true* aus, dann wird mit der Auswertung von e_2 fortgefahren; wertet e_1 zu *false* aus, dann wird e_3 ausgewertet. Im folgenden Beispiel werden zwei Alternativen ausgewertet (‘<’ ist die „kleiner“ Relation;

die Operation wird im nächsten Abschnitt formal eingeführt):

$$\dfrac{\dfrac{\overline{4711 < 815 \Downarrow \mathit{false}} \qquad \overline{\mathit{true} \Downarrow \mathit{true}}}{\textbf{\textit{if}}\ 4711 < 815\ \textbf{\textit{then}}\ \mathit{false}\ \textbf{\textit{else}}\ \mathit{true} \Downarrow \mathit{true}} \qquad \overline{\texttt{"yes"} \Downarrow \texttt{"yes"}}}{\textbf{\textit{if}}\ (\textbf{\textit{if}}\ 4711 < 815\ \textbf{\textit{then}}\ \mathit{false}\ \textbf{\textit{else}}\ \mathit{true})\ \textbf{\textit{then}}\ \texttt{"yes"}\ \textbf{\textit{else}}\ \texttt{"no"} \Downarrow \texttt{"yes"}}$$

Alternativen dürfen beliebig geschachtelt werden. Im obigen Beispiel ist die Bedingung der äußeren Alternative wiederum eine Alternative. Die freie Kombinierbarkeit gilt nicht nur für Alternativen, sondern wie gesagt für alle Ausdrücke, die wir im Laufe der Vorlesung kennenlernen.

Abstrakte Syntax Unser erstes Sprachfeature umfasst somit drei Konstrukte:

$e \in \mathsf{Expr} ::=$		*Boolesche Ausdrücke:*
	$\mid \mathit{false}$	falsch
	$\mid \mathit{true}$	wahr
	$\mid \textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3$	Alternative

Der Teilausdruck e_1 nach dem Schlüsselwort[1] ***if*** heißt *Bedingung*; die Teilausdrücke e_2 und e_3 heißen *Zweige* der Alternative.

Statische Semantik Ein Boolescher Ausdruck hat den Typ Bool.

$t \in \mathsf{Type} ::=$		*Typen:*
	$\mid \mathit{Bool}$	Typ der Booleschen Werte

Um Typen gut von Ausdrücken und Werten unterscheiden zu können, verwenden wir Farben: Typen werden in königsblau (engl. royal blue) gesetzt, Ausdrücke und Werte in schwarz.

Die Wahrheitswerte *false* und *true* sind vom Typ Bool.

$$\overline{\mathit{false} : \mathit{Bool}} \qquad \overline{\mathit{true} : \mathit{Bool}}$$

$$\dfrac{e_1 : \mathit{Bool} \qquad e_2 : t \qquad e_3 : t}{\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3 : t}$$

Die Bedingung e_1 muss ein Boolescher Ausdruck sein; die Zweige der Alternative müssen den gleichen Typ besitzen, dieser ist auch der Typ der gesamten Alternative.

[1] Ein Schlüsselwort ist ein Konzept der lexikalischen Syntax: eine Folge von Buchstaben, die reserviert ist und somit nicht als Bezeichner verwendet werden kann. Schlüsselwörter heben wir farblich hervor.

Dynamische Semantik Boolesche Ausdrücke werten zu den Wahrheitswerten *false* und *true* aus.

$\nu \in \mathsf{Val} ::=$	*Boolesche Werte:*
$\mid$ *false*	falsch
$\mid$ *true*	wahr

Die Wahrheitswerte *false* und *true* werten zu sich selbst aus. Dies gilt wie gesagt allgemein für alle Werte.

$$\frac{}{\mathit{false} \Downarrow \mathit{false}} \qquad \frac{}{\mathit{true} \Downarrow \mathit{true}}$$

$$\frac{e_1 \Downarrow \mathit{true} \qquad e_2 \Downarrow \nu}{\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3 \Downarrow \nu} \qquad \frac{e_1 \Downarrow \mathit{false} \qquad e_3 \Downarrow \nu}{\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3 \Downarrow \nu}$$

Die dynamische Semantik der Alternative wird mit Hilfe zweier Regeln spezifiziert; diese legen fest, dass die Bedingung ausgewertet wird und in Abhängigkeit vom Ergebnis dieser Auswertung genau einer der beiden Zweige.

Vertiefung Ein Boolescher Ausdruck modelliert einen Sachverhalt oder eine Aussage. Wir sind gewohnt, einfache Aussagen zu komplexen Aussagen zusammenzusetzen: Der Kunde ist *nicht* kreditwürdig (*Negation*). Das Konto ist überzogen *und* der Kunde ist nicht kreditwürdig (*Konjunktion*). Das Netzteil ist defekt *oder* die Leitung ist unterbrochen (*Disjunktion*). Diese sogenannten Booleschen Verknüpfungen lassen sich mit Hilfe der Alternative programmieren. Ist e ein Boolescher Ausdruck, dann programmiert

$$\textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ \mathit{false}\ \textbf{\textit{else}}\ \mathit{true}$$

die *Negation* von e. Die *Konjunktion* von e_1 und e_2 wird durch

$$\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ \mathit{false}$$

und die *Disjunktion* durch

$$\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ \mathit{true}\ \textbf{\textit{else}}\ e_2$$

realisiert.

Boolesche Verknüpfungen werden oft mit Hilfe von sogenannten Wahrheitstafeln eingeführt, die das Ein-/Ausgabeverhalten tabellieren.

	Negation
false	*true*
true	*false*

Konjunktion	*false*	*true*
false	*false*	*false*
true	*false*	*true*

Disjunktion	*false*	*true*
false	*false*	*true*
true	*true*	*true*

Wir können uns leicht davon überzeugen, dass die Miniprogramme die entsprechenden Verknüpfungen realisieren. Wertet zum Beispiel e zu *false* aus, dann wertet der Ausdruck für die Negation *if* e *then false else true* zu *true* aus usw.

Negation, Konjunktion und Disjunktion werden häufig bei der Formulierung von Bedingungen eingesetzt. Deshalb erlauben wir auch eine abkürzende Schreibweise:

- *not e* statt ***if*** *e* ***then*** *false* ***else*** *true*,
- e_1 `&&` e_2 statt ***if*** e_1 ***then*** e_2 ***else*** *false* und
- e_1 `||` e_2 statt ***if*** e_1 ***then*** *true* ***else*** e_2.

3.2. Natürliche Zahlen

Die ganze Zahl schuf der liebe Gott,
alles übrige ist Menschenwerk.
— Leopold Kronecker (1823–1891)

Die natürliche Zahl schuf der liebe Gott,
alles übrige ist Menschenwerk.
— Ralf Hinze (1965)

Den Begriff Rechnen werden die meisten mit Zahlen in Verbindung bringen. In diesem Abschnitt führen wir einige elementare Konstrukte zum Rechnen mit Zahlen ein. Die Sprachunterstützung ist zunächst recht spartanisch; „liebgewonnene“ Operationen wie etwa die Potenzfunktion oder die Quadratwurzel werden erst in einem späteren Abschnitt formal definiert. Genauer: Wir werden später zeigen, wie sich diese Operationen programmieren lassen.

Abstrakte Syntax Neben den natürlichen Zahlen selbst und den gängigen arithmetischen Operatoren kommen *Vergleichsoperatoren* zur Syntax hinzu.

$n \in \mathbb{N} ::= 0 \mid 1 \mid 2 \mid \cdots$	*natürliche Zahlen*
$e ::= \cdots$	*Arithmetische Ausdrücke:*
$\mid n$	natürliche Zahl
$\mid e_1 + e_2$	Addition
$\mid e_1 \dot{-} e_2$	natürliche Subtraktion („monus“)
$\mid e_1 * e_2$	Multiplikation
$\mid e_1 \div e_2$	natürliche Division
$\mid e_1 \,\%\, e_2$	Divisionsrest
$\mid e_1 < e_2$	kleiner
$\mid e_1 \leqslant e_2$	kleiner gleich
$\mid e_1 = e_2$	gleich
$\mid e_1 <> e_2$	ungleich
$\mid e_1 \geqslant e_2$	größer gleich
$\mid e_1 > e_2$	größer

Das Auslassungszeichen ($\cdots$) soll deutlich machen, dass wir die syntaktische Kategorie $e \in$ Expr *erweitern*: die oben aufgeführten Konstrukte kommen zu den im voherigen Abschnitt eingeführten hinzu.

In der konkreten Syntax bestehen die Vergleichsoperatoren $\leqslant$ und $\geqslant$ jeweils aus zwei Zeichen: `<=` und `>=`. Die konkrete Syntax für $\dot{-}$ ist `-` und für $\div$ ist `/`. (Die Subtraktion auf den natürlichen Zahlen und die Subtraktion auf den reellen Zahlen haben sehr unterschiedliche Eigenschaften. Aus diesem Grund ist es hilfreich, sie einfach syntaktisch unterscheiden zu können. Gleiches gilt für die Division. Mehr dazu in Kürze.)

Statische Semantik Ein arithmetischer Ausdruck hat den Typ *Nat*.

$t ::= \cdots$	*Typen:*
$\mid \mathit{Nat}$	Typ der natürlichen Zahlen

Die arithmetischen Operatoren erwarten jeweils zwei Argumente vom Typ *Nat* und haben ein Ergebnis vom Typ *Nat*.

$$\frac{}{n : \mathit{Nat}}$$

$$\frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1 + e_2 : \mathit{Nat}} \qquad \frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1 \dot{-} e_2 : \mathit{Nat}} \qquad \text{usw.}$$

Die Vergleichsoperatoren erwarten ebenfalls zwei Argumente vom Typ *Nat*, geben aber ein Element vom Typ *Bool* zurück.

$$\frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1 < e_2 : \mathit{Bool}} \qquad \frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1 \leqslant e_2 : \mathit{Bool}} \qquad \text{usw.}$$

Die statische Semantik bringt Ordnung in die Welt der Programme: Ausdrücke werden hier zunächst unterteilt in Boolesche Ausdrücke vom Typ *Bool* und in arithmetische Ausdrücke vom Typ *Nat*. Auf diese Weise werden unsinnige Ausdrücke wie etwa $\mathit{false} + 1$ oder ***if*** 0 ***then*** 0 ***else*** *true* abgefangen. Diese Ausdrücke sind unsinnig, da die dynamische Semantik ihnen keinen Wert zuordnet.

Die Umkehrung gilt übrigens nicht: $1 + (\textbf{\textit{if}}\ \mathit{true}\ \textbf{\textit{then}}\ 0\ \textbf{\textit{else}}\ \mathit{false})$ wertet zu 1 aus, ist aber nicht wohlgetypt, da beide Zweige der Alternative einen unterschiedlichen Typ besitzen. Dieses Phänomen ist typisch für statische Typsysteme. Dynamische Eigenschaften von Programmen zählen in der Regel zu den formal unentscheidbaren Problemen. Aus diesem Grund ist die statische Semantik nur eine Annäherung an das tatsächliche Verhalten, allerdings von der sicheren Seite. Wertlose Programme, denen die dynamische Semantik keinen Wert zuordnet, werden von den Typregeln herausgefischt; in den Maschen bleiben aber auch Programme hängen, deren Wert definiert ist. Mehr zu unentscheidbaren Problemen erfahren Sie in späteren Semestern aus der Abteilung der Theoretischen Informatik.

Dynamische Semantik Ein arithmetischer Ausdruck wertet zu einer natürlichen Zahl aus. Entsprechend erweitern wir den Bereich der Werte um natürliche Zahlen.

$\nu ::= \cdots$	*Werte:*
$\mid n$	natürliche Zahlen

Da alle Operatoren auf den natürlichen Zahlen und nicht etwa auf den ganzen, rationalen oder gar reellen Zahlen arbeiten, müssen wir bei der Formulierung der Auswertungsregeln Vorsicht walten lassen. Zum Beispiel evaluiert $0 \dot{-} 1$ zu 0. Die Metavariablen k, q und r rangieren jeweils über den natürlichen Zahlen: $k, q, r \in \mathbb{N}$.

$$\frac{}{n \Downarrow n} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 + n_2} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 * e_2 \Downarrow n_1 n_2}$$

$$\frac{e_1 \Downarrow n + k \qquad e_2 \Downarrow n}{e_1 \dot{-} e_2 \Downarrow k} \qquad \frac{e_1 \Downarrow n \qquad e_2 \Downarrow n + k + 1}{e_1 \dot{-} e_2 \Downarrow 0}$$

Die Differenz zweier Ausdrücke ist 0, wenn das zweite Argument größer ist als das erste.

Die Operatoren '÷' und '%' implementieren die Division mit Rest: $a \div b$ ist der Quotient von a und b und $a \,\%\, b$ ist der Divisionsrest.

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \div e_2 \Downarrow q} \qquad n_1 = qn_2 + r \text{ und } r < n_2$$

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \,\%\, e_2 \Downarrow r} \qquad n_1 = qn_2 + r \text{ und } r < n_2$$

Die Auswertungsregeln haben jeweils eine Nebenbedingung, die neben der Regel aufgeführt ist. Die Nebenbedingung formuliert Einschränkungen an die Belegung der Metavariablen. So ist zum Beispiel $47 \,\%\, 11 \Downarrow r$ nur ableitbar, wenn $47 = 11q + r$ und $r < 11$ gilt. Da q und r natürliche Zahlen sein müssen, schränkt die Nebenbedingung r auf 3 ein und q auf 4. Für $b > 0$ gilt stets

$$a = (a \div b) * b + (a \,\%\, b) \quad \text{und} \quad 0 \leqslant a \,\%\, b < b$$

Jede Zahl a lässt sich eindeutig in einen Quotienten und in einen Rest zerlegen (für ein festes $b > 0$). Das ist eine merkenswerte Eigenschaft, von der wir wiederholt Gebrauch machen werden.

Ist $b = 0$, dann sind beide Operationen nicht definiert — die obigen Regeln sind nicht anwendbar, da es kein r mit $r < 0$ gibt. Dies ist natürlich keine befriedigende Lösung; wir werden auf dieses Problem in einem sehr viel späteren Abschnitt zurückkommen. Ein *Problem* ist das undefinierte Verhalten tatsächlich, hatten wir doch bei der Besprechung der statischen Semantik angemerkt, dass die Typregeln alle wertlosen Programme herausfischen. Nun ist zum Beispiel das Programm $1 \div 0$ wohlgetypt; die dynamische Semantik ordnet ihm aber keinen Wert zu — das wird sich in Abschnitt 7.4 ändern.

Für jede Vergleichsoperation gibt es zwei Auswertungsregeln.

$$\frac{e_1 \Downarrow n + k \qquad e_2 \Downarrow n}{e_1 < e_2 \Downarrow \mathit{false}} \qquad \frac{e_1 \Downarrow n \qquad e_2 \Downarrow n + k + 1}{e_1 < e_2 \Downarrow \mathit{true}} \qquad \text{usw.}$$

Wertet e_1 zu $n + k$ und e_2 zu n aus, dann ist $e_1 < e_2$ äquivalent zu $n + k < n$, einer falschen Aussage, da k vereinbarungsgemäß eine natürliche Zahl ist.

Vertiefung Unsere Programmiersprache hat nunmehr die Funktionalität eines einfachen Taschenrechners. Lassen wir ihn, den *Interpreter* für Mini–F#, also rechnen. Die Benutzungsschnittstelle des Interpreters ist einfach und intuitiv. Man gibt einen Ausdruck ein, der Interpreter wertet den Ausdruck aus und gibt den resultierenden Wert aus. Dann geht es von vorne los — dieser Zyklus des Einlesens, Auswertens und Ausgebens heißt im Englischen *read-eval-print loop* (kurz *REPL*).

```
Mini⟩  4711 * 2 + 815
10237
```

Die Zeichenfolge ‘Mini⟩ ’, das sogenannte *Prompt*, fordert den Benutzer oder die Benutzerin auf, eine Eingabe zu tätigen; das Ergebnis wird in der nächsten Zeile ausgegeben.

```
Mini⟩  11 * 11
121
Mini⟩  111 * 111
12321
Mini⟩  111111111 * 111111111
12345678987654321
```

Die Ergebnisse werden mit mathematischer Genauigkeit berechnet, so wie die Semantik es vorsieht. (Die Genauigkeit ist insbesondere nicht auf die native Genauigkeit von Rechnern, sei es 32 Bit oder 64 Bit, eingeschränkt: Wir rechnen mit mathematischen Zahlen, nicht mit Maschinenzahlen.)

3.3. Wertedefinitionen

> *The primary purpose of the* `DATA` *statement is to give names to constants; instead of referring to* π *as 3.141592653589793 at every appearance, the variable* `PI` *can be given that value with a* `DATA` *statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of* π *change.*
>
> — Fortran *manual for Xerox computers*

Rechnungen sind in der Regel nicht linear, sie enthalten Zwischen- oder Hilfsrechnungen: Soll zum Beispiel der Flächeninhalt eines Quadrats berechnet werden, wird man zunächst die Länge einer Seite bestimmen und dann das Ergebnis mit sich selbst multiplizieren. Ein modularer Aufbau von Rechnungen ist auch für unsere Programmiersprache wünschenswert. Um das Ergebnis einer Zwischenrechnung gegebenenfalls mehrfach verwenden zu können, geben wir ihm einen Namen. Ist e der Ausdruck, der die Länge der Seite berechnet, dann wird mit

let $s = e$

der Name oder im Fachjargon der *Bezeichner* s an den *Wert* von e gebunden. Aus diesem Grund heißt ***let*** $s = e$ auch *Wertedefinition* oder *Wertebindung*.

Die Syntax ist an die natürlichsprachliche Formulierung angelehnt, die man häufig in handschriftlichen Rechnungen und in mathematischen Texten findet: „sei $d = 2\pi$, …“. Nur ist, wie in der Programmierwelt üblich, das Schlüsselwort ***let*** dem Englischen entnommen.

Ein Bezeichner ist ein Ausdruck und kann somit überall dort verwendet werden, wo ein Ausdruck verlangt wird. Der Ausdruck

$$s * s$$

berechnet zum Beispiel den gewünschten Flächeninhalt des Quadrats. Der Bezeichner s wird in diesem Ausdruck zweimal verwendet.

Woher wissen wir, welche Bezeichner wir in einem Ausdruck verwenden können? Oder anders ausgedrückt, wie verbinden wir die Definition ***let*** $s = e$ mit dem Ausdruck $s * s$, in dem s verwendet wird? Die Antwort ist einfach: Wir führen ein weiteres linguistisches Konstrukt ein, das den Zusammenhang herstellt.

$$\textbf{\textit{let}}\ s = e\ \textbf{\textit{in}}\ s * s$$

Die vor dem Schlüsselwort ***in*** aufgeführte Wertedefinition darf in dem Ausdruck, der nach dem Schlüsselwort aufgeführt wird, verwendet werden. Man sagt auch, die definierten Bezeichner sind dort *sichtbar*. In unserem Beispiel erstreckt sich der Sichtbarkeitsbereich (engl. scope) des Bezeichners s auf den Ausdruck $s * s$.

Die Einschränkung der Sichtbarkeit von Bezeichnern hat den Vorteil, dass wir an verschiedenen Stellen im Programm den gleichen Bezeichner — gegebenenfalls auch für unterschiedliche Zwecke — verwenden können, ohne dass sich die verschiedenen Vorkommen ins Gehege kommen, eine wichtige Hygienemaßnahme. (Die Bedeutsamkeit dieses Features lässt sich erahnen, wenn man sich vergegenwärtigt, dass größere Softwaresysteme aus Millionen von Programmzeilen bestehen. Der Linux Kernel umfasst zum Beispiel zur Zeit circa 25 Millionen Codezeilen.)

Der ***in***-Ausdruck ist, wie der Name sagt, ein Ausdruck und kann somit überall dort verwendet werden, wo ein Ausdruck verlangt wird:

$$(\textbf{\textit{let}}\ s = e\ \textbf{\textit{in}}\ s * s) + 1$$

In diesem Beispiel ist der erste Summand ein ***in***-Ausdruck. Beachte, dass der Bezeichner s im zweiten Summanden *nicht* sichtbar ist. Wollen wir s auch dort verwenden, müssen wir die Addition in die Klammern schieben bzw. diese weglassen:

$$\textbf{\textit{let}}\ s = e\ \textbf{\textit{in}}\ s * s + s$$

Teilausdrücke zu benennen und mit Hilfe der vergebenen Namen wiederzuverwenden, gehört zu den grundlegenden Eigenschaften jeder Programmiersprache. Die Bezeichner können dabei frei gewählt werden: ***let*** $s = e$ ***in*** $s * s$ ist gleichwertig zu

$$\textbf{\textit{let}}\ \mathit{size} = e\ \textbf{\textit{in}}\ \mathit{size} * \mathit{size}$$

oder auch

$$\textbf{\textit{let}}\ \mathit{seitenlänge} = e\ \textbf{\textit{in}}\ \mathit{seitenlänge} * \mathit{seitenlänge}$$

Für das Ausrechnen und somit auch für den Rechner spielen Namen keine Rolle, wohl aber für den menschlichen Betrachter eines Programms. Deshalb sollte man sich bemühen, möglichst aussagekräftige Bezeichner zu vergeben. Wer viel programmiert, weiß, dass dies nicht immer leicht ist. Allgemein gilt: Je größer der Sichtbarkeitsbereich eines Namens, desto mehr Sorgfalt sollte man bei der Namenswahl walten lassen.

Wir haben bereits angemerkt, dass die gleichen Namen an unterschiedlichen Stellen eines Programms verwendet werden dürfen und auch unterschiedliches meinen können. Wenn sich die Sichtbarkeitsbereiche *nicht* überlappen, ist dies kein Problem:

$$(\textbf{\textit{let}}\ s = 4711\ \textbf{\textit{in}}\ s * s) + (\textbf{\textit{let}}\ s = 815\ \textbf{\textit{in}}\ s * s)$$

Wenn sich die Sichtbarkeitsbereiche überlagern wie in

$$\textbf{\textit{let}}\ s = 4711\ \textbf{\textit{in}}\ (\textbf{\textit{let}}\ s = 815\ \textbf{\textit{in}}\ (s * s))$$

dann müssen wir festlegen, auf welche Definition sich der Bezeichner s in $s * s$ bezieht. Wir vereinbaren, dass jeweils die nächste Definition — von innen nach außen gelesen — die Bedeutung eines Bezeichners festlegt. (Wir werden später sehen, dass dies tatsächlich die einzig sinnvolle Festlegung ist.) Das obige Programm ist somit zu

$$\textbf{\textit{let}}\ s_1 = 4711\ \textbf{\textit{in}}\ (\textbf{\textit{let}}\ s_2 = 815\ \textbf{\textit{in}}\ (s_2 * s_2))$$

gleichwertig.

Abstrakte Syntax Wir führen eine neue syntaktische Kategorie ein: *Deklarationen*. Für's Erste ist eine Deklaration eine nicht-leere Sequenz von Wertedefinitionen.

$x \in \mathsf{Id}$	*Bezeichner*
$d \in \mathsf{Decl} ::=$	*Deklarationen:*
$\mid\ \textbf{\textit{let}}\ x = e$	Wertedefinition
$\mid\ d_1\ d_2$	Sequenz von Deklarationen

Der Bereich der Bezeichner wird später in der lexikalischen Syntax genau definiert. Für den Moment halten wir fest, dass ein Bezeichner mit einem Buchstaben anfängt; danach können weitere Buchstaben, Ziffern und Sonderzeichen wie ein Unterstrich oder ein Apostroph folgen.

Sequenzen von Deklarationen werden in der *konkreten* Syntax untereinander geschrieben: $\textbf{\textit{let}}\ s = 4711\ \textbf{\textit{let}}\ a = s * s$ zum Beispiel wird konkret notiert

$$\textbf{\textit{let}}\ s = 4711$$
$$\textbf{\textit{let}}\ a = s * s$$

Die Kategorie der Ausdrücke wird um Bezeichner und lokale Definitionen erweitert.

$e ::= \cdots$	*lokale Definitionen:*
$\mid\ x$	Bezeichner
$\mid\ d\ \textbf{\textit{in}}\ e$	lokale Definition

Ein ***in***-Ausdruck verknüpft eine Definition mit einem Ausdruck. Der Teilausdruck e heißt Rumpf des ***in***-Ausdrucks.

Statische Semantik Die Semantikregeln, statische wie dynamische, sind idealerweise *kompositional*. Die Bedeutung eines Ausdrucks wird auf die Bedeutung seiner Teilausdrücke zurückgeführt. Diese wünschenswerte Eigenschaft stellt uns bei der Typisierung von ***in***-Ausdrücken wie ***let*** $s = 4711 + 815$ ***in*** $s * s$ vor ein Problem: Wenn wir den Teilausdruck $s*s$ typisieren, woher kennen wir den Typ von s? Technisch gesprochen enthält der isolierte Ausdruck $s * s$ einen sogenannten *freien* Bezeichner, ein Bezeichner der in dem Ausdruck selbst nicht definiert wird. Nun kann ein Bezeichner wie gesagt an unterschiedlichen Stellen im Programm unterschiedliches bedeuten und insbesondere auch einen unterschiedlichen Typ besitzen.

$$(\textbf{\textit{let}}\ s = \mathit{true}\ \textbf{\textit{in}}\ s)\ \texttt{\&\&}\ (\textbf{\textit{let}}\ s = 815\ \textbf{\textit{in}}\ s * s > 4711)$$

Die Lösung dieses Problems ist vielleicht naheliegend: Wir müssen uns die Typen der jeweils sichtbaren Bezeichner merken. Diesem Zweck dient eine sogenannte *Signatur* (engl. signature oder interface), eine endliche Abbildung von Bezeichnern auf Typen.

$$\Sigma \in \mathsf{Sig} = \mathsf{Id} \to_{\text{fin}} \mathsf{Type} \qquad \textit{Signatur}$$

Wir erweitern unsere Typregeln entsprechend um Signaturen. Aus der zweistelligen Relation $e : t$ wird die dreistellige Relation

$$\Sigma \vdash e : t$$

zwischen Signaturen, Ausdrücken und Typen. *Lies:* „bezüglich der Signatur Σ hat e den Typ t“. (Der Hammer „$\vdash$“ dient traditionell als Trennsymbol, ähnlich wie ein Punkt oder ein Semikolon im Deutschen.) Die ursprüngliche Relation $e : t$ gilt ab sofort als Abkürzung für $\varnothing \vdash e : t$ — wir nehmen an, dass die Signatur leer ist. Die bisherigen Typregeln müssen entsprechend angepasst werden, zum Beispiel:

$$\frac{\Sigma \vdash e_1 : \mathit{Bool} \qquad \Sigma \vdash e_2 : t \qquad \Sigma \vdash e_3 : t}{\Sigma \vdash \textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3 : t}$$

Die Signatur wird unverändert an die Teilausdrücke „weitergereicht“.

Kommen wir zu den Wertedefinitionen. Die statische Semantik ordnet einem Ausdruck einen Typ zu: dem Ausdruck $4711+815$ wird zum Beispiel der Typ Nat zugeordnet. Analog wird einer Wertedefinition eine Signatur zugeordnet: der Definition ***let*** $x = 4711+815$ wird zum Beispiel die Signatur $\{x \mapsto \mathit{Nat}\}$ zugeordnet. Eine Signatur repräsentiert — ähnlich wie ein Typ — das, was wir über eine Definition statisch wissen wollen.

$$\frac{\Sigma \vdash e : t}{\Sigma \vdash (\textbf{\textit{let}}\ x = e) : \{x \mapsto t\}} \qquad \frac{\Sigma \vdash d_1 : \Sigma_1 \qquad \Sigma, \Sigma_1 \vdash d_2 : \Sigma_2}{\Sigma \vdash d_1\ d_2 : \Sigma_1, \Sigma_2}$$

Die Typregeln für die Sequenz von Deklarationen verdient besondere Beachtung. Wir haben schon besprochen, dass in einer Folge von Deklarationen die späteren Deklarationen die früheren „sehen“: In ***let*** $s = e$ ***let*** $a = s * s$ bezieht sich s in $s * s$ auf die

vorangegangene Bindung ***let*** $s = e$. Die gesamte Deklaration wird wie folgt typisiert:

$$\dfrac{\dfrac{\dfrac{\vdots}{\varnothing \vdash e : \mathit{Nat}}}{\varnothing \vdash (\textbf{\textit{let}}\ s = e) : \{s \mapsto \mathit{Nat}\}} \qquad \dfrac{\dfrac{\overline{\{s \mapsto \mathit{Nat}\} \vdash s : \mathit{Nat}} \qquad \overline{\{s \mapsto \mathit{Nat}\} \vdash s : \mathit{Nat}}}{\{s \mapsto \mathit{Nat}\} \vdash s * s : \mathit{Nat}}}{\{s \mapsto \mathit{Nat}\} \vdash (\textbf{\textit{let}}\ a = s * s) : \{a \mapsto \mathit{Nat}\}}}{\varnothing \vdash (\textbf{\textit{let}}\ s = e\ \textbf{\textit{let}}\ a = s * s) : \{s \mapsto \mathit{Nat}, a \mapsto \mathit{Nat}\}}$$

In der Signatur vor dem Symbol '$\vdash$' wird protokolliert, welche Bezeichner in einem Ausdruck oder einer Deklaration sichtbar sind. Wir starten „unten" mit der leeren Signatur $\varnothing$. Die erste Wertedefinition resultiert in der Signatur $\{s \mapsto \mathit{Nat}\}$. Die zweite Wertedefinition wird bezüglich $\varnothing, \{s \mapsto \mathit{Nat}\} = \{s \mapsto \mathit{Nat}\}$ typisiert und resultiert in der Bindung $\{a \mapsto \mathit{Nat}\}$. Zusammen erhalten wir $\{s \mapsto \mathit{Nat}\}, \{a \mapsto \mathit{Nat}\} = \{s \mapsto \mathit{Nat}, a \mapsto \mathit{Nat}\}$. Das Komma ist der Kommaoperator aus Abschnitt 2.1.

Spätere Wertedefinitionen können frühere „verschatten", wenn sie die gleichen Bezeichner verwenden. Der Deklaration ***let*** $s = \mathit{false}$ ***let*** $s = 4711$ wird die Signatur $\{s \mapsto \mathit{Nat}\}$ zugeordnet. Man spricht von Verschattung, weil die erste Definition prinzipiell zwar sichtbar ist, aber nicht mehr auf sie zugegriffen werden kann, da der Bezeichner s inzwischen anderweitig verwendet wird. Formal wird die Verschattung durch den Kommaoperator implementiert: in Σ_1, Σ_2 werden bei Überschneidungen den Bindungen in Σ_2 Vorrang eingeräumt.

$$\dfrac{\dfrac{\overline{\varnothing \vdash \mathit{false} : \mathit{Bool}}}{\varnothing \vdash (\textbf{\textit{let}}\ s = \mathit{false}) : \{s \mapsto \mathit{Bool}\}} \qquad \dfrac{\overline{\{s \mapsto \mathit{Bool}\} \vdash 4711 : \mathit{Nat}}}{\{s \mapsto \mathit{Bool}\} \vdash (\textbf{\textit{let}}\ s = 4711) : \{s \mapsto \mathit{Nat}\}}}{\varnothing \vdash (\textbf{\textit{let}}\ s = \mathit{false}\ \textbf{\textit{let}}\ s = 4711) : \{s \mapsto \mathit{Nat}\}}$$

Die Signatur der zwei Bindungen ist $\{s \mapsto \mathit{Nat}\}$, da $\{s \mapsto \mathit{Bool}\}, \{s \mapsto \mathit{Nat}\} = \{s \mapsto \mathit{Nat}\}$.

Wenden wir uns den Ausdrücken zu, die wir um Bezeichner und ***in***-Ausdrücke erweitert haben. Der Typ eines Bezeichners wird in der Signatur nachgeschlagen.

$$\dfrac{}{\Sigma \vdash x : \Sigma(x)} \qquad x \in \mathit{dom}\ \Sigma$$

Zur Erinnerung: Σ ist eine endliche Abbildung; damit $\Sigma(x)$ definiert ist, muss x im Definitionsbereich von Σ enthalten sein. Mit anderen Worten, zu jedem Bezeichner muss eine definierende Bindung existieren. Der Ausdruck $4711 * (\mathit{a11} + 815)$ aus Abschnitt 2.2 ist zum Beispiel nicht wohlgetypt (bezüglich der leeren Signatur $\varnothing$), da der Bezeichner $\mathit{a11}$ nicht definiert ist. Als Teil eines größeren Programms, zum Beispiel ***let*** $\mathit{a11} = 271828$ ***in*** $4711 * (\mathit{a11} + 815)$, ist der Ausdruck aber okay.

Die Typregel für lokale Definitionen ähnelt der Regel für die Sequenz von Deklarationen.

$$\dfrac{\Sigma \vdash d : \Sigma' \qquad \Sigma, \Sigma' \vdash e : t}{\Sigma \vdash (d\ \textbf{\textit{in}}\ e) : t}$$

Der Kommaoperator hat einen weiteren Auftritt: Der Rumpf des ***in***-Ausdrucks wird bezüglich der erweiterten Signatur Σ, Σ' typisiert.

Fassen wir zusammen: Die statische Semantik ordnet

- einem Ausdruck einen Typ zu, $\Sigma \vdash e : t$, und
- einer Definition eine Signatur, $\Sigma \vdash d : \Sigma'$.

Die statische Semantik typisiert nur Ausdrücke und Definitionen, deren freie Bezeichner in der Signatur aufgeführt werden. — Bevor ein Teilausdruck, der freie Bezeichner enthält, typisiert wird, wird zunächst die Signatur um die Typen der freien Bezeichner erweitert. Dies ist die große *Invariante der statischen Semantik*.

Dynamische Semantik Die dynamische Semantik ordnet einem Ausdruck einen Wert zu: $4711 + 815$ wird zum Beispiel der Wert 5526 zugeordnet. Analog wird der Wertebindung $\mathit{let}\ x = 4711 + 815$ die sogenannte *Umgebung* $\{x \mapsto 5526\}$ zugeordnet. Ähnlich wie eine Signatur ist eine Umgebung (engl. environment) eine endliche Abbildung; im Unterschied zur Signatur bildet eine Umgebung Bezeichner auf Werte ab.

$$\delta \in \mathsf{Env} = \mathsf{Id} \to_{\text{fin}} \mathsf{Val} \qquad \textit{Umgebung}$$

Wir erweitern unsere Auswertungsregeln entsprechend um Umgebungen. Aus der zweistelligen Relation $e \Downarrow \nu$ wird die dreistellige Relation

$$\delta \vdash e \Downarrow \nu$$

zwischen Umgebungen, Ausdrücken und Werten. *Lies:* „bezüglich der Umgebung δ wertet e zu dem Wert ν aus“. Die ursprüngliche Relation $e \Downarrow \nu$ dient ab sofort als Abkürzung für $\varnothing \vdash e \Downarrow \nu$ — wir nehmen an, dass die Umgebung leer ist. Die bisherigen Auswertungsregeln müssen entsprechend angepasst werden, zum Beispiel:

$$\frac{\delta \vdash e_1 \Downarrow \mathit{true} \qquad \delta \vdash e_2 \Downarrow \nu}{\delta \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow \nu} \qquad \frac{\delta \vdash e_1 \Downarrow \mathit{false} \qquad \delta \vdash e_3 \Downarrow \nu}{\delta \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow \nu}$$

Die Umgebung wird jeweils unverändert an die Teilausdrücke „weitergereicht“.

Für jedes neu eingeführte Konstrukt gibt es jeweils eine Auswertungsregel.

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash (\mathit{let}\ x = e) \Downarrow \{x \mapsto \nu\}} \qquad \frac{\delta \vdash d_1 \Downarrow \delta_1 \qquad \delta, \delta_1 \vdash d_2 \Downarrow \delta_2}{\delta \vdash d_1\ d_2 \Downarrow \delta_1, \delta_2}$$

$$\frac{}{\delta \vdash x \Downarrow \delta(x)} \qquad \frac{\delta \vdash d \Downarrow \delta' \qquad \delta, \delta' \vdash e \Downarrow \nu}{\delta \vdash (d\ \mathit{in}\ e) \Downarrow \nu}$$

In der Auswertungsregel für Bezeichner ist durch die statische Semantik sichergestellt, dass der Bezeichner x stets definiert ist: $x \in \mathit{dom}\ \delta$. In der Regel für *in*-Ausdrücke regelt der Kommaoperator analog zur statischen Semantik Überschneidungen.

Für unser laufendes Beispiel erhalten wir

$$\frac{\dfrac{\dfrac{\vdots}{\varnothing \vdash e \Downarrow 4711}}{\varnothing \vdash \mathit{let}\ s = e \Downarrow \{s \mapsto 4711\}} \qquad \dfrac{\dfrac{\{s \mapsto 4711\} \vdash s \Downarrow 4711 \qquad \{s \mapsto 4711\} \vdash s \Downarrow 4711}{\{s \mapsto 4711\} \vdash s * s \Downarrow 22193521}}{\{s \mapsto 4711\} \vdash \mathit{let}\ a = s * s \Downarrow \{a \mapsto 22193521\}}}{\varnothing \vdash \mathit{let}\ s = e\ \mathit{let}\ a = s * s \Downarrow \{s \mapsto 4711, a \mapsto 22193521\}}$$

Die erste Wertedefinition ***let*** $s = e$ resultiert in der Umgebung $\{s \mapsto 4711\}$. Bezüglich dieser Umgebung wird die zweite Wertedefinition ***let*** $a = s*s$ ausgerechnet. Abschließend werden beide Umgebungen mit dem Kommaoperator verknüpft.

Fassen wir zusammen: Die dynamische Semantik ordnet

- einem Ausdruck einen Wert zu, $\delta \vdash e \Downarrow \nu$ und
- einer Definition eine Umgebung, $\delta \vdash d \Downarrow \delta'$.

Die dynamische Semantik legt nur die Bedeutung von Ausdrücken und Definitionen fest, deren freie Bezeichner in der Umgebung aufgeführt werden. — Bevor ein Teilausdruck, der freie Bezeichner enthält, ausgewertet wird, wird zunächst die Umgebung um die Werte der freien Bezeichner erweitert. Dies ist die große *Invariante der dynamischen Semantik*.

Vertiefung Wir haben schon angesprochen, dass Bezeichner frei gewählt werden können: ***let*** $s = e$ ***in*** $s * s$ ist gleichwertig zu

let *size* $= e$ ***in*** *size* $*$ *size* oder ***let*** *seitenlänge* $= e$ ***in*** *seitenlänge* $*$ *seitenlänge*

Dass Namen Schall und Rauch sind, ist vielleicht klar, wenn wir die obigen Ausdrücke isoliert für sich betrachten. Die Bedeutung der Ausdrücke bleibt aber ebenso unverändert, wenn sie *Teil* eines größeren Programms sind. Zum Beispiel: Der Bezeichner s im rechten ***in***-Ausdruck

let $s = 4711$ ***in let*** $s = 815$ ***in*** $s * s$

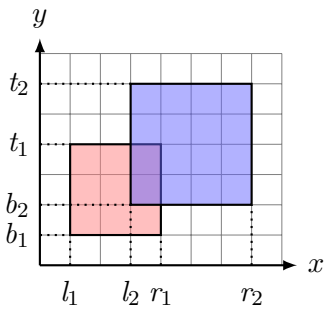
kann zu *size* umbenannt werden,

let $s = 4711$ ***in let*** *size* $= 815$ ***in*** *size* $*$ *size*

ohne die Bedeutung des Programms zu ändern. Das liegt daran, dass die Auswertungsregeln „neuen“ Definitionen Vorrang einräumen. (Würden wir in der Auswertungsregel für ***in***-Ausdrücke die Argumente des Kommaoperators vertauschen, aus $\delta, \delta' \vdash e \Downarrow \nu$ wird $\delta', \delta \vdash e \Downarrow \nu$, dann hätten die beiden Ausdrücke tatsächlich eine unterschiedliche Bedeutung — eine grausame Vorstellung.)

module
Values.
Area

Vertiefung: Wohnfläche Obwohl unsere Programmiersprache noch vergleichsweise spartanisch ausgestattet ist, können wir mit ihrer Hilfe schon erste kleinere Aufgaben lösen. Schauen wir uns ein — nur auf den ersten Blick einfaches — Beispiel an. Die Fläche einer Wohnung mit dem unten skizzierten Grundriss soll berechnet werden.




```
let l1 = 1
let r1 = 4
let b1 = 1
let t1 = 4
let l2 = 3
let r2 = 7
let b2 = 2
let t2 = 6
```

Geometrisch gesehen besteht die zu berechnende Fläche aus zwei sich überlappenden Quadraten. (Das ist aller Wahrscheinlichkeit nach eine Vereinfachung, das Ergebnis des schon angesprochenen Abstraktionsprozesses, mit dem wir Aufgaben in Rechenaufgaben verwandeln.) Wir nehmen weiterhin an, dass die Quadrate durch die eingezeichneten x- und y-Koordinaten gegeben sind (*l*eft, *r*ight, *b*ottom, *t*op).

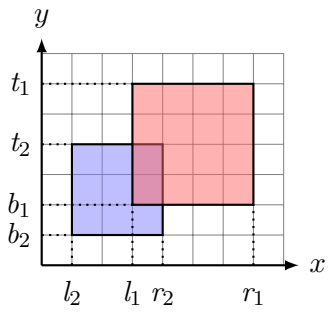
Die Gesamtfläche ergibt sich dann als Summe der beiden Quadratflächen minus der Schnittfläche, die die Form eines Rechtecks hat. Ausgedrückt in Mini–F#:

```
let s1 = r1 - l1
let s2 = r2 - l2
let w = r1 - l2
let h = t1 - b2
s1 * s1 + s2 * s2 - w * h
```

Für die Seitenlänge der Quadrate (*s*ize) führen wir zwei Bezeichner ein, ebenso für die Breite (*w*idth) und Höhe des Rechtecks (*h*eight). Letzteres dient dazu, die Lesbarkeit des Programms zu verbessern; ersteres vermeidet Mehrfachberechnungen: Formuliert man $(r_1 - l_1) * (r_1 - l_1)$ statt $s_1 * s_1$, wird die Teilrechnung $r_1 - l_1$ zweimal durchgeführt. Die vier Bezeichner werden dann in der Flächenformel verwendet. (Allerdings benötigen wir weder b_1 noch t_2. Warum?)

```
Mini⟩ ...
Mini⟩ s1 * s1 + s2 * s2 - w * h
23
```

Die Wohnfläche beträgt somit $23\,m^2$ — ein kurzer Blick auf den Grundriss bestätigt das Ergebnis. Ändern wir die Daten, indem wir zum Beispiel die Koordinaten der beiden Quadrate “vertauschen”,

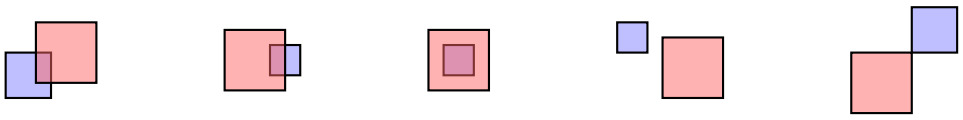


let $l_1 = 3$
let $r_1 = 7$
let $b_1 = 2$
let $t_1 = 6$
let $l_2 = 1$
let $r_2 = 4$
let $b_2 = 1$
let $t_2 = 4$

erleben wir eine unangenehme Überraschung.

```
Mini⟩ ...
Mini⟩ s1 * s1 + s2 * s2 ÷ w * h
0
```

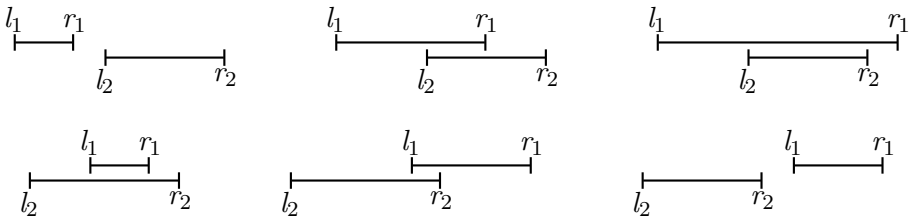
Was läuft schief? Nun, wir haben uns zu sehr von der Skizze des Grundrisses leiten lassen: Unser Programm macht unausgesprochene Annahmen über die relative Lage der beiden Quadrate. (Welche?) Wollen wir das Programm "robuster" machen, müssen wir die Lage der Quadrate berücksichtigen. Das Problem ist nur, dass viele unterschiedliche Konstellationen denkbar sind. Hier ist eine kleine Auswahl:



Wie werden wir Herr oder Frau der Lage?

Eine entscheidende Einsicht ist, dass wir Überschneidungen *getrennt* für jede der beiden Koordinatenachsen ausrechnen können und auf diese Weise ein 2-dimensionales Problem auf ein 1-dimensionales Problem zurückführen können. Jedes Quadrat definiert ein Intervall auf der x-Achse, in unserem Fall (l_1, r_1) bzw. (l_2, r_2), und ein Intervall auf der y-Achse, (b_1, t_1) bzw. (b_2, t_2). Die beiden Quadrate überlappen sich genau dann, wenn sich sowohl die beiden x-Intervalle als auch die beiden y-Intervalle überlappen.

Konzentrieren wir uns auf die x-Achse — die Überlegungen gelten entsprechend für die y-Achse. Wenn wir annehmen, dass $l_1 < r_1$ und $l_2 < r_2$ gilt, dann ergeben sich sechs mögliche Konstellationen:



Die beiden Intervalle haben also einen *leeren* Durchschnitt genau dann, wenn die rechte Grenze des einen vor der linken Grenze des anderen Intervalls liegt: $r_1 < l_2 \mid\mid r_2 < l_1$.

Umgekehrt liegt eine Überschneidung vor, wenn $r_1 \geqslant l_2$ `&&` $r_2 \geqslant l_1$. Hier kommen die Gesetze der Logik und der Arithmetik zum Einsatz: die eine Formel ergibt sich aus der anderen durch Negation, Anwendung der *De Morganschen Gesetze*,

$$not\ (a\ \texttt{\&\&}\ b) = not\ a\ \texttt{||}\ not\ b$$
$$not\ (a\ \texttt{||}\ b) = not\ a\ \texttt{\&\&}\ not\ b$$

und der Beziehungen:

$$not\ (a \leqslant b) = a > b$$
$$not\ (a < b) = a \geqslant b$$

Nun sind wir nicht nur an der Frage interessiert, ob es eine Überschneidung gibt, sondern wir benötigen insbesondere die Länge der Überschneidung. Diese ist gegeben durch die Formel $min\ r_1\ r_2 \dot{-} max\ l_1\ l_2$, wobei $min\ a\ b$ das Minimum von a und b und $max\ a\ b$ entsprechend deren Maximum berechnet. Der Ausdruck $min\ a\ b$ ist eine Abkürzung für die Alternative ***if*** $a \leqslant b$ ***then*** a ***else*** b und $max\ a\ b$ kürzt entsprechend ***if*** $a \leqslant b$ ***then*** b ***else*** a ab. Nach diesen Vorarbeiten erhalten wir das folgende Programm.

$$\begin{array}{l}
\textbf{\textit{let}}\ s_1 = r_1 \dot{-} l_1 \\
\textbf{\textit{let}}\ s_2 = r_2 \dot{-} l_2 \\
\textbf{\textit{let}}\ w\ = min\ r_1\ r_2 \dot{-} max\ l_1\ \ l_2 \\
\textbf{\textit{let}}\ h\ \ = min\ t_1\ \ t_2 \dot{-} max\ b_1\ b_2 \\
s_1 * s_1 + s_2 * s_2 \dot{-} w * h
\end{array}$$

Jetzt erhalten wir für beide Datensätze das gleiche Ergebnis: 23.

Wir werden auf das Problem der Berechnung der Wohnfläche zu einem späteren Zeitpunkt noch einmal zurückkommen — wenn wir etwas mehr Vokabular zur Verfügung haben, um die Lösung flexibler und modularer zu gestalten.

3.4. Funktionsdefinitionen

Im letzten Abschnitt haben wir mit Hilfe des Programms

$$\textbf{\textit{let}}\ s = e\ \textbf{\textit{in}}\ s * s$$

den Flächeninhalt eines Quadrats mit der Seitenlänge e berechnet. Wir können das obige Programm verallgemeinern, indem wir von der gegebenen Seitenlänge e *abstrahieren*. Aus dem Ausdruck $s * s$ wird eine Funktion in s.

$$\textbf{\textit{let}}\ area\ (s : Nat) : Nat = s * s$$

Der Bezeichner *area* ist der Name der Funktion, s ist der *formale Parameter* und $s * s$ heißt *Rumpf* der Funktion. Um jetzt den Flächeninhalt für eine gegebene Seitenlänge e zu berechnen, wenden wir die Funktion auf e an.

$area\ (e)$

Den Ausdruck e nennen wir das Argument oder den *aktuellen Parameter* der Funktion.

Ein Funktionsaufruf wie $area\ (e)$ wird ausgerechnet, indem im Rumpf der formale Parameter durch den aktuellen Parameter ersetzt und der resultierende Ausdruck dann ausgewertet wird. Mit anderen Worten, der Funktionsaufruf $area\ (e)$ ist gleichwertig zu dem Ausdruck $\mathbf{let}\ s = e\ \mathbf{in}\ s * s$, dem Ausgangspunkt unserer Überlegungen. Der Vorteil einer Funktionsdefinition ist, dass die Funktion mehrfach angewendet werden kann.

$area\ (e_1) + area\ (e_2)$

Ohne Funktionen im Repertoire müssten wir formulieren

$(\mathbf{let}\ s = e_1\ \mathbf{in}\ s * s) + (\mathbf{let}\ s = e_2\ \mathbf{in}\ s * s)$

Je größer der Funktionsrumpf und je öfter man eine Funktion verwendet, desto größer ist die Ökonomie einer Funktionsdefinition.

Funktionsdefinitionen gehören wie Wertedefinitionen zu den Deklarationen und können wie diese in $\mathbf{in}$-Ausdrücken verwendet werden.

$\mathbf{let}\ area\ (s : Nat) : Nat = s * s$
$\mathbf{in}\ \ area\ (e_1) + area\ (e_2)$

Die Sichtbarkeit des Funktionsbezeichners *area* erstreckt sich auf den Rumpf des $\mathbf{in}$-Ausdrucks; der formale Parameter s ist hingegen *nur* im Funktionsrumpf sichtbar.

Funktionen kennen wir aus der Mathematik; die Funktionsdefinitionen in diesem und in den nächsten beiden Kapiteln beschreiben Funktionen im mathematischen Sinne — dies wird sich in den darauffolgenden Kapiteln ändern.[2] Liebgewonnene Funktionen aus Kurvendiskussionen wie $f(x) = 2x^2 + x$ lassen sich in unserer Sprache einfach nachprogrammieren.

$\mathbf{let}\ f\ (x : Nat) : Nat = 2 * x * x + x$

Der einzige Unterschied ist, dass unsere Funktionen auf den natürlichen Zahlen arbeiten, und nicht auf den reellen Zahlen. (Reelle Zahlen lassen sich auf dem Rechner nur unvollkommen nachbilden; es gibt einfach zu viele davon. Deshalb beschränken wir uns in der Vorlesung im Wesentlichen auf die natürlichen Zahlen.)

Abstrakte Syntax Wir erweitern Deklarationen um Funktionsdefinitionen und Ausdrücke um Funktionsapplikationen (das ist der vornehme Name für Funktionsaufrufe oder Funktionsanwendungen).

$f \in \mathsf{Id}$

$d ::= \cdots$	*Deklarationen:*
$\quad \mid\ \mathbf{let}\ f\ (x : t_1) : t_2 = e$	Funktionsdefinition
$e ::= \cdots$	*Funktionsausdrücke:*
$\quad \mid\ f\ (e)$	Funktionsapplikation

[2] Eine Ausnahme gibt es auch in diesem Kapitel: Die instrumentierte Version von *player-A* in Abschnitt 3.6 ist keine Funktion im mathematischen Sinne.

Statische Semantik Eine Funktion mit den Argumenttyp t_1 und dem Ergebnistyp t_2 erhält den Typ $t_1 \to t_2$. *Lies:* t_1 nach t_2.

$t ::= \cdots$	*Typen:*
$\mid t_1 \to t_2$	Funktionstyp

Bei der Definition einer Funktion müssen der Argument- und der Ergebnistyp angegeben werden. (In der Mathematik würde man statt vom Argument- und Ergebnistyp, vom Definitions- und Wertebereich sprechen.)

$$\frac{\Sigma, \{x_1 \mapsto t_1\} \vdash e_2 : t_2}{\Sigma \vdash (\textbf{\textit{let}}\ f\ (x_1 : t_1) : t_2 = e_2) : \{f \mapsto t_1 \to t_2\}}$$

$$\frac{\Sigma(f) = t_1 \to t_2 \qquad \Sigma \vdash e_1 : t_1}{\Sigma \vdash f\ (e_1) : t_2} \qquad f \in dom\ \Sigma$$

Der Rumpf einer Funktion wird in der um den formalen Parameter erweiterten Signatur getypt (‘,’ ist der Kommaoperator). Bei der Funktionsanwendung schlagen wir den Typ der Funktion in der Signatur nach und stellen dann sicher, dass der aktuelle Parameter den gleichen Typ hat wie der formale. Der Typ der Funktionsanwendung entspricht dem Ergebnistyp der Funktion.

Dynamische Semantik Was ist der Wert einer Funktion? Können wir Funktionen überhaupt auswerten? Die Antwort ist vielleicht überraschend: Da wir die Bindung des formalen Parameters bei der Definition nicht kennen, ist es nicht möglich, eine Funktion auszurechnen; allerdings ist dies auch nicht notwendig. Wir verzögern die Auswertung einfach, bis wir den aktuellen Parameter kennen. Mit anderen Worten, eine Funktion wertet im Wesentlichen zu sich selbst aus.

„Im Wesentlichen“, weil wir noch berücksichtigen müssen, dass der Rumpf einer Funktionsdefinition möglicherweise freie Bezeichner enthält, deren Werte in der Umgebung protokolliert sind. Betrachten wir ein Beispiel:

$$\frac{\dfrac{\varnothing \vdash 2 \Downarrow 2}{\varnothing \vdash \textbf{\textit{let}}\ d = 2 \Downarrow \{d \mapsto 2\}} \qquad \dfrac{}{\{d \mapsto 2\} \vdash \textbf{\textit{let}}\ next\ (n) = n + d \Downarrow \{next \mapsto ?\}}}{\varnothing \vdash \textbf{\textit{let}}\ d = 2\ \textbf{\textit{let}}\ next\ (n) = n + d \Downarrow \{d \mapsto 2, next \mapsto ?\}}$$

Wenn wir die Auswertung der Funktion *next* “einfrieren”, um sie zu einem späteren Zeitpunkt fortzusetzen (“aufzutauen”), müssen wir uns nicht nur die Funktionsdefinition selbst, sondern auch die aktuelle Umgebung $\{d \mapsto 2\}$ merken.

Entsprechend erweitern wir den Bereich der Werte um sogenannte *Funktionsabschlüsse*.

$\nu ::= \cdots$	*Werte:*
$\mid \langle \delta, x, e \rangle$	Funktionsabschluss

Ein Funktionsabschluss besteht aus einer Umgebung, dem formalen Parameter und dem Rumpf der definierten Funktion. Mit der Einführung von Funktionsabschlüssen hängt der Bereich der Werte somit vom Bereich der Ausdrücke ab!

Damit können wir das obige Beispiel vervollständigen:

$$\dfrac{\dfrac{\varnothing \vdash 2 \Downarrow 2}{\varnothing \vdash \textbf{\textit{let}}\ d = 2 \Downarrow \{d \mapsto 2\}} \qquad \dfrac{}{\{d \mapsto 2\} \vdash \textbf{\textit{let}}\ next\ (n) = n + d \Downarrow \{next \mapsto \gamma\}}}{\varnothing \vdash \textbf{\textit{let}}\ d = 2\ \textbf{\textit{let}}\ next\ (n) = n + d \Downarrow \{d \mapsto 2, next \mapsto \gamma\}}$$

wobei $\gamma = \langle \{d \mapsto 2\},\ n,\ n + d \rangle$

Der Funktionsabschluss repräsentiert eine Funktion, die ihr Argument um 2 erhöht.

Eine Funktionsdefinition wertet somit zu einer Bindung aus, in der der Funktionsname an einen Funktionsabschluss gebunden ist.

$$\dfrac{}{\delta \vdash (\textbf{\textit{let}}\ f\ (x) = e) \Downarrow \{f \mapsto \langle \delta,\ x,\ e \rangle\}}$$

$$\dfrac{\delta \vdash e_1 \Downarrow \nu_1 \qquad \delta', \{x_1 \mapsto \nu_1\} \vdash e \Downarrow \nu}{\delta \vdash f\ (e_1) \Downarrow \nu} \qquad \delta(f) = \langle \delta',\ x_1,\ e \rangle$$

Wenn eine Funktion aufgerufen wird, haben wir alles beisammen, um die verzögerte Berechnung fortzusetzen: den formalen Parameter x_1, den aktuellen Parameter e_1 und den Funktionsrumpf e. Der Funktionsaufruf $f\ (e_1)$ mit $f\ (x_1) = e$ ist gleichwertig zu $\textbf{\textit{let}}\ x_1 = e_1\ \textbf{\textit{in}}\ e$ und wird entsprechend ausgewertet. (Die statische Semantik stellt sicher, dass $f \in dom\ \delta$.) Das folgende Beispiel illustriert die einzelnen Auswertungsschritte.

$$\dfrac{\dfrac{}{\varnothing \vdash \textbf{\textit{let}}\ area\ (s) = s * s \Downarrow \delta} \qquad \dfrac{\dfrac{}{\delta \vdash 4711 + 815 \Downarrow 5526} \quad \dfrac{}{\{s \mapsto 5526\} \vdash s * s \Downarrow 30536676}}{\delta \vdash area\ (4711 + 815) \Downarrow 30536676}}{\varnothing \vdash \textbf{\textit{let}}\ area\ (s) = s * s\ \textbf{\textit{in}}\ area\ (4711 + 815) \Downarrow 30536676}$$

wobei $\delta = \{area \mapsto \langle \varnothing,\ s,\ s * s \rangle\}$

Der Bezeichner *area* wird an den Funktionsabschluss $\langle \varnothing,\ s,\ s * s \rangle$ gebunden. Der Abschluss repräsentiert eine Funktion, die jeder natürlichen Zahl ihren Quadratwert zuordnet.

Vertiefung Auf die Funktion *area* werden wir im Folgenden des Öfteren zurückgreifen. Deswegen benennen wir sie in *square* um — *area* ist mehrdeutig und könnte auch den Flächeninhalt eines Kreises berechnen.

$$\textbf{\textit{let}}\ square\ (n : Nat) : Nat = n * n$$

In Abschnitt 3.1 haben wir gezeigt, wie sich Boolesche Verknüpfungen mit Hilfe von Fallunterscheidungen programmieren lassen. Es ist verlockend, die dort eingeführten Abkürzungen *not* e, e_1 `&&` e_2 und e_1 `||` e_2 (Konstrukte der Metasprache) durch ordentliche Funktionsdefinitionen (Konstrukte der Objektsprache Mini–F#) zu ersetzen.

$$\textbf{\textit{let}}\ not\ (a : Bool) : Bool = \textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ false\ \textbf{\textit{else}}\ true$$

Formal gesehen tritt an die Stelle der Metavariablen e — in Abschnitt 3.1 hatten wir den Ausdruck ***if*** *e* ***then*** *false* ***else*** *true* für die Berechnung der Negation eingeführt — der Bezeichner *a* der Programmiersprache — ***if*** *a* ***then*** *false* ***else*** *true* im Rumpf der Funktion.

Konjunktion und Disjunktion sind binäre Operationen, sie hängen von *zwei* Argumenten ab. Wir erweitern entsprechend Funktionendefinitionen und Funktionsapplikationen auf mehrparametrige Funktionen.

let *and-also* $(a : \mathit{Bool}, b : \mathit{Bool}) : \mathit{Bool} =$ ***if*** *a* ***then*** *b* ***else*** *false*
let *or-else* $(a : \mathit{Bool}, b : \mathit{Bool}) : \mathit{Bool} =$ ***if*** *a* ***then*** *true* ***else*** *b*

Die Formalisierung dieser Erweiterung holen wir in Abschnitt 4.1.1 nach.

Sind die programmierten Funktionen ein gleichwertiger Ersatz für die liebgewonnenen Abkürzungen? Im Fall der Negation ja, bei den binären Operationen ergeben sich Unterschiede. Zunächst einmal syntaktische. Wenn wir mehr als eine Bedingung konjunktiv verknüpfen, müssen wir die Aufrufe von *and-also* entsprechend schachteln:

and-also $(e_1,$ *and-also* $(e_2, e_3))$

oder alternativ

and-also $($*and-also* $(e_1, e_2), e_3)$

Beide Ausdrücke haben stets das gleiche Ergebnis: die Konjunktion ist *assoziativ*. Assoziative Funktionen notiert man vorteilhafter *infix*: Der Funktionsname kommt zwischen die Argumente.

e_1 `&&` e_2 `&&` e_3

Neben diesem syntaktischen Unterschied gibt es noch einen gewichtigeren semantischen Unterschied zwischen `&&` und *and-also*. Damit beschäftigt sich Aufgabe 3.3. Wir greifen die Diskussion auch im nächsten Abschnitt noch einmal auf.

3.5. Funktionsausdrücke

Für die Auswertung von Funktionen haben wir Funktionsabschlüsse eingeführt. Ein Funktionsabschluss repräsentiert im Prinzip eine anonyme Funktion. Es bietet sich an, anonyme Funktionen der Programmierer*in auch explizit zur Verfügung zu stellen. Wir werden im Laufe der Vorlesung sehen, dass anonyme Funktionen sehr bequem sind — sie befreien die Programmierer*in von der Last, sich für jede Funktion einen Namen überlegen zu müssen.

Im gleichen Atemzug verallgemeinern wir die Syntax für die Funktionsanwendung. Da Funktionen normale Werte sind, können sie auch durch einen Ausdruck berechnet werden: aus $f\ (e_1)$ wird $e_2\ (e_1)$, an Stelle des Funktionsbezeichners f tritt ein vollwertiger Ausdruck e_2.

Abstrakte Syntax Wir erweitern die abstrakte Syntax um die folgenden Konstrukte:

$e ::= \cdots$	*Funktionsausdrücke:*
$\mid$ ***fun*** $(x : t) \to e$	Funktionsabstraktion
$\mid$ $e\ e_1$	Funktionsapplikation

Der Ausdruck ***fun*** $(x : t) \to e$ heißt anonyme Funktion (oder λ-Ausdruck), da er eine Funktion bezeichnet, diese aber keinen Namen hat. Der formale Parameter der Funktion ist x, das Symbol '$\to$' trennt den formalen Parameter vom Funktionsrumpf e, der durch einen Ausdruck gegeben ist. Beachte, dass wir die Funktionsanwendung nunmehr ohne Klammern schreiben. (Klammern sind ein Hilfsmittel der konkreten Syntax.)

Statische Semantik Die Funktionsabstraktion führt ein Element vom Typ $t_1 \to t_2$ ein; die Funktionsapplikation eliminiert ein Element dieses Typs.

$$\frac{\Sigma, \{ x_1 \mapsto t_1 \} \vdash e : t_2}{\Sigma \vdash (\textbf{\textit{fun}}\ (x_1 : t_1) \to e) : t_1 \to t_2} \qquad \frac{\Sigma \vdash e : t_1 \to t_2 \qquad \Sigma \vdash e_1 : t_1}{\Sigma \vdash e\ e_1 : t_2} \tag{3.1}$$

Wenn der Typ des formalen Parameters x_1 aus dem Kontext abgeleitet werden kann — etwa weil x_1 als Argument einer arithmetischen Operation verwendet wird — lassen wir die Typangabe oft weg und schreiben kurz ***fun*** $x \to e$.

Dynamische Semantik Eine anonyme Funktion wertet zu einem Funktionsabschluss aus.

$$\frac{}{\delta \vdash (\textbf{\textit{fun}}\ x \to e) \Downarrow \langle \delta, x, e \rangle}$$

Die Auswertung der Funktionsapplikation ist jetzt etwas umfangreicher, da zusätzlich die Funktion e ausgerechnet werden muss — vorher stand an dieser Stelle bereits ein Wert.

$$\frac{\delta \vdash e \Downarrow \langle \delta', x_1, e' \rangle \qquad \delta \vdash e_1 \Downarrow \nu_1 \qquad \delta', \{ x_1 \mapsto \nu_1 \} \vdash e' \Downarrow \nu'}{\delta \vdash e\ e_1 \Downarrow \nu'} \tag{3.2}$$

Die Auswertung vollzieht sich in drei Schritten:

(1) Die Funktion e wird ausgerechnet; das Ergebnis ist ein Funktionsabschluss.

(2) Das Argument e_1 wird ausgerechnet; das Ergebnis ist ein Wert.

(3) Der Funktionsrumpf e' wird in der Umgebung $\delta', \{ x_1 \mapsto \nu_1 \}$ ausgerechnet. Die in dem Funktionsabschluss abgelegte Umgebung wird um die Parameterbindung erweitert: Dem formalen Parameter x_1 wird der *Wert* ν_1 des aktuellen Parameters zugeordnet. Das Ergebnis dieser Rechnung ist der Wert des Funktionsaufrufs $e\ e_1$.

Wir sehen, die Anwendung einer Funktion auf ein Argument ist einigermaßen involviert. Zwei Punkte verdienen besondere Beachtung.

Zunächst ist es wichtig festzuhalten, dass der Parameter einer Funktion immer ausgerechnet wird. Da an die Funktion der *Wert* des Parameters übergeben wird, spricht man im Englischen auch von „*call by value*“. Welche alternativen Parameterübergabemechanismen sind denkbar? Wir könnten alternativ den aktuellen Parameter unausgewertet, das heißt den Ausdruck selbst, an den formalen Parameter binden. Warum wäre das eine sinnvolle Alternative? Nun, die Funktion benötigt das Argument vielleicht nicht immer. Wir würden das Argument erst ausrechnen, wenn es im Rumpf der Funktion tatsächlich benötigt wird. Man spricht auch von bedarfsgetriebener Auswertung oder etwas plakativer von „fauler“ Auswertung (engl. call by need oder lazy evaluation). Wir haben im letzten Abschnitt die Unterschiede zwischen e_1 `&&` e_2 und *and-also* (e_1, e_2) diskutiert. Schauen wir uns ein konkretes Beispiel an:

$$(b > 0)\ \texttt{\&\&}\ (a \div b \geqslant 10) \quad \text{versus} \quad \textit{and-also}\ (b > 0, a \div b \geqslant 10)$$

In der Umgebung $\{a \mapsto 99, b \mapsto 0\}$ wertet der linke Ausdruck zu *false* aus; der rechte Ausdruck hingegen ist undefiniert: Da Parameter call-by-value übergeben werden, wird der Wert sowohl von $b > 0$ als auch von $a \div b \geqslant 10$ benötigt — die dynamische Semantik ordnet letzterem aber keinen Wert zu. Mit anderen Worten: Im Falle der Funktion *and-also* würden wir von einer „faulen“ Auswertung profitieren.

Der zweite Punkt der Beachtung verdient, hat mit Umgebungen zu tun. Die Auswertungsregel (3.2) involviert zwei verschiedene Umgebungen: δ, die aktuelle Umgebung, bezüglich der die *Anwendung* der Funktion ausgerechnet wird, und δ', die Umgebung, die bei der *Definition* der Funktion aktuell war und die im Funktionsabschluss abgelegt wurde. Der Funktionsrumpf wird bezüglich der erweiterten Umgebung $\delta', \{x_1 \mapsto \nu_1\}$ abgearbeitet, *nicht* $\delta, \{x_1 \mapsto \nu_1\}$. Der Unterschied ist bedeutsam, wenn der Rumpf der Funktion freie Bezeichner enthält. Greifen wir noch einmal unser obiges Beispiel auf.

let $d = 2$
let $\mathit{next}\ (n) = n + d$
$\ldots$
$\ldots$ ***let*** $d = 4711$ ***in*** $\mathit{next}\ (1) \ldots$
$\ldots$ ***let*** $d = 0815$ ***in*** $\mathit{next}\ (1) \ldots$

Der Bezeichner d wird insgesamt dreimal definiert: einmal vor der Definition von *next* und zweimal vor der Anwendung von *next*. Welcher Wert von d ist im Rumpf gemeint? Darauf gibt die Semantik eine präzise Antwort. Es ist der Wert, der *statisch* bei der Definition von *next* an d gebunden ist, also 2. Würden wir in der Auswertungsregel die dritte Voraussetzung durch $\delta, \{x_1 \mapsto \nu_1\} \vdash e' \Downarrow \nu'$ ersetzen (in diesem Fall müssen wir in Funktionsabschlüssen die Bindung nicht mehr mitführen), dann wäre d jeweils der Wert, der *dynamisch* beim jeweiligen Aufruf an d gebunden ist, also 4711 bzw. 0815. Man sieht, ein Apostroph macht einen großen Unterschied. Im ersten Fall spricht man übrigens von statischer Bindung (engl. static binding), im zweiten Fall von dynamischer Bindung (engl. dynamic binding). Mini–F# verwendet statische Bindung und das aus

einem guten Grund: Programme sind einfacher zu lesen und zu verstehen. Wird zum Beispiel eine Funktion mit den gleichen Argumenten aufgerufen, so erhält man den gleichen Wert — mit dynamischer Bindung gilt das nicht, siehe obiges Beispiel. Allgemein sind statische Eigenschaften ungleich einfacher zu verstehen, zu vermitteln und zu verifizieren als dynamische Eigenschaften. Für erstere genügt ein Blick in den Programmtext; für letztere muss man alle möglichen Rechnungen betrachten, alle Verläufe, die ein Programm nehmen kann — das sind in der Regel unendlich viele.

Vertiefung Mit der Einführung von Funktionsabstraktionen benötigen wir Funktionsdefinitionen streng genommen nicht mehr. Die Funktionsdeklaration

$$\textbf{\textit{let}}\ not\ (a : Bool) : Bool = \textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ false\ \textbf{\textit{else}}\ true$$

entspricht exakt der Wertedeklaration

$$\textbf{\textit{let}}\ not = \textbf{\textit{fun}}\ (a : Bool) \to \textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ false\ \textbf{\textit{else}}\ true$$

Funktionen sind Werte!

Umgekehrt entspricht die Funktionsabstraktion

$$\textbf{\textit{fun}}\ (x_1 : t_1) \to e$$

exakt dem ***in***-Ausdruck

$$\textbf{\textit{let}}\ f\ (x_1 : t_1) : t_2 = e\ \textbf{\textit{in}}\ f$$

Beide Konstrukte sind also austauschbar. Für welches Konstrukt man sich bei der Programmierung entscheidet, ist daher eine Frage des guten Geschmacks, über den sich ja bekanntlich nicht streiten lässt.

Funktionsabstraktionen können auch geschachtelt oder *gestaffelt* werden.

$$\textbf{\textit{let}}\ add = \textbf{\textit{fun}}\ (m : Nat) \to \textbf{\textit{fun}}\ (n : Nat) \to m + n$$

Die Funktion *add* hat den merkwürdigen Typ $Nat \to (Nat \to Nat)$. Sie bildet eine natürliche Zahl vom Typ Nat auf eine Funktion vom Typ $Nat \to Nat$ ab. Wenden wir *add* an, müssen entsprechend die Funktionsaufrufe gestaffelt werden: (*add* 815) 4711. Der Teilausdruck *add* 815 berechnet eine Funktion, die dann auf 4711 angewendet wird. In der konkreten Syntax erlauben wir, die Klammern auszulassen und (*add* 815) 4711 bzw. allgemein $(e\ e_1)\ e_2$ kurz als *add* 815 4711 bzw. $e\ e_1\ e_2$ aufzuschreiben. Naiv lässt sich eine solche Abfolge lesen als Anwendung der Funktion e auf die zwei Parameter e_1 und e_2. In Wirklichkeit handelt es sich, wie gesagt, um eine gestaffelte Funktionsanwendung: e wertet zu einer Funktion aus, diese wird auf e_1 angewendet, die Anwendung resultiert wiederum in einer Funktion, die ihrerseits auf e_2 angewendet wird.

Noch eine Bemerkung zur konkreten Syntax: so wie wir $(e\ e_1)\ e_2$ zu $e\ e_1\ e_2$ abkürzen, so erlauben wir auch den Typ $t_1 \to (t_2 \to t_3)$ kurz als $t_1 \to t_2 \to t_3$ aufzuschreiben. Die Funktionsanwendung ist übrigens *nicht* assoziativ: $e\ (e_1\ e_2)$ ist völlig verschieden

von $(e\ e_1)\ e_2$; ebenso ist $(t_1 \to t_2) \to t_3$ ein völlig anderer Typ als $t_1 \to (t_2 \to t_3)$, siehe auch Aufgabe 3.4.

Eine gestaffelte Funktion vom Typ $Nat \to (Nat \to Nat)$ ist im Vergleich zu einer mehrparametrigen Funktion vom Typ $Nat * Nat \to Nat$ flexibler (den „Paartyp“ $t_1 * t_2$ führen wir im nächsten Kapitel ein). Sie kann zunächst nur mit einem Parameter versorgt werden: *add* 815 ist ein gültiger Ausdruck, der überall dort verwendet werden kann, wo eine Funktion des Typs $Nat \to Nat$ verlangt wird. Im Gegensatz dazu muss eine mehrparametrige Funktion stets mit allen Parametern aufgerufen werden.

Eine Funktion, die eine Funktion als Argument erwartet oder — wie *add* — als Ergebnis zurückgibt, heißt *Funktion höherer Ordnung*. Im Laufe der Vorlesung werden uns einige dieser Exemplare über den Weg laufen.

3.6. Rekursive Funktionen

To iterate is human;
to recurse, divine.
— L. Peter Deutsch

Die folgende Erweiterung motivieren wir mit einer Knobelaufgabe: Wieviele Möglichkeiten gibt es, n verschiedene Objekte in einer Reihe zu arrangieren? Nun, für die erste Position können wir zwischen n Objekten auswählen, die zweite Position lässt sich mit $n-1$ Objekten besetzen usw. Für die letzte Position bleibt schließlich nur ein einziges Objekt übrig. Da die jeweiligen Entscheidungen für ein bestimmtes Objekt unabhängig voneinander sind, ergibt sich als Gesamtzahl aller Arrangements das Produkt der Zahlen von 1 bis n: als Formel $n! = 1 * 2 * \cdots * (n-1) * n$. Diese Zahl heißt auch n *Fakultät*.

module Values. Factorial

Im Folgenden wollen wir überlegen, wie wir die Fakultätsfunktion in unserer Sprache programmieren können.

let *factorial* $(n : Nat) : Nat =$

Wenn wir wissen, welchen konkreten Wert der formale Parameter n hat, können wir das Ergebnis jeweils einfach angeben:

if $n = 0$ ***then*** 1
else ***if*** $n = 1$ ***then*** 1
else if $n = 2$ ***then*** $1 * 2$
else if $n = 3$ ***then*** $1 * 2 * 3$
else $\ldots$

Der Fall für $n = 0$ bedarf noch einer kurzen Klärung. Der Wert von 0 Fakultät ist 1, da das leere Produkt von Zahlen vereinbarungsgemäß 1 ist. Und in der Tat: 0 Objekte können auf genau eine Art und Weise angeordnet werden, als leeres Arrangement.

Die Ellipse $(\ldots)$ zeigt an, dass wir noch kein vollständiges Programm vor uns haben. Trotzdem lässt sich bereits ein Muster ausmachen: In jedem der Fälle $n > 0$ ist der letzte Faktor die Zahl n selbst. (Klar, oder?)

$$
\begin{array}{l}
\textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\
\textbf{\textit{else}}\quad \textbf{\textit{if}}\ n = 1\ \textbf{\textit{then}}\ n \\
\quad \textbf{\textit{else if}}\ n = 2\ \textbf{\textit{then}}\ 1 * n \\
\quad \textbf{\textit{else if}}\ n = 3\ \textbf{\textit{then}}\ 1 * 2 * n \\
\quad \textbf{\textit{else}}\ \ldots
\end{array}
$$

Mit Hilfe der Eigenschaft $\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2 * e\ \textbf{\textit{else}}\ e_3 * e = (\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3) * e$ können wir das Programm bzw. den Funktionsrumpf umschreiben. Der Faktor n wird sozusagen aus den Zweigen der Alternativen „herausgezogen“.

$$
\begin{array}{l}
\textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\
\textbf{\textit{else}}\ (\quad \textbf{\textit{if}}\ n = 1\ \textbf{\textit{then}}\ 1 \\
\quad \textbf{\textit{else if}}\ n = 2\ \textbf{\textit{then}}\ 1 \\
\quad \textbf{\textit{else if}}\ n = 3\ \textbf{\textit{then}}\ 1 * 2 \\
\quad \textbf{\textit{else}}\ \ldots) * n
\end{array}
$$

Der Ausdruck in Klammern sieht dem ursprünglichen Funktionsrumpf sehr ähnlich, nur dass die Bedingungen $n = 1$, $n = 2$ usw. lauten statt $n = 0$, $n = 1$ usw. Wir können die Konstanten in Übereinstimmung bringen, indem wir links und rechts jeweils 1 subtrahieren. (Dabei ist Vorsicht geboten, da $0 \dot{-} 1 = 0$.)

$$
\begin{array}{l}
\textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\
\textbf{\textit{else}}\ (\quad \textbf{\textit{if}}\ n \dot{-} 1 = 0\ \textbf{\textit{then}}\ 1 \\
\quad \textbf{\textit{else if}}\ n \dot{-} 1 = 1\ \textbf{\textit{then}}\ 1 \\
\quad \textbf{\textit{else if}}\ n \dot{-} 1 = 2\ \textbf{\textit{then}}\ 1 * 2 \\
\quad \textbf{\textit{else}}\ \ldots) * n
\end{array}
$$

Somit ist der Ausdruck in Klammern gleich dem Aufruf $\mathit{factorial}\ (n \dot{-} 1)$. Erlauben wir bei der Definition einer Funktion den Rückgriff auf die definierte Funktion selbst (!), so erhalten wir das folgende, kompakte Programm.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{factorial}\ (n : \mathit{Nat}) : \mathit{Nat} = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \mathit{factorial}\ (n \dot{-} 1) * n
\end{array}
$$

In Worten ausgedrückt: Die Fakultät von 0 ist 1; ist $n > 0$, dann ist n Fakultät gleich dem Produkt von $n \dot{-} 1$ Fakultät und n. Greift man — wie hier — bei der Definition auf das definierte Objekt selbst zurück, spricht man von einer *rekursiven* Definition. Rekursive Definitionen werden mit dem Schlüsselwort ***rec*** gekennzeichnet. (Warum?)

Abbildung 3.3 illustriert die Abarbeitung des Funktionsaufrufs *factorial* (3). Um uns nicht in Details zu verlieren, haben wir die Auswertung als Folge von Gleichungen notiert: $\mathit{factorial}\ (3) = \cdots = 6$. Innerhalb der geschweiften Klammern geben wir jeweils an, warum eine Umformung gültig ist: Die unkommentierte Gleichung $e_1 = e_2$ wird zu

$$
\begin{array}{ll}
 & e_1 \\
= & \quad \{\ \text{Rechtfertigung}\ \} \\
 & e_2
\end{array}
$$

$$
\begin{aligned}
& \mathit{factorial}\ (3) \\
= \quad & \{ \text{ Definition von } \mathit{factorial} \ \} \\
& \textbf{\textit{if}}\ 3 = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \mathit{factorial}\ (3 \dot{-} 1) * 3 \\
= \quad & \{ \text{ Auswertung der Alternative } \} \\
& \mathit{factorial}\ (3 \dot{-} 1) * 3 \\
= \quad & \{ \text{ Definition der natürlichen Subtraktion } \} \\
& \mathit{factorial}\ 2 * 3 \\
= \quad & \{ \text{ Definition von } \mathit{factorial} \ \} \\
& (\textbf{\textit{if}}\ 2 = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \mathit{factorial}\ (2 \dot{-} 1) * 2) * 3 \\
= \quad & \{ \text{ Auswertung der Alternative } \} \\
& (\mathit{factorial}\ (2 \dot{-} 1) * 2) * 3 \\
= \quad & \{ \text{ Definition der natürlichen Subtraktion } \} \\
& (\mathit{factorial}\ 1 * 2) * 3 \\
= \quad & \{ \text{ Definition von } \mathit{factorial} \ \} \\
& ((\textbf{\textit{if}}\ 1 = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \mathit{factorial}\ (1 \dot{-} 1) * 1) * 2) * 3 \\
= \quad & \{ \text{ Auswertung der Alternative } \} \\
& ((\mathit{factorial}\ (1 \dot{-} 1) * 1) * 2) * 3 \\
= \quad & \{ \text{ Definition der natürlichen Subtraktion } \} \\
& ((\mathit{factorial}\ 0 * 1) * 2) * 3 \\
= \quad & \{ \text{ Definition von } \mathit{factorial} \ \} \\
& (((\textbf{\textit{if}}\ 0 = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \mathit{factorial}\ (0 \dot{-} 1) * 0) * 1) * 2) * 3 \\
= \quad & \{ \text{ Auswertung der Alternative } \} \\
& ((1 * 1) * 2) * 3 \\
= \quad & \{ \text{ Definition der Multiplikation } \} \\
& (1 * 2) * 3 \\
= \quad & \{ \text{ Definition der Multiplikation } \} \\
& 2 * 3 \\
= \quad & \{ \text{ Definition der Multiplikation } \} \\
& 6
\end{aligned}
$$

Abbildung 3.3.: Auswertung des Funktionsaufrufs *factorial* (3).

wobei wir die Rechtfertigung für die Umformung mitliefern. Auf diese Weise wird jeder einzelne Rechenschritt hoffentlich leicht nachvollziehbar. Da *factorial* eine Funktion im mathematischen Sinne ist, können wir einen Funktionsaufruf ausrechnen, indem wir wiederholt die linke Seite der Definition durch die rechte Seite ersetzen und dabei für die formalen Parameter die aktuellen Parameter einsetzen. (Wenn eine Mini–F# Funktion keine mathematische Funktion ist, dann ist das nicht möglich — in Kapitel 7 verlassen wir den Pfad der Tugend.)

Zwei Phasen lassen sich bei der Abarbeitung von *factorial* ausmachen: Beim *rekursiven Abstieg* wird Schritt für Schritt ein Turm von Multiplikationen aufgebaut; beim *rekursiven Aufstieg* wird dieser Turm Schritt für Schritt wieder abgebaut. (Bei der Herleitung der Funktion haben wir im letzten Schritt die Definition „eingeklappt“. Bei der Abarbeitung der Funktion wird die Definition sozusagen wieder „ausgeklappt“, gegebenenfalls mehrfach.) Bei der Darstellung in Abbildung 3.3 vereinfachen bzw. mogeln wir bewusst: Die formalen Parameter werden natürlich nicht durch die aktuellen Parameter *textuell* ersetzt, sondern die jeweiligen Bindungen werden mit Hilfe von Umgebungen protokolliert. Dazu in Kürze mehr.

Abstrakte Syntax Deklarationen werden um rekursive Funktionsdefinitionen erweitert.

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***let rec*** $f\ (x_1 : t_1) : t_2 = e$	rekursive Funktionsdefinition

Die Gleichung ***let rec*** $f\ (x) = e$ definiert genau wie ***let*** $f\ (x) = e$ eine Funktion, mit dem Unterschied, dass in e zusätzlich der Bezeichner f selbst sichtbar ist.

Statische Semantik Um prüfen zu können, ob eine rekursiv definierte Funktion wohlgetypt ist, müssen wir den Typ der Funktion für die Prüfung des Funktionsrumpfes bereits kennen. Aus diesem Grund muss der Argument- *und* der Ergebnistyp angegeben werden — bei nicht-rekursiven Funktionen genügt tatsächlich der Argumenttyp.

$$\frac{\Sigma, \{f \mapsto t_1 \to t_2, x_1 \mapsto t_1\} \vdash e_2 : t_2}{\Sigma \vdash (\textbf{\textit{let rec}}\ f\ (x_1 : t_1) : t_2 = e_2) : \{f \mapsto t_1 \to t_2\}}$$

Dynamische Semantik Eine rekursive Funktionsdefinition wertet ähnlich wie eine nichtrekursive im Wesentlichen zu sich selbst aus. Die Deklaration der Fakultätsfunktion ergibt zum Beispiel die Umgebung

$$\{\mathit{factorial} \mapsto \langle \varnothing, \mathit{factorial}, n, \ \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \mathit{factorial}\ (n \dot{-} 1) * n \rangle\}$$

An die Stelle eines Funktionsabschlusses ist ein *rekursiver Funktionsabschluss* getreten, in dem *zusätzlich* der Name der rekursiven Funktion aufgeführt wird. Bei der Anwendung einer rekursiven Funktion auf ein Argument müssen wir den formalen Parameter an den Wert des aktuellen Parameters binden *und* zusätzlich den Funktionsbezeichner an den rekursiven Funktionsabschluss.

Wir erweitern den Bereich der Werte um rekursive Funktionsabschlüsse.

$\nu ::= \cdots$	*Werte:*
$\mid \langle \delta, f, x, e \rangle$	rekursiver Funktionsabschluss

Eine rekursive Funktionsdefinition wertet zu einer Bindung aus, in der der Funktionsname an einen rekursiven Funktionsabschluss gebunden ist.

$$\frac{}{\delta \vdash (\textbf{\textit{let rec}}\ f\ x = e) \Downarrow \{f \mapsto \langle \delta, f, x, e \rangle\}}$$

Schließlich benötigen wir eine weitere Regel für die Funktionsapplikation, die sich um rekursive Funktionen kümmert.

$$\frac{}{\delta \vdash (\textbf{\textit{let rec}}\ f\ x = e) \Downarrow \{f \mapsto \langle \delta, f, x, e \rangle\}}$$

$$\frac{\delta \vdash e \Downarrow \nu \qquad \delta \vdash e_1 \Downarrow \nu_1 \qquad \delta', \{f \mapsto \nu, x_1 \mapsto \nu_1\} \vdash e' \Downarrow \nu'}{\delta \vdash e\ e_1 \Downarrow \nu'} \quad \text{mit } \nu = \langle \delta', f, x_1, e' \rangle$$

Wie auch bei nicht-rekursiven Funktionsabschlüssen vollzieht sich die Auswertung in drei Schritten:

(1) Die Funktion e wird ausgerechnet; die obige Regel kommt zur Anwendung, wenn das Ergebnis ein rekursiver Funktionsabschluss ist.

(2) Das Argument e_1 wird ausgerechnet; das Ergebnis ist ein Wert.

(3) Der Funktionsrumpf e' wird in der Umgebung $\delta', \{f \mapsto \nu, x_1 \mapsto \nu_1\}$ ausgerechnet. Der Bezeichner f wird an den rekursiven Funktionsabschluss gebunden, in dem f selbst aufgeführt wird. (Aus der Rekursion wird ein zyklisches Geflecht.) Zusätzlich wird dem formalen Parameter x_1 der *Wert* ν_1 des aktuellen Parameters zugeordnet. Das Ergebnis dieser Rechnung ist der Wert des Funktionsaufrufs $e\ e_1$.

Schauen wir uns die Abarbeitung des Aufrufs *factorial* (3) noch einmal und jetzt im Detail an — konstruieren wir einen Beweisbaum. Damit die Formeln nicht zu groß geraten, kürzen wir *factorial* mit *fac* ab. Weiterhin vereinbaren wir Abkürzungen für den rekursiven Funktionsabschluss und für die beteiligten Umgebungen.

$$\gamma := \langle \varnothing, fac, n, \ \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ fac\ (n \dot{-} 1) * n \rangle$$
$$\delta := \{fac \mapsto \gamma\}$$
$$\delta_i := \varnothing, \{fac \mapsto \gamma, n \mapsto i\} = \{fac \mapsto \gamma, n \mapsto i\}$$

Wir haben bereits in Abbildung 3.3 gesehen, dass der initiale Aufruf *fac* (3) drei weitere Aufrufe nach sich zieht. Entsprechend mächtig gerät der Beweisbaum — tatsächlich der größte Beweisbaum, der uns in diesem Skript unterkommen wird. Wie für jeden Auswertungsbaum gilt: Die Form der Teilbäume wird jeweils durch die Form der Teilausdrücke bestimmt. Für die meisten Ausdrücke gibt es genau eine Auswertungsregel — damit ist klar, welche Regel zur Anwendung kommt. Ausnahmen von dieser Faustregel sind die Alternative und die Vergleichsoperationen. Für sie gibt es jeweils zwei Auswertungsregeln;

entsprechend unterschiedlich können die Beweisbäume aussehen. Beim initialen Aufruf kommen die Auswertungsregeln für rekursive Funktionsdefinitionen und für Aufrufe rekursiv definierter Funktionen zum Tragen.

$$\dfrac{\dfrac{}{\varnothing \vdash \textbf{\textit{let rec}}\ fac\ (n) = \cdots \Downarrow \delta} \qquad \dfrac{\dfrac{}{\delta \vdash fac \Downarrow \gamma} \quad \dfrac{}{\delta \vdash 3 \Downarrow 3} \quad \dfrac{\vdots}{\delta_3 \vdash \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ fac\ (n \dot{-} 1) * n}}{\delta \vdash fac\ (3) \Downarrow 6}}{\varnothing \vdash \textbf{\textit{let rec}}\ fac\ (n) = \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ fac\ (n \dot{-} 1) * n\ \textbf{\textit{in}}\ fac\ (3) \Downarrow 6}$$

Die Auswertung des Funktionsaufrufs umfasst wie bereits angesprochen drei Schritte: zunächst wird der Bezeichner fac in der Umgebung δ nachgeschlagen; dann wird der aktuelle Parameter ausgerechnet, und schließlich wird der Funktionsrumpf in der Umgebung δ_3 ausgewertet. Diese Umgebung ergibt sich als Erweiterung der im Funktionsabschluss abgelegten Umgebung — diese ist leer — um Bindungen für den Namen der rekursiv definierten Funktion und deren Parameter. Entwickeln wir den Beweisbaum für den Funktionsrumpf weiter — im obigen Baum durch das Auslassungszeichen markiert.

$$\dfrac{\dfrac{}{\delta_3 \vdash n = 0 \Downarrow \mathit{false}} \qquad \dfrac{\dfrac{\dfrac{}{\delta_3 \vdash fac \Downarrow \gamma} \quad \dfrac{}{\delta_3 \vdash n \dot{-} 1 \Downarrow 2} \quad \dfrac{\vdots}{\delta_2 \vdash \textbf{\textit{if}}\ n = 0 \cdots \Downarrow 2}}{\delta_3 \vdash fac\ (n \dot{-} 1) \Downarrow 2} \quad \dfrac{}{\delta_3 \vdash n \Downarrow 3}}{\delta_3 \vdash fac\ (n \dot{-} 1) * n \Downarrow 6}}{\delta_3 \vdash \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ fac\ (n \dot{-} 1) * n \Downarrow 6}$$

Die Auswertung zieht einen weiteren Aufruf der Fakultätsfunktion nach sich. Die Abarbeitungsschritte für den Rumpf sind jeweils gleich, nur dass die Umgebung eine andere ist: Aus δ_3 wird δ_2. Sie protokolliert den Wert des Parameters n beim rekursiven Abstieg.

$$\dfrac{\dfrac{}{\delta_2 \vdash n = 0 \Downarrow \mathit{false}} \qquad \dfrac{\dfrac{\dfrac{}{\delta_2 \vdash fac \Downarrow \gamma} \quad \dfrac{}{\delta_2 \vdash n \dot{-} 1 \Downarrow 1} \quad \dfrac{\vdots}{\delta_1 \vdash \textbf{\textit{if}}\ n = 0 \cdots \Downarrow 1}}{\delta_2 \vdash fac\ (n \dot{-} 1) \Downarrow 1} \quad \dfrac{}{\delta_2 \vdash n \Downarrow 2}}{\delta_2 \vdash fac\ (n \dot{-} 1) * n \Downarrow 2}}{\delta_2 \vdash \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ fac\ (n \dot{-} 1) * n \Downarrow 2}$$

Es erfolgt ein weiterer rekursiver Aufruf — der resultierende Beweisbaum ist vom Aufbau wiederum identisch; nur die Umgebungen und berechneten Werte sind unterschiedlich.

$$\dfrac{\dfrac{}{\delta_1 \vdash n = 0 \Downarrow \mathit{false}} \qquad \dfrac{\dfrac{\dfrac{}{\delta_1 \vdash fac \Downarrow \gamma} \quad \dfrac{}{\delta_1 \vdash n \dot{-} 1 \Downarrow 0} \quad \dfrac{\vdots}{\delta_0 \vdash \textbf{\textit{if}}\ n = 0 \cdots \Downarrow 1}}{\delta_1 \vdash fac\ (n \dot{-} 1) \Downarrow 1} \quad \dfrac{}{\delta_1 \vdash n \Downarrow 1}}{\delta_1 \vdash fac\ (n \dot{-} 1) * n \Downarrow 1}}{\delta_1 \vdash \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ fac\ (n \dot{-} 1) * n \Downarrow 1}$$

Vergleicht man die drei obigen Beweisbäume, sieht man, dass zum Beispiel der Teilausdruck n, das zweite Argument der Multiplikation, insgesamt dreimal ausgewertet wird,

aber jeweils in unterschiedlichen Umgebungen: $\delta_3 \vdash n \Downarrow 3$, $\delta_2 \vdash n \Downarrow 2$ und $\delta_1 \vdash n \Downarrow 1$. Der letzte Aufruf mit $n = 0$ führt schließlich unmittelbar zum Ziel: Die Alternative wird zu 1 ausgewertet.

$$\frac{\overline{\delta_0 \vdash n = 0 \Downarrow \mathit{true}} \qquad \overline{\delta_0 \vdash 1 \Downarrow 1}}{\delta_0 \vdash \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \mathit{fac}\ (n \dot{-} 1) * n \Downarrow 1}$$

Eigentlich müsste man jetzt alle fünf Beweisbäume zusammenstecken; davon wollen wir aus Platzgründen absehen.

Die Fakultät wächst übrigens sehr schnell, wie die folgenden Beispielaufrufe zeigen.

```
Mini⟩ factorial 10
3.628.800
Mini⟩ factorial 100
93.326.215.443.944.152.681.699.238.856.266.700.490.715.968.264.381.621.468.592.
963.895.217.599.993.229.915.608.941.463.976.156.518.286.253.697.920.827.223.758.
251.185.210.916.864.000.000.000.000.000.000.000.000
```

Die Zahl der Atome im sichtbaren Weltall wird auf ungefähr 10^{79} geschätzt; *factorial* 100 mit seinen 158 Stellen übersteigt diese Zahl um ein Vielfaches.

3.7. Entwurfsmuster

Few things are harder to put up with than the annoyance of a good example.
— Mark Twain (1835–1910), *Pudd'nhead Wilson (1894) ch. 19*

Der Schritt von den nicht-rekursiven zu den rekursiven Funktionen ist ein gewaltiger. Nunmehr ist unsere Sprache *berechnungsuniversell*. Mit ihr können wir die prinzipiellen Möglichkeiten eines Rechners ausnutzen. Mehr dazu später in der Theoretischen Abteilung der Informatik. Wir wenden uns an dieser Stelle den vergnüglichen Dingen zu, der Programmierung.

module Values. Peano

Peano Entwurfsmuster Bei der Auflistung der vordefinierten arithmetischen Operatoren in Abschnitt 3.2 fehlt unter anderem die Potenzfunktion. Vervollständigen wir unser Repertoire an arithmetischen Funktionen, indem wir ein Programm dafür schreiben. Bevor wir die Aufgabe angehen, ist es hilfreich, sich noch einmal die Definition der Fakultät ins Gedächtnis zu rufen.

```
let rec factorial (n : Nat) : Nat =
  if n = 0 then 1
           else factorial (n ∸ 1) * n
```

Die Definition macht Gebrauch von der Tatsache, dass eine natürliche Zahl entweder 0 oder größer als 0 ist. Im ersten Fall können und müssen wir unmittelbar die Lösung

angeben (*Rekursionsbasis* oder Rekursionsverankerung). Im zweiten Fall bestimmen wir rekursiv eine Lösung für $n \dot{-} 1$ und erweitern dann die Teillösung zu einer Gesamtlösung für n (*Rekursionsschritt*). Wenden wir dieses Schema auf die Potenzfunktion x^n an.

$$\begin{array}{l} \textbf{\textit{let rec}}\ \mathit{power}\ (x : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} = \\ \quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}} \ldots \\ \qquad\qquad \textbf{\textit{else}}\ \ldots \mathit{power}\ (x, n \dot{-} 1) \ldots \end{array}$$

(Beachte, dass wir über n rekurrieren, nicht über x. Warum?) Die Rekursionsbasis ist einfach: x^0 ist 1. Der Rekursionsschritt ist nicht viel schwieriger: Der rekursive Aufruf ermittelt x^{n-1}; wir erweitern die Teillösung zur Gesamtlösung x^n, indem wir $x^{n \dot{-} 1}$ mit x multiplizieren. Insgesamt erhalten wir das folgende Programm.

$$\begin{array}{l} \textbf{\textit{let rec}}\ \mathit{power}\ (x : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} = \\ \quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\ \qquad\qquad \textbf{\textit{else}}\ \ \mathit{power}\ (x, n \dot{-} 1) * x \end{array}$$

Die Potenzfunktion wird auf wiederholte Multiplikation zurückgeführt. Auf die gleiche Art und Weise können wir auch die Multiplikation auf die Addition zurückführen

$$\begin{array}{l} \textbf{\textit{let rec}}\ \mathit{mul}\ (m : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} = \\ \quad \textbf{\textit{if}}\ m = 0\ \textbf{\textit{then}}\ 0 \\ \qquad\qquad \textbf{\textit{else}}\ \ \mathit{mul}\ (m \dot{-} 1, n) + n \end{array}$$

und die Addition auf die Nachfolgerfunktion.

$$\begin{array}{l} \textbf{\textit{let rec}}\ \mathit{add}\ (m : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} = \\ \quad \textbf{\textit{if}}\ m = 0\ \textbf{\textit{then}}\ n \\ \qquad\qquad \textbf{\textit{else}}\ \ \mathit{add}\ (m \dot{-} 1, n) + 1 \end{array}$$

Die Beispielprogramme zeigen, dass wir theoretisch mit einigen wenigen vordefinierten Funktionen auskommen: der Zahl 0, dem Test „gleich 0“, der Nachfolger- und der Vorgängerfunktion. Praktisch gesehen ist ‘+’ allerdings vorteilhafter als *add*, da schneller: $4711 + 815$ benötigt einen Auswertungsschritt, $\mathit{add}\ (4711, 815)$ hingegen mehrere zehntausend Schritte. (Die Anzahl der Rechenschritte entspricht der Größe des Beweisbaums für $\delta \vdash e \Downarrow n$ — aus wievielen Regeln bzw. Regelinstanzen setzt sich der Baum zusammen?)

Nichtsdestotrotz war die Exkursion sehr lehrreich, zeigt sie uns doch ein allgemeines *Entwurfsmuster* (engl. design pattern) auf, um Funktionen über den natürlichen Zahlen zu programmieren: Haben wir die Aufgabe eine Funktion $f : \mathit{Nat} \to t$ zu erstellen, dann sieht ein erster Entwurf folgendermaßen aus.

$\textbf{\textit{let rec}}\ f\ (n : \mathit{Nat}) : t =$	*Peano Entwurfsmuster:*
$\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}} \ldots$	*Rekursionsbasis*
$\qquad\qquad \textbf{\textit{else}}\ \ldots f\ (n \dot{-} 1) \ldots$	*Rekursionsschritt*

Der italienische Mathematiker und Logiker Giuseppe Peano entwickelte, an die Algebra der Logik von Boole, Jevons, Schröder und Porezki anknüpfend, die mathematische Logik weiter.

Von Peano stammt ein bekanntes und noch heute verwendetes *Axiomensystem* für die natürlichen Zahlen:

- 0 ist eine natürliche Zahl.
- Für alle n gilt, dass, wenn n eine natürliche Zahl ist, auch die auf n folgende Zahl eine natürliche Zahl ist.
- Wenn auf zwei Zahlen dieselbe Zahl folgt, sind sie identisch.
- 0 kann nicht auf eine natürliche Zahl folgen.
- Das Induktionsaxiom: Wenn 0 eine Eigenschaft hat, und wenn jede auf eine natürliche Zahl folgende Zahl die Eigenschaft besitzt, sofern die Zahl selbst die Eigenschaft hat, dann haben alle natürlichen Zahlen die betreffende Eigenschaft.

Abbildung 3.4.: Giuseppe Peano (1858–1932).

Die Ellipsen müssen wir sodann mit Leben füllen: an die Stelle des ersten Auslassungszeichens muss ein Ausdruck des Typs t treten (Rekursionsbasis); die zweite Stelle müssen wir mit einem Ausdruck füllen, der die Teillösung $f\ (n \dot{-} 1)$ vom Typ t zu einer Gesamtlösung vom Typ t erweitert (Rekursionsschritt). Der Parameter n heißt übrigens auch *Rekursionsvariable*.

Um uns später auf dieses Entwurfsmuster beziehen zu können, geben wir ihm einen Namen: *Peano Entwurfsmuster* nach dem italienischen Mathematiker Giuseppe Peano (1858–1932), der sich mit der Axiomatisierung der natürlichen Zahlen beschäftigt hat, siehe Abbildung 3.4.

Wenden wir das Peano Entwurfsmuster auf ein weiteres Beispiel an. Die Funktion *square-root* n soll die Quadratwurzel der Zahl n bestimmen. Die Wurzel geht nicht immer glatt auf, so dass wir die Aufgabe präzisieren müssen: Gesucht wird die *größte* Zahl r, so dass $r * r$ kleiner oder gleich n ist. Zum Beispiel: *square-root* 143 $\Downarrow$ 11 und *square-root* 144 $\Downarrow$ 12 usw. In eine Formel gegossen suchen wir $\lfloor\sqrt{n}\rfloor$.[3] Das Entwurfsmu-

[3] Die Gaußklammer $\lfloor x \rfloor$ (engl. floor function) bezeichnet die größte ganze Zahl, die kleiner oder gleich

ster gibt das Skelett vor:

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then ...
           else  ... square-root (n ∸ 1) ...
```

Die Wurzel von 0 ist 0, somit ist die Rekursionsbasis abgehakt. Zum Rekursionsschritt: Wenn wir die Wurzel von $n \dot{-} 1$ kennen, wie lässt sich daraus die Wurzel von n herleiten? Nun, die Quadratwurzel von n ist entweder identisch zur Quadratwurzel von $n \dot{-} 1$ oder um eins größer (Aufgabe 3.11 fragt nach einem Beweis). Um herauszufinden, welcher Fall vorliegt, testen wir einfach, ob wir mit der Erhöhung über das Ziel hinausschießen.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then 0
           else  let r = square-root (n ∸ 1)
                 in  if n < square (r + 1) then r else r + 1
```

Dem Ergebnis des rekursiven Aufrufs geben wir einen Namen (r), da dieser mehrfach benötigt wird. Die Berechnung von *square-root* $(n \dot{-} 1)$ ist aufwändig, es ist keine gute Idee, *if* $n <$ *square* (*square-root* $(n \dot{-} 1)+1$) *then square-root* $(n \dot{-} 1)$ *else square-root* $(n \dot{-} 1)+1$ zu schreiben, da so die aufwändige Rechnung doppelt durchgeführt wird.

Wenden wir uns noch einmal dem Peano Entwurfsmuster selbst zu. Interessanterweise können wir das Peano Entwurfsmuster selbst als Programm formulieren. Aus einem informellen Schema wird eine wiederverwendbare Funktion, eine sogenannte *Bibliotheksfunktion*! Die wesentliche Einsicht ist, dass die Erweiterung der Teillösung zu einer Gesamtlösung der Natur nach eine Funktion ist. Geben wir also den fehlenden Programmteilen im Schema einen Namen.

```
let rec f (n : Nat) : Nat =
  if n = 0 then zero
           else  succ (f (n ∸ 1))
```

Da wir uns auf keine konkreten Werte für die Bezeichner *zero* und *succ* festlegen wollen — wir entwickeln ja ein Lösungs*schema*, nicht eine Lösung für *ein* konkretes Problem — machen wir sie zum Parameter einer Funktion. Der Abstraktionsschritt ist der gleiche wie bei der Einführung der Funktion *area*, nur dass jetzt von zwei Werten abstrahiert wird und dass der eine Wert selbst eine Funktion ist.

```
let peano-pattern (zero : Nat, succ : Nat → Nat) : Nat → Nat =
  let rec f (n : Nat) : Nat =
        if n = 0 then zero
                 else  succ (f (n ∸ 1))
  in f
```

der reellen Zahl x ist. Formal ist sie definiert durch: $\lfloor x \rfloor$ ist eine ganze Zahl, so dass $n \leqslant \lfloor x \rfloor \iff n \leqslant x$ für alle ganzen Zahlen n und für alle reellen Zahlen x.

Die Funktion *peano-pattern* ist ungewöhnlich. Sie nimmt als Argument eine Funktion (*succ*) und gibt als Ergebnis eine Funktion (f) zurück: *peano-pattern* ist eine Funktion *höherer Ordnung*. Mit Hilfe von *peano-pattern* können wir *power* usw. sehr viel kürzer aufschreiben.

$$\begin{array}{l} \textbf{\textit{let}}\ power\ (x,\ n) = (peano\text{-}pattern\ (1, \textbf{\textit{fun}}\ s \to s * x\ ))\ n \\ \textbf{\textit{let}}\ mul\quad (m, n) = (peano\text{-}pattern\ (0, \textbf{\textit{fun}}\ s \to s + n))\ m \\ \textbf{\textit{let}}\ add\quad (m, n) = (peano\text{-}pattern\ (n, \textbf{\textit{fun}}\ s \to s + 1))\ m \end{array}$$

Der aktuelle Parameter für *succ* wird jeweils durch eine anonyme Funktion spezifiziert. Man sieht sehr schön, wie jeweils die Teillösung s (wie *s*olution) zu einer Gesamtlösung erweitert wird. Die rechten Seiten der Funktionsdefinitionen sind eine weitere Betrachtung wert; wir finden jeweils eine geschachtelte Funktionsapplikation vor: im Fall von *power* zum Beispiel $(e\ e_1)\ e_2$ mit $e = peano\text{-}pattern$, $e_1 = (1, \textbf{\textit{fun}}\ s \to s * x)$ und $e_2 = n$. Aus den Typen der Ausdrücke lässt sich ablesen, dass $e\ e_1$ in der Tat zu einer Funktion auswertet; die resultierende Funktion wird dann auf e_2 angewendet.

Zurück zur Funktion *peano-pattern*: ganz perfekt ist die Umsetzung des Entwurfsmusters noch nicht. Bei der Angabe der Parameterliste haben wir $zero : Nat$ und $succ : Nat \to Nat$ spezifiziert. Diese Festlegung ist relativ willkürlich, genausogut würde auch $zero : Bool$ und $succ : Bool \to Bool$ funktionieren bzw. allgemein $zero : t$ und $succ : t \to t$. Und in der Tat, bei der Besprechung des Entwurfsmusters haben wir uns auf keinen bestimmten Typ kapriziert. Dieses Manko können wir mit unseren bisherigen Sprachmitteln nicht beheben; wir kommen aber später in Abschnitt 4.3.2 darauf zurück. Ein weiteres Manko betrifft die Ausdruckskraft von *peano-pattern*: Nicht alle bisher betrachteten Funktionen lassen sich damit definieren. Ein Negativbeispiel ist die Fakultätsfunktion. Versuchen Sie es mal. Aufgabe 3.15 geht der Ursache auf den Grund und weist einen Ausweg.

module Values. Leibniz

Leibniz Entwurfsmuster Eine Funktion, die mit Hilfe des Peano Entwurfsmusters erstellt oder die direkt mit Hilfe von *peano-pattern* definiert wurde, benötigt für die Lösung eines Problems n rekursive Aufrufe. Das Problem für n wird auf das Problem für $n \div 1$ zurückgeführt, dieses wird auf das Problem für $n \div 2$ zurückgeführt usw. Ist diese Vorgehensweise zwingend? Keineswegs. Wir können zum Beispiel alternativ versuchen, das Problem für n auf das Problem für $n \div 2$ zurückzuführen, also in jedem Schritt n zu halbieren. Das heißt umgekehrt, dass wir aus einer Lösung für $n \div 2$ eine Lösung für n ableiten müssen. Bevor wir loslegen, sei daran erinnert, dass '$\div$' die Division auf den natürlichen Zahlen bezeichnet: $4 \div 2 \Downarrow 2$ aber $5 \div 2 \Downarrow 2$. Programmieren wir die Potenzfunktion neu.

$$\begin{array}{l} \textbf{\textit{let rec}}\ power\ (x : Nat, n : Nat) : Nat = \\ \quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}} \ldots \\ \qquad\qquad \textbf{\textit{else}}\ \ldots power\ (x, n \div 2) \ldots \end{array}$$

Können wir aus $x^{n \div 2}$ das gewünschte Ergebnis x^n ableiten? Ja! Dazu greifen wir auf die Eigenschaft $n = (n \div 2) \cdot 2 + (n \,\%\, 2)$, siehe Abschnitt 3.2, und etwas Schulmathematik zurück (Potenzgesetze).

$$x^n = x^{(n \div 2) \cdot 2 + (n \% 2)} = (x^{n \div 2})^2 * x^{n \% 2}$$

Wir müssen also $power\ (x, n \div 2)$ quadrieren und das Ergebnis mit $x^{n\%2}$ multiplizieren. Dieser Faktor ist entweder 1 oder x, je nachdem, ob n gerade ($n \,\%\, 2 = 0$) oder ungerade ($n \,\%\, 2 = 1$) ist.

$$\begin{array}{l} \textbf{\textit{let rec}}\ power\ (x : Nat, n : Nat) : Nat = \\ \quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\ \qquad\qquad \textbf{\textit{else if}}\ n \,\%\, 2 = 0\ \textbf{\textit{then}}\ square\ (power\ (x, n \div 2)) \\ \qquad\qquad\qquad\qquad\ \ \textbf{\textit{else}}\ \ square\ (power\ (x, n \div 2)) * x \end{array}$$

Wieviele rekursive Aufrufe benötigt $power\ (x, n)$ jetzt? In jedem Schritt wird n halbiert; das können wir insgesamt $\lg n$ mal machen.[4] Möchte man die Anzahl der Rechenschritte nur grob klassifizieren, so sagt man, *power* hat eine *logarithmische Laufzeit*, im Unterschied zur ersten Version, die eine *lineare Laufzeit* hat. Die folgende Tabelle zeigt, dass Programme mit logarithmischer Laufzeit einen erheblichen Geschwindigkeitsvorteil gegenüber Programmen mit linearer Laufzeit haben.

n	$\lg n$
100	$\approx 6,6$
1.000	$\approx 10,0$
10.000	$\approx 13,3$
100.000	$\approx 16,6$
1.000.000	$\approx 20,0$

Für eine Million Elemente ist die binäre Herangehensweise also 50.000 mal schneller als die lineare Implementierung der Potenzfunktion. Natürlich ist das nur eine Abschätzung; für eine präzise Aussage müsste man die tatsächliche Anzahl der Rechenschritte ermitteln. Da uns der binäre Logarithmus wiederholt begegnen wird, lohnt es sich die Tabelle einzuprägen. Dazu reicht ein Fakt, $\lg 1.000 \approx 10,0$, und eine Formel, $\lg(a \cdot b) = \lg a + \lg b$. Zum Beispiel ist $\lg 1.000.000 = 2 \cdot \lg 1.000 \approx 20,0$. Die Approximation ist übrigens recht ordentlich, da $2^{10} = 1.024$.

Mit dem gleichen Entwurfsmuster lässt sich auch die Multiplikation verbessern.

$$m * n = (2 * (m \div 2) + (m \,\%\, 2)) * n = 2 * ((m \div 2) * n) + (m \,\%\, 2) * n$$

Jetzt müssen wir die Teillösung verdoppeln und gegebenfalls n addieren.

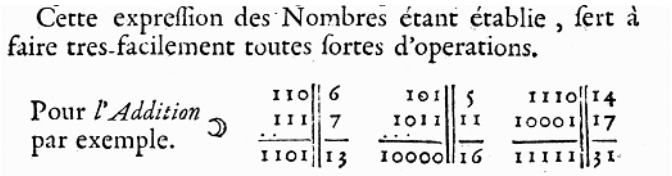
$$\begin{array}{l} \textbf{\textit{let rec}}\ mul\ (m : Nat, n : Nat) : Nat = \\ \quad \textbf{\textit{if}}\ m = 0\ \textbf{\textit{then}}\ 0 \\ \qquad\qquad \textbf{\textit{else if}}\ m \,\%\, 2 = 0\ \textbf{\textit{then}}\ 2 * mul\ (m \div 2, n) \\ \qquad\qquad\qquad\qquad\ \ \textbf{\textit{else}}\ \ 2 * mul\ (m \div 2, n) + n \end{array}$$

Es ist übrigens wichtig, dass wir das Ergebnis verdoppeln, nicht etwa die Rechnung: der Ausdruck $mul\ (m \div 2, n) + mul\ (m \div 2, n)$ würde $mul\ (m \div 2, n)$ zweimal ausrechnen.

[4] lg ist der binäre Logarithmus: $\lg x = \log_2 x$.

Gottfried Wilhelm Leibniz war ein deutscher Philosoph und Wissenschaftler, Mathematiker, Diplomat, Physiker, Historiker, Bibliothekar und Doktor des weltlichen und des Kirchenrechts. Er gilt als der universale Geist seiner Zeit und war einer der bedeutendsten Philosophen des ausgehenden 17. und beginnenden 18. Jahrhunderts.

Auszug aus „Explication de l'Arithmétique Binaire“:



Cette expreſſion des Nombres étant établie, ſert à faire tres-facilement toutes ſortes d'operations.

Pour *l'Addition* par exemple.

110	6		101	5		1110	14	
111	7		1011	11		10001	17	
1101	13		10000	16		11111	31	

Abbildung 3.5.: Gottfried Wilhelm Leibniz (1646 – 1716).

Die obige Implementierung der Multiplikation ist sehr hardware-nah; ähnlich geht auch der Computer vor — die Operationen $e*2$, $e \div 2$, $e \,\%\, 2$ sind sehr leicht in Hardware zu implementieren, da die Arithmetik eines Computers auf dem sogenannten *Dualsystem* basiert. Mehr dazu später aus der Technischen Abteilung der Informatik, siehe auch Abschnitt 4.2.2.

Auch dem neuen Entwurfsmuster geben wir einen Namen: *Leibniz Entwurfsmuster* nach dem deutschen Universalgelehrten Gottfried Wilhelm Leibniz (1646 – 1716), der unter anderem das Dualsystem entwickelt hat, siehe Abbildung 3.5.

```
let rec f (n : Nat) : t =                 Leibniz Entwurfsmuster:
  if n = 0 then ...                       Rekursionsbasis
           else ... f (n ÷ 2) ...         Rekursionsschritt
```

Versuchen wir das Leibniz Entwurfsmuster auf die Wurzelfunktion anzuwenden. Können wir aus $\lfloor\sqrt{n \div 2}\rfloor$ das gewünschte Ergebnis $\lfloor\sqrt{n}\rfloor$ ableiten? Vielleicht, unmittelbar drängt sich jedoch keine Lösung auf. Nun ist es nicht zwingend durch zwei zu dividieren, die Zerlegung $n = (n \div b) * b + (n \,\%\, b)$ gilt für jedes beliebige $b > 0$. Für unser Problem ist eine Quadratzahl, zum Beispiel vier, eine geschickte Wahl. Da weiterhin $2\lfloor\sqrt{n \div 4}\rfloor$ und $\lfloor\sqrt{n}\rfloor$ höchstens um eins differieren (Aufgabe 3.11 fragt nach einem Beweis), können wir unsere ursprüngliche Lösung einfach adaptieren.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then 0
           else let r = 2 * square-root (n ÷ 4)
                in if n < square (r + 1) then r else r + 1
```

Diese Implementierung ist schon recht ordentlich, einzig die in jedem Rekursionsschritt durchgeführte Multiplikation bremst das Programm etwas aus — unser Rechenmodell setzt für eine Multiplikation einen Rechenschritt an, Multiplikationen sind aber teure Operationen in Hardware. Für's Erste haben wir genug Arithmetik betrieben, wenden wir uns spielerischen Dingen zu.

module Values. Game

Ratespiel Programmieren wir ein kleines Spiel: Spieler A denkt sich eine Zahl aus, höchstens sechsstellig, die Spielerin B erraten muss. Spielerin B darf dazu Fragen der Form „Ist die gesuchte Zahl gleich oder kleiner als 815?“ stellen, die Spieler A wahrheitsgemäß beantworten muss. Unsere Aufgabe ist es, die Logik von Spielerin B zu entwerfen und zu implementieren.

Bevor wir mit der Programmierung beginnen, müssen wir uns zunächst Gedanken über die Schnittstelle machen. Spieler A muss zu einem gegebenen n Auskunft geben, ob die gesuchte Zahl gleich oder kleiner als n ist. Wir können ihn somit durch eine Funktion des Typs $Nat \to Bool$ repräsentieren, ein sogenanntes *Orakel*. Zum Beispiel:

```
let player-A (guess : Nat) : Bool = 4711 ⩽ guess
```

Das Orakel machen wir Spielerin B bekannt, die somit durch eine Funktion des Typs $(Nat \to Bool) \to Nat$ implementiert wird: Ein Orakel wird abgebildet auf die gesuchte Zahl. Die Repräsentation des Orakels stellt dabei sicher, dass keine der beteiligten Parteien mogelt: Die gesuchte Zahl ist fest verdrahtet — Spieler A kann sie nicht nachträglich ändern — kann aber auch nicht eingesehen werden — Spielerin B kann das Orakel nur auf eine Zahl anwenden und aus dem Ergebnis ihre Schlüsse ziehen.

Kommen wir zur Logik von Spielerin B. Wir können systematisch alle Zahlen beginnend mit 0 durchprobieren, bis das Orakel eine positive Antwort liefert.

```
let player-B (oracle : Nat → Bool) : Nat =
  let rec search (n : Nat) : Nat =
    if oracle n then n
                else search (n + 1)
  in search 0
```

Dieser Ansatz sieht dem Peano Entwurfsmuster ähnlich, ist es aber nicht: Um das Problem für n zu lösen, wird auf die Lösung für $n + 1$ zurückgegriffen! Auch die Rekursionsbasis ist nicht erkennbar. Ein mulmiges Gefühl ist in der Tat angebracht. Der Aufruf *player-B* (**fun** $k \to$ *false*) hat keinen Wert. Der Aufruf *search* 0 führt zum Aufruf *search* 1, dieser zu *search* 2 usw. Man sagt auch, dass Programm *terminiert nicht*. Die Semantik ordnet dem Aufruf entsprechend keinen Wert zu. Um einen Beweisbaum für

search 0 zu konstruieren, benötigen wir einen Beweisbaum für *search* 1; für die Konstruktion dieses Beweisbaums benötigen wir einen Beweisbaum für *search* 2 usw. Ein unendlicher Regress.

Mit dem Konstrukt der Rekursion haben wir uns das Problem der Nichtterminierung eingehandelt. An dieser Stelle kommt wieder das Peano Entwurfsmuster ins Spiel. Es bändigt die Rekursion und stellt insbesondere die Terminierung sicher: Das Problem für n wird auf das Problem für $n \dot{-} 1$ zurückgeführt; irgendwann wird auf diese Weise der Basisfall $n = 0$ erreicht. Auf unser Beispiel übertragen heißt das, dass wir die Suche nach oben beschränken müssen. In der Aufgabenstellung war von einer höchstens sechsstelligen Zahl die Rede, also müssen wir nur die Zahlen bis maximal 999.999 abklappern. Um das Programm flexibel zu halten, parametrisieren wir *player-B* mit der oberen, sowie der unteren Grenze des Suchintervalls.

```
let player-B (oracle : Nat → Bool,
              lower : Nat, upper : Nat) : Nat =
  let rec search (n : Nat) : Nat =
    if n = upper then upper
                 else if oracle n then n
                                  else search (n + 1)
  in search lower
```

Was machen wir, wenn die Suche die obere Grenze erreicht hat? Wir geben einfach die Grenze selbst zurück — im Vertrauen darauf, dass Spieler A nicht mogelt und sich tatsächlich eine Zahl im Suchintervall ausgedacht hat.

Terminiert die neue Version von *player-B stets*? Vielleicht. Wir können ganz sichergehen, indem wir das Peano Entwurfsmuster beherzigen und nicht über den Ratekandidaten selbst rekurrieren, sondern über den *Abstand* des Kandidaten zur oberen Schranke.

```
let player-B (oracle : Nat → Bool,
              lower : Nat, upper : Nat) : Nat =
  let rec search (d : Nat) : Nat =
    if d = 0 then upper
             else if oracle (upper ∸ d) then upper ∸ d
                                        else search (d ∸ 1)
  in search (upper ∸ lower)
```

Voilà. Die Terminierung ist sichergestellt. Die Terminierung der ursprünglichen Version damit auch? Nein! Diese weicht tatsächlich in einem kleinen Detail ab. Der Aufruf *player-B* (**fun** $k \to 2 \leqslant k, 9, 0$) terminiert zum Beispiel nicht. Die „untere“ Schranke liegt über der „oberen“, somit wertet der Test $n = upper$ nie zu wahr aus. Glücklicherweise ist das einfach zu reparieren: Wir ersetzen den Test $n = upper$ durch $n \geqslant upper$. In der letzten Version von *player-B* wird dieser Fall automatisch mitbehandelt, da $upper \dot{-} lower$ zu 0 auswertet, wenn die untere Schranke über der oberen liegt. (Zur Erinnerung: ‘$\dot{-}$’ ist „monus“, die Subtraktion auf den natürlichen Zahlen, nicht minus.)

Einstweiliges Fazit: nie ohne guten Grund von den Entwurfsmustern abweichen! Programmierfehler sind oft sehr subtil und aus einem Programmierfehler wird heutzutage schnell ein „Sicherheitsloch“.

Probieren wir das Programm aus.

```
Mini⟩ player-B (player-A, 0, 999.999)
4711
Mini⟩ player-B (fun k → 815 ⩽ k, 0, 999)
815
```

Es klappt! Werden wir etwas wagemutiger: Spieler A kann sich natürlich auch eine Zahl ausdenken, die durch eine Formel gegeben ist.

```
Mini⟩ player-B (fun k → 815 < square (k + 1), 0, 999)
28
```

Jetzt hat Spielerin B die natürliche Quadratwurzel von 815 bestimmt: $28^2 = 784 \leqslant 815 < 841 = 29^2$. Diese Erkenntnis können wir als Programm festhalten.

```
let square-root (n : Nat) : Nat =
  linear-search ((fun k → n < square (k + 1)), 0, n)
```

Da aus der Implementierung eines Spiels ein Programmstück von allgemeinem Nutzen geworden ist, haben wir *player-B* in *linear-search* umbenannt. Der Name charakterisiert die Suchstrategie: Das Suchintervall wird linear von links nach rechts durchforstet. Im schlimmsten Fall benötigt *linear-search* n rekursive Aufrufe, durchschnittlich werden immerhin noch $n \div 2$ Rekursionsschritte benötigt.

Können wir die Laufzeit mit Hilfe des Leibniz Entwurfsmusters verbessern? Das Entwurfsmuster sieht vor, den Parameter, in unserem Fall die Größe des Suchintervalls (l, u), in jedem Rekursionsschritt zu halbieren. Die Mitte des Suchintervalls ist durch $m = l + (u \dot{-} l) \div 2$ oder gleichwertig durch $m = (l + u) \div 2$ gegeben (falls $l \leqslant u$). Welche Schlüsse können wir aus der Befragung des Orakels an der Stelle m ziehen? Wertet der Aufruf *oracle m* zu wahr aus — die gesuchte Zahl ist gleich oder kleiner als die geratene —, dann müssen wir im Intervall (l, m) suchen, anderenfalls muss die gesuchte Zahl im Intervall $(m + 1, u)$ liegen. Der Basisfall ist erreicht, wenn die untere und die oberer Intervallgrenze zusammenfallen.

```
let binary-search (oracle : Nat → Bool,
                   lower : Nat, upper : Nat) : Nat =
  let rec search (l : Nat, u : Nat) : Nat =
    if l ⩾ u then u
             else let m = (l + u) ÷ 2
                  in if oracle m then search (l,     m)
                                 else search (m + 1, u)
  in search (lower, upper)
```

Es ist instruktiv, sich die Abfolge der Rateversuche anzuschauen. Zu diesem Zweck lassen wir Spieler A die geratenen Zahlen am Bildschirm ausgeben. (Wir greifen hier etwas vor: Ausgaben werden erst in Abschnitt 7.1 besprochen, da es sich um einen Effekt handelt.)

```
let player-A (guess : Nat) : bool =
  putline ("Geratene Zahl: " ^ show guess);
  4711 ⩽ guess
```

Die Funktion *putline* gibt einen String auf dem Bildschirm aus; *show* wandelt eine natürliche Zahl in einen String um, '^' konkateniert zwei Strings und ';' führt zwei Rechnungen hintereinander aus.

Testen wir die Implementierung mit der instrumentierten Version von *player-A*.

```
Mini⟩ binary-search (player-A, 0, 999999)
Geratene Zahl : 499999
Geratene Zahl : 249999
Geratene Zahl : 124999
Geratene Zahl : 62499
Geratene Zahl : 31249
Geratene Zahl : 15624
Geratene Zahl : 7812
Geratene Zahl : 3906
Geratene Zahl : 5859
Geratene Zahl : 4883
Geratene Zahl : 4395
Geratene Zahl : 4639
Geratene Zahl : 4761
Geratene Zahl : 4700
Geratene Zahl : 4731
Geratene Zahl : 4716
Geratene Zahl : 4708
Geratene Zahl : 4712
Geratene Zahl : 4710
Geratene Zahl : 4711
4711
```

Die gesuchte Zahl wird systematisch eingekreist; nach 20 Versuchen ist sie erraten.

Mit der verbesserten Suchstrategie lässt sich auch *square-root* auf eine logarithmische Laufzeit beschleunigen.

```
let square-root (n : Nat) : Nat =
  binary-search ((fun k → n < square (k + 1)), 0, n)
```

Fassen wir zusammen: Das gleiche Problem lässt sich auf verschiedene Art und Weise lösen. Hier liegt die kreative gedankliche Leistung der Programmiererin oder des Programmierers. Die Entwurfsmuster helfen einen ersten Ansatz systematisch zu entwickeln.

Beiden Entwurfsmustern ist gemeinsam, dass Probleme auf Teilprobleme reduziert werden. Programmieren wir eine rekursive Funktion „freihändig“, müssen wir selbst darauf achten, dass die Parameter der rekursiven Aufrufe kleiner werden, so dass die Terminierung sichergestellt ist. Und noch einmal: Programmierfehler sind oft sehr subtil und aus einem Programmierfehler wird heutzutage schnell ein „Sicherheitsloch“.

Schauen wir uns die Funktion *binary-search* noch einmal durch die Terminierungsbrille an. Die Intervallgröße muss stets schrumpfen: Im Rekursionsschritt wird ein Intervall der Größe $n = r \dot{-} l + 1$ in zwei Intervalle der Größen $(n+1) \div 2$ und $n \div 2$ unterteilt — es gilt stets $n = (n+1) \div 2 + n \div 2$. Nun ist $(n+1) \div 2$ nur echt kleiner als n, wenn n echt größer als 1 ist. Das bedeutet, dass $n = 1$ nicht im Rekursionsschritt behandelt werden darf, sondern neben $n = 0$ einen weiteren Basisfall darstellt. Im Programm fängt der Test $l \geqslant r$ beide Basisfälle ab; im Rekursionsschritt werden nur Suchintervalle behandelt, die mindestens zwei Elemente umfassen.

Übungen.

1. Machen Sie sich noch einmal den Unterschied zwischen Syntax und Semantik klar.

1. Wieviele Boolesche Ausdrücke gibt es?
2. Wieviele Boolesche Werte gibt es?

2. Die Vergleichsoperationen ($<$, $\leqslant$, $=$, $<>$, $\geqslant$, $>$) arbeiten auf den natürlichen Zahlen. Übertragen Sie die Operationen auf Boolesche Werte unter der Prämisse, dass $\mathit{false} < \mathit{true}$. Für die Operation ‘$\leqslant$’ definieren Sie zum Beispiel eine Funktion

$$\textbf{\textit{let}}\ \mathit{leq}\ (a : \mathit{Bool}, b : \mathit{Bool}) : \mathit{Bool}$$

die die folgende Wahrheitstabelle implementiert:

		b	
	$a \leqslant b$	*false*	*true*
a	*false*	*true*	*true*
	true	*false*	*true*

Zu welchen Booleschen Verknüpfungen korrespondieren die Vergleichsoperationen?

3. Wir haben e_1 `&&` e_2 als Abkürzung für $\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ \mathit{false}$ eingeführt. Was ist der Unterschied zwischen e_1 `&&` e_2 und dem Funktionsaufruf *and-also* (e_1, e_2)? *Hinweis:* Bestimmen Sie die Semantik beider Ausdrücke für $e_1 = \mathit{false}$ und $e_2 = 1 \div 0$.

4. Finden Sie zu jedem der folgenden Typen einen passenden Ausdruck:

1. $\mathit{Bool} \to \mathit{Bool}$,
2. $\mathit{Bool} * \mathit{Bool} \to \mathit{Bool}$,
3. $\mathit{Bool} \to (\mathit{Bool} \to \mathit{Bool})$,
4. $(\mathit{Bool} \to \mathit{Bool}) \to \mathit{Bool}$.

Um die Aufgabe etwas interessanter zu machen, verlangen wir, dass alle formalen Parameter im jeweiligen Funktionsrumpf auch verwendet werden müssen. Somit scheidet $\textbf{\textit{fun}}\ (x : \mathit{Bool}) \to \mathit{false}$ als Lösung für Teil 1 aus.

Wieviele semantisch unterschiedliche Funktionen gibt es jeweils?

5. Mit Hilfe der dynamischen Semantik kann man zeigen, dass zwei Ausdrücke äquivalent sind, dass beide Ausdrücke zum gleichen Wert ausrechnen, etwa $(2+3)*4$ und $2*4+3*4$. (Zu diesem Zweck kann auch der Mini–F# Interpreter herangezogen werden.) Es lassen sich darüber hinaus auch allgemeine Aussagen über *Ausdrucksschemata* treffen, das sind Ausdrücke, die Metavariablen enthalten. Zum Beispiel gilt das Distributivgesetz:

$$(e_1 + e_2) * e_3 \quad = \quad (e_1 * e_3) + (e_2 * e_3)$$

für beliebige Ausdrücke e_1, e_2 und e_3. Dazu muss man zeigen, dass sich jeder Beweisbaum für $(e_1+e_2)*e_3 \Downarrow \nu$ in einen Beweisbaum für $(e_1*e_3)+(e_2*e_3) \Downarrow \nu$ überführen lässt und umgekehrt. Da es für '+' und '$*$' nur jeweils eine Auswertungsregel gibt, ist der Beweis recht einfach: Ist $\mathcal{P}_i$ ein Beweisbaum mit der Wurzel $e_i \Downarrow \nu_i$, dann lassen sich die Beweisbäume

$$\dfrac{\dfrac{\mathcal{P}_1 \qquad \mathcal{P}_2}{e_1 + e_2 \Downarrow \nu_1 + \nu_2} \qquad \mathcal{P}_3}{(e_1 + e_2) * e_3 \Downarrow (\nu_1 + \nu_2)\nu_3} \qquad \text{und} \qquad \dfrac{\dfrac{\mathcal{P}_1 \qquad \mathcal{P}_3}{e_1 * e_3 \Downarrow \nu_1\nu_3} \qquad \dfrac{\mathcal{P}_2 \qquad \mathcal{P}_3}{e_2 * e_3 \Downarrow \nu_2\nu_3}}{(e_1 * e_3) + (e_2 * e_3) \Downarrow \nu_1\nu_3 + \nu_2\nu_3}$$

direkt ineinander überführen. (Dass $(\nu_1 + \nu_2)\nu_3 = \nu_1\nu_3 + \nu_2\nu_3$ ist, folgt aus den algebraischen Eigenschaften der Addition und der Multiplikation.) Zeigen Sie auf ähnliche Art und Weise die Äquivalenz der folgenden Ausdrucksschemata.

$$\begin{aligned}
&\textbf{\textit{if}}\ \mathit{true}\ \textbf{\textit{then}}\ e_1\ \textbf{\textit{else}}\ e_2 && = && e_1 \\
&\textbf{\textit{if}}\ \mathit{false}\ \textbf{\textit{then}}\ e_1\ \textbf{\textit{else}}\ e_2 && = && e_2 \\
&\textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ \mathit{true}\ \textbf{\textit{else}}\ \mathit{false} && = && e \\
&\textbf{\textit{if}}\ (\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3)\ \textbf{\textit{then}}\ e_4\ \textbf{\textit{else}}\ e_5 && && \\
&\qquad && = && \textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ \textbf{\textit{if}}\ e_2\ \textbf{\textit{then}}\ e_4\ \textbf{\textit{else}}\ e_5\ \textbf{\textit{else}}\ \textbf{\textit{if}}\ e_3\ \textbf{\textit{then}}\ e_4\ \textbf{\textit{else}}\ e_5 \\
&(\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3) * e_4 && = && \textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2 * e_4\ \textbf{\textit{else}}\ e_3 * e_4
\end{aligned}$$

Lassen sich die letzten beiden Regeln verallgemeinern?

6. Geben Sie für jeden der folgenden Ausdrücke an, welche Bezeichner er beinhaltet, und welche der Bezeichner frei bzw. gebunden sind.

- $\textbf{\textit{let}}\ a = c\ \textbf{\textit{in}}\ \textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ b\ \textbf{\textit{else}}\ \mathit{true}$,
- $\textbf{\textit{let}}\ a = 1\ \textbf{\textit{let}}\ b = \mathit{true}\ \textbf{\textit{in}}\ \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ a + 1\ \textbf{\textit{else}}\ c + 1$,
- $\textbf{\textit{let}}\ a = c\ \textbf{\textit{let}}\ b = \mathit{true}\ \textbf{\textit{in}}\ \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ a + 1\ \textbf{\textit{else}}\ d + 1$,
- $\textbf{\textit{let}}\ a = \mathit{true}\ \textbf{\textit{in}}\ \textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ 2$.

7. Werten Sie die folgenden Ausdrücke nacheinander mit dem Mini–F# Interpreter aus:

- $\textbf{\textit{let}}\ b = 5\ \textbf{\textit{in}}\ \textbf{\textit{let}}\ a = 4\ \textbf{\textit{in}}\ a * b$,
- $\textbf{\textit{let}}\ d = \mathit{true}\ \textbf{\textit{in}}\ \textbf{\textit{let}}\ b = d\ \textbf{\textit{let}}\ a = \mathit{false}\ \textbf{\textit{let}}\ c = \mathit{true}\ \textbf{\textit{in}}\ \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ a\ \textbf{\textit{else}}\ c$,
- $\textbf{\textit{let}}\ c = 7\ \textbf{\textit{in}}\ \textbf{\textit{let}}\ b = 5\ \textbf{\textit{let}}\ c = 4\ \textbf{\textit{in}}\ b + c$.

Geben Sie für jeden Ausdruck die vollständige Rechnung in Form eines Beweisbaums an.

8. Definieren Sie eine Funktion

$$\textbf{\textit{let}}\ \mathit{total\text{-}area}\ (l_1, r_1, b_1, t_1, l_2, r_2, b_2, t_2) : \mathit{Nat}$$

die die Gesamtfläche einer aus zwei *Rechtecken* bestehenden Fläche berechnet — wie im Beispiel aus Abschnitt 3.3 können sich die Flächen überlappen. Verallgemeinern Sie die Funktion auf *drei* sich möglicherweise überlappende Rechtecke.

9. Folgt die folgende Version der Potenzfunktion

$$\begin{array}{l}\textbf{\textit{let}}\ \mathit{power}\ (x : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} = \\ \quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\ \qquad\qquad \textbf{\textit{else}}\ \ \textbf{\textit{if}}\ n \,\%\, 2 = 0\ \textbf{\textit{then}}\ \mathit{power}\ (x * x, n \div 2) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{\textit{else}}\ \ \mathit{power}\ (x * x, n \div 2) * x\end{array}$$

dem Leibniz Entwurfsmuster?

10. Die dynamische Semantik schreibt vor, dass beide Argumente der arithmetischen Operationen und der Vergleichsoperationen ausgewertet werden. Muss das zwangsläufig so sein?

11. Die nach den Entwurfsmustern konstruierten Implementierungen von *square-root* beruhen auf den folgenden Eigenschaften:

1. $0 \leqslant \lfloor\sqrt{n}\rfloor - \lfloor\sqrt{n-1}\rfloor \leqslant 1$ für alle $n > 0$,
2. $0 \leqslant \lfloor\sqrt{n}\rfloor - 2\lfloor\sqrt{n \div 4}\rfloor \leqslant 1$ für alle $n > 0$.

Zeigen Sie die Eigenschaften.

12. Werten Sie die folgenden Ausdrücke bzw. Deklarationen nacheinander mit dem Mini–F# Interpreter aus:

- $(\textbf{\textit{fun}}\ (n : \mathit{Nat}) \to n)\ 4711$,
- $(\textbf{\textit{fun}}\ (n : \mathit{Nat}) \to n * n)\ 4711$,
- $\textbf{\textit{let}}\ \mathit{twice} = \textbf{\textit{fun}}\ (f : \mathit{Nat} \to \mathit{Nat}) \to \textbf{\textit{fun}}\ (x : \mathit{Nat}) \to f\ (f\ x)$,
- $\mathit{twice}\ (\textbf{\textit{fun}}\ (n : \mathit{Nat}) \to n * n)\ 4711$,
- $\mathit{twice}\ (\mathit{twice}\ (\textbf{\textit{fun}}\ (n : \mathit{Nat}) \to n * n))\ 4711$.

Geben Sie für jeden Ausdruck die vollständige Rechnung als Beweisbaum an. Beachten Sie, dass die letzten beiden Ausdrücke die Definition $\textbf{\textit{let}}\ \mathit{twice} = \cdots$ sehen.

13. Programmieren Sie zwei Funktionen $\mathit{even} : \mathit{Nat} \to \mathit{Bool}$ und $\mathit{odd} : \mathit{Nat} \to \mathit{Bool}$, die bestimmen, ob eine Zahl gerade bzw. ungerade ist. Geben Sie für jede Funktion zwei verschiedene Definitionen an:

- Gehen Sie einmal *streng* nach dem Peano Entwurfsmuster vor.
- Programmieren Sie „freihändig“ unter Zuhilfenahme der Operatoren $\div$ und $\%$.

14. Schreiben Sie eine rekursive Funktion, die die Quersumme einer Zahl ermittelt.

15. Warum kann man die Funktion *factorial* nicht mit Hilfe von *peano-pattern* definieren? Lässt sich die Funktion *peano-pattern* erweitern, so dass dies möglich wird?

16. Die Funktion *binary-search* setzt voraus, dass die Größe des zu untersuchenden Bereichs bekannt ist. Natürlich gibt es auch Fälle, in denen der Suchbereich vorher nicht eingeschränkt werden kann. In diesem Fall muss zunächst die obere Grenze des Suchintervalls bestimmt werden. Diese Aufgabe übernimmt die Funktion *exponential-search*: In exponentiell wachsenden Schritten (1, 2, 4, 8, 16, …) wird der Bereich bestimmt, in dem das gesuchte Element liegen muss. Nachdem das Suchintervall gefunden wurde, kann dieses mit der Funktion *binary-search* durchsucht werden. Implementieren Sie die Funktion *exponential-search*.

4. Datentypen ⧹ Rechnen mit Daten

I think that I shall never see
A poem lovely as a tree.
— Joyce Kilmer (1886–1918), *Trees*

Informatikerinnen und Informatiker bilden Modelle der Wirklichkeit; einen nicht unwesentlichen Teil dieser Modelle machen Daten aus. Daten repräsentieren alles Wissenswerte über einen Weltausschnitt, alles was für die jeweilige Anwendung relevant ist oder zumindest relevant erscheint. So werden aus Personen Stammdaten, aus An- und Verkäufen Börsendaten, aus Wolken Luftströmungen und aus pittoresken Sonnenuntergängen Wetterdaten.

Modellbildung ist immer Vereinfachung, Beschränkung und Abstraktion. Für die Verwaltung eines Unternehmens wird zum Beispiel ein Mitarbeiter oder eine Mitarbeiterin auf einige wenige Angaben reduziert. In der Regel werden Name, Geburtsdatum, Geschlecht und Familienstand erfasst; vielleicht die Dauer des Beschäftigungsverhältnisses oder die Stellung im Unternehmen; eher unwahrscheinlich ist die Erfassung von Statur, Bekannten- oder Freundeskreis oder des Lieblingsitalieners (insbesondere nach der WM 2006).

In diesem Kapitel beschäftigen wir uns damit, wie man Daten strukturiert und verarbeitet. Die Modellbildung selbst wird dabei eine eher untergeordnete Rolle spielen. Zu diesem Thema erfahren Sie im weiteren Verlauf des Studiums mehr aus der Abteilung „Software Engineering“. (Wir beschäftigen uns im Wesentlichen mit der „Programmierung im Kleinen“; die „Programmierung im Großen“ streifen wir dabei nur gelegentlich, siehe aber Kapitel 8.) Unser bisheriges Repertoire an Datentypen ist bescheiden: Wir haben Boolesche Werte, natürliche Zahlen, Funktionen und Zeichenketten im Angebot. Was uns fehlt, sind Möglichkeiten

- mehrere Daten zu einem Datum[1] zusammenzufassen: etwa einen Straßennamen, eine Postleitzahl und einen Ortsnamen zu einer Adresse;
- mehrere alternative Angaben als Einheit zu behandeln: etwa den Familienstand mit den Alternativen ledig, verheiratet mit Angabe des Datums der Trauung oder geschieden ebenfalls mit Datumsangabe.

Wenn größere Datenmengen zusammengefasst werden, wird die Frage interessant, wie gut oder auch wie schnell man auf ein einzelnes Datum zugreifen kann. Die Organisation von Daten tritt in den Vordergrund: Aus Daten werden Datenstrukturen. Zwei grundlegende Datenstrukturen werden wir ebenfalls in diesem Kapitel kennenlernen: Listen und Arrays.

[1] Datum ist der Singular von Daten.

4.1. Records

Mit Hilfe der Alternative können wir von zwei gegebenen Zahlen einfach das Minimum (die kleinere Zahl) und das Maximum (die größere Zahl) bestimmen.

$$\begin{array}{l}\textbf{\textit{let}}\ \mathit{minimum}\ (a : \mathit{Nat}, b : \mathit{Nat}) = \textbf{\textit{if}}\ a \leqslant b\ \textbf{\textit{then}}\ a\ \textbf{\textit{else}}\ b \\ \textbf{\textit{let}}\ \mathit{maximum}\ (a : \mathit{Nat}, b : \mathit{Nat}) = \textbf{\textit{if}}\ a \leqslant b\ \textbf{\textit{then}}\ b\ \textbf{\textit{else}}\ a\end{array}$$

Benötigen wir beide Informationen auf einen Schlag — das heißt, wollen wir die Argumente a und b der Größe nach ordnen — müssen wir ein *Paar* von Zahlen zurückgeben.

$$\begin{array}{l}\textbf{\textit{let}}\ \mathit{sort2}\ (a : \mathit{Nat}, b : \mathit{Nat}) : \mathit{Nat} * \mathit{Nat} = \\ \quad \textbf{\textit{if}}\ a \leqslant b\ \textbf{\textit{then}}\ (a, b)\ \textbf{\textit{else}}\ (b, a)\end{array}$$

Paare erlauben es uns, Daten zu aggregieren, zwei verschiedene Daten als Einheit zu behandeln. Die zwei Komponenten eines Paares müssen dabei nicht den gleichen Typ besitzen: $(\texttt{"Lisa"}, 9)$ vom Typ $\mathit{String} * \mathit{Nat}$ fasst zum Beispiel einen String und eine natürliche Zahl zusammen, etwa Name und Alter einer Person; $(7, \textbf{\textit{fun}}\ (i : \mathit{Nat}) \to 2 * i)$ vom Typ $\mathit{Nat} * (\mathit{Nat} \to \mathit{Nat})$ aggregiert eine natürliche Zahl und eine Funktion und repräsentiert vielleicht eine endliche Abbildung.

Paare sind tatsächlich kein neues Konzept; sie sind uns schon in einem früheren Abschnitt begegnet, nämlich als Argumente von Funktionen. Wir haben uns bisher auf den Standpunkt gestellt, dass Funktionen wie *minimum* und *maximum* zwei Argumente erhalten (a und b). Eine alternative Sichtweise ist, dass Funktionen stets genau ein Argument verarbeiten, im Fall von *minimum* und *maximum* ist dieses eine Argument eben das Paar (a, b). Da die zweite Sichtweise vorteilhafter ist, werden wir sie uns zu eigen machen. Mehr zu diesem Thema im Abschnitt 4.1.2.

Zurück zu unserem einleitenden Beispiel: wir haben die Funktion *sort2* „von Grund auf“ neu programmiert; alternativ können wir bei der Definition auf *minimum* und *maximum* zurückgreifen.

$$\textbf{\textit{let}}\ \mathit{sort2}\ (a : \mathit{Nat}, b : \mathit{Nat}) : \mathit{Nat} * \mathit{Nat} = (\mathit{minimum}\ (a, b), \mathit{maximum}\ (a, b))$$

Der Programmtext ist etwas kürzer, hat aber den kleinen Nachteil, dass beim Ausrechnen *zwei* Alternativen abgearbeitet werden müssen.

Lassen sich umgekehrt *minimum* und *maximum* auch mit Hilfe von *sort2* programmieren? Ja! So geht's:

$$\begin{array}{l}\textbf{\textit{let}}\ \mathit{minimum}\ (a : \mathit{Nat}, b : \mathit{Nat}) : \mathit{Nat} = \mathit{fst}\ \ (\mathit{sort2}\ (a, b)) \\ \textbf{\textit{let}}\ \mathit{maximum}\ (a : \mathit{Nat}, b : \mathit{Nat}) : \mathit{Nat} = \mathit{snd}\ (\mathit{sort2}\ (a, b))\end{array}$$

Ist e ein Ausdruck, der zu einem Paar auswertet, so kann mit *fst* e auf die erste und mit *snd* e auf die zweite Komponente zugegriffen werden. Die alternativen Definitionen von *minimum* und *maximum* sind ebenfalls etwas prägnanter, haben aber den kleinen Nachteil, dass *sort2* ein Paar konstruiert, von dem stets nur eine Komponente benötigt wird.

4.1.1. Binäre Tupel⁄Paare

Im Folgenden formalisieren wir Syntax und Semantik von Paaren. Alle Konstrukte verallgemeinern sich in natürlicher Weise auf *Tupel*, Aggegrationen von n verschiedenen Komponenten. In den Beispielprogrammen werden wir ausgiebig Gebrauch von Tupeln machen.

Abstrakte Syntax Wir erweitern Ausdrücke um Sprachkonstrukte, die Paare konstruieren und analysieren.

$e ::= \cdots$	*Paarausdrücke:*
$\mid (e_1, e_2)$	Konstruktion⁄Paarbildung
$\mid \mathit{fst}\ e$	Projektion auf die erste Komponente
$\mid \mathit{snd}\ e$	Projektion auf die zweite Komponente

Die Ausdrücke e_1 und e_2 heißen *Komponenten* des Paares (e_1, e_2).

Statische Semantik Im Typ eines Paares wird festgehalten, von welchem Typ die Komponenten sind. Mit anderen Worten, der Typ eines Paares ist ein Paar von Typen, das sogenannte kartesische Produkt der Typen.

$t ::= \cdots$	*Typen:*
$\mid t_1 * t_2$	Paartyp

Die Typregeln für Paare sind vergleichsweise einfach: Ein Paar erhält den Paartyp $t_1 * t_2$; Projektionen erwarten einen Ausdrucks des Typs $t_1 * t_2$ und selektieren die entsprechende Komponente.

$$\frac{\Sigma \vdash e_1 : t_1 \qquad \Sigma \vdash e_2 : t_2}{\Sigma \vdash (e_1, e_2) : t_1 * t_2} \qquad \frac{\Sigma \vdash e : t_1 * t_2}{\Sigma \vdash \mathit{fst}\ e : t_1} \qquad \frac{\Sigma \vdash e : t_1 * t_2}{\Sigma \vdash \mathit{snd}\ e : t_2}$$

Dynamische Semantik Wir erweitern den Bereich der Werte um Paare von Werten.

$\nu ::= \cdots$	*Werte:*
$\mid (\nu_1, \nu_2)$	Paare

Die Auswertungsregeln folgen der statischen Semantik.

$$\frac{\delta \vdash e_1 \Downarrow \nu_1 \qquad \delta \vdash e_2 \Downarrow \nu_2}{\delta \vdash (e_1, e_2) \Downarrow (\nu_1, \nu_2)} \qquad \frac{\delta \vdash e \Downarrow (\nu_1, \nu_2)}{\delta \vdash \mathit{fst}\ e \Downarrow \nu_1} \qquad \frac{\delta \vdash e \Downarrow (\nu_1, \nu_2)}{\delta \vdash \mathit{snd}\ e \Downarrow \nu_2}$$

Bevor wir uns den Beispielprogrammen zuwenden, überlegen wir noch kurz, wie die Konstrukte und Regeln allgemein für n-Tupel aussehen.

Ist $n = 0$, hat das Tupel keine Komponenten und entsprechend gibt es keine Projektionsfunktionen. Mit anderen Worten, der 0-Tupeltyp, genannt *Unit*, umfasst genau ein Element, nämlich (). Gegenwärtig ist der 0-Tupeltyp von keinem großen Nutzen; später in Kapitel 7 werden wir ihn oft als Platzhaltertyp verwenden: Programme, die um

ihres Effektes und nicht um des Wertes willen ausgerechnet werden, geben oft '()' als Dummywert zurück. Aber wir greifen vor.

Im Fall $n = 1$ haben wir ein 1-Tupel und eine einzige Projektionsfunktion. Dieser Fall ist als einziger wenig sinnvoll, da anstelle des 1-Tupels stets die einzige Komponente treten kann. Dieser Fall wird auch von der konkreten Syntax nicht unterstützt, da '(e)' zur Gruppierung bzw. Klammerung von Ausdrücken dient.

Ist $n = 3$, so konstruieren wir 3-Tupel oder Tripel und analysieren diese mit Hilfe dreier Projektionsfunktionen.

Ist $n = 4$, so konstruieren wir 4-Tupel oder Quadrupel …

Vertiefung Die Funktion *sort2* ordnet zwei natürliche Zahlen; wie lassen sich drei natürliche Zahlen sortieren?

```
let sort3 (a : Nat, b : Nat, c : Nat) : Nat * Nat * Nat =
  if a ⩽ b then
    if b ⩽ c then (a, b, c)
             else if a ⩽ c then (a, c, b) else (c, a, b)
  else
    if a ⩽ c then (b, a, c)
             else if b ⩽ c then (b, c, a) else (c, b, a)
```

Die obige Lösung benötigt bis zu drei Vergleiche: *sort3* $(1, 3, 2)$ zum Beispiel testet zunächst $1 \leqslant 3$, dann $3 \leqslant 2$ und schließlich $1 \leqslant 2$. Gibt es eine Lösung, die mit weniger Vergleichen auskommt? Nein! Drei Zahlen können auf 3 Fakultät Arten angeordnet werden: $3! = 6$. Mit zwei ineinander geschachtelten Alternativen können aber nur vier Fälle unterschieden werden. Ergo werden im schlechtesten Fall drei Vergleiche benötigt. Gleichwohl lässt sich *sort3* etwas kompakter aufschreiben, indem wir auf *sort2* zurückgreifen.

```
let sort3 (a : Nat, b : Nat, c : Nat) : Nat * Nat * Nat =
  let x = sort2 (a, b)
  in if snd x ⩽ c then (fst x, snd x, c)
                  else if fst x ⩽ c then (fst x, c, snd x)
                                    else (c, fst x, snd x)
```

Diese Version, die exakt dieselben Vergleiche durchführt wie die ursprüngliche, macht die Vorgehensweise deutlich: Zunächst werden a und b geordnet, dann wird die Position von c bestimmt. Je größer und umfangreicher ein Programm wird, desto wichtiger ist ein *modularer* Aufbau.

4.1.2. Unwiderlegbare Muster

Binden wir ein Paar an einen Bezeichner, so ist es bequem, nicht nur einen Namen für das Paar selbst, sondern auch Namen für die beiden Komponenten vergeben zu können. Die folgende Version von *sort3* nutzt dieses Feature.

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{sort3}\ (a : \mathit{Nat}, b : \mathit{Nat}, c : \mathit{Nat}) : \mathit{Nat} * \mathit{Nat} * \mathit{Nat} = \\
\quad \textbf{\textit{let}}\ (\mathit{min}, \mathit{max}) = \mathit{sort2}\ (a, b) \\
\quad \textbf{\textit{in}}\ \ \textbf{\textit{if}}\ \mathit{max} \leqslant c\ \textbf{\textit{then}}\ (\mathit{min}, \mathit{max}, c) \\
\qquad\qquad\qquad\quad \textbf{\textit{else}}\ \ \textbf{\textit{if}}\ \mathit{min} \leqslant c\ \textbf{\textit{then}}\ (\mathit{min}, c, \mathit{max}) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{\textit{else}}\ \ (c, \mathit{min}, \mathit{max})
\end{array}
$$

Selbst vergebene Namen für Komponenten, hier *min* und *max*, sind in der Regel prägnanter als Projektionen wie *fst* x und *snd* x.

Abstrakte Syntax Wir verallgemeinern Bezeichner in Bindungspositionen zu sogenannten *Mustern* (engl. patterns).

$d ::= \cdots$	*Deklarationen:*
$\mid\ \textbf{\textit{let}}\ p = e$	verallgemeinerte Wertedefinition

Entsprechend lassen sich alle Konstrukte verallgemeinern, die Bezeichner binden: Funktionsdefinitionen, Funktionsabstraktionen usw.

Um die abstrakte Syntax von Mustern präzise zu beschreiben, führen wir eine neue Baumsprache ein.

$p \in \mathsf{Pat} ::=$	*Muster:*
$\mid\ \_$	anonymer Bezeichner╲„don't care“ Muster
$\mid\ x$	Bezeichner
$\mid\ p_1\ \&\ p_2$	konjunktives Muster
$\mid\ (p_1, p_2)$	Paarmuster

Mit Hilfe des Musters $p_1\ \&\ p_2$ werden sowohl die Bezeichner in p_1 als auch in p_2 an die entsprechenden Werte gebunden. Ein konjunktives Muster kann zum Beispiel verwendet werden, um sowohl einen Namen für ein Paar als auch für dessen Komponenten zu vergeben: $\textbf{\textit{let}}\ x\ \&\ (\mathit{min}, \mathit{max}) = \mathit{sort2}\ (e_1, e_2)$. Das Muster '_' markiert Komponenten, an denen man nicht interessiert ist: $\textbf{\textit{let}}\ (\mathit{min}, \_) = \mathit{sort2}\ (e_1, e_2)$.

Statische Semantik Eine verallgemeinerte Wertedefinition bindet mehrere Bezeichner, unter Umständen auch keine.

$$\frac{\Sigma \vdash e : t \qquad p \sim t : \Sigma'}{\Sigma \vdash (\textbf{\textit{let}}\ p = e) : \Sigma'}$$

In der Voraussetzung der Regel wird von einer neuen Relation Gebrauch gemacht, die sicherstellt, dass der Musterabgleich (engl. pattern matching) wohlgetypt ist. Zum Beispiel ist $\textbf{\textit{let}}\ (\mathit{min}, \mathit{max}) = 4711$ keine sinnvolle Wertedefinition. Die Relation

$$p \sim t : \Sigma'$$

„ermittelt“ die Signatur aller Bezeichner, die bei dem Musterabgleich gebunden werden. *Lies:* der Abgleich eines Wertes vom Typ t mit dem Muster p ergibt die Signatur Σ'.

(Die Tilde $\sim$ soll den Musterabgleich symbolisieren.) Zum Beispiel werden beim Abgleich des Musters $x \,\&\, (min, max)$ mit einem Element des Typs $Nat * Nat$ Bindungen des Typs $\{x \mapsto Nat * Nat, min \mapsto Nat, max \mapsto Nat\}$ erzeugt. Die Muster heißen übrigens *unwiderlegbar* (engl. irrefutable), weil die statische Semantik garantiert, dass der Wert stets auf das Muster passt. In Abschnitt 4.2.3 lernen wir Muster kennen, auf die das nicht mehr zutrifft, sogenannte widerlegbare Muster (engl. refutable patterns).

$$\frac{}{\_ \sim t : \varnothing} \qquad \frac{}{x \sim t : \{x \mapsto t\}}$$

$$\frac{p_1 \sim t : \Sigma_1 \qquad p_2 \sim t : \Sigma_2}{(p_1 \,\&\, p_2) \sim t : \Sigma_1, \Sigma_2} \qquad \frac{p_1 \sim t_1 : \Sigma_1 \qquad p_2 \sim t_2 : \Sigma_2}{(p_1, p_2) \sim t_1 * t_2 : \Sigma_1, \Sigma_2}$$

Konjunktive Muster und Paarmuster binden unter Umständen mehrere Bezeichner; mögliche Überschneidungen werden wie immer durch den Kommaoperator aufgelöst.

Dynamische Semantik Bei der Abarbeitung einer verallgemeinerten Wertedefinition wird zunächst die rechte Seite evaluiert; der resultierende Wert wird dann mit dem Muster auf der linken Seite abgeglichen.

$$\frac{\delta \vdash e \Downarrow \nu \qquad p \sim \nu \Downarrow \delta'}{\delta \vdash \textbf{\textit{let}}\ p = e \Downarrow \delta'}$$

Der *Musterabgleich* (engl. pattern matching) selbst wird mit Hilfe der Relation

$$p \sim \nu \Downarrow \delta$$

formalisiert. *Lies:* der Abgleich des Musters p mit dem Wert ν ergibt die Umgebung δ. Die Auswertungregeln folgen den Typregeln.

$$\frac{}{\_ \sim \nu \Downarrow \varnothing} \qquad \frac{}{x \sim \nu \Downarrow \{x \mapsto \nu\}}$$

$$\frac{p_1 \sim \nu \Downarrow \delta_1 \qquad p_2 \sim \nu \Downarrow \delta_2}{(p_1 \,\&\, p_2) \sim \nu \Downarrow \delta_1, \delta_2} \qquad \frac{p_1 \sim \nu_1 \Downarrow \delta_1 \qquad p_2 \sim \nu_2 \Downarrow \delta_2}{(p_1, p_2) \sim (\nu_1, \nu_2) \Downarrow \delta_1, \delta_2}$$

Beim Abgleich eines konjunktiven Musters wird der *gleiche* Wert zweimal abgeglichen. Bei einem Paarmuster werden die Komponenten des Paares mit den Komponenten des Musters abgeglichen — durch die statische Semantik ist sichergestellt, dass der Wert tatsächlich ein Paar ist.

Vertiefung Schauen wir uns noch einmal die Sortierfunktion an.

```
let sort2 (a : Nat, b : Nat) : Nat * Nat =
  if a ⩽ b then (a, b) else (b, a)
```

Der Typ von *sort2* ist $Nat * Nat \to Nat * Nat$; die Funktion *sort2* bildet also ein Paar auf ein Paar ab. Die Parameterliste $(a : Nat, b : Nat)$ entspricht im Prinzip einem Paarmuster

mit der kleinen Abweichung, dass die Typangaben zu den Komponenten gerückt sind: $(a : Nat, b : Nat)$ statt $(a, b) : Nat * Nat$.

Funktionen auf Paaren haben gegenüber mehrparametrigen Funktionen den Vorteil, dass der aktuelle Parameter nicht syntaktisch ein Paar sein muss, sondern ein Ausdruck sein kann, der zu einem Paar auswertet. Nehmen wir zum Beispiel an, dass wir ein Paar von Zahlen nicht auf- sondern absteigend ordnen wollen. Das lässt sich unter anderem bewerkstelligen, indem wir das von *sort2* geordnete Paar umdrehen: $swap\ (sort2\ (e_1, e_2))$ wobei *swap* wie folgt definiert ist.

$$\textbf{\textit{let}}\ swap\ (a : Nat, b : Nat) : Nat * Nat = (b, a)$$

Wäre *swap* eine mehrparametrige Funktion, müssten wir etwas länglicher formulieren: $\textbf{\textit{let}}\ (min, max) = sort2\ (e_1, e_2)\ \textbf{\textit{in}}\ swap\ (min, max)$.

4.1.3. Records

Bei Paaren und allgemein bei Tupeln spielt die Reihenfolge der Komponenten eine Rolle: $(12, 1, 2006)$ ist etwas anderes als $(1, 12, 2006)$; (`"Stefan"`, `"Thomas"`) unterscheidet sich von (`"Thomas"`, `"Stefan"`). Beide Beispiele machen deutlich, dass die *Rolle* einer Komponente nur implizit festgelegt ist: Programmierkonvention.

Wir erlauben, die Rolle auch explizit zu machen, indem man die Komponenten benennt: $\{day = 12; month = 1; year = 2006\}$ und $\{forename =$ `"Stefan"`$; surname =$ `"Thomas"`$\}$. Die sogenannten *Labels* machen deutlich, dass 12 der Tag ist und nicht der Monat bzw. `"Stefan"` der Vor- und nicht der Nachname. Weiterhin bezeichnen $\{month = 1; day = 12; year = 2006\}$ bzw. $\{surname =$ `"Thomas"`$; forename =$ `"Stefan"`$\}$ die gleichen Werte; das heißt, die Reihenfolge, in der die benannten Komponenten aufgeschrieben werden, ist irrelevant. Mit Hilfe der Labels können Komponenten auch extrahiert werden: $date.year$ oder $person.surname$. Tupel mit benannten Komponenten heißen *Records*.

Bevor Records verwendet werden können, müssen sie zunächst mit einer sogenannten *Typdefinition* bekannt gemacht werden.

$$\textbf{\textit{type}}\ Date\ \ = \{day : Nat; month : Nat; year : Nat\}$$
$$\textbf{\textit{type}}\ Name = \{forename : String; surname : String\}$$

So wie eine Wertedefinition einen Bezeichner für einen Wert einführt, so führt eine Typdefinition einen Bezeichner für einen Typ ein — eine neue Schublade, in die wir Werte stecken können. Dieser Typname kann in anderen Typdefinitionen sowie in Typangaben verwendet werden. Im Unterschied zu einer Wertedefinition hat die rechte Seite der Typdefinition allerdings *keine* eigenständige Bedeutung, sie kann insbesondere *nicht* in Typangaben verwendet werden: $Date \to Nat$ ist ein gültiger Typ, *nicht* aber $\{day : Nat; month : Nat; year : Nat\} \to Nat$.

Neben dem Namen für den Typ selbst, $Date$ und $Name$, führt eine Typdefinition Namen ein, um Komponenten eines Records zu extrahieren: day, $month$, $year$, $forename$ und $surname$. Diese Bezeichner heißen wie gesagt *Recordlabels* oder kurz *Labels*. Die zwei obigen Definitionen führen somit zusammen sieben neue Bezeichner ein.

Abstrakte Syntax Wie auch bei Paaren beschränken wir uns auf den binären Fall und formalisieren nur Records mit exakt 2 Komponenten; alle Konstrukte verallgemeinern sich in naheliegender Weise auf Records mit n Komponenten.

Ein Recordtyp wird durch eine Definition eingeführt.

$T \in \mathsf{TyId}$	*Typbezeichner*
$\ell \in \mathsf{Lab}$	*Labels*
$d ::= \cdots$	*Deklarationen:*
$\mid$ ***type*** $T = \{\ell_1 : t_1; \ell_2 : t_2\}$	Recordtypdefinition ($\ell_1 \neq \ell_2$)

Der Bezeichner T wird durch die Definition neu eingeführt, ebenso die Labels ℓ_1 und ℓ_2. Die beiden Labels müssen verschieden sein: $\ell_1 \neq \ell_2$.

Die Kategorie der Typen wird um Typbezeichner erweitert. (Im Prinzip sind auch *Bool* und *Nat* Bezeichner für Typen.)

$t ::= \cdots$	*Typen:*
$\mid T$	Typbezeichner

Für's Erste stehen Typbezeichner für Recordtypen; im Laufe der Vorlesung werden weitere „Sorten" von Typen hinzukommen: Variantentypen, Schnittstellentypen usw.

Wir erweitern Ausdrücke um Sprachkonstrukte, die Records konstruieren und analysieren.

$e ::= \cdots$	*Recordausdrücke:*
$\mid \{\ell_1 = e_1; \ell_2 = e_2\}$	Konstruktion ($\ell_1 \neq \ell_2$)
$\mid e.\ell$	Projektion\Extraktion

Die Ausdrücke e_1 und e_2 heißen *Recordkomponenten* oder kurz *Komponenten*.

Da ein Label für sich alleine *kein* Ausdruck ist, kommen sich Labels und Bezeichner übrigens nicht ins Gehege. Die gleiche Abfolge von Buchstaben kann aus diesem Grund gleichzeitig als Bezeichner und als Label verwendet werden: ***let*** *year* = *date*.*year*. Das erste Vorkommen von *year* ist ein Bezeichner, das zweite ein Label. Ähnliches gilt für den Ausdruck $\{day = day; month = month; year = year\}$: Das erste Vorkommen ist jeweils ein Label, das zweite ein Bezeichner.

Statische Semantik: Vorüberlegungen⋆ Bei der Besprechung von Wertedefinitionen haben wir gesehen, dass der gleiche Bezeichner für verschiedene Werte verwendet werden kann: ***let*** s = *false* ***let*** $s = 4711$. Die zweite Definition verschattet die erste. Das gleiche Phänomen kann auch bei Typdefinitionen auftreten: ***type*** *Oh* = $\{je : Bool\}$ ***type*** *Oh* = $\{je : Nat\}$. Auch hier verschattet die zweite Definition die erste. Im Unterschied zu Wertedefinitionen können wir das nicht auf die leichte Schulter nehmen. Warum? Nun, Typen bringen Ordnung in die Welt der Werte. Wenn wir Unordnung in der Welt der Typen stiften, indem wir den gleichen Namen für unterschiedliche Recordtypen verwenden, dann bricht Chaos aus. In unserem Beispiel: Der Typbezeichner *Oh* kann — und wird in der Regel — in Typangaben vorkommen. Zum Beispiel:

$$\textbf{\textit{let}}\ \mathit{na\text{-}und}\ (\mathit{oh} : \mathit{Oh}) : \mathit{Bool} = \mathit{not}\ (\mathit{oh}.\mathit{je})$$

Die Funktion erhält den Typ $\mathit{Oh} \to \mathit{Bool}$. Wird der Typ Oh nach der Definition von *na-und* redefiniert, dann stimmt der Typ von *na-und* nicht mehr; der neue Typ und der alte Typ müssen ja nichts miteinander zu tun haben. Das folgende, vollständige Beispiel illustriert, was schieflaufen kann.

$$\begin{array}{l}\textbf{\textit{type}}\ \mathit{Oh} = \{\mathit{je} : \mathit{Bool}\}\\ \textbf{\textit{let}}\ \mathit{na\text{-}und}\ (\mathit{oh} : \mathit{Oh}) : \mathit{Bool} = \mathit{not}\ (\mathit{oh}.\mathit{je})\\ \textbf{\textit{type}}\ \mathit{Oh} = \{\mathit{je} : \mathit{Nat}\ \}\\ \textbf{\textit{let}}\ \mathit{egal} = \mathit{na\text{-}und}\ \{\mathit{je} = 4711\}\end{array}$$

Wir schmuggeln eine natürliche Zahl an eine Stelle, an der ein Boolescher Wert erwartet wird und das Unglück nimmt seinen Lauf.

Was ist zu tun? Die Lösung ist so einfach, wie einschränkend: Wir erlauben es *nicht* Typen zu redefinieren. Aus einem ähnlichen Grund lassen wir auch keine lokalen Typdefinitionen zu. In einem Mini–F# Modul dürfen Typdefinition nur auf dem „top-level“ des Moduls eingeführt werden, *nicht* aber lokal mit ***in***-Ausdrücken.

Statische Semantik Die folgenden Typregeln setzen voraus, dass die Typdefinition

$$\textbf{\textit{type}}\ T = \{\ell_1 : t_1; \ell_2 : t_2\}$$

bekannt ist. Die Regeln unterscheiden sich nicht wesentlich von den korrespondierenden Regeln für Paare: an die Stelle des anonymen Typs $t_1 * t_2$ tritt der benannte Typ T.

$$\frac{\Sigma \vdash e_1 : t_1 \qquad \Sigma \vdash e_2 : t_2}{\Sigma \vdash \{\ell_1 = e_1; \ell_2 = e_2\} : T} \qquad \frac{\Sigma \vdash e_1 : t_1 \qquad \Sigma \vdash e_2 : t_2}{\Sigma \vdash \{\ell_2 = e_2; \ell_1 = e_1\} : T}$$

$$\frac{\Sigma \vdash e : T}{\Sigma \vdash e.\ell_i : t_i} \quad i \in \{1, 2\}$$

Bei der Konstruktion eines Records müssen stets alle Komponenten angegeben werden; Komponenten dürfen aber nicht doppelt aufgeführt werden: $\ell_1 \neq \ell_2$. Auf diese Weise wird sichergestellt, dass Projektionen stets wohldefiniert sind. Der Ausdruck $\{\mathit{day} = 30\}.\mathit{year}$ ist zum Beispiel nicht wohlgetypt. Es gibt zwei Regeln für die Konstruktion eines Records, da die Reihenfolge, in der die beiden Komponenten aufgeführt werden, keine Rolle spielt.

Ein Label ist einer Funktion nicht unähnlich. Der Typ nach dem Label korrespondiert zum Ergebnistyp, der deklarierte Recordtyp korrespondiert zum Argumenttyp: year hat im Prinzip den Typ $\mathit{Date} \to \mathit{Nat}$ und $\mathit{surname}$ den Typ $\mathit{Name} \to \mathit{String}$. Im Unterschied zu einer Funktion hat ein Label aber keine Definition; es steht sozusagen für sich selbst. An die Stelle der Funktionsanwendung tritt die Punktnotation: $\mathit{date}.\mathit{year}$ oder $\mathit{person}.\mathit{surname}$.

Dynamische Semantik Typdefinitionen können in der dynamischen Semantik ignoriert werden.

Wir erweitern den Bereich der Werte um Records, deren Komponenten Werte sind.

$\nu ::= \cdots$	*Werte:*
$\mid \{\ell_1 = \nu_1; \ell_2 = \nu_2\}$	Records ($\ell_1 \neq \ell_2$)

Die Auswertungsregeln korrespondieren zu den Regeln für Paare, nur dass an die Stelle der Projektionsfunktionen *fst* und *snd* die Punktnotation tritt.

$$\frac{\delta \vdash e_1 \Downarrow \nu_1 \qquad \delta \vdash e_2 \Downarrow \nu_2}{\delta \vdash \{\ell_1 = e_1; \ell_2 = e_2\} \Downarrow \{\ell_1 = \nu_1; \ell_2 = \nu_2\}}$$

$$\frac{\delta \vdash e_1 \Downarrow \nu_1 \qquad \delta \vdash e_2 \Downarrow \nu_2}{\delta \vdash \{\ell_2 = e_2; \ell_1 = e_1\} \Downarrow \{\ell_1 = \nu_1; \ell_2 = \nu_2\}} \qquad \frac{\delta \vdash e \Downarrow \{\ell_1 = \nu_1; \ell_2 = \nu_2\}}{\delta \vdash e.\ell_i \Downarrow \nu_i}$$

Vertiefung: Ganze Zahlen Mini–F# bietet von Haus aus keine ganzen Zahlen an. Um mit ganzen Zahlen rechnen zu können, müssen wir sie implementieren. (Natürliche Zahlen haben wir mit Hilfe der Metasprache eingeführt; ganze Zahlen werden mit Hilfe der Objektsprache definiert. Beim ersten Ansatz hat die Sprachdesigner*in die Arbeit, beim zweiten die Programmierer*in.)

Wir können eine ganze Zahl als Differenz zweier natürlicher Zahlen repräsentieren: -4 wird zum Beispiel durch $0-4$, $5-9$ oder $4711-4715$ dargestellt; $+4$ entsprechend durch $4-0$, $9-5$ oder $4715-4711$. Wir stellen also eine ganze Zahl durch *zwei* natürliche Zahlen dar, ein Fall für Recordtypen.

$$\textbf{\textit{type}}\ \mathit{Int} = \{\mathit{pos} : \mathit{Nat}; \mathit{neg} : \mathit{Nat}\}$$

Die Bedeutung von $\{\mathit{pos} = p; \mathit{neg} = n\}$ ist $p - n$, wobei '$-$' die *mathematische* Subtraktion auf den ganzen Zahlen meint. Kurz: $\{\mathit{pos} = p; \mathit{neg} = n\}$ ist Syntax, $p - n$ ist Semantik. Da jede ganze Zahl verschiedene Repräsentationen besitzt, haben wir es mit einer *redundanten* Zahlendarstellung zu tun. (Wenn führende Nullen zugelassen werden, ist auch das uns vertraute Dezimalsystem redundant: Die Ziffernfolgen `815`, `0815` und `00815` repräsentieren jeweils die gleiche Zahl, nämlich 815.)

Wie können wir mit ganzen Zahlen rechnen? Wir müssen entsprechende Rechenregeln aufstellen, indem wir die arithmetischen Operationen und Vergleichsoperationen für ganze Zahlen zurückführen auf Operationen über den natürlichen Zahlen. Fangen wir mit der Addition an:

$$\begin{aligned}&\textbf{\textit{let}}\ \mathit{add}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Int} = \\ &\quad \{\mathit{pos} = i.\mathit{pos} + j.\mathit{pos}; \mathit{neg} = i.\mathit{neg} + j.\mathit{neg}\}\end{aligned}$$

Die Bedeutung von $\mathit{add}\ (i, j)$ ist

$$(i.\mathit{pos} - i.\mathit{neg}) + (j.\mathit{pos} - j.\mathit{neg}) = (i.\mathit{pos} + j.\mathit{pos}) - (i.\mathit{neg} + j.\mathit{neg})$$

Durch Umordnen und Zusammenfassen positiver wie negativer Komponenten erhalten wir das gewünschte Ergebnis. Auf ähnliche Art und Weise lässt sich auch die Multiplikation implementieren (zur Übung). Interessanter wird es, wenn wir Operationen betrachten, die auf den natürlichen Zahlen nicht unterstützt werden: Negation und Subtraktion. Die Implementierung der Negation ist verblüffend einfach: Da

$$-(i.pos - i.neg) = i.neg - i.pos$$

müssen wir lediglich die Komponenten vertauschen.

$$\begin{array}{l}\textbf{\textit{let}}\ negate\ (i : Int) : Int = \\ \quad \{pos = i.neg; neg = i.pos\} \\ \textbf{\textit{let}}\ sub\ (i : Int, j : Int) : Int = \\ \quad \{pos = i.pos + j.neg; neg = i.neg + j.pos\}\end{array}$$

Wann sind zwei ganze Zahlen gleich? Da wir ein redundantes Zahlensystem vor uns haben, müssen wir etwas aufpassen. (Wäre die Darstellung eindeutig, würde gelten: Zwei Zahlen sind gleich, wenn sie gleich aussehen, $i = j \iff i.pos = j.pos \wedge i.neg = j.neg$. Kurz: zwei Zahlen sind semantisch gleich, wenn sie syntaktisch gleich sind.) Die Bedeutung von i ist $i.pos - i.neg$; die Bedeutung von j entsprechend $j.pos - j.neg$. Die semantischen Werte müssen gleich sein:

$$i.pos - i.neg = j.pos - j.neg \iff i.pos + j.neg = i.neg + j.pos$$

Somit erhalten wir:

$$\begin{array}{l}\textbf{\textit{let}}\ equal\ (i : Int, j : Int) : Bool = \\ \quad i.pos + j.neg = i.neg + j.pos\end{array}$$

Die anderen Vergleichsoperationen lassen sich analog implementieren (zur Übung).

Jetzt da Mini–F# über zwei Zahlentypen verfügt, stellt sich die Frage, wie wir Elemente des einen Typs in den anderen Typ überführen (im Fachjargon: Typkonversion). Eine natürliche Zahl in eine ganze zu überführen, ist einfach:

$$\begin{array}{l}\textbf{\textit{let}}\ int\ (n : Nat) : Int = \\ \quad \{pos = n; neg = 0\}\end{array}$$

Die umgekehrte Richtung geht mit einem Informationsverlust einher: Die ganze Zahl i wird auf ihren Betrag $|i|$ abgebildet.

$$\begin{array}{l}\textbf{\textit{let}}\ abs\ (i : Int) : Nat = \\ \quad \textbf{\textit{if}}\ i.pos \leqslant i.neg\ \textbf{\textit{then}}\ i.neg \dot{-} i.pos \\ \qquad\qquad\qquad\ \textbf{\textit{else}}\ \ i.pos \dot{-} i.neg\end{array}$$

Etwas kürzer können wir formulieren:

$$\begin{array}{l}\textbf{\textit{let}}\ abs\ (i : Int) : Nat = \\ \quad (i.neg \dot{-} i.pos) + (i.pos \dot{-} i.neg)\end{array}$$

Die rechte Seite regt zum Nachdenken an: gilt nicht $(i.neg \dot{-} i.pos) + (i.pos \dot{-} i.neg) = 0$? Weit gefehlt! Das Symbol '$\dot{-}$' ist Syntax für die Subtraktion natürlicher Zahlen („monus“ nicht „minus“) und es gilt: $(a \dot{-} b) + (b \dot{-} a) = |a - b|$.

Sei $\|P\|$ der Flächeninhalt einer Menge von Punkten P.[a] Flächenberechnungen basieren auf zwei grundlegenden Annahmen: (1) Die Fläche des Einheitsquadrates ist 1; (2) die Fläche zweier *disjunkter* Punktmengen entspricht der Summe der Einzelflächen: $\|A \cup B\| = \|A\| + \|B\|$, falls $A \cap B = \emptyset$. Wenn die Mengen nicht disjunkt sind, müssen wir sie entsprechend in disjunkte Teilmengen aufteilen, zum Beispiel mit Hilfe einer der folgenden Eigenschaften:

$$\begin{aligned} A &= (A \setminus B) \cup (A \cap B) \\ B &= (B \setminus A) \cup (A \cap B) \\ A \cup B &= (A \setminus B) \cup (A \cap B) \cup (B \setminus A) \end{aligned}$$

Die Mengen auf den rechten Seite sind jeweils disjunkt, somit gilt:

$$\|A\| = \|A \setminus B\| + \|A \cap B\| \tag{4.1a}$$
$$\|B\| = \|B \setminus A\| + \|A \cap B\| \tag{4.1b}$$
$$\|A \cup B\| = \|A \setminus B\| + \|A \cap B\| + \|B \setminus A\| \tag{4.1c}$$

Die Formel für $\|A \cup B\|$ verwendet die *Mengendifferenz* auf der rechten Seite; diese Mengenoperation lässt sich vermeiden, wie die folgende Rechnung zeigt :

$$\begin{aligned} & \|A\| + \|B\| \\ = & \quad \{ \text{ siehe oben (4.1a) und (4.1b) } \} \\ & \|A \setminus B\| + \|A \cap B\| + \|B \setminus A\| + \|A \cap B\| \\ = & \quad \{ \text{ siehe oben (4.1c) } \} \\ & \|A \cup B\| + \|A \cap B\| \end{aligned}$$

Durch Umstellen erhalten wir

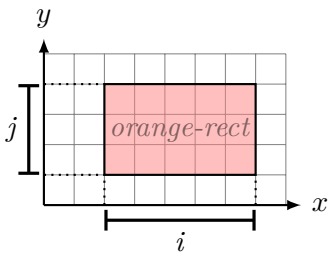
$$\|A \cup B\| = \|A\| + \|B\| - \|A \cap B\|$$

Abbildung 4.1.: Berechnung von Flächeninhalten.

[a] Einer *beliebigen* Punktmenge eine Fläche zuzuordnen, ist tatsächlich ein unlösbares mathematisches Problem. Wenn Sie mathematisch interessiert sind, erfahren Sie dazu mehr in der Maßtheorie. Wir ignorieren diese Probleme grundsätzlicher Natur, sind aber auch auf der sicheren Seite, da unsere Punktmengen bzw. Flächen sehr einfach gestrickt sind.

module
Datatypes.
Area

Vertiefung: Wohnfläche Kommen wir noch einmal auf ein Problem zurück, das wir bereits in Abschnitt 3.3 bearbeitet haben: die Berechnung einer Wohnfläche. Abstrakt gesehen gilt es, die Gesamtfläche zweier gegebener Rechtecke zu berechnen — in der ursprünglichen Aufgabenstellung ist von Quadraten die Rede, aber das ist eine unnötige Einschränkung. (Abbildung 4.1 enthält ein paar grundlegende Überlegungen zur Berechnung von Flächeninhalten.) Jetzt da wir die Möglichkeit haben, Daten zu aggregieren, verschiedene Daten zu einer Einheit zusammenzufassen, können wir die Aufgabe professioneller angehen. Erinnern wir uns: Ein Rechteck ist gegeben durch zwei Intervalle, einem Abschnitt der x-Achse und einem Abschnitt der y-Achse.




```
let orange-rect =
  let i = {lo = 2; hi = 7}
  let j = {lo = 1; hi = 4}
  {x = i; y = j}
```

Im folgenden führen wir die notwendigen Typdefinitionen ein, so dass das orangene Rechteck durch die Wertedefinition auf der rechten Seite repräsentiert werden kann. (Die Intervalldarstellung ist natürlich nicht die einzig denkbare Repräsentation von Rechtecken. Welche Alternativen gibt es? Welche Vor- und Nachteile haben die verschiedenen Darstellungen? Wenn Sie Lust haben, lösen Sie die Aufgabe ein zweites Mal unter Verwendung Ihrer favorisierten Darstellung.)

Wenden wir uns zunächst der Darstellung von Intervallen zu. Ein Intervall wird durch die beiden Intervallgrenzen festgelegt.

```
type Interval = {lo : Nat; hi : Nat}          // low und high
```

Die Länge eines Intervalls ergibt sich als Differenz der Intervallgrenzen.

```
let length (i : Interval) : Nat = i.hi ∸ i.lo
```

Da Längen nicht-negativ sind, ist die natürliche Subtraktion „∸“ die richtige Wahl. Würden wir zum Beispiel ganze Zahlen als Intervallgrenzen verwenden, müssten wir „monus“ nachprogrammieren und die Länge als $max\ 0\ (i.hi - i.lo)$ definieren.

Jetzt können wir die 1-dimensionale Variante unseres ursprünglichen Problems lösen und die Gesamtlänge zweier Intervalle berechnen. Überlappen sich die Intervalle nicht, ist die Gesamtlänge die Summe der Einzellängen. Im Fall einer Überschneidung müssen wir von der Summe die Länge der Überlappung abziehen. Mathematisch betrachtet entspricht die Überlappung einem Mengendurchschnitt.

Bildet man den Durchschnitt zweier Intervalle, erhält man wieder ein Intervall (das klappt bei der Vereinigung von Intervallen nicht).

```
let intersection (i : Interval, j : Interval) : Interval =
  {lo = max i.lo j.lo; hi = min i.hi j.hi}
```

Es ist nützlich sich zu überlegen, welche Eigenschaften eine Funktion hat. Die Funktion *intersection* ist zum Beispiel kommutativ, *intersection* (i, j) = *intersection* (j, i), da sowohl *min* als auch *max* kommutativ sind. Gibt es noch weitere merkenswerte oder bemerkenswerte Eigenschaften?

Jetzt haben wir das Vokabular zusammen, um die umgangssprachliche Lösung in Mini–F# zu transliterieren: Die Gesamtlänge zweier Intervalle ist die Summe der Einzellängen minus der Länge des Durchschnitts.

```
let length2 (i : Interval, j : Interval) =
    length i + length j - length (intersection (i, j))
```

Um unser usprüngliches Problem zu lösen, müssen wir die 1-dimensionalen Konzepte im Wesentlichen in den 2-dimensionalen Raum übertragen. Fangen wir mit der Darstellung von Rechtecken an: Ein Rechteck wird durch zwei Intervalle festgelegt.

```
type Rectangle = { x : Interval; y : Interval }
```

Der Flächeninhalt eines Rechtecks ergibt sich als Produkt der Intervalllängen.

```
let area (r : Rectangle) = length r.x * length r.y
```

Analog zum 1-dimensionalen Fall definieren wir den Durchschnitt zweier Rechtecke.

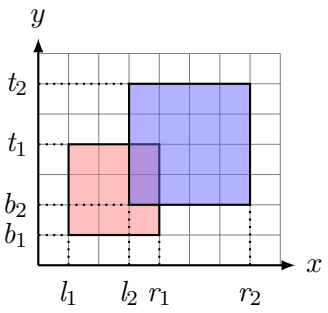
```
let Intersection (r : Rectangle, s : Rectangle) =
    { x = intersection (r.x, s.x);
      y = intersection (r.y, s.y) }
```

Der Durchschnitt wird koordinatenweise berechnet und stützt sich auf die entsprechende Funktion für Intervalle ab. Somit „erbt“ *Intersection* die Eigenschaften ihrer kleinen Schwester, zum Beispiel, *Intersection* (r, s) = *Intersection* (s, r) — die Funktion ist kommutativ. Somit erhalten wir für die Gesamtfläche zweier Rechtecke:

```
let area2 (r : Rectangle, s : Rectangle) =
    area r + area s - area (Intersection (r, s))
```

Wie im 1-dimensionalen Fall transliterieren wir die umgangssprachliche Formulierung in Mini–F#. Man erkennt die Fortschritte, die wir erzielt haben, wenn man das obige Programm mit dem Code aus Abschnitt 3.3 vergleicht. Letzterer löst eine spezielle Probleminstanz; der Code besteht aus einem monolithischen Block. Im Gegensatz dazu löst das obige Programm das Problem im Allgemeinen; es ist modular aufgebaut und besteht aus mehreren, kleinen Bausteinen. Der kompositionale Aufbau bringt viele Vorteile mit sich: Jeder Baustein, jede Typdefinition und jede Funktion, kann isoliert betrachtet, verstanden, getestet und verifiziert werden. Je größer ein Programm, je schwieriger ein Problem, je größer ein Softwaresystem, je umfangreicher die Anforderungen, desto wichtiger wird ein modularer Aufbau.

Schauen wir uns das Programm bzw. die Funktionen in Aktion an. Der Grundriss aus Abschnitt 3.3 wird durch zwei Rechtecke beschrieben.



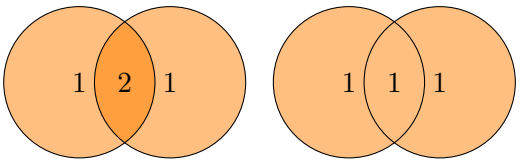

```
let shift (d : Nat, i : Interval) =
  {lo = i.lo + d; hi = i.hi + d}
let red-rect =
  let i = {lo = 1; hi = 4}
  {x = i; y = i}
let blue-rect =
  let i = {lo = 2; hi = 6}
  {x = shift (1, i); y = i}
```

Der Durchschnitt der Quadrate ergibt das kleine, violette Rechteck in der Mitte.

```
Mini⟩ Intersection (red-rect, blue-rect)
{x = {lo = 3; hi = 4}; y = {lo = 2; hi = 4}}
Mini⟩ area2 (red-rect, blue-rect)
23
Mini⟩ area2 (blue-rect, red-rect)
23
```

Es ist beruhigend, dass es keine Rolle spielt, in welcher Reihenfolge wir die Rechtecke an *area2* übergeben. Vielleicht erinnern Sie sich: Der erste Ansatz aus Abschnitt 3.3 funktionierte für eine Reihenfolge, versagte aber bei der anderen. Es gilt allgemein: *area2* (r, s) = *area2* (s, r). Mit anderen Worten, auch *area2* ist kommutativ, eine Eigenschaft, die sich schnell nachweisen lässt. (Versuchen Sie es.)

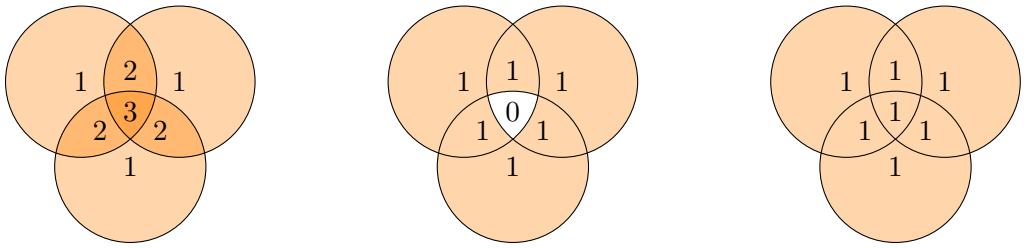
Werden wir etwas ambitionierter: Wie können wir die Gesamtfläche von *drei* Rechtecken bestimmen? Machen wir uns noch einmal klar, wie wir im Fall von zwei Rechtecken vorgegangen sind. Unserem Programm liegt das *Prinzip der Einschließung und Ausschließung* zugrunde (Inklusion und Exklusion). Wenn wir die Fläche von $A \cup B$ berechnen, indem wir die Summe der Einzelflächen addieren, dann wird die Schnittfläche doppelt gezählt, muss also wieder abgezogen werden.



$\|A\| + \|B\|$ $\quad\quad$ $\ldots - \|A \cap B\|$

Wenn wir drei Flächen addieren, dann werden die Schnitte von zwei Flächen doppelt,

der Schnitt von allen drei Flächen wird dreifach gezählt.



$\|A\| + \|B\| + \|C\| \quad \ldots - \|A \cap B\| - \|A \cap C\| - \|B \cap C\| \quad \ldots + \|A \cap B \cap C\|$

Wenn wir die doppelt gezählten Flächen wieder abziehen, geht aber die gemeinsame Schnittfläche verloren; diese müssen wir dann wieder hinzufügen. Man sieht: Die Flächen werden alternierend ein- und ausgeschlossen, daher der Name des Prinzips. Damit ergibt sich das folgende Mini–F# Programm.

```
let Intersection3 (r : Rectangle, s : Rectangle, t : Rectangle) =
  Intersection (r, Intersection (s, t))
let area3 (r : Rectangle, s : Rectangle, t : Rectangle) =
  area r + area s + area t
  - area (Intersection (r, s)) - area (Intersection (r, t)) - area (Intersection (s, t))
  + area (Intersection3 (r, s, t))
```

Das Programm ist schon etwas umfangreicher. Das Prinzip der Einschließung und Ausschließung lässt sich auch anwenden, um die Gesamtfläche von vier oder mehr Rechtecken auszurechnen. Aber ist das auch eine gute Idee? (Abbildung 4.2 erklärt das Prinzip noch einmal aus dem Blickwinkel der Kombinatorik.)

4.2. Varianten

module
Datatypes.
Person

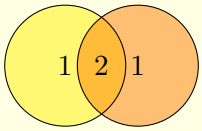
Eine Person ist entweder weiblich oder männlich[2]; eine männliche Person hat Attribute, die eine weibliche nicht hat (und vielleicht umgekehrt, das modellieren wir aber nicht).

```
type Woman = {name : String}
type Man   = {name : String; bald : Bool}
```

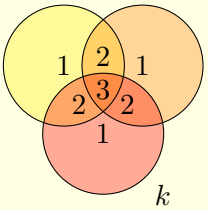
Daten, die unterschiedliche Ausprägungen besitzen, können wir in unserer Programmiersprache mit sogenannten *Varianten* modellieren. Ein einfacher Variantentyp für Personen ist zum Beispiel

[2] Erinnern wir uns: Informatiker*innen bilden abstrakte Modelle der konkreten Wirklichkeit. Die Wirklichkeitstreue dieser Modelle macht den Reiz und die Verantwortung bei der Software-Entwicklung aus. Dieses Modell dient als einführendes Beispiel und ist aus diesem Grund bewusst einfach gehalten, auf Kosten der Wirklichkeitstreue. Auch das Recht konstruiert Modelle der Wirklichkeit. Ende 2018 wurde im Zuge der Reform des Personenstandsgesetzes die zusätzliche Geschlechtsoption „divers“ geschaffen, die Identitäten jenseits des binären Geschlechtermodells abdecken soll.

Etwas Kombinatorik gefällig? Gegeben seien n nicht notwendigerweise disjunkte Flächen A_1, …, A_n. Wir wollen die Gesamtfläche $\|A_1 \cup \cdots \cup A_n\|$ bestimmen. (Um die folgenden Formeln kompakt zu halten, verwenden wir zum Beispiel $\|134\|$ als Abkürzung für die Schnittfläche $\|A_1 \cap A_3 \cap A_4\|$.) Ein systematischer Ansatz zur Bestimmung der Gesamtfläche beruht auf dem *Prinzip der Einschließung und Ausschließung*, nachfolgend illustriert für die beiden einfachsten, nicht-trivialen Probleminstanzen.



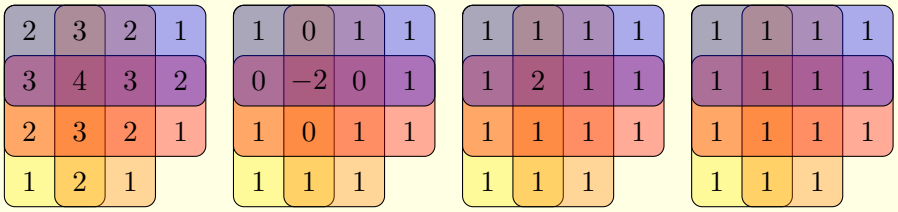
	k	1	2
$\|1\| + \|2\|$		$+1$	$+2$
$- \|12\|$			-1



	k	1	2	3
$\|1\| + \|2\| + \|3\|$		$+1$	$+2$	$+3$
$- \|12\| - \|13\| - \|23\|$			-1	-3
$+ \|123\|$				$+1$

Betrachten wir eine Teilfläche, die in *genau* k der Flächen $A_1, \ldots, A_n$ enthalten ist. Ihr Beitrag zur Gesamtfläche ist jeweils in der k-ten Spalte aufgeführt. Ist zum Beispiel $n = 3$ und $k = 2$, dann wird die Fläche doppelt in $\|1\| + \|2\| + \|3\|$ und einfach in $\|12\| + \|13\| + \|23\|$ gezählt. Addiert man die k-te Spalte auf, erkennt man dass diese Fläche summa summarum genau einmal gezählt wird.

Für $n = 4$ ergibt sich das folgende Bild — die Illustration der Überschneidungen mit Hilfe von Venn-Diagrammen stößt langsam aber sicher an ihre Grenzen.



	k	1	2	3	4
$\|1\| + \|2\| + \|3\| + \|3\|$		$+1$	$+2$	$+3$	$+4$
$- \|12\| - \|13\| - \|14\| - \|23\| - \|24\| - \|34\|$			-1	-3	-6
$+ \|123\| + \|124\| + \|134\| + \|234\|$				$+1$	$+4$
$- \|1234\|$					-1

Die Diagramme, von links nach rechts gelesen, illustrieren wie oft eine Fläche gezählt wird, wenn die Flächenformel von oben nach unten abgearbeitet wird. Bleibt zu klären, warum die Summe jeder Spalte 1 ergibt. Die Einträge sind sogenannte Binomialkoeffizienten (warum?); die Summe alternierender Koeffizienten ist 0, eine Konsequenz aus dem Binomischen Satz (die Formel berechnet 1 *minus* der Summe der Spalte):

$$(-1)^0 \binom{k}{0} + (-1)^1 \binom{k}{1} + (-1)^1 \binom{k}{2} + \cdots + (-1)^k \binom{k}{k} = (1-1)^k = 0$$

Abbildung 4.2.: Das Prinzip der Einschließung und Ausschließung.

```
type Person =
  | Female of Woman
  | Male   of Man
```

Variantentypen ähneln der Notation, mit der wir die abstrakte Syntax unserer Programmiersprache beschreiben. Diese Ähnlichkeit ist nicht zufällig. Die abstrakte Syntax beschreibt Alternativen: Ein Ausdruck ist *entweder* eine Boolesche Konstante *oder* eine Alternative *oder* eine Funktionsabstraktion *oder* eine Funktionsapplikation usw.

Zurück zu unserem Beispiel: Die Deklaration legt fest, dass eine Person *entweder* weiblich *oder* männlich ist; beide haben einen Namen, bei männlichen Personen wird zusätzlich der Zustand des Haupthaares modelliert.

Ähnlich wie Records müssen auch Varianten mit einer *Typdefinition* bekannt gemacht werden. Eine Variantentypdefinition führt zwei Arten von Bezeichnern ein. Zunächst wird nach dem Schlüsselwort **type** ein Typbezeichner festgelegt, in unserem Beispiel *Person*. Dieser Typname kann in anderen Typdefinitionen sowie in Typangaben verwendet werden. Auf der rechten Seite werden dann Bezeichner für die Elemente des neu definierten Variantentyps eingeführt, in unserem Beispiel *Female* und *Male*. Diese Bezeichner heißen auch *Datenkonstruktoren* oder kurz *Konstruktoren*, da mit ihrer Hilfe Elemente des Variantentyps konstruiert werden: *Female* angewendet auf ein Element vom Typ *Woman* ist ein Element vom Typ *Person*; analog für *Male*. Hier sind zwei Beispielausdrücke vom Typ *Person*.

```
Female {name = "Lisa"}
Male {name = "Florian"; bald = false}
```

Wie immer können wir Ausdrücke auch an Bezeichner binden.

```
let ralf    = Male   {name = "Ralf";   bald = true}
let melanie = Female {name = "Melanie"}
let julia   = Female {name = "Julia"}
let andres  = Male   {name = "Andres"; bald = false}
```

Ein Variantentyp beschreibt Alternativen; mit Hilfe der *Fallunterscheidung* **match** können wir in einem Programm feststellen, welche konkrete Alternative vorliegt. Das folgende Programm bestimmt zum Beispiel in Abhängigkeit vom Geschlecht eine Anrede.

```
let dear (person : Person) : String =
  match person with
    | Female female → "Liebe " ^ female.name
    | Male   male   → (if male.bald
                         then "Lieber glatzköpfiger "
                         else "Lieber ") ^ male.name
```

Nach dem Schlüsselwort **match** steht der Ausdruck, der analysiert werden soll; die Zweige der Fallunterscheidung führen die verschiedenen Fälle des Variantentyps auf. Wertet

etwa der Parameter *person* zu $\mathit{Female}\ \{\mathit{name} = \texttt{"Lisa"}\}$ aus, dann wird mit der Auswertung des ersten Zweigs fortgefahren: *female* wird an $\{\mathit{name} = \texttt{"Lisa"}\}$ gebunden und anschließend der Ausdruck nach dem Pfeil ausgerechnet.

$$
\begin{aligned}
& \mathit{dear}\ (\mathit{Female}\ \{\mathit{name} = \texttt{"Lisa"}\}) \\
= \quad & \{\ \text{Definition von } \mathit{dear}\ \} \\
& \textbf{\textit{match}}\ \mathit{Female}\ \{\mathit{name} = \texttt{"Lisa"}\}\ \textbf{\textit{with}}\ \cdots \\
= \quad & \{\ \text{Auswertung der Fallunterscheidung}\ \} \\
& \texttt{"Liebe "} \mathbin{\texttt{\^{}}} \texttt{"Lisa"} \\
= \quad & \{\ \text{Definition der Konkatenation}\ \} \\
& \texttt{"Liebe Lisa"}
\end{aligned}
$$

Die Fallunterscheidung ist der Alternative $\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3$ sehr ähnlich; wir werden später sehen, dass die Fallunterscheidung die Alternative verallgemeinert.

4.2.1. Binäre Varianten

Wie auch bei den Tupeln und Records führen wir nur die binäre Version der Varianten ein; alle Konstrukte verallgemeinern sich in naheliegender Weise auf Varianten mit n Alternativen.

Abstrakte Syntax Ein Variantentyp (engl. union oder variant type) wird durch eine *Definition* eingeführt.

$C \in \mathsf{Con}$	*Datenkonstruktoren*
$d ::= \cdots$	*Deklarationen:*
$\mid\ \textbf{\textit{type}}\ T = \mid C_1\ \textbf{\textit{of}}\ t_1$	Variantentypdefinition ($C_1 \neq C_2$)
$\qquad\qquad\quad \mid C_2\ \textbf{\textit{of}}\ t_2$	

Der Bezeichner T wird durch die Definition neu eingeführt, ebenso die Konstruktoren C_1 und C_2. Die beiden Konstruktoren müssen verschieden sein: $C_1 \neq C_2$. Konstruktornamen müssen im Unterschied zu Bezeichnern mit einem *großen* Buchstaben anfangen. (Zur Erinnerung: Bezeichner können mit einem beliebigen Buchstaben anfangen.) Danach können weitere Buchstaben, kleine und große, Ziffern, und Sonderzeichen wie ein Unterstrich oder ein Apostroph folgen.

Auf Ebene der Ausdrücke führen wir Sprachkonstrukte ein, die Varianten konstruieren und analysieren.

$e ::= \cdots$	*Ausdrücke:*
$\mid\ C\ e$	Konstruktion↘Injektion
$\mid\ \textbf{\textit{match}}\ e\ \textbf{\textit{with}} \mid C_1\ x_1 \to e_1$	Fallunterscheidung ($C_1 \neq C_2$)
$\qquad\qquad\qquad\qquad \mid C_2\ x_2 \to e_2$	

Der Ausdruck e zwischen den Schlüsselwörtern ***match*** und ***with*** heißt *Diskriminatorausdruck*; $C_1\ x_1 \to e_1$ und $C_2\ x_2 \to e_2$ sind *Zweige* der Fallunterscheidung. Die Konstruktoren der beiden Zweige müssen verschieden sein: $C_1 \neq C_2$. Die einzelnen Zweige

binden Bezeichner: In $C_i\ x_i \to e_i$ wird der Bezeichner x_i an das Argument des Konstruktors C_i gebunden und ist in e_i sichtbar.

Statische Semantik: Vorüberlegungen⋆ Wie Recordtypen dürfen auch Variantentypen weder redefiniert noch lokal definiert werden. Das folgende Beispiel, eine Adaption des entsprechenden Programms für Recordtypen, illustriert, was schieflaufen kann.

```
type Oh = | Je of Bool
let na-und (oh : Oh) : Bool = match oh with | Je b → not b
type Oh = | Je of Nat
let egal = na-und (Je 4711)
```

Wieder schmuggeln wir eine natürliche Zahl an eine Stelle, an der ein Boolescher Wert erwartet wird.

Ein ähnliches Problem tritt auf, wenn Typen lokal definiert werden und ein Typbezeichner aus dem lokalen Kontext entweicht.

```
(type Oh = | Je of Bool in fun (oh : Oh) : Bool → match oh with | Je b → not b)
  (type Oh = | Je of Nat in Je 4711)
```

Der erste Teilausdruck der Funktionsanwendung hat den Typ $Oh \to Bool$, der zweite den Typ Oh. Aber wiederum handelt es sich um zwei verschiedene Typen. Deshalb: Variantentypen dürfen *weder* redefiniert *noch* lokal definiert werden.

Statische Semantik Die folgenden Typregeln setzen voraus, dass der Variantentyp

```
type T = | C1 of t1 | C2 of t2
```

bekannt ist. Für jeden Variantentyp gibt es einen entsprechenden Satz von Regeln.

Die Regel für die Fallunterscheidung verallgemeinert im gewissen Sinne die Regel für die Alternative — in der Vertiefung kommen wir darauf noch einmal zurück.

$$\frac{\Sigma \vdash e : t_i}{\Sigma \vdash C_i\ e : T}$$

$$\frac{\Sigma \vdash e : T \qquad \Sigma, \{x_1 \mapsto t_1\} \vdash e_1 : t' \qquad \Sigma, \{x_2 \mapsto t_2\} \vdash e_2 : t'}{\Sigma \vdash (\textbf{match}\ e\ \textbf{with} \mid C_1\ x_1 \to e_1 \mid C_2\ x_2 \to e_2) : t'}$$

$$\frac{\Sigma \vdash e : T \qquad \Sigma, \{x_1 \mapsto t_1\} \vdash e_1 : t' \qquad \Sigma, \{x_2 \mapsto t_2\} \vdash e_2 : t'}{\Sigma \vdash (\textbf{match}\ e\ \textbf{with} \mid C_2\ x_2 \to e_2 \mid C_1\ x_1 \to e_1) : t'}$$

Alle Zweige der Fallunterscheidung müssen den gleichen Typ besitzen; dieser ist auch der Typ des gesamten Ausdrucks. Die einzelnen Zweige binden wie gesagt Bezeichner: In $C_i\ x_i \to e_i$ ist der Bezeichner x_i im Rumpf e_i sichtbar. Die Reihenfolge der beiden Zweige ist irrelevant.

Ein Konstruktor ist einer Funktion nicht unähnlich. Der Typ nach dem Konstruktor korrespondiert zum Argumenttyp, der deklarierte Variantentyp korrespondiert zum Ergebnistyp. Der Konstruktor *Female* hat somit im Prinzip den Typ $Woman \to Person$ und *Male* den Typ $Man \to Person$. Im Unterschied zu einer Funktion hat ein Konstruktor aber keine Definition; er steht sozusagen für sich selbst.

Dynamische Semantik Wie Recordtypdefinitionen können auch Variantentypdefinitionen in der dynamischen Semantik ignoriert werden.

Konstruktoren konstruieren Werte, entsprechend müssen wir den Bereich der Werte erweitern.

$\nu ::= \cdots$	*Werte:*
$\mid\ C\ \nu$	Konstruktion⧹Injektion in einen Variantentyp

Ein Konstruktoraufruf wird ausgerechnet, indem das Argument ausgewertet wird und der resultierende Wert mit dem Konstruktor „markiert“ wird (engl. tagging).

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash C\ e \Downarrow C\ \nu}$$

In einer Fallunterscheidung wird zunächst der Diskriminatorausdruck ausgerechnet; abhängig vom Ergebnis wird dann genau ein Zweig ausgewertet. Dabei wird der Bezeichner x_i an das Argument von C_i gebunden.

$$\frac{\delta \vdash e \Downarrow C_i\ \nu_i \qquad \delta, \{x_i \mapsto \nu_i\} \vdash e_i \Downarrow \nu}{\delta \vdash (\textbf{\textit{match}}\ e\ \textbf{\textit{with}} \mid C_1\ x_1 \to e_1 \mid C_2\ x_2 \to e_2) \Downarrow \nu}$$

$$\frac{\delta \vdash e \Downarrow C_i\ \nu_i \qquad \delta, \{x_i \mapsto \nu_i\} \vdash e_i \Downarrow \nu}{\delta \vdash (\textbf{\textit{match}}\ e\ \textbf{\textit{with}} \mid C_2\ x_2 \to e_2 \mid C_1\ x_1 \to e_1) \Downarrow \nu}$$

Bevor wir uns den Beispielprogrammen zuwenden, überlegen wir noch kurz, wie die Konstrukte und Regeln für Varianten mit n Alternativen aussehen.

Ist $n = 0$, so haben wir einen Typ ohne Alternativen, sprich einen *leeren* Typ. Die Fallunterscheidung hat entsprechend keine Zweige: ***match*** *e* ***with***. Es gibt auch keine Auswertungsregeln für die leere Fallunterscheidung — wir können ja keinen Diskriminatorausdruck e konstruieren, der den passenden Typ hat. Allgemein signalisiert die leere Fallunterscheidung *toten Code*.

```
type Empty = |
let you-cannot-call-me (x : Empty) : Nat = match x with
```

(F# erlaubt keine leeren Variantentypen.)

Ist $n = 1$, so umfasst der Typ genau einen Konstruktor und die Fallunterscheidung entsprechend einen Zweig. Zum Beispiel:

```
type Price    = | Cent of Nat
type Postcode = | Code of Nat
```

Im Gegensatz zu 1-Tupeln sind 1-Varianten sehr nützlich: sind *Price* und *Postcode* wie oben definiert, so stellt die statische Semantik sicher, dass wir in einem Programm nicht aus Versehen Preise und Postleitzahlen addieren — *Price* und *Postcode* sind unterschiedliche, inkompatible Typen. Auch lässt sich ein Preis p nicht mit $2 * p$ verdoppeln. Zu diesem Zweck müssen wir extra eine Funktion schreiben.

```
let double (price : Price) : Price =
  match price with | Price n → Price (2 ∗ n)
```

Der Gewinn an Sicherheit wird mit einem Verlust an Bequemlichkeit erkauft.

Ist $n = 3$, so haben wir drei Alternativen und entsprechend ***match***-Ausdrücke mit drei Zweigen.

Ist $n = 4$, so haben wir vier Alternativen …

Vertiefung Wir haben schon kurz anklingen lassen, dass Fallunterscheidungen Alternativen verallgemeinern. Der Typ *Bool* kann in der Tat als Variantentyp aufgefasst werden bzw. durch einen Variantentyp implementiert werden.

```
type Bool = | False of Unit | True of Unit
```

Die Wahrheitswerte *false* und *true* werden durch die Ausdrücke *False* () und *True* () repräsentiert. Die Konstruktoren sind sozusagen „nullstellig“, formalisiert durch *Unit*, den Typ des leeren Tupels. Die Alternative ***if*** e_1 ***then*** e_2 ***else*** e_3 kann entsprechend durch die Fallunterscheidung ***match*** e_1 ***with*** | *False* () → e_3 | *True* () → e_2 realisiert werden. „Nullstellige“ Konstruktoren wie *False* oder *True* kommen relativ häufig vor. Aus diesem Grund erlauben wir, das Dummyargument auch wegzulassen, sowohl bei der Deklaration, als auch bei der Konstruktion und der Analyse. Die Definition der Wahrheitswerte verkürzt sich damit zu

```
type Bool = | False | True
```

Variantentypen, die nur nullstellige Konstruktoren umfassen, nennt man auch *Aufzählungstypen* (engl. enumeration types), da sie ihre Element gewissermaßen nacheinander aufzählen: ein Wahrheitswert ist entweder *False* oder *True*.

Mit Tupeln bzw. Records und Varianten haben wir zwei grundlegende Strukturierungselemente für Daten kennengelernt. Statt von Tupeln und Varianten spricht man auch von *Produkten* und *Summen*. Produkt deshalb, weil die Anzahl der Elemente von $t_1 * t_2$, die sogenannte *Kardinalität*, gleich dem Produkt der Kardinalitäten von t_1 und t_2 ist.[3] Summe deshalb, weil die Kardinalität von T mit ***type*** T = | C_1 ***of*** t_1 | C_2 ***of*** t_2, gleich der Summe der Kardinalitäten von t_1 und t_2 ist. Die Korrespondenz zwischen den arithmetischen Operationen und den Typkonstruktoren geht aber noch tiefer: in der gleichen Weise wie ‘+’ und ‘∗’ interagieren — ausgedrückt im *Distributivgesetz* $(a + b) * c = a * c + b * c$ —, so interagieren auch Records und Varianten. Kommen wir noch einmal auf die Definition von *Person* zurück.

```
type Woman = {name : String}
type Man   = {name : String; bald : Bool}
type Person = | Female of Woman | Male of Man
```

[3] Wir setzen stillschweigend voraus, dass die beteiligten Typen nur endlich viele Elemente enthalten.

Beiden Alternativen von *Person* ist der Name gemeinsam; diesen gemeinsamen „Faktor“ können wir mit Hilfe des Distributivgesetzes auch herausziehen und Personen alternativ darstellen durch

```
type Person' = {name : String; gender : Gender}
type Male'   = {bald : Bool}
type Gender  = | Female' | Male' of Male'
```

Das Geschlecht umfasst die trennenden Merkmale, die gemeinsamen sind in *Person′* zusammengefasst.

Die Typen *Person* und *Person′* sind wechselseitig austauschbar, sie sind im Fachjargon *isomorph*: jedem Element aus *Person* lässt sich eineindeutig[4] ein Element aus *Person′* zuordnen. Die Funktionen *from-Person* und *to-Person* belegen diese Tatsache.

```
let from-Person (p : Person) : Person' =
  match p with
  | Female f  → {name = f.name;  gender = Female'}
  | Male   m → {name = m.name; gender = Male' {bald = m.bald}}
let to-Person (p' : Person') : Person =
  match p'.gender with
    | Female'  → Female {name = p'.name}
    | Male' m' → Male   {name = p'.name; bald = m'.bald}
```

Welcher Repräsentation beim Programmieren der Vorzug gegeben wird, hängt vom Zugriffsmuster ab: wird oft auf den Namen zugegriffen, dann ist die zweite Version vorteilhaft; hängt vieles vom Geschlecht ab, dann ist die erste Version vorzuziehen.

4.2.2. Rekursive Varianten

Mit den Konstrukten, die wir bisher eingeführt haben, können wir nur eine beschränkte Anzahl von Daten zusammenfassen: ein 7-Tupel aggregiert 7 Daten, ein 128-Tupel 128 Daten; beide sind ungeeignet um 6, 8, 127 oder 129 Daten aufzunehmen. Nun kommt es häufig vor, dass man zum Zeitpunkt des Programmierens die genaue Anzahl von Daten nicht kennt: Wieviele Personen immatrikulieren sich im WS 2019/2020? Wieviele Unternehmen sind an der Börse notiert? Wieviele Mitarbeiter hat eine Abteilung? usw. Auch wenn man die Anzahl kennt — weil vielleicht nur 128 Personen zum Studium angenommen werden — ist ein n-Tupel zu unhandlich: ein Ausdruck, der ein 128-Tupel konstruiert, hat eben 128 Teilausdrücke, die erst einmal aufgeschrieben werden wollen.

Im Folgenden überlegen wir, wie wir eine beliebige Anzahl von natürlichen Zahlen repräsentieren können. Wenn wir naiv für jede Anzahl eine Alternative definieren, erhalten wir den folgenden Variantentyp (wir lassen zunächst die Datenkonstruktoren weg).

```
type Nats = | Unit | Nat | Nat * Nat | Nat * Nat * Nat | ···
```

[4] Das Adjektiv „eineindeutig“ bedeutet umkehrbar eindeutig, eindeutig in beide Richtungen. Eine eineindeutige Abbildung nennt man auch *Bijektion*.

In Worten ausgedrückt: Eine Folge von natürlichen Zahlen ist entweder die leere Folge (ein 0-Tupel vom Typ *Unit*), oder eine einelementige Folge (ein 1-Tupel), oder eine zweielementige Folge (ein 2-Tupel) usw. Das Plural 's' im Bezeichner *Nats* soll andeuten, dass ein Element dieses Typs viele natürliche Zahlen umfasst.

Die Ellipse $(\cdots)$ zeigt an, dass wir noch keine vollständige Definition vor uns haben. Wir müssen die Definition noch etwas massieren. Es ist klar, dass alle Alternativen bis auf die erste mindestens eine *Nat* Komponente haben. Wenn wir diese Komponente „herausziehen", erhalten wir (das ist keine gültige Typdefinition, aber das soll uns im Moment nicht stören)

$$\textbf{\textit{data}}\ \mathit{Nats} = \ |\ \mathit{Unit} \mid \mathit{Nat}\ *\ (\mathit{Unit} \mid \mathit{Nat} \mid \mathit{Nat} * \mathit{Nat} \mid \cdots)$$

Aus einem $n+1$-Tupel wird jeweils ein n-Tupel. Formal liegt dem „Herausziehen" das Distributivgesetz zugrunde, das uns schon im letzten Abschnitt begegnet ist. Jetzt sind wir fast am Ziel. Der Typ der zweiten Komponente ist identisch mit der rechten Seite der ursprünglichen Definition von *Nats*. Erlauben wir bei der Definition eines Variantentyps den Rückgriff auf den definierten Typ selbst (!), so erhalten wir die folgende, kompakte Definition (jetzt mit Datenkonstruktoren).

$$\textbf{\textit{type}}\ \mathit{Nats} = \ |\ \mathit{Nil} \mid \mathit{Cons}\ \textbf{\textit{of}}\ \mathit{Nat} * \mathit{Nats}$$

In Worten ausgedrückt: Eine Folge von natürlichen Zahlen ist entweder die leere Folge *Nil* oder eine mindestens einelementige Folge *Cons* (n, ns) bestehend aus einer natürlichen Zahl n und einer Folge von natürlichen Zahlen *ns*. Statt von einer Folge von natürlichen Zahlen spricht man auch kurz von einer *Liste*. Entsprechend heißt die Zahl n *Kopfelement* der Liste *Cons* (n, ns) und *ns* *Restliste*. Die Folge der ersten vier Primzahlen wird durch den Ausdruck *Cons* $(2,$ *Cons* $(3,$ *Cons* $(5,$ *Cons* $(7,$ *Nil*$))))$ repräsentiert; die natürliche Zahl 2 ist das Kopfelement dieser Liste und *Cons* $(3,$ *Cons* $(5,$ *Cons* $(7,$ *Nil*$)))$ die Restliste. Bei Listen ist wie bei Tupeln die Reihenfolge der Elemente signifikant: *Cons* $(7,$ *Cons* $(2,$ *Cons* $(5,$ *Cons* $(3,$ *Nil*$))))$ ist eine gänzlich andere Liste.

Listen sind die erste und einfachste *Datenstruktur*, die uns begegnet. Eine Datenstruktur verwaltet Daten und unterstützt Zugriff und Manipulation dieser Daten.

Rekursive Typen sind für uns eigentlich nichts Neues. Bei der Beschreibung der abstrakten Syntax unserer Programmiersprache sind sie uns schon wiederholt begegnet: Ein Ausdruck ist zum Beispiel eine Alternative, diese besteht aus drei Teilausdrücken; ein Muster hat zum Beispiel die Form p_1 & p_2, wobei p_1 und p_2 wiederum Muster sind. Fast alle syntaktischen Bereiche sind rekursiv definiert. Mit Hilfe rekursiver Variantentypen haben wir sozusagen Baumsprachen vollständig in unsere Programmiersprache integriert!

Wie gehen wir nun mit einem rekursiven Variantentyp um? Wir konstruieren und analysieren rekursive Varianten mit Hilfe rekursiver Funktionen! Wenden wir uns zunächst der Verarbeitung von Listen zu und übertragen die Funktion *sort2* bzw. *sort3* auf Listen.

Mini⟩ *sort* (*Cons* $(7,$ *Cons* $(2,$ *Cons* $(5,$ *Cons* $(3,$ *Nil*$)))))$
Cons $(2,$ *Cons* $(3,$ *Cons* $(5,$ *Cons* $(7,$ *Nil*$))))$

Der Variantentyp gibt das folgende Skelett für die Sortierfunktion vor.

```
let rec sort (nats : Nats) : Nats =
  match nats with
    | Nil          → ...
    | Cons (n, ns) → ...
```

Die leere Liste zu sortieren ist einfach; sie ist bereits geordnet. Wie aber füllen wir den Fall der mindestens einelementigen Liste mit Leben? Getreu dem Motto „rekursive Funktionen für rekursive Typen“ können wir *sort* auf die Restliste *ns* anwenden. Wir erhalten eine geordnete Liste, bei der lediglich das Kopfelement n fehlt. (Die Vorgehensweise ist ähnlich wie bei der Definition von *sort3*, nur dass *sort3* sich auf *sort2* abstützt.) Das Element n ist nun nicht notwendigerweise das kleinste Element, wir müssen es an die richtige Stelle einordnen. Ein Element in eine geordnete Liste einfügen, das hört sich nach einer anspruchsvolleren Teilaufgabe an. Wir geben dieser Teilaufgabe einen Namen, *insert*, und können damit die Definition von *sort* vervollständigen.

```
let rec sort (nats : Nats) : Nats =
  match nats with
    | Nil          → Nil
    | Cons (n, ns) → insert (n, sort ns)
```

Die Definition der Hilfsfunktion *insert* gehen wir auf die gleiche Art und Weise an.[5]

```
let rec insert (nat : Nat, nats : Nats) : Nats =
  match nats with
    | Nil          → ...
    | Cons (n, ns) → ...
```

Ist die Liste leer, so geben wir die einelementige Liste *Cons* (*nat*, *Nil*) zurück. Im anderen Fall wissen wir, dass das Kopfelement n das Minimum der Liste *nats* ist. Wenn nun das einzufügende Element *nat* gleich oder kleiner als n ist, müssen wir *nat* an den Anfang der Liste stellen:

```
let rec insert (nat : Nat, nats : Nats) : Nats =
  match nats with
    | Nil          → Cons (nat, Nil)
    | Cons (n, ns) → if nat ⩽ n then Cons (nat, nats)
                                else ...
```

Ist *nat* größer als n, so bleibt n das kleinste Element: die resultierende Liste hat somit die Form *Cons* $(n, \ldots)$. Es verbleibt, *nat* in die Restliste *ns* einzufügen. Getreu dem Motto „rekursive Funktionen für rekursive Typen“ können wir dafür *insert* verwenden. Die vollständige Definition von *insert* lautet somit

[5] Im tatsächlichen Programmtext muss *insert* vor *sort* definiert werden, da letztere Funktion sich auf erstere abstützt.

```
let rec insert (nat : Nat, nats : Nats) : Nats =
  match nats with
    | Nil          → Cons (nat, Nil)
    | Cons (n, ns) → if nat ⩽ n then Cons (nat, nats)
                                else Cons (n, insert (nat, ns))
```

Die gezeigte Vorgehensweise ist typisch für die Verarbeitung von Listen und lässt sich als Problemlösungsstrategie formulieren: Wir lösen das Problem zunächst für die leere Liste (*Rekursionsbasis*); um das Problem für eine mindestens einelementige Liste zu lösen, bestimmen wir zunächst rekursiv die Lösung für die Restliste und erweitern diese dann zu einer Gesamtlösung (*Rekursionsschritt*). Mit anderen Worten, wir haben ein *Entwurfsmuster* gefunden, um listenverarbeitende Funktionen zu definieren.

```
let rec f (nats : Nats) : t =                    Struktur Entwurfsmuster
  match nats with
    | Nil          → ...                         Rekursionsbasis
    | Cons (n, ns) → ... n ... f ns ...          Rekursionsschritt
```

Da wir uns eng an der Struktur des Listentyps orientieren — eine Liste ist entweder leer oder besteht aus einem Kopf*element* und einer Rest*liste* —, nennen wir das Schema *Struktur Entwurfsmuster* für *Nats*.

Erproben wir das Struktur Entwurfsmuster. Nehmen wir an, wir wollen eine gegebene Liste nicht aufsteigend, sondern absteigend sortieren. Wir können natürlich das Programm für *sort* hernehmen und '⩽' systematisch durch '⩾' ersetzen. Eine solche Duplizierung des Programmcodes ist aber unökonomisch. Alternativ können wir die Liste zunächst aufsteigend sortieren und sie dann einfach umdrehen.

```
let decreasing-sort (nats : Nats) : Nats = reverse (sort nats)
```

Die Funktion *reverse* ist genau wie *insert* von allgemeinem Nutzen. Wir sehen, die Identifizierung von Teilproblemen und die Programmierung von Teillösungen beschert uns eine Menge nützlicher Funktionen.

```
let rec reverse (nats : Nats) : Nats =
  match nats with
    | Nil          → Nil
    | Cons (n, ns) → ... reverse ns ...
```

Nachdem wir *reverse* rekursiv auf *ns* anwendet haben, müssen wir noch *n* hinter die resultierende Liste setzen. Listen — wie wir sie definiert haben — sind allerdings asymmetrisch. Auf das erste Element einer Liste können wir leicht zugreifen, auf das letzte Element nicht. Ein Element einer Liste voranzustellen ist einfach; um ein Element hinten anzustellen, müssen wir Aufwand betreiben. Sprich, wir müssen eine entsprechende

Funktion programmieren. Da wir mittlerweile schon etwas im Umgang mit Listen geübt sind, hier gleich die vollständige Definition.

```
let rec put-last (nats : Nats, nat : Nat) : Nats =
  match nats with
    | Nil          → Cons (nat, Nil)
    | Cons (n, ns) → Cons (n, put-last (ns, nat))
```

Die Funktion erledigt ihren Job, ist aber unnötig speziell: *put-last* bildet im Basisfall die leere Liste auf eine einelementige Liste ab. Verallgemeinern wir die einelementige Liste zu einer beliebigen Liste, erhalten wir eine flexiblere Funktion.

```
let rec append (nats₁ : Nats, nats₂ : Nats) : Nats =
  match nats₁ with
    | Nil          → nats₂
    | Cons (n, ns) → Cons (n, append (ns, nats₂))
```

Die Funktion *append* hängt zwei Listen aneinander; *put-last* lässt sich einfach mit Hilfe von *append* implementieren: *append* (*nats*, *Cons* (*nat*, *Nil*)). Die Umkehrung gilt nicht.

Somit können wir die Definition von *reverse* vervollständigen.

```
let rec reverse (nats : Nats) : Nats =
  match nats with
    | Nil          → Nil
    | Cons (n, ns) → append (reverse ns, Cons (n, Nil))
```

module Datatypes. Sort

Kommen wir noch einmal auf die ursprüngliche Aufgabenstellung zurück: die absteigende Sortierung einer Liste. Wenn wir die Aufgabenstellung etwas verallgemeinern und von der speziellen Ordnungsrelation abstrahieren, ergibt sich ein weiterer Lösungsansatz. Von einer speziellen Ordnungsrelation abstrahieren heißt, dass wir '$\leqslant$' bzw. '$\geqslant$' nicht fest im Programm verdrahten, sondern zum Parameter der Sortierfunktion machen.

```
let sort-by (less-equal : Nat * Nat → Bool) : Nats → Nats =
  let rec insert (nat : Nat, nats : Nats) : Nats =
    match nats with
      | Nil          → Cons (nat, Nil)
      | Cons (n, ns) → if less-equal (nat, n) then Cons (nat, nats)
                                              else Cons (n, insert (nat, ns))
  let rec sort (nats : Nats) : Nats =
    match nats with
      | Nil          → Nil
      | Cons (n, ns) → insert (n, sort ns)
  in sort
```

Mit Hilfe einer lokalen Deklaration werden die Hilfsfunktionen *insert* und *sort* eingeführt. Aufgrund der Sichtbarkeitsregeln ist der formale Parameter von *sort-by*, die Ordnungsrelation *less-equal*, auch im Rumpf der Hilfsfunktionen sichtbar. Die Funktion *sort-by* gibt als Ergebnis die für die jeweilige Ordnungsrelation spezialisierte Sortierfunktion zurück.

Die ursprünglichen Sortierfunktionen sind jetzt hausbackene Spezialfälle:

$$\textbf{\textit{let}}\ \mathit{increasing\text{-}sort} = \mathit{sort\text{-}by}\ (\textbf{\textit{fun}}\ (m, n) \to m \leqslant n)$$
$$\textbf{\textit{let}}\ \mathit{decreasing\text{-}sort} = \mathit{sort\text{-}by}\ (\textbf{\textit{fun}}\ (m, n) \to m \geqslant n)$$

Die Funktion *sort-by* ist ein weiteres Beispiel für eine Funktion höherer Ordnung: *sort-by* nimmt eine Funktion als Argument (*less-equal*) und gibt eine Funktion (*sort*) als Ergebnis zurück. Genau wie *append* aus *put-last* hervorgeht, so ensteht *sort-by* aus *sort*: durch Verallgemeinerung der Aufgabenstellung. In beiden Fällen bleibt der Implementierungsaufwand der gleiche; wir erhalten aber Funktionen, die nützlicher und flexibler sind.

Wenden wir uns nach dem Studium der *listenverarbeitenden* Funktionen kurz den *listenerzeugenden* Funktionen zu. Programmieren wir eine Funktion, die eine Liste aller Elemente in einem gegebenen Intervall erzeugt. In Abschnitt 3.6 haben wir schon etwas Erfahrung im Umgang mit Intervallen gesammelt. Die Lektionen legen nahe, das Peano Entwurfsmuster auf die Intervallgröße anzuwenden.

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{between}\ (l : \mathit{Nat}, u : \mathit{Nat}) : \mathit{Nats} = \\ \quad \textbf{\textit{if}}\ l > u\ \textbf{\textit{then}}\ \mathit{Nil} \\ \qquad\qquad \textbf{\textit{else}}\ \ \mathit{Cons}\ (l, \mathit{between}\ (l + 1, u))\end{array}$$

Im Basisfall geben wir die leere Liste zurück; im Rekursionsfall setzen wir die linke Intervallgrenze vor die rekursiv erzeugte Liste.

Abstrakte Syntax … Weder die abstrakte Syntax noch die statische oder dynamische Semantik von Varianten ändert sich. Die Tatsache, dass wir einem Variantentyp immer einen eindeutigen Namen geben — was wir ja zum Beispiel bei Tupeltypen der Form $t_1 * t_2$ nicht machen — ermöglicht gerade die problemlose Erweiterung von Variantentypen auf *rekursive* Variantentypen.

Vertiefung: Unäre Zahlendarstellung Im Abschnitt 4.2.1 haben wir gesehen, dass die Booleschen Werte durch einen Variantentyp implementiert werden können. Überraschenderweise gilt dies auch für die natürlichen Zahlen. Die Definition lässt sich leicht motivieren, wenn wir uns noch einmal das Peano Entwurfsmuster ins Gedächtnis rufen. Das Entwurfsmuster basiert auf der Tatsache, dass jede natürliche Zahl entweder 0 oder von der Form $n + 1$ ist, wobei n wiederum eine natürliche Zahl ist. Machen wir Null und die Nachfolgerfunktion zu Datenkonstruktoren, so erhalten wir

$$\begin{array}{l}\textbf{\textit{type}}\ \mathit{Peano} = \\ \quad \mid \mathit{Zero} \\ \quad \mid \mathit{Succ}\ \textbf{\textit{of}}\ \mathit{Peano}\end{array}$$

Die Zahlendarstellung unserer Urahnen: für jedes erlegte Bison einen Strich. Die Zahl n wird durch n Anwendungen des Konstruktors *Succ* auf den Konstruktor *Zero* repräsentiert. Wir zählen: *Zero*, *Succ Zero*, *Succ* (*Succ Zero*), *Succ* (*Succ* (*Succ Zero*)) usw. Diese Repräsentation heißt auch *unäre Zahlendarstellung*.

Die Implementierungen von Addition, Multiplikation usw. lassen sich einfach aus Abschnitt 3.6 adaptieren. Die Konstante 0 wird zu *Zero*, der Ausdruck $e+1$ wird zu *Succ* e und die Alternative ***if*** $n = 0$ ***then*** $\ldots$ ***else*** $\ldots n \dot{-} 1 \ldots$ wird zur Fallunterscheidung ***match*** n ***with*** $\mid$ *Zero* $\rightarrow \ldots \mid$ *Succ* $n' \rightarrow \ldots n' \ldots$. Zum Beispiel: Die ursprünglichen Definition der Addition

```
let rec add (m : Nat, n : Nat) : Nat =
  if m = 0 then n
           else add (m ∸ 1, n) + 1
```

lässt sich wie folgt auf die neue Zahlendarstellung übertragen:

```
let rec add (m : Peano, n : Peano) : Peano =
  match m with
    | Zero     → n
    | Succ m' → Succ (add (m', n))
```

Kurzum: das Peano Entwurfsmuster entspricht dem Struktur Entwurfsmuster für den Variantentyp *Peano*.

Vertiefung: Binäre Zahlendarstellung Können wir aus dem Leibniz Entwurfsmuster eine alternative Darstellung der natürlichen Zahlen ableiten? Überlegen wir: Das Leibniz Entwurfsmuster basiert auf der Tatsache, dass jede natürliche Zahl entweder 0, oder von der Form $2n$ bzw. $2n+1$ ist, wobei n wiederum eine natürliche Zahl ist.[6] Anders ausgedrückt, eine natürliche Zahl ist entweder Null, gerade oder ungerade. Fangen wir die drei Fälle mit Datenkonstruktoren ein, so erhalten wir

```
type Leibniz =
  | Null
  | Even of Leibniz
  | Odd  of Leibniz
```

Jetzt zählen wir wie folgt: *Null*, *Odd Null*, *Even* (*Odd Null*), *Odd* (*Odd Null*) usw. Um das Bildungsgesetz zu sehen, ist es am einfachsten, wenn wir die Nachfolgerfunktion programmieren.

```
let rec succ (nat : Leibniz) : Leibniz =
  match nat with
    | Null    → Odd Null
    | Even n → Odd n
    | Odd  n → Even (succ n)
```

In Worten und Formeln: der Nachfolger von 0 ist 1, der Nachfolger von $2n$ ist $2n+1$, der Nachfolger von $2n+1$ ist $2n+2 = 2(n+1)$. Mit Hilfe von *succ* können wir jede natürliche Zahl in die neue Darstellung überführen, siehe Übung 4.6.

[6] Diese drei Fälle sind nicht disjunkt; in der Vorlesung diskutieren wir eine alternative Darstellung mit disjunkten Fällen: Jede natürliche Zahl ist entweder 0 oder von der Form $2n+1$ bzw. $2n+2$.

Die umgekehrte Fragestellung ist genauso interessant: Welche natürliche Zahl wird zum Beispiel durch $\mathit{Odd}\ (\mathit{Odd}\ (\mathit{Even}\ (\mathit{Odd}\ \mathit{Null})))$ repräsentiert? Nun, $\mathit{Even}\ e$ repräsentiert $2n$, $\mathit{Odd}\ e$ repräsentiert $2n + 1$, wenn e die Zahl n repräsentiert. Somit stellt $\mathit{Odd}\ (\mathit{Odd}\ (\mathit{Even}\ (\mathit{Odd}\ \mathit{Null})))$ die Zahl 11 dar: $2 * (2 * (2 * (2 * 0 + 1)) + 1) + 1 = 11$. Diese Umformung können wir ebenfalls in Rechenregeln gießen.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{natural}\ (\mathit{nat} : \mathit{Leibniz}) : \mathit{Nat} = \\
\quad \textbf{\textit{match}}\ \mathit{nat}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Null}\ \ \ \ \ \ \to 0 \\
\qquad \mid \mathit{Even}\ n \to 2 * \mathit{natural}\ n \\
\qquad \mid \mathit{Odd}\ \ n \to 2 * \mathit{natural}\ n + 1
\end{array}
$$

Die Funktion *natural* ordnet einer Zahlendarstellung ihren Zahlenwert zu, einer konkreten Repräsentation ihre abstrakte Bedeutung. (Der Typ *Leibniz* ist konkret: Wir kennen seine Definition und wissen welche Elemente er umfasst. Der Typ *Nat* hingegen ist abstrakt: Seine Implementierung ist uns nicht bekannt.)

Die Funktion *natural* hat noch einen anderen Verwendungszweck: Mit ihrer Hilfe können wir arithmetische Operationen auf dem Typ *Leibniz* *spezifizieren*. Eine Spezifikation beschreibt, *was* eine Funktion leisten soll; im Unterschied zu einer Implementierung, die genau festlegen, *wie* eine Funktion ihr Ergebnis ermittelt. Die Nachfolgerfunktion und die Addition lassen sich wie folgt spezifizieren:

$$\mathit{natural}\ (\mathit{succ}\ e) \quad = \quad \mathit{natural}\ e + 1 \tag{4.2a}$$

$$\mathit{natural}\ (\mathit{add}\ (e_1, e_2)) \quad = \quad \mathit{natural}\ e_1 + \mathit{natural}\ e_2 \tag{4.2b}$$

Die zweite mathematische Gleichung können wir benutzen, um eine Implementierung der Addition *herzuleiten*. Der Typ *Leibniz* gibt zunächst einmal das folgende Schema vor:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{add}\ (m : \mathit{Leibniz}, n : \mathit{Leibniz}) : \mathit{Leibniz} = \\
\quad \textbf{\textit{match}}\ m\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Null}\ \ \ \ \ \ \to \ldots \\
\qquad \mid \mathit{Even}\ m' \to \ldots \\
\qquad \mid \mathit{Odd}\ \ m' \to \ldots
\end{array}
$$

Um die Definition für den Basisfall $m = \mathit{Null}$ zu bestimmen, rechnen wir

$$
\begin{array}{ll}
 & \mathit{natural}\ (\mathit{add}\ (\mathit{Null}, n)) \\
= & \quad \{\ \text{Spezifikation von } \mathit{add}\ (4.2\text{b})\ \} \\
 & \mathit{natural}\ \mathit{Null} + \mathit{natural}\ n \\
= & \quad \{\ \text{Definition von } \mathit{natural}\ \} \\
 & 0 + \mathit{natural}\ n \\
= & \quad \{\ 0 \text{ ist das neutrale Element von '+'}\ \} \\
 & \mathit{natural}\ n
\end{array}
$$

Es muss also $natural\ (add\ (Null, n)) = natural\ n$ gelten; diese Gleichung können wir erfüllen, wenn wir im Basisfall n zurückgeben. Klar: 0 ist das neutrale Element der Addition. In den beiden anderen Fällen $m = Even\ m'$ und $m = Odd\ m'$ müssen wir eine Fallunterscheidung über das zweite Argument vornehmen:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ add\ (m : Leibniz, n : Leibniz) : Leibniz = \\
\quad \textbf{\textit{match}}\ m\ \textbf{\textit{with}} \\
\qquad \mid Null \qquad \to n \\
\qquad \mid Even\ m' \to \textbf{\textit{match}}\ n\ \textbf{\textit{with}} \mid Null \quad \to \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Even\ n' \to \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Odd\ \ n' \to \ldots \\
\qquad \mid Odd\ \ m' \to \textbf{\textit{match}}\ n\ \textbf{\textit{with}} \mid Null \quad \to \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Even\ n' \to \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Odd\ \ n' \to \ldots
\end{array}
$$

Die beiden Fälle $n = Null$ sind symmetrisch zum Fall $m = Null$. Sind beide Zahlen gerade, so können wir herleiten.

$$
\begin{array}{ll}
 & natural\ (add\ (Even\ m', Even\ n')) \\
= & \quad \{\ \text{Spezifikation von } add\ (4.2\text{b})\ \} \\
 & natural\ (Even\ m') + natural\ (Even\ n') \\
= & \quad \{\ \text{Definition von } natural\ \} \\
 & 2 * natural\ m' + 2 * natural\ n' \\
= & \quad \{\ \text{Distributivgesetz}\ \} \\
 & 2 * (natural\ m' + natural\ n') \\
= & \quad \{\ \text{Spezifikation von } add\ (4.2\text{b})\ \} \\
 & 2 * (natural\ (add\ (m', n'))) \\
= & \quad \{\ \text{Definition von } natural\ \} \\
 & natural\ (Even\ (add\ (m', n')))
\end{array}
$$

Die Rechnung legt nahe, dass $add\ (Even\ m', Even\ n')$ zu $Even\ (add\ (m', n'))$ auswerten sollte. Ähnliche Rechnungen ergeben sich, wenn eine Zahl gerade, die andere ungerade

ist. Interessant wird es, wenn beide Zahlen ungerade sind.

$$
\begin{aligned}
& \mathit{natural}\ (\mathit{add}\ (\mathit{Odd}\ m', \mathit{Odd}\ n')) \\
= & \quad \{\ \text{Spezifikation von } \mathit{add}\ (4.2\text{b})\ \} \\
& \mathit{natural}\ (\mathit{Odd}\ m') + \mathit{natural}\ (\mathit{Odd}\ n') \\
= & \quad \{\ \text{Definition von } \mathit{natural}\ \} \\
& 2 * \mathit{natural}\ m' + 1 + 2 * \mathit{natural}\ n' + 1 \\
= & \quad \{\ \text{Distributivgesetz}\ \} \\
& 2 * (\mathit{natural}\ m' + \mathit{natural}\ n' + 1) \\
= & \quad \{\ \text{Spezifikation von } \mathit{add}\ (4.2\text{b})\ \} \\
& 2 * (\mathit{natural}\ (\mathit{add}\ (m', n')) + 1) \\
= & \quad \{\ \text{Spezifikation von } \mathit{succ}\ (4.2\text{a})\ \} \\
& 2 * (\mathit{natural}\ (\mathit{succ}\ (\mathit{add}\ (m', n')))) \\
= & \quad \{\ \text{Definition von } \mathit{natural}\ \} \\
& \mathit{natural}\ (\mathit{Even}\ (\mathit{succ}\ (\mathit{add}\ (m', n'))))
\end{aligned}
$$

Die Rechnung legt nahe, dass $\mathit{add}\ (\mathit{Odd}\ m', \mathit{Odd}\ n')$ zu $\mathit{Even}\ (\mathit{succ}\ (\mathit{add}\ (m', n')))$ auswerten sollte. Die Summe zweier ungerader Zahlen ist ebenfalls gerade. Im Unterschied zum vorherigen Fall müssen wir das Ergebnis des rekursiven Aufrufs mit *succ* um eins erhöhen, sprich wir müssen den sogenannten *Übertrag* (engl. carry) berücksichtigen. Insgesamt erhalten wir das folgende Programm.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
    | Null      → n
    | Even m' → match n with | Null     → m
                             | Even n' → Even (add (m', n'))
                             | Odd  n' → Odd  (add (m', n'))
    | Odd  m' → match n with | Null     → m
                             | Even n' → Odd  (add (m', n'))
                             | Odd  n' → Even (succ (add (m', n')))
```

Der Datentyp *Leibniz* implementiert übrigens das Binär- oder *Dualsystem*. Wie unser *Dezimalsystem* ist das Dualsystem ein *Stellenwertsystem*; im Unterschied zum Dezimalsystem verwendet es nicht zehn Ziffern, sondern nur zwei. Der Zusammenhang zum Dualsystem wird noch deutlicher, wenn wir den Datentyp *Leibniz* mit Hilfe des Distributivgesetzes umschreiben (wir lassen die Datenkonstruktoren zunächst weg).

$$
\begin{aligned}
\mathit{Leibniz} &\cong \mathit{Unit} \mid \mathit{Leibniz} \mid \mathit{Leibniz} \\
&\cong \mathit{Unit} \mid \mathit{Unit} * \mathit{Leibniz} \mid \mathit{Unit} * \mathit{Leibniz} \\
&\cong \mathit{Unit} \mid (\mathit{Unit} \mid \mathit{Unit}) * \mathit{Leibniz}
\end{aligned}
$$

Im vorletzten Schritt nutzen wir aus, dass $\mathit{Unit} * t \cong t$. Vergleichen wir den resultierenden Typ mit der Definition von *Nats*, so sehen wir, dass *Leibniz* im Prinzip ein Listentyp

ist. Im Fall von *Nats* sind die Listenelemente natürliche Zahlen, im Fall von *Leibniz* sind sie Bits, Elemente des zweielementigen Typs $Unit \mid Unit$. Eine alternative Definition von *Leibniz* sieht somit folgendermaßen aus (jetzt mit Datenkonstruktoren).

$$\begin{aligned} &\textbf{\textit{type}}\ Bit\ \ = \mid Zero \mid One \\ &\textbf{\textit{type}}\ Bits = \mid Nil \mid Cons\ \textbf{\textit{of}}\ Bit * Bits \end{aligned}$$

In einem Stellenwertsystem wird eine Zahl durch eine Folge von Ziffern repräsentiert, hier eine Folge von Binärziffern (das englische Wort bit ist eine Verschmelzung von binary und digit). Der Wert einer Ziffer wird durch seine Position in der Liste bestimmt. Die Ziffern im Listenrest wiegen doppelt so viel wie die Ziffer im Listenkopf (siehe Definition von *natural*), daher der Name *Stellenwert*system.

4.2.3. Widerlegbare Muster

Die Zweige einer Fallunterscheidung führen Bezeichner ein. Wie im Fall von Wertebindungen können wir auch hier auf den linken Seiten anstelle eines einzelnen Bezeichners (x in $C\ x$) ein Muster (p in $C\ p$) erlauben. In der Tat lässt sich sogar die gesamte linke Seite als Muster lesen: $C\ p$ ist ein Muster, auf das ein Wert der Form $C\ \nu$ passt, sofern ν auf p passt. Im Unterschied zu den Mustern, die wir bisher kennengelernt haben, kann der Musterabgleich im Fall von $C\ p$ auch fehlschlagen: Der Wert $Male\ \nu$ passt nicht auf das Muster $Female\ x$. Man sagt auch, das Muster $Female\ x$ ist *widerlegbar* (engl. refutable pattern). Die Fallunterscheidung lässt sich unter diesem Hintergrund als sukzessives Durchprobieren von Mustern deuten.

Nehmen wir Konstruktoranwendungen wie $C\ p$ zu den Mustern hinzu, dann können wir Konstruktoranwendungen auch schachteln. Wir werden später sehen, dass dies sehr bequem und nützlich ist.

Abstrakte Syntax Wir erweitern zunächst die Syntax von Fallunterscheidungen.

$e ::= \cdots$	*Ausdrücke:*
$\mid$ ***match*** e ***with*** m	erweiterte Fallunterscheidung

Der Rumpf einer erweiterten Fallunterscheidung ist eine Folge von sogenannten *Regeln*[7] der Form $p \to e$.

$m \in \mathsf{Match} ::= p \to e$	Regel
$\mid\ m_1 \mid m_2$	Sequenz von Regeln

Schließlich wird der Bereich der Muster um disjunktive[8] Muster und Konstruktoranwendungen erweitert.

[7] Verwechseln Sie den Begriff „Regel“ nicht mit „Typregel“ oder „Auswertungsregel“. Eine Regel ist ein Element der Objektsprache Mini–F#; Typ- und Auswertungsregeln sind Elemente der Metasprache, mit der wir Mini–F# formalisieren.

[8] Das Adjektiv „disjunktiv“ meint hier nicht, dass die alternativen Muster einander ausschließen (exklusiv). Wie die logische Disjunktion ‘||’ ist auch das alternative Muster einschließend (inklusiv).

$p ::= \cdots$	*Muster:*
$\mid\ p_1 \mid p_2$	disjunktives Muster
$\mid\ C\ p$	Konstruktoranwendung

Eine Regel mit einem disjunktiven Muster $(p_1 \mid p_2) \to e$ entspricht der Regelsequenz $(p_1 \to e) \mid (p_2 \to e)$, hat aber den Vorteil, dass der Ausdruck e nicht dupliziert werden muss. Das Muster $\mathit{Female}\ \{\mathit{name} = s\} \mid \mathit{Male}\ \{\mathit{name} = s; \mathit{bald} = \_\}$ bindet zum Beispiel den Bezeichner s an den Namen einer Person. In einem disjunktiven Muster müssen beide Teilmuster die gleichen Bezeichner enthalten.

Statische Semantik Der Rumpf einer erweiterten Fallunterscheidung ist einer Funktion nicht unähnlich; der ***match***-Ausdruck selbst ist dem Wesen nach eine Funktionsapplikation. Die Notation

$$\Sigma \vdash m\ (t) : t'$$

für die Typisierung von Regeln soll entsprechend suggerieren, dass die Regel m auf einen Wert vom Typ t angewendet zu einem Element vom Typ t' auswertet.

$$\frac{\Sigma \vdash e : t \qquad \Sigma \vdash m\ (t) : t'}{\Sigma \vdash \textbf{\textit{match}}\ e\ \textbf{\textit{with}}\ m : t'}$$

Der Typ des Diskriminatorausdrucks e muss übrigens kein Variantentyp sein — wir werden uns später einige Beispiele dafür anschauen.

Die Typregeln für den Rumpf einer Fallunterscheidung stellen sicher, dass die rechten Seiten den gleichen Typ besitzen und dass die Muster dem Typ des Diskriminatorausdrucks entsprechen.

$$\frac{p \sim t : \Sigma' \qquad \Sigma, \Sigma' \vdash e : t'}{\Sigma \vdash (p \to e)\ (t) : t'} \qquad \frac{\Sigma \vdash m_1\ (t) : t' \qquad \Sigma \vdash m_2\ (t) : t'}{\Sigma \vdash (m_1 \mid m_2)\ (t) : t'}$$

Die folgenden Typregeln erweitern den Regelsatz für den Musterabgleich, den wir in Abschnitt 4.1.2 eingeführt haben.

$$\frac{p_1 \sim t : \Sigma' \qquad p_2 \sim t : \Sigma'}{(p_1 \mid p_2) \sim t : \Sigma'}$$

Beachte: p_1 und p_2 in dem disjunktiven Muster $p_1 \mid p_2$ müssen die gleiche Signatur Σ' haben, sprich sie müssen die gleichen Bezeichner des gleichen Typs binden. Die folgende Typregel setzt voraus, dass der Variantentyp

$$\textbf{\textit{type}}\ T = \mid C_1\ \textbf{\textit{of}}\ t_1 \mid C_2\ \textbf{\textit{of}}\ t_2$$

bekannt ist.

$$\frac{p \sim t_i : \Sigma'}{C_i\ p \sim T : \Sigma'}$$

Wenn t_i der Einheitstyp Unit ist, dann ist C_i selbst ein Muster, da das Dummyargument () wie üblich weggelassen wird. Aus diesem Grund werden übrigens Konstruktoren groß geschrieben: damit sich das Muster x in der konkreten Syntax optisch leicht von dem Muster C unterscheiden lässt.

Dynamische Semantik Mit der Einführung von Konstruktormustern kann der Abgleich eines Musters mit einem Wert fehlschlagen. Diesen Fall signalisieren wir durch einen Blitz:

$$p \sim \nu \Downarrow \lightning$$

Der Blitz symbolisiert, dass das Muster p *nicht* auf den Wert ν passt.

Die erweiterte Fallunterscheidung wird wie folgt ausgewertet.

$$\frac{\delta \vdash e \Downarrow \nu \qquad \delta \vdash m\ (\nu) \Downarrow \nu'}{\delta \vdash \textbf{\textit{match}}\ e\ \textbf{\textit{of}}\ m \Downarrow \nu'}$$

Die Hilfsrelation

$$\delta \vdash m\ (\nu) \Downarrow \nu'$$

modelliert das Durchprobieren der Regeln. Die Notation $m\ (\nu)$ suggeriert wie bei den Typregeln, dass m auf den Wert ν angewendet wird. Trifft keine Regel zu, so ist die Fallunterscheidung undefiniert: Die dynamische Semantik ordnet dem Ausdruck keinen Wert zu — dieses Problem werden wir erst in Abschnitt 7.4 beheben.

$$\frac{p \sim \nu \Downarrow \delta' \qquad \delta, \delta' \vdash e \Downarrow \nu'}{\delta \vdash (p \to e)\ (\nu) \Downarrow \nu'} \qquad \frac{p \sim \nu \Downarrow \lightning}{\delta \vdash (p \to e)\ (\nu) \Downarrow \lightning}$$

Die Zweige werden von links nach rechts durchprobiert, bis die erste Regel passt.

$$\frac{\delta \vdash m_1\ (\nu) \Downarrow \nu'}{\delta \vdash (m_1 \mid m_2)\ (\nu) \Downarrow \nu'}$$

$$\frac{\delta \vdash m_1\ (\nu) \Downarrow \lightning \qquad \delta \vdash m_2\ (\nu) \Downarrow \nu'}{\delta \vdash (m_1 \mid m_2)\ (\nu) \Downarrow \nu'}$$

$$\frac{\delta \vdash m_1\ (\nu) \Downarrow \lightning \qquad \delta \vdash m_2\ (\nu) \Downarrow \lightning}{\delta \vdash (m_1 \mid m_2)\ (\nu) \Downarrow \lightning}$$

Die folgenden Regeln erweitern den Regelsatz für den Musterabgleich, den wir in Abschnitt 4.1.2 eingeführt haben. Disjunktive Muster werden von links nach rechts durchprobiert, bis das erste Muster passt.

$$\frac{p_1 \sim \nu \Downarrow \delta}{(p_1 \mid p_2) \sim \nu \Downarrow \delta}$$

$$\frac{p_1 \sim \nu \Downarrow \lightning \qquad p_2 \sim \nu \Downarrow \delta}{(p_1 \mid p_2) \sim \nu \Downarrow \delta}$$

$$\frac{p_1 \sim \nu \Downarrow \lightning \qquad p_2 \sim \nu \Downarrow \lightning}{(p_1 \mid p_2) \sim \nu \Downarrow \lightning}$$

$$\frac{}{(C\ p) \sim (C'\ \nu) \Downarrow \lightning} \quad C \neq C'$$

$$\frac{p \sim \nu \Downarrow \delta}{(C\ p) \sim (C'\ \nu) \Downarrow \delta}\ C = C' \qquad \frac{p \sim \nu \Downarrow \lightning}{(C\ p) \sim (C'\ \nu) \Downarrow \lightning}\ C = C'$$

Der Abgleich mit einem Konstruktormuster schlägt fehl, wenn der Wert einen unterschiedlichen Konstruktor hat oder wenn die jeweiligen Argumente nicht passen. Die statische Semantik stellt sicher, dass der Wert in den letzten beiden Regeln tatsächlich ein Element eines Variantentyps ist.

Vertiefung Der größte Vorteil der erweiterten Fallunterscheidung ist, dass wir Muster — insbesondere Konstruktoranwendungen — schachteln können. Geschachtelte Muster können oft an die Stelle geschachtelter Fallunterscheidungen treten. Sie helfen, den Programmcode kürzer und klarer zu formulieren. Betrachten wir noch einmal die Implementierung der Addition zweier Binärzahlen. Im Prinzip müssen wir nur vier Fälle unterscheiden: Ein Argument ist Null, beide Argumente sind gerade, ein Argument ist ungerade, beide sind ungerade. Die geschachtelte Fallunterscheidung zwingt uns aber insgesamt sieben Fälle zu behandeln und verschleiert zudem die vorhandene Symmetrie. Mit Hilfe disjunktiver Muster können wir sehr viel natürlicher formulieren:

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match (m, n) with
    | (Null,     k       )
    | (k,        Null    ) → k
    | (Even m', Even n')   → Even (add (m', n'))
    | (Even m', Odd n'  )
    | (Odd m',  Even n')   → Odd  (add (m', n'))
    | (Odd m',  Odd n'  )  → Even (succ (add (m', n')))
```

Auch Konstruktoranwendungen können in Mustern geschachtelt werden. Ein schönes Beispiel hierfür liefert die folgende Implementierung der Funktion *ordered*, die überprüft, ob eine Liste von natürlichen Zahlen aufsteigend geordnet ist. (Zur Erinnerung: *sort* bringt eine Liste in eine aufsteigende Reihenfolge.)

```
let rec ordered (nats : Nats) : Bool =
  match nats with
    | Nil                          → true
    | Cons (_, Nil)                → true
    | Cons (n1, ns & Cons (n2, _)) → n1 ⩽ n2 && ordered ns
```

Zwei benachbarte Listenelemente müssen jeweils in der Relation '$\leqslant$' stehen. Das Muster $\mathit{Cons}\ (n_1, \mathit{Cons}\ (n_2, \_))$ passt auf eine mindestens zweielementige Liste; die ersten beiden Listenelemente werden an n_1 und n_2 gebunden. Um die Eigenschaft rekursiv für die Restliste überprüfen zu können, vergeben wir mit Hilfe eines konjunktiven Musters zusätzlich einen Namen für die Liste ohne das erste Element: $\mathit{Cons}\ (n_1, ns \,\&\, \mathit{Cons}\ (n_2, \_))$. Da der Rekursionsfall nur mindestens zweielementige Listen behandelt, wird der Fall der einelementigen Liste, $\mathit{Cons}\ (\_, \mathit{Nil})$, zu einem weiteren Basisfall.

Die größere Bequemlichkeit bei der Formulierung von Mustern geht mit einer größeren Verantwortung einher: Die Programmiererin oder der Programmierer muss sicherstellen, dass die Muster tatsächlich auch alle Fälle abdecken. Einfache Fallunterscheidungen stellen dies automatisch sicher. Für die Lesbarkeit von Programmen ist es weiterhin wünschenswert, dass die Muster paarweise disjunkt sind, so dass jedes Muster einen anderen Fall abdeckt. Ist das gewährleistet, dann kann man jeden Zweig einzeln für sich lesen und verstehen. Da die Reihenfolge des Musterabgleichs dann keine Rolle spielt, lassen sich die Zweige auch beliebig umordnen.

4.3. Parametrisierte Typen und Polymorphie

4.3.1. Parametrisierte Typen

Wir haben bisher zwei verschiedene Listentypen eingeführt: Listen von natürlichen Zahlen *Nats* und Listen von Binärziffern *Bits*. Viele andere Listentypen sind denkbar: Listen von Personen, Listen von Adressen, Listen von Listen von natürlichen Zahlen um 2-dimensionale Tabellen zu repräsentieren usw. Für jeden Elementtyp einen neuen Listentyp einzuführen ist mühsam und unökonomisch. Werden in einem Kontext zwei verschiedene Listentypen benötigt, müssen wir uns zudem unterschiedliche Namen für den Typ und insbesondere für die Datenkonstruktoren ausdenken.

Listen sind ein typisches Beispiel für *Behälter* (engl. container). Sie enthalten Daten, die sie in einer bestimmten Art und Weise organisieren. Im Fall von Listen werden die Daten linear angeordnet. Da der Typ der Elemente für die Organisation keine Rolle spielt, liegt es nahe von dem Elementtyp zu abstrahieren und ihn zum formalen Parameter einer Typdefinition zu machen.

```
type List ⟨'a⟩ = | Nil | Cons of 'a * List ⟨'a⟩
```

Der formale *Typparameter* wird in spitze Klammern gesetzt; die *Typvariable* selbst fängt mit einem Apostroph an, um sie von Record- und Variantentypen einfach unterscheiden zu können. Die parametrisierte Typdefinition macht deutlich, dass Listen eine *homogene* Datenstruktur sind: Das Kopfelement hat den gleichen Typ wie die Elemente der Restliste.

Da *List* einen Typparameter hat, entspricht *List* im Prinzip einer Funktion auf Typen. Wenden wir *List* auf einen konkreten Typ an, so erhalten wir einen speziellen Listentyp: *List* ⟨*Nat*⟩ umfasst Listen von natürlichen Zahlen, *List* ⟨*Bit*⟩ entsprechend Listen von Binärziffern. Aktuelle *Typparameter* werden ebenfalls in spitze Klammern gesetzt, um sie von Werteparametern auf den ersten Blick unterscheiden zu können. Die Typen *List* ⟨*Nat*⟩ und *List* ⟨*Bit*⟩ können wie gewohnt in Typangaben verwendet werden.

```
let rec append (list1 : List ⟨Nat⟩, list2 : List ⟨Nat⟩) : List ⟨Nat⟩ =
  match list1 with
  | Nil          → list2
  | Cons (x, xs) → Cons (x, append (xs, list2))
```

Wir formalisieren im Folgenden nur Typdefinitionen mit *einem* Typparameter. Alle Konstrukte verallgemeinern sich aber in naheliegender Weise auf n Typparameter.

Abstrakte Syntax Wir erweitern Deklarationen um parametrisierte Recordtyp- und Variantentypdefinitionen.

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***type*** $T\,\langle {'a} \rangle = \{\ell_1 : t_1; \ell_2 : t_2\}$	parametrisierter Recordtyp
$\mid$ ***type*** $T\,\langle {'a} \rangle = \mid C_1$ ***of*** $t_1 \mid C_2$ ***of*** t_2	parametrisierter Variantentyp

Die Typvariablen auf der rechten Seite einer Typdefinition müssen auf der linken Seite eingeführt werden: ***type*** $List = \mid Nil \mid Cons$ ***of*** ${'a} * List$ ist *nicht* zulässig.

Die Kategorie der Typen wird um Typvariablen und Typapplikationen erweitert.

${'a} \in \mathsf{TyVar}$

$t ::= \cdots$	*Typen:*
$\mid {'a}$	Typvariable
$\mid T\,\langle t \rangle$	Typapplikation

Mit der Einführung parametrisierter Typen sind nicht mehr alle Typausdrücke wohlgeformt und müssten im Prinzip einer statischen Prüfung unterzogen werden: Zum Beispiel sind $Bool\,\langle Nat \rangle$ und $List \to Nat$ unsinnig; $Bool$ wird mit einem Parameter versorgt, erwartet aber keinen; $List$ hingegen benötigt einen Parameter, erhält aber keinen. Wir wollen die Formalisierung an dieser Stelle nicht zu weit treiben und sehen von einer präzisen Definition von „wohlgetypten Typausdrücken“ ab.

Statische Semantik Wir beschränken uns im Folgenden auf die Formalisierung von parametrisierten Variantentypen.

Bei einer Konstruktoranwendung wird der Typparameter des Variantentyps mit einem konkreten Typ instantiiert: $Cons\ (1, Nil)$ zum Beispiel hat den Typ $List\,\langle Nat \rangle$, $Cons\ (false, Nil)$ hat den Typ $List\,\langle Bool \rangle$. Der formale Typparameter aus der Definition des Variantentyps wird jeweils durch den aktuellen Typparameter ersetzt.

Die textuelle Ersetzung von Typvariablen durch Typen nennt man *Typsubstitution*. Ist t ein Typ und $\sigma \in \mathsf{TyVar} \to_{\text{fin}} \mathsf{Type}$ eine endliche Abbildung von Typvariablen auf Typen, dann bezeichnet $t\sigma$ den Typ, in dem die Typvariablen aus $dom\ \sigma$ durch die zugeordneten Typen ersetzt werden.

$$
\begin{aligned}
{'a}\sigma &= \begin{cases} \sigma({'a}), & \text{falls } {'a} \in dom\ (\sigma) \\ {'a}, & \text{sonst} \end{cases} \\
T\sigma &= T \\
(t_1 * t_2)\sigma &= t_1\sigma * t_2\sigma \\
(t_1 \to t_2)\sigma &= t_1\sigma \to t_2\sigma \\
(T\,\langle t \rangle)\sigma &= T\,\langle t\sigma \rangle
\end{aligned}
$$

Für den parametrisierten Variantentyp

$$\textbf{\textit{type}}\ T\ \langle {}'a \rangle = \ \mid\ C_1\ \textbf{\textit{of}}\ t_1 \mid\ C_2\ \textbf{\textit{of}}\ t_2$$

muss die Typregel für die Konstruktoranwendung wie folgt angepasst werden.

$$\frac{\Sigma \vdash e : t_i\{{}'a \mapsto t\}}{\Sigma \vdash C_i\ e : T\ \langle t \rangle}$$

Diese Typregel verdient eine nähere Betrachtung. Wenn C_i den Typ $t_i \to T\ \langle {}'a \rangle$ hat, dann erhält C_i e den Typ $T\ \langle t \rangle$, sofern e den passenden Typ, nämlich $t_i\{{}'a \mapsto t\}$, hat. Der Typparameter t kann frei gewählt werden und somit kann ein einzelner Ausdruck mehrere, ja unendlich viele Typen besitzen. Dies ist unter anderem dann der Fall, wenn ${}'a$ in t_i nicht auftritt, wie zum Beispiel im Fall der leeren Liste *Nil*. (Zur Erinnerung: Ein nullstelliger Konstruktor entspricht einem einstelligen Konstruktor mit dem Argumenttyp *Unit*.) Der Datenkonstruktor *Nil* hat den Typ $List\ \langle t \rangle$ für einen beliebigen Grundtyp t: In *Cons* $(1, Nil)$ zum Beispiel hat *Nil* den Typ $List\ \langle Nat \rangle$, in *Cons* $(false, Nil)$ entsprechend $List\ \langle Bool \rangle$. Der Ausdruck *Cons* (Nil, Nil) hat hingegen unendlich viele Typen: $List\ \langle List\ \langle t \rangle \rangle$ für einen beliebigen Typ t.

$$\frac{\dfrac{\overline{\Sigma \vdash Nil : List\ \langle t \rangle} \qquad \overline{\Sigma \vdash Nil : List\ \langle List\ \langle t \rangle \rangle}}{\Sigma \vdash (Nil, Nil) : List\ \langle t \rangle * List\ \langle List\ \langle t \rangle \rangle}}{\Sigma \vdash Cons\ (Nil, Nil) : List\ \langle List\ \langle t \rangle \rangle}$$

Wir sehen, das erste Vorkommen von *Nil* hat den Typ $List\ \langle t \rangle$, das zweite den Typ $List\ \langle List\ \langle t \rangle \rangle$. Der Ausdruck *Cons* (Nil, Nil) ist eine einelementige Liste, dessen einziges Element die leere Liste ist.

Die Regel für die Fallunterscheidung ändert sich nicht wesentlich, nur dass die Buchhaltung etwas aufwändiger ist (die Typvariable ${}'a$ ist der bei der Deklaration von T spezifizierte Typparameter).

$$\frac{\Sigma \vdash e : T\ \langle t \rangle \qquad \Sigma, \{x_1 \mapsto t_1\{{}'a \mapsto t\}\} \vdash e_1 : t' \qquad \Sigma, \{x_2 \mapsto t_2\{{}'a \mapsto t\}\} \vdash e_2 : t'}{\Sigma \vdash (\textbf{\textit{match}}\ e\ \textbf{\textit{with}} \mid\ C_1\ x_1 \to e_1 \mid\ C_2\ x_2 \to e_2) : t'}$$

Den Bezeichnern x_1 und x_2 werden Instanzen der Argumenttypen von C_1 und C_2 zugeordnet: Hat der Diskriminatorausdruck e zum Beispiel den Typ $List\ \langle Bit \rangle$, dann erhält der Bezeichner x in *Cons* x den Typ $Bit * List\ \langle Bit \rangle$.

Dynamische Semantik Die dynamische Semantik ändert sich nicht.

4.3.2. Polymorphie

Parametrisierte Typen erhöhen — wie auch Funktionen — die Wiederverwendbarkeit von Programmen. Ein parametrisierter Variantentyp wie zum Beispiel *List* kann in vielen verschiedenen Kontexten verwendet werden; im Fall von Listen überall da, wo Folgen

von Elementen verwaltet werden müssen. Eine einzige Definition, unendlich viele Verwendungsmöglichkeiten. Kurioserweise halten Funktionen auf parametrisierten Typen mit dieser Entwicklung nicht Schritt. Betrachten wir einmal die Funktion *length*, die die Länge einer Liste bestimmt.

```
let rec length (list : List ⟨Nat⟩) : Nat =
  match list with
    | Nil          → 0
    | Cons (_, xs) → 1 + length xs
```

Der Typ des Parameters, $List\,\langle Nat\rangle$, schränkt die Anwendung von *length* auf Listen von *natürlichen Zahlen* ein. Um zum Beispiel die Länge einer Liste von *Personen* zu bestimmen, müssen wir eine zweite Funktion programmieren.

```
let rec length (list : List ⟨Person⟩) : Nat =
  match list with
    | Nil          → 0
    | Cons (_, xs) → 1 + length xs
```

Diese Definition ist baugleich zur ersten, der Rumpf ist sogar identisch, nur die Typangabe hat sich geändert. Dabei spielt für das Ausrechnen der Listenlänge der Typ der Elemente gar keine Rolle — die anonyme Variable „_“ in dem Muster $Cons\ (\_, xs)$ macht ja sogar explizit, dass die Elemente ignoriert werden.

Um parametrisierte Typen in ihrer vollen Schönheit genießen zu können, drängt es sich auf, auch Funktionen mit Typen zu parametrisieren.

```
let rec length ⟨'a⟩(list : List ⟨'a⟩) : Nat =
  match list with
    | Nil          → 0
    | Cons (_, xs) → 1 + length xs
```

Die Funktion *length* hat jetzt zwei Parameter: einen Typparameter ($'a$) und einen Werteparameter (*list*). Der Typparameter wird verwendet, um den Typ des Werteparameters, sprich den Elementtyp der Liste, festzulegen. Wird die Funktion *length* aufgerufen, müssen für die formalen Parameter entsprechend aktuelle Parameter angegeben werden: einen Typ und eine Liste mit Elementen des angegebenen Typs. Der Aufruf $length\,\langle Person\rangle staff$ bestimmt zum Beispiel die Länge einer Liste von Personen, $length\,\langle Nat\rangle nats$ entsprechend die Länge einer Liste von natürlichen Zahlen. Eine einzige Definition, unendliche viele Anwendungen.

Die Längenfunktion ist natürlich sehr speziell, da sie die Listenelemente gar nicht benötigt. Wenn wir zurückschauen, erkennen wir aber, dass viele Funktionen allgemeiner sind, als die Typangaben es vermuten lassen und die von einer Parametrisierung mit Typen profitieren. Zum Beispiel die Funktion *peano-pattern* aus Abschnitt 3.6:

```
let peano-pattern ⟨'soln⟩(zero : 'soln, succ : 'soln → 'soln) : Nat → 'soln =
```

Bei der Diskussion der Funktion haben wir besprochen, dass der ursprüngliche Typ zu speziell ist. Jetzt haben wir das Sprachmittel in der Hand, um dieses Manko zu beseitigen: Der spezielle Typ wird durch den Typparameter $'soln$ ersetzt. Wir können das fleischgewordene Entwurfsmuster zum Beispiel verwenden, um eine natürliche Zahl in eine der Zahlendarstellungen aus Abschnitt 4.2.2 zu überführen.

$$\mathit{peano\text{-}pattern}\ \langle \mathit{Peano} \rangle\ \ (\mathit{Zero}, \textbf{\textit{fun}}\ n \to \mathit{Succ}\ n)$$
$$\mathit{peano\text{-}pattern}\ \langle \mathit{Leibniz} \rangle (\mathit{Null}, \textbf{\textit{fun}}\ n \to \mathit{succ}\ n)$$

Es gibt zahlreiche weitere Beispiele für Funktionen, die sich verallgemeinern lassen:

$$\textbf{\textit{let}}\ \mathit{swap}\ \langle 'a, 'b \rangle (x : 'a, y : 'b) : 'b * 'a$$
$$\textbf{\textit{let}}\ \mathit{put\text{-}last}\ \langle 'a \rangle (\mathit{xs} : \mathit{List}\ \langle 'a \rangle, x : 'a) : \mathit{List}\ \langle 'a \rangle$$
$$\textbf{\textit{let}}\ \mathit{append}\ \langle 'a \rangle (\mathit{xs}_1 : \mathit{List}\ \langle 'a \rangle, \mathit{xs}_2 : \mathit{List}\ \langle 'a \rangle) : \mathit{List}\ \langle 'a \rangle$$
$$\textbf{\textit{let}}\ \mathit{reverse}\ \langle 'a \rangle (\mathit{xs} : \mathit{List}\ \langle 'a \rangle) : \mathit{List}\ \langle 'a \rangle$$
$$\textbf{\textit{let}}\ \mathit{sort\text{-}by}\ \langle 'a \rangle (\mathit{less\text{-}equal} : 'a * 'a \to \mathit{Bool}) : \mathit{List}\ \langle 'a \rangle \to \mathit{List}\ \langle 'a \rangle$$

Die Funktion *swap* abstrahiert über zwei Typparameter. Interessanterweise nageln die Typangaben die Implementierung der Funktion eindeutig fest.

$$\textbf{\textit{let}}\ \mathit{swap}\ \langle 'a, 'b \rangle (x : 'a, y : 'b) : 'b * 'a = (y, x)$$

Der Rumpf muss die Form (y, x) haben; Tippfehler wie (x, y), (x, x), $(x, 0)$ usw. fallen samt und sonders durch die statische Typprüfung.

Besonders beeindruckend ist die Verallgemeinerung von *sort-by*. Jetzt können wir beliebige Listen sortieren, sofern wir nur eine Ordnungsrelation, eine Vergleichsfunktion, auf den Listenelementen angeben können. Listen von Personen können zum Beispiel wie folgt nach ihrem Nachnamen geordnet werden.

$$\mathit{sort\text{-}by}\ \langle \mathit{Person} \rangle (\textbf{\textit{fun}}\ (\mathit{ann}, \mathit{bob}) \to \mathit{ann}.\mathit{surname} \leqslant \mathit{bob}.\mathit{surname})$$

Funktionen, die mit einem oder mehreren Typen parametrisiert sind, heißen auch *polymorphe* Funktionen nach dem griechischen Wort $\pi o \lambda \upsilon \mu o \rho \phi \iota \alpha$ für Vielgestaltigkeit.

Pragmatik Wir sehen davon ab, Syntax und Semantik polymorpher Funktionen formal zu definieren. Stattdessen geben wir uns pragmatisch und erlauben umgekehrt sogar *Typparameter auszulassen*: sowohl bei der Definition polymorpher Funktionen

$$\textbf{\textit{let rec}}\ \mathit{length}\ (\mathit{list} : \mathit{List}\ \langle 'a \rangle) : \mathit{Nat} = \ldots$$
$$\textbf{\textit{let}}\ \mathit{sort\text{-}by}\ (\mathit{less\text{-}equal} : 'a * 'a \to \mathit{Bool}) : \mathit{List}\ \langle 'a \rangle \to \mathit{List}\ \langle 'a \rangle = \ldots$$

als auch bei der Anwendung polymorpher Funktionen

$$\mathit{sort\text{-}by}\ (\textbf{\textit{fun}}\ (m, n) \to m \leqslant n)$$
$$\mathit{sort\text{-}by}\ (\textbf{\textit{fun}}\ (\mathit{ns1}, \mathit{ns2}) \to \mathit{length}\ \mathit{ns1} \leqslant \mathit{length}\ \mathit{ns2})$$

Insbesondere letzteres stellt eine erhebliche Schreiberleichterung dar.

Tatsächlich ist es möglich, auch die Typen von Funktionsparametern und -ergebnissen auszulassen — die fehlenden Informationen werden automatisch inferiert! Wir können sehr flexibel entscheiden, wie viele Typinformationen wir bereitstellen:

let rec *length* $\langle'a\rangle$ $(xs : List\ \langle'a\rangle) : Nat =$ ***match*** *xs* ***with*** ...

let rec *length* $(xs : List\ \langle'a\rangle) : Nat =$ ***match*** *xs* ***with*** ...

let rec *length* $(xs : List\ \langle'a\rangle) =$ ***match*** *xs* ***with*** ...

let rec *length* $xs =$ ***match*** *xs* ***with*** ...

let rec *length* $=$ ***function*** ...

Der Ausdruck ***function*** m ist eine beliebte Abkürzung für ***fun*** $x \rightarrow$ ***match*** x ***with*** m. Beliebt, weil man sich keinen Namen für den Funktionsparameter ausdenken muss — neue Namen zu erfinden ist bekanntlich schwer.

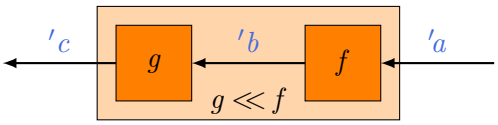
Typinferenz erhöht nicht nur die Bequemlichkeit beim Programmieren, sondern führt gelegentlich auch zu angenehmen Überraschungen, nämlich dann, wenn der inferierte Typ einer Funktion allgemeiner ist als erwartet. Schauen wir uns ein Beispiel an: Die Komposition von Funktionen, $f \circ g$, lässt sich in Mini–F# wie folgt definieren.

let *compose* $g\ f\ x = g\ (f\ x)$

Entgegen unserer sonstigen Gepflogenheiten haben wir weder die Typen der Argumente noch den Typ des Resultats angegeben. Den Mini–F# Interpreter ficht diese Nachlässigkeit nicht an; er inferiert den folgenden Typ.

$$compose : (\,'b \rightarrow\, 'c) \rightarrow (\,'a \rightarrow\, 'b) \rightarrow (\,'a \rightarrow\, 'c)$$

Der hergeleitet Typ enthält drei Typvariablen: $'a$, $'b$ und $'c$. Wenn wir die Funktion g mit f komponieren, dann muss lediglich der Ergebnistyp von f mit dem Argumenttyp von g übereinstimmen; alle anderen Typen sind beliebig. Die Funktionskomposition ist übrigens als Infixoperator vordefiniert: $g \ll f$. *Lies:* g nach f — der „Pfeil“ $\ll$ zeigt an, dass das Ergebnis von f in die Funktion g eingespeist wird.



Da die Komposition hochgradig polymorph ist, wäre eine verpflichtende Angabe von Typparametern fatal. Statt *compose reverse sort* oder infix *reverse* $\ll$ *sort* müsste man länglicher formulieren: *compose* $\langle List\ \langle Nat\rangle, List\ \langle Nat\rangle, List\ \langle Nat\rangle\rangle (reverse\ \langle Nat\rangle)$ *sort*, wahrscheinlich mit dem Ergebnis, dass man *compose* erst gar nicht verwenden würde.

Die Bequemlichkeit der Typinferenz birgt natürlich auch eine Gefahr: Typen gleich vollständig auszulassen. *Zur Erinnerung:* Typen dienen der Dokumentation und sind eine wichtige Hilfe bei der systematischen Erstellung von Programmen (Stichwort: Entwurfsmuster). Wie so oft im Leben gilt es, eine Balance zwischen zwei Extremen zu finden: der vollständigen Angabe aller Typen und dem vollständigen Weglassen aller Typangaben.

Vertiefung Listen so wie wir sie definiert haben, erinnern an Stapel (engl. stacks), zum Beispiel Akten- oder Tellerstapel. Wir können einfach und schnell auf das erste bzw. oberste Element zugreifen. Wenn wir das n-te Elemente benötigen, müssen wir Aufwand betreiben, sprich, wir müssen eine entsprechende Funktion definieren.

$$\textbf{\textit{let rec}}\ \mathit{nth}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle, n : \mathit{Nat}) : 'a =$$

Für den zweiten Parameter, den Index, sollte gelten: $0 \leqslant n < \mathit{length}\ \mathit{list}$, eine Eigenschaft, die wir in Mini–F# selbst nicht ausdrücken können. Was machen wir, wenn die Liste weniger als n Elemente enthält? Eine Möglichkeit ist, die beiden möglichen Resultate des Zugriffs, erfolglos und erfolgreich, mit einem Datentyp darzustellen.

$$\textbf{\textit{type}}\ \mathit{Option}\ \langle'a\rangle = \ |\ \mathit{None}\ |\ \mathit{Some}\ \textbf{\textit{of}}\ 'a$$

Wie der Name des Typs andeutet, können wir damit optionale Werte repräsentieren. Der Typ $\mathit{Option}\ \langle\mathit{Bool}\rangle$ enthält zum Beispiel drei Elemente: None, $\mathit{Some}\ \mathit{False}$ und $\mathit{Some}\ \mathit{True}$. Der Variantentyp Option ist wie List ein parametrisierter Typ: der Typparameter legt den Typ der Elemente fest, die im Erfolgsfall zurückgegeben werden. Mit diesem neuen Typ können wir die Signatur von nth verfeinern.

$$\textbf{\textit{let rec}}\ \mathit{nth}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle, n : \mathit{Nat}) : \mathit{Option}\ \langle'a\rangle =$$

Für den Fall, dass der Index n zu groß ist, geben wir None zurück, anderenfalls $\mathit{Some}\ x$, wobei x das n-te Element ist. Das Struktur Entwurfsmuster führt unmittelbar zum Ziel:

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{nth}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle, n : \mathit{Nat}) : \mathit{Option}\ \langle'a\rangle = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\ \quad |\ \mathit{Nil} \qquad\quad \to \mathit{None} \\ \quad |\ \mathit{Cons}\ (x, xs) \to \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ \mathit{Some}\ x \\ \qquad\qquad\qquad\qquad\quad \textbf{\textit{else}}\ \ \mathit{nth}\ (xs, n \dot{-} 1)\end{array}$$

Die Rekursionsbasis ist zwingend: Wenn die Liste leer ist, dann ist der Index zu groß — kein Index n erfüllt $0 \leqslant n < \mathit{length}\ \mathit{list} = 0$ — und wir geben None zurück. Im Rekursionsschritt führen wir eine zusätzliche Fallunterscheidung über n. (Eine Bitte: Die Funktion nth sollte nicht *missbraucht* werden, um über alle Element einer Liste zu iterieren, siehe Aufgabe 4.15.)

Über den Tellerrand Listen sind eine der beliebtesten Datenstrukturen. Sie bieten sich bei der Modellierung an, wenn es gilt, Folgen von Objekten, Sequenzen oder Reihungen zu repräsentieren. Listen sind in den meisten höheren Programmiersprachen vordefiniert und werden entsprechend gut unterstützt. F# bildet da keine Ausnahme. Die vordefinierten Listen unterscheiden sich lediglich in der Syntax, nicht aber in der Semantik. Man schreibt

- $T\ \mathit{list}$ statt $\mathit{List}\ \langle T\rangle$ — an die Stelle der Präfixnotation tritt die Postfixnotation;
- $[\,]$ statt Nil; und

- $x :: xs$ statt $Cons\ (x, xs)$ — an die Stelle der Präfixnotation tritt die Infixnotation.

Darüber hinaus gibt es eine Menge „*syntaktischen Zucker*“, der das Programmieren erleichtert und versüsst: $[x_1; x_2; x_3; x_4; x_5; x_6]$ ist zum Beispiel eine kompakte Notation für die Liste $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: [])))))$; die Intervallschreibweise $[l\,..\,u]$ erzeugt die Liste aller Elemente von l bis einschließlich u. Insbesondere letztere Abkürzung ist beim Testen von listenverarbeitenden Funktionen sehr nützlich.

Wir werden in den folgenden Kapiteln den vordefinierten Listen den Vorzug geben, einfach weil die Syntax angenehmer ist. Lassen Sie uns deshalb ein paar listenverarbeitende Funktionen in F# re-implementieren, damit wir uns an die Syntax gewöhnen. Die Funktion *nth* bestimmt das n-te Element einer Liste:

```
let rec nth (list : 'a list, n : int) : 'a option =
  match list with
  | []      → None
  | x :: xs → if n = 0 then Some x
                       else  nth (xs, n - 1)
```

Um auf das n-te Element der Liste xs zuzugreifen, kann man auch kurz $xs.[n]$ schreiben. Im Unterschied zu *nth* wird das Element direkt zurückgegeben; ist der Index allerdings negativ oder zu groß, bricht die Rechnung mit einer Fehlermeldung ab.

Aus historischen Gründen werden vordefinierte, parametrisierte Typen — dazu gehört auch *option* — postfix notiert: *int option list* ist zum Beispiel eine Liste optionaler ganzer Zahlen (Präfixnotation: $List\,\langle Option\,\langle Int\rangle\rangle$); *int list option* ist hingegen eine optionale Liste (Präfixnotation: $Option\,\langle List\,\langle Int\rangle\rangle$).

Die Funktion *append* konkateniert zwei Listen:

```
let rec append list y =
  match list with
  | []      → y
  | x :: xs → x :: append xs y
```

Die Listenkonkatenation *append x y* ist als Infixoperator vordefiniert: $x @ y$, zum Beispiel ist $[4; 7] @ [1; 1] = [4; 7; 1; 1]$. (In der Definition haben wir keine Typen angegeben. Was ist der Typ von *append* bzw. @?)

Die Funktion *contains* ermittelt, ob eine Liste ein gegebenes Element enthält. Was „$\in$“ für Mengen ist, das ist *contains* für Listen.

```
let rec contains key = function
  | []      → false
  | a :: as → key = a || contains key as
```

(*Zum Knobeln:* was ist der Typ von *contains*? Für die Antwort müssten wir weiter ausholen, was wir an dieser Stelle nicht tun wollen.)

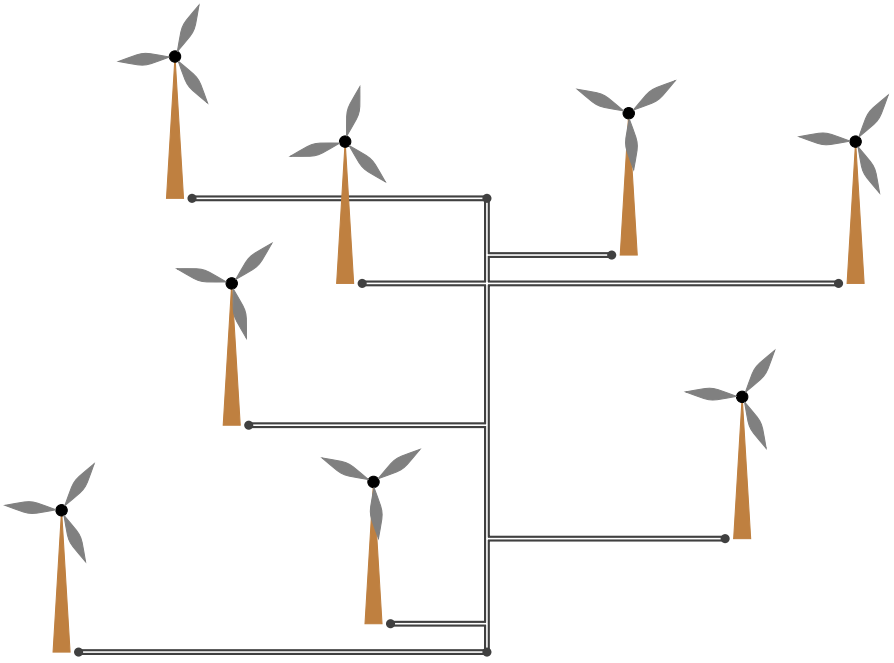


Abbildung 4.3.: Windenergieanlagen in Rheinland-Pfalz.

4.3.3. Anwendung: Planung von Stromtrassen

Prof. Ventum hat einen lukrativen Beratervertrag mit der Orkankraft GmbH und Co. KG abgeschlossen, die Stromtrassen für die Vernetzung von Windenergieanlagen plant. Bereits bestehende Anlagen, siehe Abbildung 4.3, sollen mit einer Nord-Süd-Trasse vernetzt werden, wobei jede Windkraftanlage über eine West-Ost-Trasse angebunden wird. Prof. Ventum soll herausfinden, wie die Position der Nord-Süd-Trasse gewählt werden muss, so dass die Gesamtlänge der Leitungen möglichst klein wird.

Die Position der Anlagen ist durch eine *nicht-leere* Liste von Punkten gegeben. Jeder Punkt wird durch eine x- und eine y-Koordinate festgelegt.

$$
\begin{array}{l}
\textbf{\textit{type}}\ \mathit{Point} = \{x : \mathit{Nat}; y : \mathit{Nat}\} \\
\textbf{\textit{let}}\ \mathit{wind\text{-}farm} = \\
\quad [\{x = 2;\ \ y = 2\};\ \ \{x = 6;\ \ y = 18\}; \{x = 8;\ \ y = 10\}; \{x = 13; y = 3\}; \\
\quad\ \{x = 12; y = 15\}; \{x = 26; y = 6\};\ \ \{x = 22; y = 16\}; \{x = 30; y = 15\}]
\end{array}
$$

Wir nehmen vereinfachend an, dass die y-Koordinaten unterschiedlich sind, so dass jede Anlage über eine separate Trasse angeschlossen werden muss.

Prof. Ventum ist schnell klar, dass sich die Aufgabe in zwei Teilaufgaben zergliedern lässt: (1) die Bestimmung der Länge der Nord-Süd-Trasse; (2) die *Optimierung* der Längen der West-Ost-Trassen. Gehen wir beide Teilaufgaben nacheinander an.

Länge der Nord-Süd-Trasse Für die erste Teilaufgabe müssen die Windkraftanlagen mit der kleinsten bzw. der größten y-Koordinate ermittelt werden. Prof. Ventum verallgemeinert die Aufgabe etwas und programmiert Funktionen, die die extremalen Elemente einer Liste bezüglich einer beliebigen Prioritätsfunktion bestimmen.

```
let rec minBy (priority : 'a → Nat) : 'a list → 'a = function
  | [x]    → x
  | x :: xs → let m = minBy priority xs in
              if priority x ⩽ priority m then x else m
let rec maxBy (priority : 'a → Nat) : 'a list → 'a = function
  | [x]    → x
  | x :: xs → let m = maxBy priority xs in
              if priority x ⩽ priority m then m else x
```

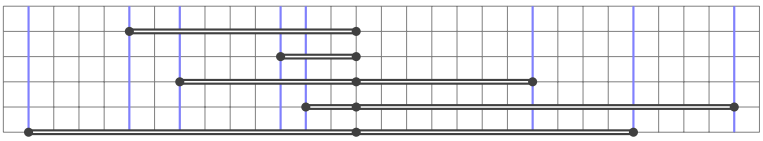
Es fällt auf, dass die Funktionen sich nicht an das Struktur Entwurfsmuster für Listen halten: Der jeweilige Basisfall löst nicht das Problem für die leere Liste, sondern für eine einelementige Liste. Mit anderen Worten, sowohl *minBy f* [] als auch *maxBy f* [] sind undefiniert. Warum? Betrachten wir den Typ der Funktion *minBy* genauer: Die Funktion erwartet als Argument eine „Prioritätsfunktion“ vom Typ $'a \to Nat$ und gibt als Ergebnis eine „Auswahlfunktion“ vom Typ $'a\ list \to 'a$ zurück — aus einer gegebenen Liste wird das Element mit der kleinsten Priorität ausgewählt. Ist die Liste nun leer, so steht kein Element zur Auswahl. Und: Wir können kein Element vom Typ $'a$ *erfinden*, da $'a$ ein Typparameter ist und kein konkreter Typ. Der polymorphe Typ $('a \to Nat) \to ('a\ list \to 'a)$ verspricht, dass *minBy* funktioniert, unabhängig davon, welcher konkrete Typ für den Typparameter $'a$ eingesetzt wird (z.B. *Bool*, *Nat*, *Nat list* oder $Nat \to Bool$). Ein Element vom Typ $'a$ zu erfinden, käme schwarzer Magie gleich!

Zurück zum Ausgangsproblem: Die Länge der Nord-Süd-Trasse ergibt sich als Differenz der kleinsten und der größten y-Koordinate.

```
let north-south-route ps =
  (maxBy (fun p → p.y) ps).y - (minBy (fun p → p.y) ps).y
```

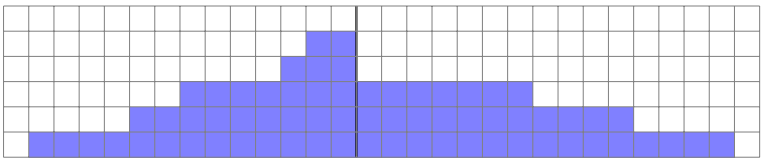
Der Aufruf *maxBy* (*fun* $p \to p.y$) *ps* gibt einen Punkt zurück, dessen y-Koordinate anschließend selektiert wird.

Optimierung der West-Ost-Trassen Wenden wir uns dem Optimierungsproblem zu: Wie muss die Position der Nord-Süd-Trasse gewählt werden, so dass die Gesamtlänge der West-Ost-Trassen möglichst klein wird? Prof. Ventum überlegt sich zunächst, dass die vertikale Position der Anlagen für diese Frage keine Rolle spielt. Wenn wir die West-Ost-Trassen entsprechend zusammenschieben, ergibt sich ein aufschlussreiches Bild.

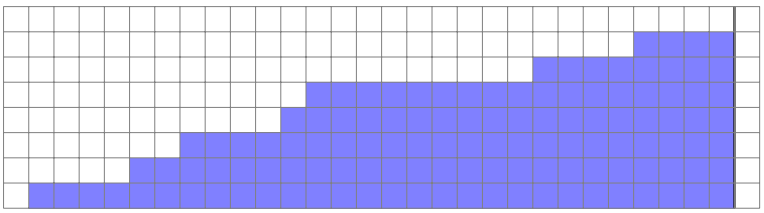


Die x-Achse lässt sich gemäß der x-Koordinaten der Anlagen in verschiedene Intervalle unterteilen. Die Grafik zeigt, dass das Intervall unmittelbar links von der Nord-Süd-Trasse insgesamt 5-mal durchquert wird — 5-mal, da 5 Anlagen links von der Trasse liegen. Das nächste Intervall zur Linken wird 4-mal durchquert usw.

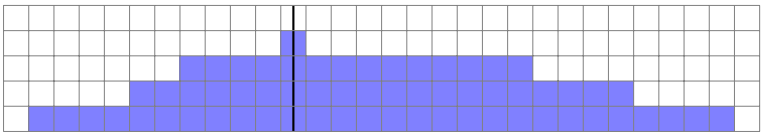
Trägt man auf der y-Achse die Gesamtzahl der Durchquerungen auf, erhält man eine Treppenfunktion. Die Fläche, die die Funktion mit der x-Achse einschließt, entspricht der Gesamtlänge der West-Ost-Trassen.



Wenn wir die Nord-Süd-Trasse weiter nach rechts verschieben, dann vergrößert sich der Flächeninhalt, da die Treppe höher und höher wird. Das folgende Diagramm zeigt den schlechtesten Fall: Die Trasse liegt am rechten Rand.



Das optimale Bild ergibt sich, wenn auf der linken Seite genauso viele Anlagen liegen, wie auf der rechten Seite.



Man beachte, dass die gesuchte Position *weder* die geometrische Mitte *noch* das arithmetische Mittel der x-Positionen ist. Wir suchen stattdessen den Median der x-Positionen. Ist die Folge x_0, …, x_{n-1} mit $n > 0$ der Größe nach geordnet, so is $x_{\lfloor (n-1)/2 \rfloor}$ der sogenannte *Untermedian* und $x_{\lceil (n-1)/2 \rceil}$ der *Obermedian*. Wenn die Gesamtzahl n der Elemente ungerade ist, fallen Unter- und Obermedian zusammen und man spricht kurz vom *Median*.[9] Für unsere Anwendung spielt es keine Rolle, ob der Unter- oder der Obermedian verwendet wird oder ein beliebiger Punkt dazwischen. Der (Unter-) Median einer beliebigen Folge lässt sich bestimmen, indem wir die Folge sortieren und dann das mittlere Element herausgreifen.

$$\textbf{\textit{let}}\ median\ xs = (sort\ xs).[(length\ xs - 1) \div 2]$$

Um die Gesamtlänge der West-Ost-Trassen zu berechnen, müssen die Abstände der Windkraftanlagen vom Median aufsummiert werden. Prof. Ventum liebt Abstraktion und programmiert eine Funktion, die etwas allgemeiner ist.

[9] Wenn die Folgenelemente rationale oder reelle Zahlen sind, wird der Median auch als arithmetisches Mittel von Unter- und Obermedian definiert.

```
let rec sum-by (cost : 'a → Nat) : 'a list → Nat = function
  | []      → 0
  | x :: xs → cost x + sum-by cost xs
```

Die Funktion *sum-by* erwartet als Argument eine „Kostenfunktion“ vom Typ $'a \to Nat$ und gibt als Ergebnis eine „Summationsfunktion“ vom Typ $'a\ list \to Nat$ zurück — diese ermittelt die Gesamtkosten für eine gegebene Liste von Elementen. Der Funktionsaufruf *sum-by* (*fun* $x \to f\ x$) *xs* entspricht in etwa der Summationsformel $\sum_{x \in X} f(x)$ — „in etwa“ weil *xs* eine Liste von Elementen ist, X hingegen eine Menge. (In einer Liste können Elemente mehrfach auftreten und werden entsprechend mehrfach addiert; in einer Menge ist das nicht der Fall.) Auch *sum-by* ist eine Funktion höherer Ordnung und als solche vielseitig einsetzbar: *sum-by id* summiert eine Liste von natürlichen Zahlen, *sum-by* (*sum-by id*) eine Liste von Listen von natürlichen Zahlen, *sum-by* (*sum-by* (*sum-by id*)) eine Liste von Listen von Listen von natürlichen Zahlen usw. Ach ja, *id* ist die sogenannte *Identitätsfunktion* *fun* $x \to x$.

Zurück zur Trassenplanung: Die optimale Länge der West-Ost-Trassen lässt sich nach diesen Vorarbeiten wie folgt bestimmen.

```
let west-east-routes ps =
  let m = (median ps).x
  (m, sum-by (fun p → (p.x ∸ m) + (m ∸ p.x)) ps)
```

Für unser ursprüngliches Beispiel ergeben sich die folgenden Werte.

```
Mini⟩ north-south-route wind-farm
16
Mini⟩ west-east-routes wind-farm
(12, 63)
```

Die Zahl 12 ist der Untermedian der x-Koordinaten: Drei Koordinaten sind kleiner (2, 6 und 8) und vier Koordinaten sind größer (13, 22, 26 und 30). Die Gesamtlänge der Trassen beläuft sich auf $16 + 63 = 79$ km.

4.4. Arrays⟍Felder⟍Reihungen

To denote the subsequence of natural numbers 2, 3, ..., 12 *without the pernicious three dots, four conventions are open to us*

a) $2 \leqslant i < 13$

b) $1 < i \leqslant 12$

c) $2 \leqslant i \leqslant 12$

d) $1 < i < 13$

Are there reasons to prefer one convention to the other? Yes, there are. The observation that conventions a) *and* b) *have the advantage that the difference between the bounds as mentioned equals the length of the subsequence is valid. So is the observation that, as a consequence, in either convention two subsequences are adjacent means that the upper bound of the one equals the lower bound of the other. Valid as these observations are, they don't enable us to choose between* a) *and* b)*; so let us start afresh.*

There is a smallest natural number. Exclusion of the lower bound as in b) *and* d) *forces for a subsequence starting at the smallest natural number the lower bound as mentioned into the realm of the unnatural numbers. That is ugly, so for the lower bound we prefer the* $\leqslant$ *as in* a) *and* c)*. Consider now the subsequences starting at the smallest natural number: inclusion of the upper bound would then force the latter to be unnatural by the time the sequence has shrunk to the empty one. That is ugly, so for the upper bound we prefer* $<$ *as in* a) *and* d)*. We conclude that convention* a) *is to be preferred.*

[...]

When dealing with a sequence of length N*, the elements of which we wish to distinguish by subscript, the next vexing question is what subscript value to assign to its starting element. Adhering to convention* a) *yields, when starting with subscript 1, the subscript range* $1 \leqslant i < N + 1$*; starting with 0, however, gives the nicer range* $0 \leqslant i < N$*. So let us let our ordinals start at zero: an element's ordinal (subscript) equals the number of elements preceding it in the sequence. And the moral of the story is that we had better regard — after all those centuries! — zero as a most natural number.*

— Edsger W. Dijkstra (1930–2002), *Why numbering should start at zero — EWD831*

Lassen wir noch einmal die Möglichkeiten, Daten zu aggregieren, Revue passieren. Um eine *feste* Anzahl von Daten *verschiedenen* Typs zusammenzufassen, können wir Tupel bzw. Records verwenden. Um eine *beliebige* Anzahl von Daten des *gleichen* Typs zusammenzufassen, können wir Listen benutzen. Die verschiedenen Typen unterscheiden sich in der Handhabung und bezüglich der Laufzeit elementarer Operationen. Auf eine Komponente eines Tupels bzw. eines Records kann in konstanter Zeit zugegriffen werden, auf ein Element einer Liste in linearer Zeit. Auf der anderen Seite kann der Zugriff auf eine Tupel- bzw. Recordkomponente nicht berechnet werden: Das Label ℓ in $e.\ell$ kann nicht das Ergebnis einer Rechnung sein. Damit gibt es keine Möglichkeit, eine *rekursive* Funktion zu schreiben, die zum Beispiel alle Komponenten eines n-Tupels oder Records aufaddiert. Dass für die Projektion kein Ausdruck angegeben werden kann, ist keine künstliche Einschränkung, sondern hat einen handfesten Grund:

Tupel wie auch Records sind *heterogene* Datenstrukturen: Die Komponenten können einen unterschiedlichen Typ besitzen. Wäre die Projektion durch einen Ausdruck gegeben, wie zum Beispiel in $e.(n \,\%\, 2)$, dann würde der *Typ* des gesamten Ausdrucks vom

Wert des Teilausdrucks $n \,\%\, 2$ abhängen. Damit wäre die strikte Trennung zwischen der statischen und der dynamischen Semantik aufgehoben. (Es gibt experimentelle Programmiersprachen, die gerade dies erlauben. So weit wollen wir aber nicht gehen.)

Für Listen gilt diese Einschränkung nicht; sie sind eine *homogene* Datenstruktur: Alle Elemente eines Containers besitzen den gleichen Typ. Der Datentyp Liste ist rekursiv definiert, entsprechend sind auch listenverarbeitende Funktionen in der Regel rekursiv definiert. Das Struktur Entwurfsmuster gibt einen ersten Ansatz vor. Problematisch, weil langsam, sind Funktionen, die *gegen* die Struktur eines Datentyps ankämpfen. Das Prinzip der binären Suche zum Beispiel lässt sich nicht ohne weiteres auf Listen übertragen, da der Zugriff auf das mittlere Element einer Liste teuer ist. (Zur Erinnerung: *nth* (xs, k) bzw. $xs.[k]$ benötigt k Schritte, um das k-te Element der Liste xs zu ermitteln. Wenn wir damit die binäre Suche füttern:

$$\mathit{binary\text{-}search}\ ((\textbf{\textit{fun}}\ k \to \mathit{number} \leqslant \mathit{list}.[k]), 0, \mathit{length}\ \mathit{list} \dot{-} 1)$$

machen wir den Geschwindigkeitsvorteil der binären Suche, $\lg n$ statt n, zunichte. Da das Orakel selbst eine lineare Laufzeit hat, ergibt sich eine Gesamtlaufzeit von $n \lg n$.)

Summa summarum, wir suchen eine Datenstruktur, die sozusagen zwischen Tupeln und Listen angesiedelt ist, die eine beliebige Anzahl von Daten des *gleichen* Typs zusammenfasst, aber einen konstanten Zugriff auf die aggregierten Daten erlaubt. Diese Lücke schließen *Arrays* (auch Felder oder Reihungen genannt). Ein Array kann ähnlich wie ein Tupel durch Aufzählung der Elemente konstruiert werden: $[|\ 2; 3; 5; 7; 11\ |]$ ist ein Array der Größe 5. Im Unterschied zu Tupeln kann der Zugriff auf Elemente berechnet werden: Ist e ein Ausdruck, der zu einem Array auswertet, und e_1 ein arithmetischer Ausdruck, dann kann mit $e.[e_1]$ auf die entsprechende Komponente zugegriffen werden. Wertet etwa e_1 zu 3 aus, dann wird die *vierte* Komponente selektiert. Das liegt daran, dass die Arrayelemente beginnend mit 0 durchnummeriert werden.

Abstrakte Syntax Arrays sind Funktionen nicht unähnlich; die sogenannte *Subskription* $e.[e_1]$ korrespondiert zur Funktionsapplikation $e\ e_1$. Im Unterschied zu Funktionen ist der Definitionsbereich stets *Int* bzw. genauer ein Anfangsstück der natürlichen Zahlen.[10] Auch zur Funktionsabstraktion gibt es ein Gegenstück: $[|\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ 0\,..\,n-1 \to e|]$ konstruiert ein Array der Größe n. Das Array $[|\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,99 \to i * i|]$ zum Beispiel umfasst die ersten hundert Quadratzahlen.

$e ::= \cdots$	*Arrays:*
$\mid\ [|\, e_0; \ldots; e_{n-1} \,|]$	Konstruktion durch Aufzählung
$\mid\ [|\, \textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1\,..\,e_2 \to e_3 \,|]$	Konstruktion durch Bildungsvorschrift
$\mid\ e.[e_1]$	Subskription
$\mid\ e.\mathit{Length}$	Größe eines Arrays

Man sieht: Eckige Klammern mit Strich, $[|$ und $|]$, sind das Markenzeichen der Sprachkonstrukte, die Arrays konstruieren. Das Konstrukt $[|\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1\,..\,e_2 \to e_3|]$ führt den

[10] Da keine negativen Zahlen als Index zulässig sind, sollte eigentlich *Nat* als Definitionsbereich verwendet werden. Aus pragmatischen Gründen rückt Mini–F# hier näher an F# heran, das ganze Zahlen als Indizes vorschreibt.

Bezeichner x neu ein; x ist in e_3 sichtbar. Der Ausdruck e_1 in $e.[\,e_1\,]$ heißt auch Arrayindex oder kurz *Index*.

Wie Listen können auch Arrays beliebig viele Elemente umfassen; insbesondere können sie leer sein oder nur ein Element enthalten: $[|\,|]$ oder $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ 1\,..\,0 \to x\,|]$ konstruieren ein leeres Array; die Ausdrücke $[|\,4711\,|]$ oder $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ 0\,..\,0 \to 4711\,|]$ konstruieren ein einelementiges Array.

Statische Semantik Der Typ eines Arrays ist mit dem Typ der Elemente parametrisiert.

$t ::= \cdots$		*Typen:*
	$\mid\ \mathit{Array}\,\langle t\rangle$	Arraytyp

Die folgenden Typregeln machen noch einmal deutlich, dass ein Array vom Typ $\mathit{Array}\,\langle t\rangle$ zu einer Funktion des Typs $\mathit{Int} \to t$ korrespondiert.

$$\frac{\Sigma \vdash e_i : t \quad \mid \quad i \in \mathbb{N}_n}{\Sigma \vdash [|\,e_0; \ldots; e_{n-1}\,|] : \mathit{Array}\,\langle t\rangle}$$

Die Notation $\phi_i \mid i \in \mathbb{N}_n$ ist eine kompakte Schreibweise für eine Regel mit den n Voraussetzungen ϕ_0, …, ϕ_{n-1}.

$$\frac{\Sigma \vdash e_1 : \mathit{Int} \qquad \Sigma \vdash e_2 : \mathit{Int} \qquad \Sigma, \{x \mapsto \mathit{Int}\} \vdash e_3 : t}{\Sigma \vdash [|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1\,..\,e_2 \to e_3\,|] : \mathit{Array}\,\langle t\rangle}$$

$$\frac{\Sigma \vdash e : \mathit{Array}\,\langle t\rangle \qquad \Sigma \vdash e_1 : \mathit{Int}}{\Sigma \vdash e.[\,e_1\,] : t} \qquad \frac{\Sigma \vdash e : \mathit{Array}\,\langle t\rangle}{\Sigma \vdash e.\mathit{Length} : \mathit{Int}}$$

Für das leere Array gilt Ähnliches wie für die leere Liste: $[|\,|]$ hat den Typ $\mathit{Array}\,\langle t\rangle$ für einen beliebigen Grundtyp t. Entsprechend hat zum Beispiel der Ausdruck $[|\,[|\,|]\,|]$ unendlich viele Typen: $\mathit{Array}\,\langle \mathit{Array}\,\langle t\rangle\rangle$ für einen beliebigen Typ t.

Dynamische Semantik Der Wert eines Arrayausdrucks ist eine endliche Abbildung des Typs $\mathbb{N} \to_{\text{fin}} \mathsf{Val}$, mit anderen Worten eine Sequenz vom Typ Val^*. Entsprechend erweitern wir den Bereich der Werte um Sequenzen von Werten.

$s \in \mathsf{Val}^*$		
$\nu ::= \cdots$		*Werte:*
	$\mid\ s$	Array (-wert)

Ähnlich wie bei Paaren werden bei der Konstruktion eines Arrays zunächst alle Elemente ausgerechnet, dann wird die endliche Abbildung erzeugt.

$$\frac{\delta \vdash e_i \Downarrow \nu_i \quad \mid \quad i \in \mathbb{N}_n}{\delta \vdash [|\,e_0, \ldots, e_{n-1}\,|] \Downarrow \{\,i \mapsto \nu_i \mid i \in \mathbb{N}_n\,\}}$$

$$\frac{\delta \vdash e_1 \Downarrow l \qquad \delta \vdash e_2 \Downarrow u \qquad \delta, \{x \mapsto i\} \vdash e_3 \Downarrow \nu_i \quad \mid \quad i \in \{l\,..\,u\}}{\delta \vdash [|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1\,..\,e_2 \to e_3\,|] \Downarrow \{\,i - l \mapsto \nu_i \mid i \in \{l\,..\,u\}\,\}}$$

$$\frac{\delta \vdash e \Downarrow s \qquad \delta \vdash e_1 \Downarrow i}{\delta \vdash e.[e_1] \Downarrow s(i)} \qquad i < len\ s \qquad \frac{\delta \vdash e \Downarrow s}{\delta \vdash e.Length \Downarrow len\ s}$$

Bei der Konstruktion mittels Bildungsvorschrift werden zunächst die Arraygrenzen ausgerechnet, dann wird der Bezeichner x nacheinander an die Werte l, …, u gebunden und bezüglich jeder Bindung wird der Rumpf e_3 ausgerechnet.

Die Subskription $e.[e_1]$ ist nur definiert, wenn der Index e_1 im Definitionsbereich der endlichen Abbildung liegt. Befindet er sich außerhalb, dann ordnet die dynamische Semantik dem Ausdruck keinen Wert zu — dieses Problem werden wir in Abschnitt 7.4 beheben. (In vielen Programmiersprachen wird *nicht* überprüft, ob der Index innerhalb der Bereichsgrenzen liegt. Die Konsequenzen sind weitreichend; werden die Probleme entdeckt, erscheinen sie oft als sogenannte „Sicherheitslöcher“ in den Kolumnen einschlägiger Zeitschriften.)

Vertiefung Der „Definitionsbereich“ des Arrays a ist durch ein Intervall gegeben. Somit können wir die Programmierung von arrayverarbeitenden Funktionen so ähnlich angehen, wie die von Funktionen auf Suchintervallen, siehe Abschnitt 3.6. Die folgende Funktion, die die Elemente eines Arrays von natürlichen Zahlen aufsummiert, illustriert die Vorgehensweise.

```
let sum (a : Array ⟨Nat⟩) : Nat =
  let rec s (i : Int) : Nat =
    if i = a.Length then 0
                    else a.[i] + s (i + 1)
  in s 0
```

Die Definition folgt dem Peano Entwurfsmuster; das Leibniz Entwurfsmuster ist ebenso anwendbar, bringt aber in diesem Fall keinen Vorteil, da jedes Element einmal angefasst werden muss.

```
let sum (a : Array ⟨Nat⟩) : Nat =
  let rec s (l : Int, u : Int) : Nat =
    match u ∸ l with
    | 0 → 0
    | 1 → a.[l]
    | d → let m = l + d ÷ 2 in s (l, m) + s (m, u)
  in s (0, a.Length)
```

Die Definition verwendet übrigens Dijkstras favorisierte Darstellung von natürlichen Intervallen, siehe Zitat am Anfang des Abschnitt. Das Intervall (l, u) umfasst alle natürlichen Zahlen i mit $l \leqslant i < u$. Somit ist $u \dot{-} l$ die Größe des Intervalls.

Fassen wir zusammen: Arrays sind zwischen Tupeln und Listen angesiedelt. Sie aggregieren eine *beliebige* Anzahl von Elementen des *gleichen* Typs. Der Zugriff auf Elemente kann berechnet werden und erfolgt in konstanter Zeit. Arrays haben auch ihre Schwächen: Um ein Array um ein Element zu erweitern, muss das komplette Array kopiert werden; der Aufwand ist also linear zur Größe des Arrays. Eine Liste hingegen kann *vorne* in konstanter Zeit erweitert werden.

4.5. Projekt: Interpreter und Maschinen⋆

Unsere Programmiersprache ist mittlerweile hinreichend ausdrucksstark, um ein kleines Projekt anzugehen: die Implementierung eines Mini–F#-Interpreters in Mini–F# selbst. Wir werden in etwa den in Kapitel 3 eingeführten Sprachumfang realisieren und dabei schrittweise, ähnlich wie in ebendiesem Kapitel, vorgehen. Eine kleine Vereinfachung nehmen wir allerdings vor: Die interpretierte Sprache ist im Unterschied zu Mini–F# nicht statisch, sondern dynamisch getypt: Eine unsinnige Kombination von Ausdrücken wie zum Beispiel $true + 1$ wird nicht *vor* der Auswertung entdeckt und moniert, sondern erst *während* der Auswertung. Aus einem statischen Typfehler wird so ein dynamischer Laufzeitfehler.

In Kapitel 3 haben wir die Auswertung von Ausdrücken mit Hilfe von Beweisbäumen formalisiert. Die zugrundeliegenden Auswertungsregeln, aus denen die Bäume zusammengesetzt werden, beschreiben aus mathematischer Sicht eine „ungerichtete“ Relation — eine Umgebung, ein Ausdruck und ein Wert werden zueinander in Beziehung gesetzt. In diesem Abschnitt richten wir den Fokus auf den Konstruktions*prozess*. Aus der Relation $\delta \vdash e \Downarrow v$ wird ein zielgerichteter Algorithmus, der Mini–F#-Interpreter, der zu einer Umgebung δ und einem Ausdruck e den zugehörigen Wert v ermittelt. Am Ende des Abschnitts steht die Einsicht, dass Beweisbäume nicht nur durch scharfes Hinsehen, sondern auch systematisch konstruiert werden können, eine Einsicht — so die Hoffnung —, die zu einem tieferen Verständnis der dynamischen Semantik beiträgt.

Die vorgestellten Programmstücke sind etwas länger und umfangreicher, als wir das bisher gewohnt sind. Länger, aber immer noch überschaubar: Die vollständige Formalisierung der abstrakten Syntax und der dynamischen Semantik passt auf zwei DIN A4 Seiten (siehe Abbildungen 4.4 und 4.5). Bei unserem kurzen Exkurs in die Welt des Übersetzerbaus lernen wir Konzepte kennen, mit deren Hilfe wir Phänomene beleuchten können, die sich mit unseren bisherigen Mitteln nicht oder nur schlecht erhellen lassen. So erklären wir, warum bei tiefer Rekursion „Stack Overflows“ auftreten, und zeigen auf, wie diese mittels endrekursiver Programme vermieden werden können.

4.5.1. Arithmetische Ausdrücke und natürliche Zahlen⋆

module Interpreter. Naturals

Konzeptionell am einfachsten sind arithmetische Ausdrücke, so dass wir uns diese als erstes vornehmen. Ausgangspunkt unserer Überlegungen ist wie gewohnt die abstrakte Syntax. In Kapitel 3 haben wir Sprachkonstrukte mit Hilfe von Baumsprachen eingeführt; an die Stelle von Baumsprachen treten jetzt rekursive Variantentypen.

In einem ersten Schritt führen wir numerische Konstanten und eine einzige Operation, die Addition, ein. Die Ausgangssprache ist bewusst einfach gehalten, um die Vorgehensweise verdeutlichen zu können, ohne dass wir uns dabei in Details verlieren. Diese einfachste aller möglichen „Programmiersprachen“ taufen wir auf den Namen Mini^2 (lies: „Mini quadriert“).

```
type Expr =
  | Num of Nat              // n
  | Add  of Expr * Expr     // e1 + e2
```

Wir haben Variantentypen eingeführt, um *Daten* modellieren zu können. Jetzt verwenden wir einen Variantentyp, um *Programme* zu repräsentieren. Die Idee von „Programmen als Daten“ spielt eine zentrale Rolle in der Informatik und steht als Geburtshelferin am Anfang ihrer Geschichte. Universelle Rechenmaschinen, Interpreter, Übersetzer, integrierte Entwicklungsumgebungen, sie alle basieren auf der Idee, Programme als Daten aufzufassen und zu verarbeiten.

Da wir uns zunächst auf arithmetische Ausdrücke beschränken, besteht der Bereich der Werte lediglich aus den natürlichen Zahlen.

```
type Value = Nat          // n
```

Metazirkulärer Interpreter In Kapitel 3 haben wir die dynamische Semantik mit Hilfe von Auswertungsregeln festgelegt. Die relevanten Regeln für Mini^2 sind:

$$\frac{}{Num\ n \Downarrow n} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{Add\ (e_1, e_2) \Downarrow n_1 + n_2}$$

Die zweistellige Relation $e \Downarrow n$ ist *funktional*: *Jedem* Ausdruck e wird *genau* ein Wert v zugeordnet. (Wir werden in Kapitel 6 Beweissysteme kennenlernen, denen diese Eigenschaft nicht zu eigen ist, die eine nicht-funktionale Relation zwischen verschiedenen mathematischen Objekten beschreiben.) Damit können wir die Auswertungs*relation* in Mini–F# als Auswertungs*funktion* definieren: Aus $e \Downarrow n$ wird *evaluate* $e = n$. Das Strukturentwurfsmuster für den rekursiven Variantentyp *Expr* führt unmittelbar zum Ziel:

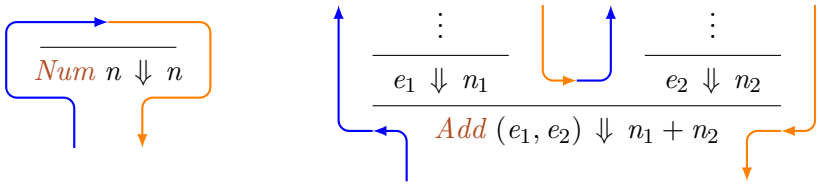
```
let rec evaluate : Expr → Value = function
    | Num n          → n
    | Add (e1, e2) → evaluate e1 + evaluate e2
```

Im Fachjargon nennt man *evaluate* einen *metazirkulären Interpreter*: Jedes Konstrukt der Objektsprache (Mini^2) wird auf das korrespondierende Konstrukt der Metasprache (Mini–F#) abgebildet.[11] Im Vergleich zu Kapitel 3 löst Mini–F# somit die Sprache der Mathematik als Metasprache ab.

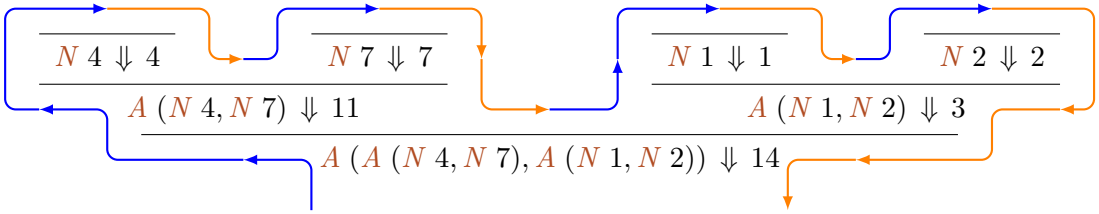
Um die Bedeutung arithmetischer Ausdrücke zu erklären, verwenden wir eine rekursive Funktionsdefinition. Das ist etwas unbefriedigend, da wir einfache Dinge wie arithmetische Ausdrücke auf komplizierte Dinge wie rekursive Funktionsdefinitionen zurückführen. Im folgenden beschreiben wir einen alternativen Ansatz, die Auswertung von Ausdrücken mit Hilfe einer *abstrakten Maschine*. Um diese zu definieren, müssen wir uns zwar etwas mehr abstrampeln, können dann aber als Lohn der Bemühungen einen gegebenen Ausdruck Schritt für Schritt, peu à peu ausrechnen. Die Maschine ist zwar abstrakt — wir abstrahieren von den zahlreichen Details realer Maschinen —, zeichnet aber dennoch ein hinreichend realistisches Bild der Wirklichkeit.

[11] Das Attribut „zirkulär“ deutet darauf hin, dass der Interpreter in der Sprache geschrieben ist, deren Programme er verarbeitet.

Abstrakte Maschine Schauen wir uns noch einmal die Regeln der dynamischen Semantik, die Auswertungsregeln, an.



Die Bedeutung eines komplexen arithmetischen Ausdrucks wird festgelegt, indem Instanzen der Beweisregeln zu einem Beweisbaum zusammengesetzt werden. Der Ausdruck $Add\ (Add\ (Num\ 4, Num\ 7), Add\ (Num\ 1, Num\ 2))$ besteht zum Beispiel aus sieben Knoten; der korrespondierende Beweisbaum entsprechend aus sieben Regelinstanzen (aus Platzgründen haben wir die Konstruktoren mit ihrem ersten Buchstaben abgekürzt).



Jeder Auswerter, menschlich oder maschinell, wird zu einem gegebenen Ausdruck einen solchen Beweisbaum konstruieren und traversieren. In der Regel wird der Beweisbaum dabei nicht *explizit* in Erscheinung treten; er wird nicht als Datum repräsentiert, sondern ist nur gedanklich, zum Beispiel *implizit* durch die Programmstruktur, gegeben — im Fall des metazirkulären Interpreters entspricht der Beweisbaum dem Rekursionsbaum.

Die Pfeile in den obigen Diagrammen sollen die Traversierung illustrieren. Entlang der blauen Pfeile klettern wir nach oben: Aus einem zusammengesetzten Ausdruck wird ein Teilausdruck herausgelöst, mit dessen Auswertung dann fortgefahren wird. Im obigen Beispiel klettern wir am Anfang dreimal nach oben: Aus dem Ausdruck $Add\ (Add\ (Num\ 4, Num\ 7), Add\ (Num\ 1, Num\ 2))$ wird der erste Summand herausgelöst; aus $Add\ (Num\ 4, Num\ 7)$ wird wiederum $Num\ 4$ herausgelöst. In jedem Schritt müssen wir uns merken, was noch zu tun ist, wenn die Auswertung des jeweiligen Teilausdrucks abgeschlossen ist. Also, wenn wir e_1 aus $Add\ (e_1, e_2)$ herauslösen, müssen wir uns den „restlichen“ Ausdruck $Add\ (\bullet, e_2)$ merken — der Platzhalter „$\bullet$“ symbolisiert dabei die Position, an der wir etwas entfernt haben. Wenn wir den Wert n_1 von e_1 kennen, fahren wir dann mit der Auswertung von e_2 fort. Jetzt müssen wir uns n_1 merken: Aus $Add\ (\bullet, e_2)$ wird $Add\ (n_1, \bullet)$. Wenn die Auswertung von e_2 ebenfalls abgeschlossen ist, können wir schließlich die Addition durchführen und das Resultat zurückgeben, sprich im Beweisbaum nach unten klettern.

Ausdrücke mit „Löchern“, Einträge auf einer *Todo-Liste*, repräsentieren wir mit Hilfe des folgenden Datentyps. Die Ziffer gibt jeweils die Position des Platzhalters an.

```
type Frame =
  | Add1 of Expr          // Add1 e2 ≅ Add (•, e2)
  | Add2 of Value         // Add2 v1 ≅ Add (v1, •)
```

Das allgemeine Prinzip lässt sich am besten mit Hilfe einer fiktiven, 3-stelligen Operation verdeutlichen.

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad e_3 \Downarrow v_3}{Op\ (e_1, e_2, e_3) \Downarrow op\ (v_1, v_2, v_3)}$$

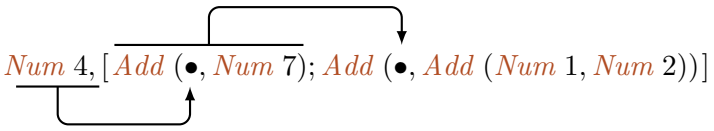
Entsprechend der fiktiven Auswertungsregel vollzieht sich die Auswertung in drei bzw. in vier Schritten — je nachdem, ob man die semantische Operation $op\ (v_1, v_2, v_3)$ als letzten Schritt hinzuzählt oder nicht.

$$Op\ (e_1, e_2, e_3) \mapsto Op\ (\bullet, e_2, e_3) \mapsto Op\ (v_1, \bullet, e_3) \mapsto Op\ (v_1, v_2, \bullet) \mapsto op\ (v_1, v_2, v_3)$$

Das Loch „•“ markiert jeweils die Grenze zwischen dem ausgewerteten und dem unausgewerteten Teil des Ausdrucks. Da das Loch an drei Positionen stehen kann, würden wir entsprechend drei Sorten von *Todo-Einträgen* zum Variantentyp *Frame* hinzufügen.

```
type Frame =
  | ...
  | Op1 of Expr  * Expr        // Op1 (e2, e3) ≅ Op (•, e2, e3)
  | Op2 of Value * Expr        // Op2 (v1, e3) ≅ Op (v1, •, e3)
  | Op3 of Value * Value       // Op3 (v1, v2) ≅ Op (v1, v2, •)
```

Natürlich genügt es nicht, sich nur einen „restlichen“ Teilausdruck, ein Element vom Typ *Frame*, zu merken. Ist ein Ausdruck tief verschachtelt, so lösen wir wiederholt Teilausdrücke heraus, die wir uns auf einer *Todo-Liste* vom Typ *Frame list* merken. Das obige Beispiel fortführend repräsentiert



die Auswertung nach den ersten drei Schritten. Im Prinzip nehmen wir einen Repräsentationswechsel vor: Aus der Todo-Liste lässt sich der ursprüngliche Ausdruck rekonstruieren, indem wir die Teilausdrücke wie oben angedeutet wieder in die Löcher einsetzen.

Jetzt haben wir fast alle Zutaten beisammen. Unsere abstrakte Maschine befindet sich stets in einem von zwei möglichen Zuständen, je nachdem, ob sie sich im Beweisbaum nach oben oder nach unten bewegt, ob sie einen Ausdruck auswertet, $e \Downarrow \cdots$, oder einen Wert zurückgibt, $\cdots \Downarrow v$. Auf dem Weg nach unten werden die semantischen Operationen durchgeführt: Aus den *Werten* der Teilausdrücke wird der *Wert* des Gesamtausdrucks berechnet.

```
type State =
  | Eval of Expr  * Frame list      // e ⇓ ...
  | Ret  of Value * Frame list      // ... ⇓ v
```

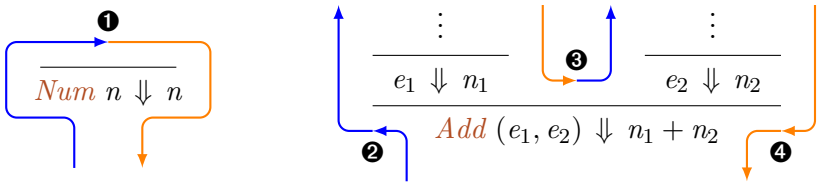
Die beiden Konstruktoren können auch als Instruktionen aufgefasst werden: Die Instruktion *Eval* $(e, stack)$ entspricht einem blauen Pfeil, *Ret* $(v, stack)$ einem orangenen, wobei

stack jeweils die Todo-Liste repräsentiert. Apropos, die Todo-Liste nennt man im Fachjargon *Stapel* (engl. stack), da sie streng von links nach rechts bzw. um im Bild zu bleiben von oben nach unten abgearbeitet wird — das unterscheidet sie von meinem Todo-Zettel. *Rahmen* (engl. frame) ist ein technischer Begriff aus dem Übersetzerbau.

Die Funktion *step* implementiert einen einzelnen Rechenschritt, einen Zustandübergang der abstrakten Maschine.

```
let step : State → State = function
// Auswertung (blau)
  | Eval (Num n,          stack) → Ret (n, stack)                  // ❶
  | Eval (Add (e1, e2), stack) → Eval (e1, Add1 e2 :: stack)       // ❷
// Rückgabe (orange)
  | Ret (v, [])                → Ret (v, [])                       // Endzustand
  | Ret (v1, Add1 e2 :: stack) → Eval (e2, Add2 v1 :: stack)       // ❸
  | Ret (v2, Add2 v1 :: stack) → Ret (v1 + v2, stack)              // ❹
```

Da es zwei prinzipielle Zustände gibt, $\rightarrow$ und $\rightarrow$, existieren insgesamt $2 \cdot 2 = 4$ Zustandübergänge: $\rightarrow\rightarrow$, $\rightarrow\rightarrow$, $\rightarrow\rightarrow$ und $\rightarrow\rightarrow$. Die beiden Auswertungsregeln



illustrieren alle möglichen Arten von Übergängen.

Ist die Todo-Liste leer, so haben wir entweder einen *Anfangszustand*, $Eval\ (e, [])$, oder einen *Endzustand*, $Ret\ (v, [])$, vor uns. Mit Hilfe von *step* können wir einen Mini^2-Ausdruck im Mini–F#-Interpreter schrittweise ausrechnen.

```
Mini⟩ step (Eval (Add (Num 4, Num 7), []))
Eval (Num 4, [Add1 (Num 7)])
Mini⟩ step it
Ret (4, [Add1 (Num 7)])
Mini⟩ step it
Eval (Num 7, [Add2 4])
Mini⟩ step it
Ret (7, [Add2 4])
Mini⟩ step it
Ret (11, [])
```

Wir rufen *step* solange auf, bis die Todo-Liste leer ist.

Zu unserem laufenden Beispiel korrespondiert die folgende Abfolge von Zuständen.

```
Eval (Add (Add (Num 4, Num 7), Add (Num 1, Num 2)), [])
Eval (Add (Num 4, Num 7), [Add1 (Add (Num 1, Num 2))])
Eval (Num 4, [Add1 (Num 7); Add1 (Add (Num 1, Num 2))])
Ret (4, [Add1 (Num 7); Add1 (Add (Num 1, Num 2))])
Eval (Num 7, [Add2 4; Add1 (Add (Num 1, Num 2))])
Ret (7, [Add2 4; Add1 (Add (Num 1, Num 2))])
Ret (11, [Add1 (Add (Num 1, Num 2))])
Eval (Add (Num 1, Num 2), [Add2 11])
Eval (Num 1, [Add1 (Num 2); Add2 11])
Ret (1, [Add1 (Num 2); Add2 11])
Eval (Num 2, [Add2 1; Add2 11])
Ret (2, [Add2 1; Add2 11])
Ret (3, [Add2 11])
Ret (14, [])
```

Die Anzahl der Zustände entspricht der Anzahl der Pfeile in dem weiter oben dargestellten Beweisbaum: Insgesamt haben wir sieben blaue und sieben orangene Pfeile.

module Interpreter. Booleans

4.5.2. Boolesche Ausdrücke und Werte⋆

Wir erweitern Mini^2 um Boolesche Ausdrücke. Zusätzlich ersetzen wir die spezielle Additionsoperation durch einen „generischen“ binären Operator, so dass wir die zahlreichen arithmetischen und Vergleichsoperationen einheitlich behandeln können.

```
type Op =
  | Add | Sub | Mul | Div | Mod | Lt | Lte | Equ | Neq | Gte | Gt
type Expr =
  | Num of Nat                    // n
  | Bin  of Expr * Op * Expr      // e1 + e2, e1 ÷ e2 etc.
  | False                         // false
  | True                          // true
  | If   of Expr * Expr * Expr    // if e1 then e2 else e3
```

Die folgenden Abkürzungen dienen dazu, die Lesbarkeit der abstrakten Syntax zu verbessern.

```
let add (e1, e2) = Bin (e1, Add, e2)
let sub (e1, e2) = Bin (e1, Sub, e2)
let mul (e1, e2) = Bin (e1, Mul, e2)
...
let equ (e1, e2) = Bin (e1, Equ, e2)
...
```

Die Funktionen illustrieren im Kleinen, dass wir Programme in der abstrakten Syntax von Mini^2 auch mit Hilfe von Mini–F#-Funktionen *generieren* können — dabei nutzen wir aus, dass Programme als Daten behandelt werden.

Der Typ der Werte wird entsprechend um Wahrheitswerte erweitert.

```
type Value =
    | Nat  of Nat          // n
    | Bool of Bool         // false oder true
```

Die Werte werden jeweils mit ihrem Typ „getaggt“. Da wir, wie bereits angesprochen, die Typprüfung nicht statisch, sondern dynamisch durchführen, werden sozusagen aus Typen Daten: *Bool false* und *Nat* 4711 sind Elemente vom Typ *Value*; der Datenkonstruktor *Bool* bzw. *Nat* verrät jeweils den Typ des Datums.

Die Funktion *primitive* implementiert die verschiedenen Operationen und führt die dynamische Typprüfung durch.

```
let primitive : Value * Op * Value → Value option = function
    | (Nat n1, Add, Nat n2) → Some (Nat  (n1  +  n2))
    | (Nat n1, Sub, Nat n2) → Some (Nat  (n1  -  n2))
    | (Nat n1, Mul, Nat n2) → Some (Nat  (n1  *  n2))
    | (Nat n1, Div, Nat n2) → Some (Nat  (n1  /  n2))
    | (Nat n1, Mod, Nat n2) → Some (Nat  (n1  %  n2))
    | (Nat n1, Lt,  Nat n2) → Some (Bool (n1  <  n2))
    | (Nat n1, Lte, Nat n2) → Some (Bool (n1  <= n2))
    | (Nat n1, Equ, Nat n2) → Some (Bool (n1  =  n2))
    | (Nat n1, Neq, Nat n2) → Some (Bool (n1  <> n2))
    | (Nat n1, Gte, Nat n2) → Some (Bool (n1  >= n2))
    | (Nat n1, Gt,  Nat n2) → Some (Bool (n1  >  n2))
    | _                     → None
```

Die Funktion setzt im Prinzip die Regeln der statischen Semantik um: Ein arithmetischer Operator erwartet zwei natürliche Zahlen als Eingabe und gibt eine natürliche Zahl als Ausgabe zurück; ein Vergleichsoperator erwartet die gleichen Eingaben, gibt aber einen Wahrheitswert zurück. Werden die Erwartungen nicht erfüllt, schlägt die Typprüfung fehl und das Ergebnis ist *None*. Man sieht jeweils sehr schön, wie Syntax (*Add* etc.) auf Semantik (+ etc.) abgebildet wird.

Kommen wir zur Auswertung der Alternative. Es ist hilfreich, sich noch einmal die Auswertungsregeln ins Gedächtnis zu rufen.

$$\frac{e_1 \Downarrow \mathit{true} \qquad e_2 \Downarrow v}{\mathit{If}\ (e_1, e_2, e_3) \Downarrow v} \qquad \frac{e_1 \Downarrow \mathit{false} \qquad e_3 \Downarrow v}{\mathit{If}\ (e_1, e_2, e_3) \Downarrow v}$$

Beiden Regeln ist gemeinsam, dass die Bedingung e_1 ausgewertet wird. In Abhängigkeit vom Ergebnis wird genau ein Zweig der Alternative ausgewertet: e_2 oder e_3. Ein weiteres Detail ist bedeutsam: Das Ergebnis v des Zweigs ist auch das Ergebnis der Alternative *If* (e_1, e_2, e_3). Im Unterschied zu den binären Operatoren gibt es nach der Auswertung von e_2 bzw. e_3 nichts mehr zu tun; der berechnete Wert wird einfach „nach unten“ weitergereicht (im Fall der Additionsoperation müssen die Werte der Summanden noch addiert werden). Da wir uns auf der Todo-Liste somit nichts merken müssen, enthält

Frame nur einen Konstruktor für die Alternative, und nicht zwei wie im Fall der binären Operatoren.

```
type Frame =
    | Bin₁ of Op * Expr          // Bin (•, op, e₂)
    | Bin₂ of Value * Op         // Bin (v₁, op, •)
    | If₁  of Expr * Expr        // If (•, e₂, e₃)
```

Zu den Fällen für arithmetische Ausdrücke gesellen sich Fälle für Boolesche Konstanten und die Alternative.

```
let step : State → State = function
// Auswertung
    | Eval (Num n,           stack)      → Ret (Nat n, stack)
    | Eval (Bin (e₁, op, e₂), stack)     → Eval (e₁, Bin₁ (op, e₂) :: stack)
    | Eval (False,           stack)      → Ret (Bool false, stack)
    | Eval (True,            stack)      → Ret (Bool true, stack)
    | Eval (If (e₁, e₂, e₃), stack)      → Eval (e₁, If₁ (e₂, e₃) :: stack)
// Rückgabe
    | Ret (v,      [])                   → Ret (v, [])
    | Ret (v₁,     Bin₁ (op, e₂) :: stack) → Eval (e₂, Bin₂ (v₁, op) :: stack)
    | Ret (v₂,     Bin₂ (v₁, op) :: stack) → match primitive (v₁, op, v₂) with
                                              | Some v → Ret (v, stack)
                                              | None   → error "type mismatch"
    | Ret (Bool b, If₁ (e₂, e₃)  :: stack) → if b then Eval (e₂, stack)
                                                else  Eval (e₃, stack)
    | Ret (_,      If₁ (e₂, e₃)  :: stack) → error "type mismatch"
```

Die beiden Regeln für die Alternative setzen die Auswertungsregeln der dynamischen Semantik buchstabengetreu um. Um *If* (e_1, e_2, e_3) auszurechnen, wird zunächst die Bedingung e_1 ausgewertet; auf der Todo-Liste, dem Stack, merken wir uns die beiden Zweige der Alternative. Ist der Boolesche Wert ermittelt, wird mit der Auswertung eines der beiden Zweige fortgefahren. Die Todo-Liste ist jetzt wieder auf dem ursprünglichen Stand. Dieses kleine Implementierungsdetail hat weitreichende Konsequenzen; es ist ein Mosaikstein einer wichtigen, ja zentralen Optimierung, der sogenannten „*Tail Call Optimization*“. Aber dazu später mehr.

An zwei Stellen brechen wir die Auswertung mit einem *Typfehler* ab: Wenn die Typprüfung der primitiven Operationen fehlschlägt oder wenn die Bedingung einer Alternative nicht zu einem Wahrheitswert auswertet.

Schauen wir uns ein Beispiel an: Die Auswertung von (*if* $4 < 11$ *then* 4 *else* $11) + 1$ in der konkreten bzw. *Bin* (*If* (*Bin* (*Num* 4, *Lt*, *Num* 11), *Num* 4, *Num* 11), *Add*, *Num* 1) in der abstrakten Syntax nimmt den folgenden Verlauf (wir führen lediglich die Zustände der abstrakten Machine auf; *step* führt uns von einem zum nächsten Zustand).

Eval (*Bin* (*If* (*Bin* (*Num* 4, *Lt*, *Num* 11), *Num* 4, *Num* 11), *Add*, *Num* 1), [])
Eval (*If* (*Bin* (*Num* 4, *Lt*, *Num* 11), *Num* 4, *Num* 11), [*Bin*$_1$ (*Add*, *Num* 1)])
Eval (*Bin* (*Num* 4, *Lt*, *Num* 11), [*If*$_1$ (*Num* 4, *Num* 11); *Bin*$_1$ (*Add*, *Num* 1)])
Eval (*Num* 4, [*Bin*$_1$ (*Lt*, *Num* 11); *If*$_1$ (*Num* 4, *Num* 11); *Bin*$_1$ (*Add*, *Num* 1)])
Ret (*Nat* 4, [*Bin*$_1$ (*Lt*, *Num* 11); *If*$_1$ (*Num* 4, *Num* 11); *Bin*$_1$ (*Add*, *Num* 1)])
Eval (*Num* 11, [*Bin*$_2$ (*Nat* 4, *Lt*); *If*$_1$ (*Num* 4, *Num* 11); *Bin*$_1$ (*Add*, *Num* 1)])
Ret (*Nat* 11, [*Bin*$_2$ (*Nat* 4, *Lt*); *If*$_1$ (*Num* 4, *Num* 11); *Bin*$_1$ (*Add*, *Num* 1)])
Ret (*Bool* *false*, [*If*$_1$ (*Num* 4, *Num* 11); *Bin*$_1$ (*Add*, *Num* 1)])

An dieser Stelle haben wir die Bedingung der Alternative ausgerechnet und fahren mit der Auswertung des ***else***-Zweigs fort.

Eval (*Num* 11, [*Bin*$_1$ (*Add*, *Num* 1)])
Ret (*Nat* 11, [*Bin*$_1$ (*Add*, *Num* 1)])
Eval (*Num* 1, [*Bin*$_2$ (*Nat* 11, *Add*)])
Ret (*Nat* 1, [*Bin*$_2$ (*Nat* 11, *Add*)])
Ret (*Nat* 12, [])

Man sieht sehr schön, dass das Ergebnis des ***else***-Zweigs unmittelbar an die Additionsoperation weitergereicht wird. Anhand der Todo-Liste lässt sich nicht mehr nachvollziehen, dass 11 dem Zweig einer Alternative entstammt. Die letzten fünf Zustände erhalten wir auch, wenn wir zum Beispiel $11 + 1$ bzw. *Bin* (*Num* 11, *Add*, *Num* 1) ausrechnen.

4.5.3. Wertedefinitionen⋆

module Interpreter. Definitions

Im nächsten Schritt erweitern wir Mini^2 um Bezeichner und Wertedefinitionen.

```
type Id = String
type Expr =
  | ...
  | Id   of Id                    // x
  | Let  of Id * Expr * Expr      // let x1 = e1 in e
```

Bezeichner repräsentieren wir durch Strings. Wertedefinitionen sind im Vergleich zu Mini–F# einfacher gestrickt: *Let* (x_1, e_1, e) entspricht in der konkreten Syntax ***let*** $x_1 = e_1$ ***in*** e, kombiniert also eine Deklaration mit einem ***in***-Ausdruck. (Auf diese Weise sparen wir uns die Einführung einer zweiten syntaktischen Kategorie: Deklarationen zusätzlich zu Ausdrücken. *Ein* Typ für Ausdrücke ist einfacher zu handhaben als *zwei* verschränkt rekursive Typen für Ausdrücke und Deklarationen.)

Die Einführung von Bezeichnern zieht den viel zitierten Rattenschwanz von Änderungen nach sich; nahezu jedes Detail unserer abstrakten Maschine muss angepasst werden.

Ausgangspunkt unserer Überlegungen sind wie immer die Auswertungsregeln der dynamischen Semantik.

$$\frac{}{\delta \vdash \mathit{Id}\ x \Downarrow \delta(x)} \qquad \frac{\delta \vdash e_1 \Downarrow v_1 \qquad \delta, \{x_1 \mapsto v_1\} \vdash e \Downarrow v}{\delta \vdash \mathit{Let}\ (x_1, e_1, e) \Downarrow v}$$

In Mini–F# garantiert die statische Semantik, dass der Bezeichner x in der Umgebung δ definiert wird. In Mini^2 führen wir diese Überprüfung dynamisch zur Laufzeit durch. Eine Wertedefinition wird in zwei Schritten abgearbeitet: Zunächst wird die rechte Seite der Gleichung $x_1 = e_1$ ausgewertet; der erhaltene Wert v_1 wird an x_1 gebunden; in der um diese Bindung erweiterten Umgebung wird der Ausdruck e ausgerechnet. Ähnlich wie im Fall der Alternative ist nach der Auswertung von e für die Wertedefinition nichts mehr zu tun; der berechnete Wert v wird „nach unten“ weitergereicht (ein weiteres Mosaikstück der *Tail Call Optimization*.)

Bevor wir uns den Änderungen an der abstrakten Maschine zuwenden, müssen wir zunächst überlegen, wie wir Umgebungen repräsentieren. Wir erinnern uns: Eine Umgebung ist eine endliche Abbildung von Bezeichnern auf Werte. Mit der Implementierung von endlichen Abbildungen beschäftigen wir uns intensiv in Abschnitt 5.3; dem wollen wir an dieser Stelle nicht vorgreifen, so dass wir eine konzeptionell einfachere Lösung wählen.[12] In der dynamischen Semantik wird eine Umgebung, ausgehend von der leeren endlichen Abbildung, schrittweise mit dem Kommaoperator um Bindungen erweitert. Diese beiden Operationen machen wir zu Konstruktoren eines Datentyps!

```
type Map ⟨'key, 'val⟩ =
  | Empty                                  // ∅
  | Comma of Map ⟨'key, 'val⟩ * 'key * 'val   // δ, {x ↦ v}
```

Der Typ ist parametrisiert mit dem Definitions- und dem Wertebereich: Wenn wir Typen mit Mengen identifizieren, dann entspricht $Map\,\langle X, Y\rangle$ der Menge $X \to_{\text{fin}} Y$ aller endlichen Abbildungen von X nach Y. Der Konstruktor *Empty* repräsentiert die leere Abbildung; wenn *env* die Umgebung δ repräsentiert, dann repräsentiert $Comma\,(env, x, v)$ die Umgebung $\delta, \{x \mapsto v\}$.

Die Funktion *lookup* entspricht der Anwendung einer endlichen Abbildung; *lookup x env* schlägt den Bezeichner x in *env* nach.

```
let rec lookup (x : 'key) : Map ⟨'key, 'val⟩ → 'val option when 'key : equality = function
  | Empty →                  None
  | Comma (env, x1, v1) → if x = x1 then Some v1 else lookup x env
```

Repräsentiert *env* die Umgebung δ, dann gilt *lookup x env* $=$ *Some* v, falls $x \in dom\ \delta$ und $\delta(x) = v$; anderenfalls erhalten wir *lookup x env* $=$ *None*. Das Datum

```
Comma (Comma (Empty, "a", Nat 4711), "b", Bool false)
```

repräsentiert zum Beispiel die Umgebung, in der `"a"` an 4711 und `"b"` an *false* gebunden ist: $\varnothing, \{a \mapsto 4711\}, \{b \mapsto false\} = \{a \mapsto 4711, b \mapsto false\}$. Weiter rechts stehende Bindungen verschatten dabei wie gewünscht weiter links stehende Bindungen, da *lookup* gemäß der Struktur des Datentyps von „rechts nach links“ vorgeht.

[12] Tatsächlich ist eine ausgefeilte Implementierung nicht notwendig, da Umgebungen nicht sehr groß werden: Die Anzahl der Bindungen hängt *statisch* vom Programmtext ab und nicht *dynamisch* vom Verlauf der Rechnung.

(Der Typ *Map* ⟨'*key*, '*val*⟩ ist übrigens isomorph zu ('*key* ∗ '*val*) *list*, einer Liste von Schlüssel-Wert Paaren. Im Unterschied zu einer Liste werden Einträge nicht vorne mit „::“, sondern hinten mit *Comma* hinzugefügt. Der Grund für die Einführung eines speziellen Typs ist so banal, wie überzeugend: Sie erlaubt uns, die Regeln der dynamischen Semantik buchstabengetreu umzusetzen.)

Todo-Einträge vom Typ *Frame* und Zustände der abstrakten Maschine vom Typ *State* müssen um Umgebungen erweitert werden. Klar: Jeder Teilausdruck kann potentiell einen Bezeichner enthalten, so dass wir Umgebungen für die Auswertung bereitstellen und zu diesem Zweck auf der Todo-Liste vermerken müssen.

```
type Env = Map ⟨Id, Value⟩
type Frame =
    | Bin1 of Env ∗ Op ∗ Expr          // δ, Bin (•, op, e2)
    | Bin2 of       Value ∗ Op         // Bin (v1, op, •)
    | If1  of Env ∗ Expr ∗ Expr        // δ, If (•, e2, e3)
    | Let1 of Env ∗ Id ∗ Expr          // δ, Let (x, •, e)
```

Jeder Konstruktor erhält die aktuelle Umgebung als zusätzlichen Parameter mit auf den Weg, bis auf eine Ausnahme: Bin_2. Für die Ausführung einer primitiven Operation wird schlicht und einfach keine Umgebung benötigt.

```
type State =
    | Eval of Env ∗ Expr ∗ Frame list   // δ ⊢ e ⇓ ··· , stack
    | Ret  of      Value ∗ Frame list   // ··· ⇓ v, stack
```

Die Instruktion *Eval* (δ, *e*, *stack*) weist die Maschine an, den Ausdruck *e* in der Umgebung δ auszuwerten; *Ret* (*v*, *stack*) instruiert die Maschine, den Wert *v* zurückzugeben, sprich für die Abarbeitung des nächsten Eintrags auf der Todo-Liste *stack* zu verwenden.

Wenden wir uns der „Schrittfunktion“ zu. (Bezeichner dürfen in F# nicht nur aus lateinischen, sondern auch aus griechischen Buchstaben bestehen; an Stelle des Bezeichners *env* verwenden wir im Folgenden stets δ.)

```
let step : State → State = function
// Auswertung
    | Eval (δ, Num n,          stack) → Ret (Nat n, stack)
    | Eval (δ, Bin (e1, op, e2), stack) → Eval (δ, e1, Bin1 (δ, op, e2) :: stack)
    | Eval (δ, False,          stack) → Ret (Bool false, stack)
    | Eval (δ, True,           stack) → Ret (Bool true, stack)
    | Eval (δ, If (e1, e2, e3), stack) → Eval (δ, e1, If1 (δ, e2, e3) :: stack)
    | Eval (δ, Id x,           stack) → match lookup x δ with
                                          | None   → error ("'" ^ x ^ "' is undefined")
                                          | Some v → Ret (v, stack)
    | Eval (δ, Let (x1, e1, e), stack) → Eval (δ, e1, Let1 (δ, x1, e) :: stack)
```

Die ersten fünf Regeln kennen wir schon — nur, dass wir uns bisher nicht um Umgebungen kümmern mussten. Im Fall von Konstanten können wir die Umgebung ignorieren; bei der Auswertung zusammengesetzter Ausdrücke propagieren wir die Umgebung

zu den Teilausdrücken. Die Umgebung kommt schließlich bei der Auswertung von Bezeichnern zum Einsatz; ist der Bezeichner nicht definiert, wird die Rechnung mit einem Laufzeitfehler abgebrochen.

Die beiden Regeln für ***let***-Ausdrücke setzen die dynamische Semantik buchstabengetreu um: Zunächst wird die rechte Seite der Gleichung $x_1 = e_1$ ausgewertet (Code oben); der erhaltene Wert v_1 wird an x_1 gebunden; in der um diese Bindung erweiterten Umgebung wird der Ausdruck e ausgerechnet (Code unten).

```
// Rückgabe
  | Ret (v,  [])                          → Ret (v, [])
  | Ret (v1, Bin1 (δ, op, e2)   :: stack) → Eval (δ, e2, Bin2 (v1, op) :: stack)
  | Ret (v2, Bin2 (v1, op)      :: stack) → match primitive (v1, op, v2) with
                                              | Some v → Ret (v, stack)
                                              | None   → error "type mismatch"
  | Ret (Bool b, If1 (δ, e2, e3) :: stack) → if b then Eval (δ, e2, stack)
                                             else  Eval (δ, e3, stack)
  | Ret (_,  If1 (δ, e2, e3))   :: stack) → error "type mismatch"
  | Ret (v1, Let1 (δ, x1, e)    :: stack) → Eval (Comma (δ, x1, v1), e, stack)
```

Schauen wir uns die abstrakte Maschine in Aktion an und werten den geschachtelten Ausdruck ***let*** $a = 47$ ***in let*** $b = a + 11$ ***in*** $a * b$ bzw. in abstrakter Syntax *Let* (`"a"`, *Num* 47, *Let* (`"b"`, *Bin* (*Id* `"a"`, *Add*, *Num* 11), *Bin* (*Id* `"a"`, *Mul*, *Id* `"b"`))) aus. Um nicht den Überblick zu verlieren, vereinbaren wir zwei Abkürzungen: eine für die vom äußeren ***let***-Ausdruck erzeugte Umgebung, δ = *Comma* (*Empty*, `"a"`, *Nat* 47), und eine weitere für den Rumpf des inneren ***let***-Ausdrucks, e = *Bin* (*Id* `"a"`, *Mul*, *Id* `"b"`). (Wie gewohnt geben wir nur die Abfolge der Zustände an.)

Eval (*Empty*, *Let* (`"a"`, *Num* 47, *Let* (`"b"`, *Bin* (*Id* `"a"`, *Add*, *Num* 11), e)), [])
Eval (*Empty*, *Num* 47, [*Let*$_1$ (*Empty*, `"a"`, *Let* (`"b"`, *Bin* (*Id* `"a"`, *Add*, *Num* 11), e))])
Ret (*Nat* 47, [*Let*$_1$ (*Empty*, `"a"`, *Let* (`"b"`, *Bin* (*Id* `"a"`, *Add*, *Num* 11), e))])
Eval (δ, *Let* (`"b"`, *Bin* (*Id* `"a"`, *Add*, *Num* 11), e), [])
Eval (δ, *Bin* (*Id* `"a"`, *Add*, *Num* 11), [*Let*$_1$ (δ, `"b"`, e)])
Eval (δ, *Id* `"a"`, [*Bin*$_1$ (δ, *Add*, *Num* 11); *Let*$_1$ (δ, `"b"`, e)])
Ret (*Nat* 47, [*Bin*$_1$ (δ, *Add*, *Num* 11); *Let*$_1$ (δ, `"b"`, e)])
Eval (δ, *Num* 11, [*Bin*$_2$ (*Nat* 47, *Add*); *Let*$_1$ (δ, `"b"`, e)])
Ret (*Nat* 11, [*Bin*$_2$ (*Nat* 47, *Add*); *Let*$_1$ (δ, `"b"`, e)])
Ret (*Nat* 58, [*Let*$_1$ (δ, `"b"`, e)])
Eval (*Comma* (δ, `"b"`, *Nat* 58), e, [])

An dieser Stelle haben wir die beiden Definitionen abgearbeitet und wenden uns der Auswertung des Rumpfs e zu. Setzen wir die zwei Abkürzungen ein, um den Zustand in voller Schönheit vor uns zu haben.

Eval (*Comma* (*Comma* (*Empty*, `"a"`, *Nat* 47), `"b"`, *Nat* 58), *Bin* (*Id* `"a"`, *Mul*, *Id* `"b"`), [])

Die Umgebung besteht aus zwei Einträgen; beide werden benötigt, um den Rumpf auszuwerten. Die Auswertung nimmt jetzt ihren überraschungsfreien Verlauf.

Eval (*Comma* (δ, "b", *Nat* 58), *Id* "a", [*Bin*$_1$ (*Comma* (δ, "b", *Nat* 58), *Mul*, *Id* "b")])
Ret (*Nat* 47, [*Bin*$_1$ (*Comma* (δ, "b", *Nat* 58), *Mul*, *Id* "b")])
Eval (*Comma* (δ, "b", *Nat* 58), *Id* "b", [*Bin*$_2$ (*Nat* 47, *Mul*)])
Ret (*Nat* 58, [*Bin*$_2$ (*Nat* 47, *Mul*)])
Ret (*Nat* 2726, [])

Die Bezeichner werden in der Umgebung nachgeschlagen und die Ergebnisse miteinander multipliziert.

4.5.4. Funktionsausdrücke und Funktionsabschlüsse⋆

module Interpreter.Functions

Was steht als nächstes auf dem Programm? Funktionsausdrücke und -applikationen!

```
type Expr =
  | ...
  | Fun of Id * Expr          // fun x → e
  | App of Expr * Expr        // e e1
```

Die meisten Programmiersprachen, und F# ist da keine Ausnahme, erlauben Schreibvereinfachungen: Zum Beispiel kann die geschachtelte Abstraktion ***fun*** $x_1 \to$ ***fun*** $x_2 \to e$ kurz ***fun*** $x_1\ x_2 \to e$ notiert werden. Man spricht auch von „*syntaktischem Zucker*“ (engl. syntactic sugar), da die verkürzte Notation die Programmerstellung erleichtert und so das Programmierleben versüßt. Für die abstrakte Syntax können wir die gleichen Schreiberleichterungen anbieten, indem wir entsprechende Funktionen programmieren — wir machen an dieser Stelle erneut von der Idee der „Programme als Daten“ Gebrauch.

```
let rec funs : Id list * Expr → Expr = function          // fun x1 x2 ··· xn → e
  | ([],      e) → e
  | (x :: xs, e) → Fun (x, funs (xs, e))
let rec apps (f : Expr) : Expr list → Expr = function    // e e1 e2 ··· en
  | []      → f
  | e :: es → apps (App (f, e)) es
```

Es lohnt sich, die beiden Funktionsdefinitionen genauer zu studieren: Mehrfachabstraktionen ***fun*** $x_1\ x_2\ \cdots\ x_n \to e$ werden auf nach *rechts* geschachtelte Abstraktionen abgebildet; Mehrfachapplikationen $e\ e_1\ e_2\ \cdots\ e_n$ auf nach *links* geschachtelte Applikationen (f ist ein sogenannter akkumulierender Parameter).

Den Bereich der Werte erweitern wir entsprechend um Funktionsabschlüsse.

```
type Value =
  | ...
  | Closure of Env * Id * Expr          // ⟨δ, x, e⟩
```

Zur Erinnerung: Hier sind die Auswertungsregeln der dynamischen Semantik.

$$\frac{}{\delta \vdash \mathit{Fun}\ (x, e) \Downarrow \langle \delta, x, e \rangle}$$

$$\frac{\delta \vdash e \Downarrow \langle \delta', x_1, e' \rangle \qquad \delta \vdash e_1 \Downarrow v_1 \qquad \delta', \{ x_1 \mapsto v_1 \} \vdash e' \Downarrow v'}{\delta \vdash \mathit{App}\ (e, e_1) \Downarrow v'}$$

Ein Funktionsausdruck wertet zu einem Funktionsabschluss aus. Die Abarbeitung eines Funktionsaufrufs vollzieht sich in drei Schritten: (1) Die Funktion e wird ausgerechnet; das Ergebnis ist hoffentlich ein Funktionsabschluss. (2) Das Argument e_1 wird ausgerechnet. (3) Der Funktionsrumpf e' wird in der erweiterten Umgebung $\delta', \{ x_1 \mapsto v_1 \}$ ausgerechnet. Ein Detail ist bedeutsam: Das Ergebnis des Funktionsrumpfes ist auch das Ergebnis der Funktionsapplikation. Genau wie bei Alternativen und Wertebindungen wird der berechnete Wert v' einfach „nach unten“ weitergereicht (dies ist das dritte und letzte Mosaikstück der *Tail Call Optimization* — mit den segensreichen Auswirkungen beschäftigen wir uns im nächsten Abschnitt).

Wir werden sehen, dass die Umsetzung der Regel etwas weniger aufwändig ist, da wir bereits ***let***-Ausdrücke implementiert haben. Aus diesem Grund benötigen wir nur einen Todo-Eintrag vom Typ *Frame*. Da *App* zwei Argumente besitzt, erwartet man vielleicht zwei Konstruktoren, einen für $\mathit{App}\ (\bullet, e_1)$ und einen weiteren für $\mathit{App}\ (v, \bullet)$ — tatsächlich benötigen wir nur den ersteren.

```
type Frame =
  | ...
  | App1 of Env * Expr            // App (•, e1)
```

Die Schrittfunktion setzt wie immer die Regeln der dynamischen Semantik um.

```
let step : State → State = function
// Auswertung
  | ...
  | Eval (δ, Fun (x, e),  stack) → Ret (Closure (δ, x, e), stack)
  | Eval (δ, App (e, e1), stack) → Eval (δ, e, App1 (δ, e1) :: stack)
```

Der Funktionsabschluss friert eine Berechnung ein: Wir merken uns alle „Zutaten“ im Funktionsabschluss, die aktuelle Umgebung, den formalen Parameter und den Funktionsrumpf. (Alle auf der linken Seiten eingeführten Bezeichner, δ, x, e, und *stack*, werden auf der rechten Seite genau einmal verwendet.) Wie schon angedeutet wird eine Funktionsanwendung in zwei, nicht in drei Schritten ausgerechnet. Im ersten Schritt bestimmen wir die Funktion (Code oben).

```
// Rückgabe
  | ...
  | Ret (Closure (δ', x1, e'), App1 (δ, e1) :: stack) → Eval (δ, e1, Let1 (δ', x1, e') :: stack)
  | Ret (v,                   App1 _ :: stack) → error "type mismatch"
```

Um den zweiten Schritt zu verstehen, ist es hilfreich, sich ins Gedächtnis zu rufen, dass die Wertedefinition ***let*** $x_1 = e_1$ ***in*** e' auch durch (***fun*** $x_1 \to e'$) e_1 ausgedrückt werden kann. Umgekehrt entspricht die Kombination aus einem Funktionsabschluss $\mathit{Closure}\ (\delta', x_1, e')$ und einer noch zu tätigen Funktionsanwendung $\mathit{App}_1\ (\delta, e_1)$ einer Wertebindung. Diesen Zusammenhang nutzt die Regel aus. Wenn wir den Zustandsübergang für ***let***-Ausdrücke

$$\mid \mathit{Eval}\ (\delta, \mathit{Let}\ (x_1, e_1, e), \mathit{stack}) \rightarrow \mathit{Eval}\ (\delta, e_1, \mathit{Let}_1\ (\delta, x_1, e) :: \mathit{stack})$$

mit der obigen Regel vergleichen, stellen wir allerdings einen kleinen, aber gewichtigen Unterschied fest: Im Fall der Funktionsanwendung haben wir es mit *zwei* unterschiedlichen Umgebungen zu tun. Dies liegt daran, dass die Definition der Funktion und der Funktionsaufruf räumlich getrennt sind und möglicherweise in unterschiedlichen Umgebungen ausgewertet werden.

Auch bei der Funktionsanwendung $e\ e_1$ kann ein Laufzeitfehler auftreten, nämlich dann, wenn e nicht zu einem Funktionsabschluss auswertet, zum Beispiel ist der Ausdruck $\mathit{App}\ (\mathit{Num}\ 1, \mathit{Num}\ 2)$ nicht wohlgetypt.

Laufzeitmesssungen Jetzt da Funktionsausdrücke in Mini^2 zur Verfügung stehen, können wir lange, sehr lange Rechnungen mit wenig Programmieraufwand durchführen lassen[13], so dass wir der interessanten Frage nachgehen wollen, wie sich unser Interpreter im Vergleich zum F# Interpreter schlägt. Die Programmiersprache F# wird ebenfalls mit Hilfe einer abstrakten Maschine implementiert, die die sogenannte „Common Intermediate Language“ (CLI) versteht. Unsere Machine wird somit von einer anderen Maschine ausgeführt, die ihrerseits auf der realen Maschine, der tatsächlichen Hardware, mittels eines Programms emuliert wird. Moderne Prozessoren legen beim Rechnen eine phänomenale Geschwindigkeit an den Tag — aber, wie weit entschleunigen wir die Rechenleistung, wenn wir Abstraktionsschicht um Abstraktionsschicht einziehen?

Wie schnell rechnet der F#-Interpreter? Mit Hilfe der folgenden Funktionen können wir ihn eine Zeitlang beschäftigen.

$$\begin{aligned}
&\textbf{\textit{let}}\ \mathit{twice} &&= \textbf{\textit{fun}}\ f\ x \rightarrow f\ (f\ x)\\
&\textbf{\textit{let}}\ \mathit{thrice} &&= \textbf{\textit{fun}}\ f\ x \rightarrow f\ (f\ (f\ x))\\
&\textbf{\textit{let}}\ \mathit{succ} &&= \textbf{\textit{fun}}\ n \rightarrow n + 1
\end{aligned}$$

Die Funktion *twice* wendet ihr erstes Argument zweimal auf ihr zweites Argument an; *thrice* entsprechend dreimal. Die Funktionen lassen sich auch kombinieren: *twice thrice* ist eine Funktion, die ihr erstes Argument neunmal auf ihr zweites Argument anwendet; *thrice twice* nur achtmal. (Erkennen Sie die Gesetzmäßigkeit?) Mit dem Kommando `#time` instruiert man den F#-Interpreter, zusätzlich zum Rechenergebnis die benötigte Rechenzeit in Sekunden mitzuteilen.

```
Mini⟩ #time
Mini⟩ twice twice twice twice succ 0
CPU: 0.010
65536
Mini⟩ twice twice twice thrice succ 0
CPU: 2.420
43046721
```

[13] Da Mini^2 als dynamisch getypte Sprache von der Disziplin und von den Zwängen eines statischen Typsystems befreit ist, haben wir tatsächlich eine *berechnungsuniverselle* Sprache vor uns, mit der man die prinzipiellen Möglichkeiten eines Rechners ausschöpfen kann — mehr dazu später aus der Abteilung der Theoretischen Informatik.

Vielleicht ist klar, dass beide Ergebnisse erzielt werden, indem die Nachfolgerfunktion *succ* ausgehend von 0 entsprechend oft angewendet wird. Da wir nur an der Rechenzeit interessiert sind, schweigen wir uns über die weiteren Details aus. An dieser Stelle nur ein kleiner Hinweis:

$$65536 = 2^{16} = 2^{2^{2^2}} \qquad \text{und} \qquad 43046721 = 3^{16} = 3^{2^{2^2}}$$

Also, in rund 2 Sekunden können wir bis 40 Millionen zählen — so hoch hängt die Messlatte.

Hier sind die gleichen Programme in der abstrakten Syntax von Mini^2.

```
let Twice  = funs (["f";"x"], App (Id "f", App (Id "f", Id "x")))
let Thrice = funs (["f";"x"], App (Id "f", App (Id "f", App (Id "f", Id "x"))))
let Succ   = Fun ("n", add (Id "n", Num 1))
```

Um den Ausdruck e auszuwerten, wenden wir die Schrittfunktion ausgehend vom Startzustand $Eval\ (Empty, e, [\,])$ solange an, bis wir einen Endzustand der Form $Ret\ (v, [\,])$ erhalten.

```
let eval (e : Expr) : Value =
  let rec loop = function
    | Ret (v, []) -> v
    | state       -> loop (step state)
  loop (Eval (Empty, e, []))
```

Wiederholen wir die obigen Rechnungen, diesmal in Mini^2, nicht in F#.

```
Mini⟩  eval (apps Twice [Twice; Twice; Twice; Succ; Num 0])
CPU: 0.070
Nat 65536
Mini⟩  eval (apps Twice [Twice; Twice; Thrice; Succ; Num 0])
CPU: 23.960
Nat 43046721
```

Jetzt benötigen wir rund 20 Sekunden, um bis 40 Millionen zu zählen. Das ist ganz erstaunlich. Es war zu erwarten, das die zusätzliche Abstraktionsschicht eine Verlangsamung um mehrere Größenordnungen nach sich zieht — die Tests deuten an, dass es nur ein Faktor im einstelligen Bereich ist. Das ist, wie gesagt, ganz erstaunlich.

Um eine Idee davon zu bekommen, wieviele Rechenschritte unser Interpreter benötigt, programmieren wir eine Variante von *eval*, die zusätzlich die Anzahl der Schritte zählt.

```
let count-steps (e : Expr) : Value * Nat =
  let rec loop = function
    | (Ret (v, []), n) -> (v, n)
    | (state,       n) -> loop (step state, n + 1)
  loop (Eval (Empty, e, []), 0)
```

Anstelle des Zeitnehmers von F# verwenden wir jetzt unseren eigenen Schrittzähler.

```
Mini⟩ count-steps (apps Twice [Twice; Twice; Twice; Succ; Num 0])
(Nat 65536, 917710)
Mini⟩ count-steps (apps Twice [Twice; Twice; Thrice; Succ; Num 0])
(Nat 43046721, 495037500)
```

Um bis 40 Millionen zu zählen, werden rund eine halbe Milliarde (!) Schritte getätigt. Allerdings: Ein Schritt entspricht *nicht* der Anwendung einer Auswertungsregel der dynamischen Semantik; unser Interpreter ist kleinschrittiger: Um eine Regel mit n Voraussetzungen abzuarbeiten, werden $n+1$ Schritte benötigt. Nichtsdestotrotz: In der Sekunde schafft unsere Maschine rund 20 Millionen Rechenschritte. Hätten Sie das gedacht?

4.5.5. Rekursive Funktionsdefinitionen⋆

module Interpreter. Functions

> *„The world, marm," said I, anxious to display my acquired knowledge, „is not exactly round, but resembles in shape a flattened orange; and it turns on its axis once in twenty-four hours."*
>
> *„Well, I don't know anything about its axes," replied she, „but I know it don't turn round, for if it did we'd be all tumbled off; and as to its being round, any one can see it's a square piece of ground, standing on a rock!"*
>
> *„Standing on a rock! but upon what does that stand?"*
>
> *„Why, on another, to be sure!"*
>
> *„But what supports the last?"*
>
> *„Lud! child, how stupid you are! There's rocks all the way down!"*
>
> — New-York Mirror

Kommen wir zum krönenden Abschluss, dem großen Finale: der Implementierung rekursiver Funktionsdefinitionen.

```
type Expr =
    | ...
    | Letrec of Id * Id * Expr * Expr          // let rec f x1 = e in e'
```

Ähnlich wie Wertedefinitionen sind rekursive Funktionsdefinitionen im Vergleich zu Mini–F# einfacher gestrickt: der Ausdruck $Letrec\ (f, x_1, e, e')$ entspricht in der konkreten Syntax **let rec** $f\ x_1 = e$ **in** e', kombiniert also eine rekursive Deklaration mit einem **in**-Ausdruck.

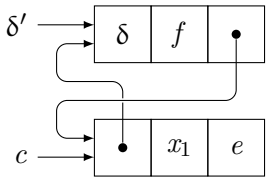
Abweichend von unserem bisherigen Vorgehen wenden wir uns unmittelbar der „Schrittfunktion" zu. Jetzt wird es abenteuerlich:

```
let step : State → State = function
// Auswertung
    | ...
    | Eval (δ, Letrec (f, x1, e, e'), stack) → let rec δ' = Comma (δ, f, c)
                                               and     c  = Closure (δ', x1, e)
                                               in Eval (δ', e', stack)
```

Der durch die rekursive Funktionsdefinition eingeführte Bezeichner f wird an einen Funktionsabschluss gebunden, der die erweiterte Umgebung als erste Komponente enthält. Dann wird mit der Auswertung von e' weitergemacht. Damit ist die Implementierung des neuen Sprachfeatures abgeschlossen.

Das war's? Yup, mehr ist nicht zu tun! Aber vielleicht stellt sich ein mulmiges Gefühl ein: Die Umgebung δ' enthält den Funktionsabschluss c, der wiederum die Umgebung δ' enthält, die wiederum den Funktionsabschluss c enthält … ein unendlicher Regress.

Im obigen Programmfragment werden die Bezeichner δ' und c durch eine rekursive *Werte*definition eingeführt — ein Feature von F#, das wir bisher aus guten Gründen nicht besprochen haben. Wird die Definition abgearbeitet, entsteht das folgende *zyklische Geflecht*.



Die meisten rekursiven *Werte*definitionen ergeben keinen Sinn: ***let rec*** $x = x+1$ wird zum Beispiel als fehlerhaft zurückgewiesen, da der Wert von x benötigt wird (rechte Seite), um den Wert von x zu bestimmen (linke Seite). Unser Fall ist aber unproblematisch, da die Konstruktoren Daten konstruieren. Unproblematisch ist vielleicht übertrieben — die Implementierung ist tatsächlich sehr trickreich: Die „Kästchen" in der obigen Graphik werden zunächst provisorisch mit Dummywerten angelegt: $\delta' = \mathit{Comma}\ (\bullet, \bullet, \bullet)$ und $c = \mathit{Closure}\ (\bullet, \bullet, \bullet)$; dann werden die Argumente der Konstruktoren ausgewertet und die Platzhalter anschließend durch die resultierenden Werte ersetzt — den letzten Schritt nennt man auch „*Back Patching*".

Im Prinzip könnten wir auf diese Weise auch die in Abschnitt 3.6 eingeführten Auswertungsregeln vereinfachen und insbesondere rekursive Funktionsabschlüsse überflüssig machen. Unsere abstrakte Maschine implementiert die unten aufgeführte Auswertungsregel. Der Vorteil des Ansatzes ist, wie gesagt, dass es nur eine Art von Funktionsabschluss gibt und somit nur eine Auswertungsregel für die Funktionsapplikation.

$$\frac{\delta' \vdash e' \Downarrow v}{\delta_0 \vdash \mathit{Letrec}\ (f, x_1, e, e') \Downarrow v} \quad \text{mit} \quad \begin{array}{lcl} \delta' & = & \delta_0, \{f \mapsto c\} \\ c & = & \langle \delta',\ x_1,\ e \rangle \end{array}$$

$$\frac{\delta \vdash e_0 \Downarrow \langle \delta',\ x_1,\ e \rangle \qquad \delta \vdash e_1 \Downarrow v_1 \qquad \delta', \{x_1 \mapsto v_1\} \vdash e \Downarrow v}{\delta \vdash \mathit{App}\ (e_0, e_1) \Downarrow v}$$

Warum nur „im Prinzip"? Nun, es ist überhaupt nicht klar, was die Definition mathematisch bedeutet — die endliche Abbildung δ' enthält ja sich selbst als Komponente ihres Wertebereichs! Eine Erklärung dieses Phänomens ist möglich, dafür muss man aber schwere mathematische Geschütze auffahren: entweder eine andere Mengenlehre verwenden („Non-wellfounded Set Theory") oder die Mengenlehre durch die Theorie der vollständigen Halbordnungen ersetzen („Domain Theory"). Beide Geschütze werden als zu schwer erachtet.

Wir sind in Abschnitt 3.6 diesem fundamentalen Problem aus dem Weg gegangen, indem wir die Bildung der Umgebung δ' sozusagen verzögern: Sie wird nicht bei der Definition vorgenommen, sondern bei *jedem* Aufruf der rekursiven Funktion, jedesmal auf's Neue. Zu diesem Zweck haben wir rekursive Funktionsabschlüsse eingeführt und eine zweite Auswertungsregel für Funktionsapplikationen formuliert.

$$\frac{\delta_0, \{f \mapsto \langle \delta_0, f, x_1, e\rangle\} \vdash e' \Downarrow v}{\delta_0 \vdash \mathit{Letrec}\ (f, x_1, e, e') \Downarrow v}$$

$$\frac{\delta \vdash e_0 \Downarrow \langle \delta_0, f, x_1, e\rangle \qquad \delta \vdash e_1 \Downarrow v_1 \qquad \delta_0, \{f \mapsto \langle \delta_0, f, x_1, e\rangle\}, \{x_1 \mapsto v_1\} \vdash e \Downarrow v}{\delta \vdash \mathit{App}\ (e_0, e_1) \Downarrow v}$$

Wenn man die beiden Regelpaare genau studiert, sieht man hoffentlich, dass der Nettoeffekt der gleiche ist.

Tail Call Optimization Kehren wir aus den Sphären der höheren Mathematik zurück in die Niederungen der abstrakten Maschine. Mit Hilfe einer rekursiven Funktionsdefinition können wir zum Beispiel die Fakultätsfunktion programmieren.

```
let Factorial =
  Letrec ("fac", "n", If (equ (Id "n", Num 0),
                          Num 1,
                          mul (Id "n", App (Id "fac", sub (Id "n", Num 1)))), Id "fac")
```

Wenn wir die Fakultät ausrechnen, dann wächst beim rekursiven Abstieg die Todo-Liste stetig an. Abbildung 3.3 illustriert die Auswertung von *factorial* 3 per Hand; unsere abstrakte Maschine durchläuft grob folgende Zustände.

```
...; Bin2 (Nat 3, Mul)])
...; Bin2 (Nat 2, Mul); Bin2 (Nat 3, Mul)])
...; Bin2 (Nat 1, Mul); Bin2 (Nat 2, Mul); Bin2 (Nat 3, Mul)])
Ret (Nat 1, [Bin2 (Nat 1, Mul); Bin2 (Nat 2, Mul); Bin2 (Nat 3, Mul)])
Ret (Nat 1, [Bin2 (Nat 2, Mul); Bin2 (Nat 3, Mul)])
Ret (Nat 2, [Bin2 (Nat 3, Mul)])
Ret (Nat 6, [])
```

Erst wenn der Basisfall erreicht ist, wird die Todo-Liste wieder peu à peu abgearbeitet. Mit anderen Worten, um $n!$ auszurechnen, benötigen wir nicht nur eine lineare Anzahl an Schritten, sondern wir haben auch einen linearen Platzbedarf für den Stack.

Raum ist im gewissen Sinne eine kostbarere Resource als Zeit. Man kann ohne große Anstrengungen doppelt so lange warten; den verfügbaren Raum zu verdoppeln, gelingt in der Regel nicht so einfach. Diesem Phänomen begegnet man auch beim Programmieren. Schauen wir uns ein paar Beispielaufrufe an — damit die Ergebnisse nicht so groß werden, verwenden wir an Stelle der Fakultätsfunktion ihre kleine Schwester.

```
let rec sum n = if n = 0 then 0 else n + sum (n - 1)
```

Während die Fakultät dem Produkt der Zahlen von 1 bis n entspricht, berechnet *sum* n deren Summe.[14] Die Tests verlaufen ernüchternd:

```
Mini⟩ sum 50000
1250025000
Mini⟩ sum 75000
Stack overflow.
```

Wenn wir in die Nähe sechsstelliger Argumente kommen, erhalten wir anstatt des Ergebnisses eine Fehlermeldung, den berüchtigten *Stack Overflow*. Was ist passiert? Der F#-Interpreter reserviert für die Todo-Liste, den Stack, eine begrenzte Menge an Speicherplatz; wenn dieser Platz ausgeschöpft ist, wird die Rechnung gnadenlos abgebrochen. Zum Vergleich: Wenn eine Rechnung viel Zeit in Anspruch nimmt, können wir entscheiden, ob wir sie abbrechen oder nicht. Bei Platzmangel wird uns die Entscheidung abgenommen — es gibt auch keine Möglichkeit, den Platz zu vergrößern und anschließend die Rechnung fortzusetzen.

Wie können wir das Problem lösen? Nun, wir kennen die Ursache für den Abbruch: Die Funktion *sum* erledigt die Hauptarbeit beim rekursiven *Auf*stieg: Nach jedem rekursiven Aufruf, *sum* $(n \dot{-} 1)$, ist noch Arbeit zu erledigen, $n + \bullet$, die wir uns auf der Todo-Liste merken müssen. Die Programmiertechnik des *Rekursionsparadoxons* hilft uns an dieser Stelle weiter: Wir programmieren eine Funktion, die die Zahlen von 1 bis n summiert und zusätzlich auf das Ergebnis eine weitere Zahl addiert.

$$accumulate\ a\ n = a + sum\ n \tag{4.3}$$

Aus dieser *Spezifikation* lässt sich systematisch eine Implementierung herleiten. Analog zur Funktion *sum* nehmen wir eine Fallunterscheidung über n vor. **Fall** $n = 0$**:**

$$\begin{aligned}
& accumulate\ a\ 0 \\
= & \quad \{\ \text{Spezifikation von } accumulate\ (4.3)\ \} \\
& a + sum\ 0 \\
= & \quad \{\ \text{Definition von } sum\ \} \\
& a + 0 \\
= & \quad \{\ 0 \text{ ist das neutrale Element von '+'}\ \} \\
& a
\end{aligned}$$

[14] Natürlich würden wir die Summe so nicht berechnen, da wir mit Hilfe der Gaußschen Summenformel $\sum_{i=1}^{n} = \binom{n+1}{2} = n \cdot (n+1)/2$ das Ergebnis sehr viel schneller bestimmen können:

```
let sum n = (n * (n + 1)) / 2
```

Fall $n > 0$**:**

$$
\begin{aligned}
& \mathit{accumulate}\ a\ n \\
= \quad & \{ \text{ Spezifikation von } \mathit{accumulate}\ (4.3) \ \} \\
& a + \mathit{sum}\ n \\
= \quad & \{ \text{ Definition von } \mathit{sum} \ \} \\
& a + (n + \mathit{sum}\ (n \dot{-} 1)) \\
= \quad & \{ \text{ `+' ist assoziativ } \} \\
& (a + n) + \mathit{sum}\ (n \dot{-} 1) \\
= \quad & \{ \text{ Spezifikation von } \mathit{accumulate}\ (4.3) \ \} \\
& \mathit{accumulate}\ (a + n)\ (n \dot{-} 1)
\end{aligned}
$$

Fassen wir den ersten und den letzten Ausdruck jeweils zu einer Rechenregel zusammen, erhalten wir das folgende Programm.

$$\textbf{\textit{let rec}}\ \mathit{accumulate}\ a\ n = \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ a\ \textbf{\textit{else}}\ \mathit{accumulate}\ (a + n)\ (n \dot{-} 1)$$

Die Funktion erledigt die gesamte Arbeit während des rekursiven *Ab*stiegs: Nach dem rekursiven Aufruf $\mathit{accumulate}\ (a+n)\ (n \dot{-} 1)$ ist nichts mehr zu tun. Der erste Parameter heißt übrigens aus naheliegenden Gründen auch *Akkumulator*. Der Lohn für unsere Mühen: Der begrenzte Stackplatz bereitet nicht länger Probleme.

```
Mini⟩ accumulate 0 50000
1250025000
Mini⟩ accumulate 0 75000
2812537500
```

Im Fachjargon nennt man *accumulate* eine *endrekursive* Funktion (engl. tail recursive), da alle rekursiven Aufrufe an einer *Endposition* stehen: direkt in den Zweigen einer Alternative, einer Fallunterscheidung mit ***match*** oder im Rumpf einer Wertebindung. Zum Vergleich: *sum* ist *nicht* endrekursiv, da der rekursive Aufruf Argument einer Additionsoperation ist.

Die Optimierung beliebiger Aufrufe an Endposition nennt man *Tail Call Optimization* (TCO); werden nur end*rekursive* Aufrufe optimiert, spricht man von *Tail Recursion Elimination* (TRE). Viele gängige Programmiersprachen (z.B. C, C++, Java) unterstützen weder die eine noch die andere Optimierung.[15] Das liegt daran, dass Programmieraufgaben in diesen Sprachen aus historischen Gründen bevorzugt iterativ und nicht rekursiv angegangen werden — Abschnitt 7.3 stellt Kontrollstrukturen für die iterative Programmierung vor.

Unsere abstrakte Maschine unterstützt wie schon mehrfach angedeutet Tail Call Optimization. Die Definition einer Endposition lässt sich unmittelbar aus den Auswertungsregeln der dynamischen Semantik ableiten. Die Regeln für Alternativen, Wertdefinitionen

[15] Im Fall von C und C++ werden die Optimierungen vom Sprachstandard nicht vorgeschrieben. Viele gängige Übersetzer wie z.B. GCC oder Clang implementieren aber die Optimierungen.

und Funktionsaufrufe haben alle die gleiche schematische Form:

$$\frac{\delta_1 \vdash e_1 \Downarrow v_1 \qquad \cdots \qquad \delta_n \vdash e_n \Downarrow v}{\delta \vdash e \Downarrow v}$$

Der Wert v des letzten Teilausdrucks wird jeweils unverändert nach unten weitergereicht. Da die abstrakte Machine die Regeln buchstabengetreu umsetzt, werden Funktionsaufrufe an Endpositionen *ohne zusätzlichen Stackverbrauch* effizient abgearbeitet.

Um die Optimierung plastisch vor Augen zu führen, verwenden wir nicht *accumulate*, sondern die einfachste aller endrekursiven Funktionen, ***let rec*** *forever* n = *forever* n bzw. in abstrakter Syntax:

let *Forever* = *Letrec* (`"forever"`, `"n"`, *App* (*Id* `"forever"`, *Id* `"n"`), *Id* `"forever"`)

Für das zyklische Geflecht aus Umgebung und Funktionsabschluss vereinbaren wir Abkürzungen: δ = *Comma* (*Empty*, `"f"`, c) und c = *Closure* (δ, `"n"`, *App* (*Id* `"f"`, *Id* `"n"`)).[16] Die Auswertung von *forever* 0 nimmt folgenden Verlauf (wir kürzen *forever* mit f ab).

Eval (*Empty*, *App* (*Letrec* (`"f"`, `"n"`, *App* (*Id* `"f"`, *Id* `"n"`), *Id* `"f"`), *Num* 0), [])
Eval (*Empty*, *Letrec* (`"f"`, `"n"`, *App* (*Id* `"f"`, *Id* `"n"`), *Id* `"f"`), [*App*$_1$ (*Empty*, *Num* 0)])
Eval (δ, *Id* `"f"`, [*App*$_1$ (*Empty*, *Num* 0)])
Ret (c, [*App*$_1$ (*Empty*, *Num* 0)])
Eval (*Empty*, *Num* 0, [*Let*$_1$ (δ, `"n"`, *App* (*Id* `"f"`, *Id* `"n"`))])
Ret (*Nat* 0, [*Let*$_1$ (δ, `"n"`, *App* (*Id* `"f"`, *Id* `"n"`))])

An dieser Stelle ist die Funktion und ihr Argument ausgerechnet und es wird mit der Auswertung des Funktionsrumpfes fortgefahren.

Eval (*Comma* (δ, `"n"`, *Nat* 0), *App* (*Id* `"f"`, *Id* `"n"`), [])
Eval (*Comma* (δ, `"n"`, *Nat* 0), *Id* `"f"`, [*App*$_1$ (*Comma* (δ, `"n"`, *Nat* 0), *Id* `"n"`)])
Ret (c, [*App*$_1$ (*Comma* (δ, `"n"`, *Nat* 0), *Id* `"n"`)])
Eval (*Comma* (δ, `"n"`, *Nat* 0), *Id* `"n"`, [*Let*$_1$ (δ, `"n"`, *App* (*Id* `"f"`, *Id* `"n"`))])
Ret (*Nat* 0, [*Let*$_1$ (δ, `"n"`, *App* (*Id* `"f"`, *Id* `"n"`))])

Nach fünf weiteren Schritten landen wir wieder im identischen Zustand! Mit anderen Worten, die Auswertung terminiert nicht; sie wird aber auch nicht durch einen „*Stack Overflow*“ abgebrochen. Zum Vergleich: Der Aufruf von ***let rec*** *forever* n = n+*forever* n terminiert zwar in der Theorie nicht, aber in der Praxis ist irgendwann der Stackplatz erschöpft.

Wir haben bisher nicht-terminierende Programme geächtet — dieses Urteil ist aber zu engstirnig. Viele Programme sollen ewig oder zumindest lange laufen: das Betriebssystem eines Rechners, der Webserver eines Online-Shops oder, um ein geläufigeres Beispiel

[16] Man könnte argumentieren, dass diese Definitionen tatsächlich notwendig sind, da man ein zyklisches Geflecht nicht oder zumindest nicht einfach ausgeben kann. Der F#-Interpreter geht diesem Problem mit einem Trick aus dem Weg: Er zeigt Datenstrukturen nur bis zu einer festgelegten Schachtelungstiefe an; ist diese erreicht, wird das Auslassungszeichen „…“ ausgegeben.

zu nennen, der Kommandozeileninterpreter (CLI) von F#. Damit diese Programme tatsächlich ewig laufen, muss man neben vielen anderen Dingen darauf achten, dass kein Stack Overflow auftritt — verwendet man endrekursive Funktionen, so muss die Programmiersprache TCO oder zumindest TRE unterstützen. (In Abschnitt 7.4.4 schauen wir uns konkrete Beispiele an: Kommandozeileninterpreter für einfache Taschenrechner.) Diese Beispiele gehören übrigens zur Gruppe der sogenannten *reaktiven Programme*; sie reagieren auf Stimuli der Umwelt. Von einem reaktiven Programm fordert man zwar nicht, dass es terminiert, aber man wünscht sich, dass es auf einen Stimulus in angemessener oder zumindest in endlicher Zeit reagiert. (Wenn man möchte, kann man Funktionen als Spezialfall reaktiver Programme auffassen: Auf einen Stimulus, den Funktionsaufruf, folgt eine Reaktion, das Funktionsergebnis; damit ist die Interaktion allerdings beendet.) Zu diesem Thema erfahren Sie in höheren Semestern mehr.

Kommen wir zum Schluß noch einmal auf das Ausgangsproblem zurück, dem Stack Overflow als Folge „tiefer“ Rekursion. Interessanterweise kann unsere Maschine auch die ursprüngliche Definition von *sum*

```
let Sum =
  Letrec ("sum", "n", If (equ (Id "n", Num 0),
                          Num 0,
                          add (Id "n", App (Id "sum", sub (Id "n", Num 1)))), Id "sum")
```

für beliebig große Argumente ausrechnen.

```
Mini⟩  eval (App (Sum, Num 50000))
Nat 1250025000
Mini⟩  eval (App (Sum, Num 75000))
Nat 2812537500
Mini⟩  eval (App (Sum, Num 1000000))
Nat 500000500000
```

Obwohl unser Interpreter vom F#-Interpreter ausgeführt wird, tritt kein Stack Overflow auf. Wie ist das zu erklären? Zum ersten ist die Funktion *eval* endrekursiv; somit kann der F#-Interpreter beliebig tiefe rekursive Aufrufe problemlos abarbeiten — sie erinnern sich: eine Milliarde Schritte kommen schnell zusammen. Zum zweiten wird der Stack in unserer Maschine mittels einer *dynamischen* Datenstruktur verwaltet, einer Liste, die problemlos wachsen und auch schrumpfen kann. Der F#-Interpreter verwendet im Gegensatz dazu eine *statische* Datenstruktur, ein Array, das eine feste, vorher festgelegte Größe hat.

Die Erklärung wirft vielleicht eine andere Frage auf. Wo wird die Liste oder allgemeiner wo werden eigentlich Daten und Datenstrukturen gespeichert? Neben dem Stack gibt es zu diesem Zweck einen zweiten Speicherbereich, den sogenannten *Heap*[17]. Der Name „Haufen“ (engl. heap) deutet die chaotische Organisationsstruktur an: Daten werden dort abgelegt, wo Platz ist; geht der freie Platz zur Neige, räumt man auf und wirft

[17] Dieser Heap hat nichts mit der gleichnamigen Datenstruktur zu tun, mit der Prioritätswarteschlangen implementiert werden, siehe Abschnitt 5.4.

nicht mehr benötigte Daten weg; lässt sich nichts mehr freiräumen, versucht man den Speicherbereich zu vergrößern; erst wenn man an die physischen Grenzen der Hardware stößt, muss man aufgeben und meldet einen „*Heap Overflow*“. (So chaotisch die Struktur des Heaps, so clever und ausgefeilt sind die Algorithmen, die ihn verwalten.)

Abbildungen 4.4 und 4.5 führen den vollständigen Code für die letzte Ausbaustufe unseres Interpreters auf. Das Programm ist erstaunlich kompakt: Um die abstrakte Syntax und die dynamische Semantik von Mini^2 zu definieren, sind lediglich zwei Seiten nötig — immerhin handelt es sich bei Mini^2 um eine berechnungsuniverselle Sprache.

module
Interpreter.
Continuations

4.5.6. Fortsetzungen⋆⋆

I can resist everything except temptation.

— Oscar Fingal O'Flahertie Wills Wilde (1854–1900), *Lady Windermere's Fan*

> *For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all „higher level“ programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.*
>
> *My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the „making“ of the corresponding process is delegated to the machine.*
>
> *My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.*
>
> — Edsger W. Dijkstra (1930–2002), *Go To Statement Considered Harmful*

Jetzt da wir über eine eigene Implementierung von Mini^2 verfügen, ist die Versuchung groß, zusätzliche Sprachfeatures zu verwirklichen, Konzepte, die weder von Mini–F# noch von F# selbst unterstützt werden. Wenn Sie allerdings Dijkstra zustimmen, dann sollten Sie *nicht* weiterlesen.

Unsere abstrakte Maschine setzt die Auswertungsregeln der dynamischen Semantik recht buchstabengetreu um. Mit Hilfe der Todo-Liste, des Stacks, wird dabei die Auswertung geschachtelter Ausdrücke sequentialisiert, so dass ein Ausdruck peu à peu abgearbeitet werden kann. Zu jedem Zeitpunkt der Abarbeitung repräsentiert der Stack den „Rest der Rechnung“ — wie muss mit der Rechnung fortgefahren werden, wenn

```
// endliche Abbildungen
type Map ⟨'key, 'val⟩ =
    | Empty
    | Comma of Map ⟨'key, 'val⟩ * 'key * 'val
let rec lookup (x : 'key) : Map ⟨'key, 'val⟩ → 'val option when 'key : equality = function
    | Empty                  → None
    | Comma (env, x1, v1) → if x = x1 then Some v1 else lookup x env
// abstrakte Syntax
type Op =| Add | Sub | Mul | Div | Mod | Lt | Lte | Equ | Neq | Gte | Gt
type Id = String
type Expr =
    | Num    of Nat                    // n
    | Bin    of Expr * Op * Expr       // e1 + e2, e1 - e2 etc.
    | False                            // false
    | True                             // true
    | If     of Expr * Expr * Expr     // if e1 then e2 else e3
    | Id     of Id                     // x
    | Let    of Id * Expr * Expr       // let x1 = e1 in e
    | Fun    of Id * Expr              // fun x → e
    | App    of Expr * Expr            // e e1
    | Letrec of Id * Id * Expr * Expr  // let rec f x1 = e in e'
type Value =
    | Bool    of Bool                  // n
    | Nat     of Nat                   // false oder true
    | Closure of Env * Id * Expr       // ⟨δ, x, e⟩
and Env = Map ⟨Id, Value⟩
// dynamische Semantik
let primitive : Value * Op * Value → Value option = function
    | (Nat n1, Add, Nat n2) → Some (Nat  (n1 +  n2))
    | (Nat n1, Sub, Nat n2) → Some (Nat  (n1 -  n2))
    | (Nat n1, Mul, Nat n2) → Some (Nat  (n1 *  n2))
    | (Nat n1, Div, Nat n2) → Some (Nat  (n1 /  n2))
    | (Nat n1, Mod, Nat n2) → Some (Nat  (n1 %  n2))
    | (Nat n1, Lt,  Nat n2) → Some (Bool (n1 <  n2))
    | (Nat n1, Lte, Nat n2) → Some (Bool (n1 <= n2))
    | (Nat n1, Equ, Nat n2) → Some (Bool (n1 =  n2))
    | (Nat n1, Neq, Nat n2) → Some (Bool (n1 <> n2))
    | (Nat n1, Gte, Nat n2) → Some (Bool (n1 >= n2))
    | (Nat n1, Gt,  Nat n2) → Some (Bool (n1 >  n2))
    | _                     → None
```

Abbildung 4.4.: Ein-Schritt Auswerter für Mini^2 (1. Teil).

```
type Frame =
  | Bin1 of Env * Op * Expr          // δ, Bin (•, op, e2)
  | Bin2 of Value * Op               // Bin (v1, op, •)
  | If1  of Env * Expr * Expr        // δ, If (•, e2, e3)
  | Let1 of Env * Id * Expr          // δ, Let (x1, •, e)
  | App1 of Env * Expr               // δ, App (•, e1)
type State =
  | Eval of Env * Expr * Frame list  // δ ⊢ e ⇓ ··· , stack
  | Ret  of Value      * Frame list  // ··· ⇓ v, stack
let step : State → State = function
// Auswertung
  | Eval (δ, Num n,          stack)  → Ret (Nat n, stack)
  | Eval (δ, Bin (e1, op, e2), stack) → Eval (δ, e1, Bin1 (δ, op, e2) :: stack)
  | Eval (δ, False,          stack)  → Ret (Bool false, stack)
  | Eval (δ, True,           stack)  → Ret (Bool true, stack)
  | Eval (δ, If (e1, e2, e3), stack) → Eval (δ, e1, If1 (δ, e2, e3) :: stack)
  | Eval (δ, Id x,           stack)  → match lookup x δ with
                                         | None   → error ("'" ^ x ^ "' is undefined")
                                         | Some v → Ret (v, stack)
  | Eval (δ, Let (x1, e1, e), stack) → Eval (δ, e1, Let1 (δ, x1, e) :: stack)
  | Eval (δ, Fun (x, e),     stack)  → Ret (Closure (δ, x, e), stack)
  | Eval (δ, App (e, e1),    stack)  → Eval (δ, e, App1 (δ, e1) :: stack)
  | Eval (δ, Letrec (f, x1, e, e'), stack) → let rec δ' = Comma (δ, f, c)
                                             and     c = Closure (δ', x1, e)
                                             in Eval (δ', e', stack)
// Rückgabe
  | Ret (v, [])                          → Ret (v, [])
  | Ret (v1, Bin1 (δ, op, e2)   :: stack) → Eval (δ, e2, Bin2 (v1, op) :: stack)
  | Ret (v2, Bin2 (v1, op)      :: stack) → match primitive (v1, op, v2) with
                                              | Some v → Ret (v, stack)
                                              | None   → error "type mismatch"
  | Ret (Bool b, If1 (δ, e2, e3) :: stack) → if b then Eval (δ, e2, stack)
                                             else Eval (δ, e3, stack)
  | Ret (_, If1 (δ, e2, e3)     :: stack) → error "type mismatch"
  | Ret (v1, Let1 (δ, x1, e)    :: stack) → Eval (Comma (δ, x1, v1), e, stack)
  | Ret (Closure (δ', x1, e'), App1 (δ, e1) :: stack) → Eval (δ, e1, Let1 (δ', x1, e') :: stack)
  | Ret (v,                App1 _       :: stack) → error "type mismatch"
```

Abbildung 4.5.: Ein-Schritt Auswerter für Mini^2 (2. Teil).

die aktuelle Teilaufgabe erledigt ist. Wenn zum Beispiel der Teilausdruck e in $e * 3 + 4$ ausgerechnet wird, dann repräsentiert der Stack das restliche Programm $\bullet * 3 + 4$. Aus naheliegenden Gründen nennt man $\bullet * 3 + 4$ auch *Fortsetzung*.

Jetzt kommen wir zu einer vielleicht abenteuerlichen Idee: Wir ermöglichen der Programmier*in, einen Schnappschuss vom Stack zu machen und diesen zu einem späteren Zeitpunkt an Stelle des aktuellen Stacks zu verwenden. Dazu führen wir wir zwei neue Konstrukte ein: ***letcc*** x ***in*** e und ***continue*** e e_1. Mit Hilfe von ***letcc*** wird die aktuelle Fortsetzung (engl. *c*urrent *c*ontinuation, ein anderer Name für die Todo-Liste) an den Bezeichner x gebunden, der im Rumpf e sichtbar ist. Zum Beispiel wird *hop* in dem Ausdruck (***letcc*** *hop* ***in*** $\cdots$) $* 3 + 4$ an die Fortsetzung $\bullet * 3 + 4$ gebunden. Die eingefangene Fortsetzung kann anschließend mit ***continue*** *hop* e_1 verwendet werden und ersetzt den aktuellen Stack, so dass mit der Auswertung von $e_1 * 3 + 4$ weitergemacht wird. Zum Beispiel wertet

$$(\textbf{\textit{letcc}}\ \textit{hop}\ \textbf{\textit{in}}\ 1 + \textbf{\textit{continue}}\ \textit{hop}\ 2) * 3 + 4$$

zu $2 * 3 + 4 = 10$ aus. Die Rechnung wird also mit Hilfe von ***continue*** faktisch abgebrochen und an einer anderen Stelle fortgesetzt. Da auf diese Weise der Programmablauf kontrolliert bzw. gesteuert wird, nennt man ***letcc*** und ***continue*** *Kontrolloperatoren*.

Das war ein kleiner Hopser; das nächste Beispiel illustriert einen größeren Sprung aus einer Rekursion heraus.

```
let cross-product n =
  letcc exit
  in let rec f i = if i = 0 then 1
                           else  let r = i % 10
                                 in if r = 0 then continue exit 0
                                             else  r * f (i ÷ 10)
     in f n
```

Die Funktion *cross-product* berechnet das *Querprodukt* einer natürlichen Zahl, das Produkt ihrer Ziffern. Das Querprodukt von 4711 ist zum Beipiel $4 \cdot 7 \cdot 1 \cdot 1 = 28$. Das Programm implementiert eine kleine Optimierung — hier kommen die Kontrolloperatoren ins Spiel. Wenn die Ziffer 0 auftritt, wird die Rechnung abgebrochen und unmittelbar 0 als Ergebnis zurückgegeben. Zu diesem Zweck wird direkt im Rumpf der Funktion die aktuelle Fortsetzung eingefangen, also die Stelle im Programm, die *cross-product* aufgerufen hat und an die die Funktion ihr Ergebnis zurückgibt. Wollen wir zum Beispiel $4711 +$ *cross-product* 10987654321 ausrechnen, dann wird *exit* an $4711 + \bullet$ gebunden. Die Arbeitsfunktion f ist nach dem Leibniz Entwurfsmuster gestrickt und baut einen Turm von Multiplikationen auf: Sind die ersten neuen Ziffern (von rechts) abgearbeitet, repräsentiert der Stack die Fortsetzung $4711 + (1 * (2 * (3 * (4 * (5 * (6 * (7 * (8 * (9 * (\bullet)))))))))$. An dieser Stelle der Rechnung trifft f auf die Ziffer 0; anstatt das Produkt mit vorhersehbarem Ausgang auszurechnen, nimmt f den Seitenausgang, so dass mit der Auswertung von $4711 + 0$ fortgefahren wird. (Übrigens, warum definieren wir eine Hilfsfunktion?)

Kommen wir zur Implementierung der Kontrolloperatoren. Wir erweitern zunächst den Datentyp der Ausdrücke um zwei Konstruktoren.

```
type Expr =
  | ...
  | Letcc     of Id * Expr          // letcc x in e
  | Continue of Expr * Expr         // continue e e1
```

Der Typ der Werte wird entsprechend um Fortsetzungen erweitert.

```
type Value =
  | ...
  | Continuation of Frame list
and Env = Map<Id, Value>
and Frame =
  | ...
  | Continue1 of Env * Expr         // δ, Continue (•, e1)
```

Da ***continue*** ähnlich wie ein Funktionsaufruf schrittweise ausgewertet wird, fügen wir weiterhin einen neuen Todo-Eintrag zum Typ *Frame* hinzu. Die Typen *Value*, *Env* und *Frame* sind verschränkt rekursiv — eine Fortsetzung besteht aus Todo-Einträgen vom Typ *Frame* und umgekehrt enthalten Todo-Einträge Komponenten vom Typ *Value* —, somit müssen die Typdefinitionen mit ***and*** verbunden werden.

Die Schrittfunktion setzt die umgangssprachliche Beschreibung der Kontrolloperatoren ***letcc*** und ***continue*** um.

```
let step : State → State = function
// Auswertung
  | ...
  | Eval (δ, Letcc (x, e),       stack) → Eval (Comma (δ, x, Continuation stack), e, stack)
  | Eval (δ, Continue (e, e1), stack) → Eval (δ, e, Continue1 (δ, e1) :: stack)
```

In der Regel für ***letcc*** *x* ***in*** *e* wird der aktuelle Stack *dupliziert*: Wir fahren mit der Auswertung von *e* bei unverändertem Stack fort; zusätzlich wird *x* an den Stack gebunden. Ähnlich wie bei einem Funktionsaufruf wird im Fall von ***continue*** *e* e_1 zunächst mit der Auswertung von *e* weitergemacht.

```
// Rückgabe
  | ...
  | Ret (Continuation stack, Continue1 (δ, e1) :: _) → Eval (δ, e1, stack)
  | Ret (_,                 Continue1 (δ, e1) :: _) → error "type mismatch"
```

Das erste Argument von ***continue*** muss zu einer Fortsetzung auswerten. Ist das gegeben, wird der aktuelle Stack verworfen (anonymer Bezeichner „_“) bzw. durch den vorher eingefangenen Stack ersetzt und mit der Auswertung des zweiten Arguments von ***continue*** fortgefahren.

Schauen wir uns die Auswertung des „Hopsers“ an.

```
add (mul (Letcc ("hop", add (Num 1, Continue (Id "hop", Num 2))), Num 3), Num 4)
```

Aus Gründen der Übersichtlichkeit verwenden wir Abkürzungen für die Fortsetzung, an die *hop* gebunden wird, und die daraus resultierende Umgebung.

$c = \mathit{Continuation}\ [\mathit{Bin}_1\ (\mathit{Empty}, \mathit{Mul}, \mathit{Num}\ 3); \mathit{Bin}_1\ (\mathit{Empty}, \mathit{Add}, \mathit{Num}\ 4)]$
$\delta = \mathit{Comma}\ (\mathit{Empty}, \texttt{"hop"}, c)$

Zunächst arbeiten wir die arithmetischen Operationen ab, die zu Todo-Einträgen auf dem Stack führen (einige Konstruktoren kürzen wir mit ihrem ersten Buchstaben ab).

$\mathit{Eval}\ (E, B\ (B\ (\mathit{Letcc}\ (\texttt{"hop"}, B\ (N\ 1, A, \mathit{Continue}\ (\mathit{Id}\ \texttt{"hop"}, N\ 2))), M, N\ 3), A, N\ 4), [])$
$\mathit{Eval}\ (E, B\ (B\ (\mathit{Letcc}\ (\texttt{"hop"}, B\ (N\ 1, A, \mathit{Continue}\ (\mathit{Id}\ \texttt{"hop"}, N\ 2))), M, N\ 3), [\mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Eval}\ (E, \mathit{Letcc}\ (\texttt{"hop"}, B\ (N\ 1, A, \mathit{Continue}\ (\mathit{Id}\ \texttt{"hop"}, N\ 2))), [\mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$

An dieser Stelle machen wir einen Schnappschuss des Stacks und binden ihn an den Bezeichner *hop*.

$\mathit{Eval}\ (\delta, B\ (N\ 1, A, \mathit{Continue}\ (\mathit{Id}\ \texttt{"hop"}, N\ 2)), [\mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$

Mit der Auswertung des Rumpfs von ***letcc*** geht es weiter.

$\mathit{Eval}\ (\delta, N\ 1, [\mathit{Bin}_1\ (\delta, A, \mathit{Continue}\ (\mathit{Id}\ \texttt{"hop"}, N\ 2)); \mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Ret}\ (\mathit{Nat}\ 1, [\mathit{Bin}_1\ (\delta, A, \mathit{Continue}\ (\mathit{Id}\ \texttt{"hop"}, N\ 2)); \mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Eval}\ (\delta, \mathit{Continue}\ (\mathit{Id}\ \texttt{"hop"}, N\ 2), [\mathit{Bin}_2\ (\mathit{Nat}\ 1, A); \mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Eval}\ (\delta, \mathit{Id}\ \texttt{"hop"}, [\mathit{Continue}_1\ (\delta, N\ 2); \mathit{Bin}_2\ (\mathit{Nat}\ 1, A); \mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Ret}\ (c, [\mathit{Continue}_1\ (\delta, N\ 2); \mathit{Bin}_2\ (\mathit{Nat}\ 1, A); \mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$

Der Stack ist etwas angewachsen und enthält neben der Addition auch den unvollendeten Aufruf von ***continue***. Die Ausführung des Kontolloperators bewirkt, dass dieser Stack verworfen wird (insbesondere wird $1 + \bullet$ vergessen) und durch die Fortsetzung c ersetzt wird. Der Rest ist Routine.

$\mathit{Eval}\ (\delta, N\ 2, [\mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Ret}\ (\mathit{Nat}\ 2, [\mathit{Bin}_1\ (E, M, N\ 3); \mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Eval}\ (E, N\ 3, [\mathit{Bin}_2\ (\mathit{Nat}\ 2, M); \mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Ret}\ (\mathit{Nat}\ 3, [\mathit{Bin}_2\ (\mathit{Nat}\ 2, M); \mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Ret}\ (\mathit{Nat}\ 6, [\mathit{Bin}_1\ (E, A, N\ 4)])$
$\mathit{Eval}\ (E, N\ 4, [\mathit{Bin}_2\ (\mathit{Nat}\ 6, A)])$
$\mathit{Ret}\ (\mathit{Nat}\ 4, [\mathit{Bin}_2\ (\mathit{Nat}\ 6, A)])$
$\mathit{Ret}\ (\mathit{Nat}\ 10, [])$

Last thing I remember, I was
Running for the door
I had to find the passage back to the place I was before
'Relax' said the night man,
'We are programmed to receive.
You can check out any time you like,
But you can never leave!'

— Eagles (Glenn Lewis Frey, Don Felder, Donald Hugh Henley), *Hotel California*

Eine Fortsetzung ist einer Funktion nicht unähnlich; $\bullet * 3 + 4$ kann als Funktion in dem Platzhalter aufgefasst werden: ***fun*** $x \to x * 3 + 4$. Entsprechend schmeckt ***continue*** nach einem Funktionsaufruf. Kombinieren wir Funktionsabstraktion und ***continue*** können wir die Illusion einer Funktion erzeugen: Ist k eine Fortsetzung, dann ist ***fun*** $x \to$ ***continue*** $k\ x$ eine „normale“ Funktion. Das es sich um eine Illusion handelt, merkt man schnell, wenn man die Funktion aufruft: ***fun*** $x \to$ ***continue*** $k\ x$ ist eine Funktion ohne Wiederkehr, ihren Funktionswert bekommen wir nie zu Gesicht. Vielleicht dämmert es langsam: Mit der Einführung von Kontrolloperatoren verlassen wir die behagliche Welt der mathematischen Funktionen.

Dijkstra mahnt in dem Artikel „Go To Statement Considered Harmful“, die konzeptionelle Lücke zwischen dem statischen Programmtext und dem dynamischen Prozess seiner Auswertung so klein wie möglich zu halten. Die Mathematik macht es vor und treibt es im gewissen Sinne auf die Spitze: Durch ihre Brille ist eine Funktion eine statische Relation zwischen Eingaben und Ausgaben.[18] Der dynamische Prozess, der die Eingaben in die Ausgaben überführt, wird gänzlich ignoriert. (Besonders deutlich wird die Ignoranz, wenn eine Funktion mit ihrem Graphen identifiziert wird.) Im Gegensatz dazu wird in der Informatik das dynamische Wesen einer Funktion betont: Durch ihre Brille ist eine Funktion ein Algorithmus, der präzise beschreibt, wie Ein- in Ausgaben transformiert werden und welche Arbeiten auf diesem Weg zu verrichten sind. Ignoranz ist gemeinhin negativ konnotiert, aber man sollte nicht vergessen, dass sie hilft, Komplexität zu meistern. Um Dijkstras Warnung nachvollziehen zu können, schauen wir uns im Folgenden kurz an, welche weitreichenden Konsequenzen die Einführung der Kontrolloperatoren nach sich zieht, welchen Geist wir aus der Flasche gelassen haben.

Der Ausdruck ***letcc*** k ***in fun*** $v \to$ ***continue*** $k\ v$ fängt die aktuelle Fortsetzung ein und gibt sie unmittelbar als Funktion verkleidet zurück. Wenden wir den Ausdruck auf *sich selbst* an,

(***letcc*** k ***in fun*** $v \to$ ***continue*** $k\ v$) (***letcc*** k ***in fun*** $v \to$ ***continue*** $k\ v$)

erleben wir eine unangenehme Überraschung: Die Auswertung terminiert nicht! Der statische Programmtext ist kurz, aber die Dynamik seiner Ausführung ist unklar. Um eine Idee von dem dynamischen Auswertungsprozess zu bekommen, reichern wir die Funktionsrümpfe mit Ausgabeanweisungen an — ähnlich wie wir das in Abschnitt 3.7 im Kontext des Ratespiels für *player-A* gemacht haben.

let *yin* = ***letcc*** k ***in fun*** $v \to$ *putstring* `"\n"`; ***continue*** $k\ v$
let *yang* = ***letcc*** k ***in fun*** $v \to$ *putstring* `"|"` ; ***continue*** $k\ v$

Zur Erinnerung: Die Funktion *putstring* gibt einen String auf dem Bildschirm aus und ‘;’ führt zwei Rechnungen nacheinander aus. Bevor wir einen Testlauf starten können,

[18] Siehe Anhang A.1: Eine Relation $f \subseteq A \times B$ mit der Eigenschaft, dass es zu jedem $x \in A$ genau ein $y \in B$ mit $(x, y) \in f$ gibt, heißt *Abbildung* oder *Funktion*.

müssen wir natürlich zunächst unsere abstrakte Maschine erweitern — die Erweiterung ist aber nicht allzu schwer, so dass wir uns die Details sparen.

```
let Yin  = Letcc ("k", Fun ("v", Seq (Putstring "\n", Continue (Id "k", Id "v"))))
let Yang = Letcc ("k", Fun ("v", Seq (Putstring "|",  Continue (Id "k", Id "v"))))
```

Wenn wir jetzt *yin* auf *yang* anwenden, erleben wir eine faustdicke Überraschung.

```
Mini⟩ eval (App (Yin, Yang))

|
||
|||
||||
|||||
||||||
|||||||
||||||||
|||||||||
...
```

Der nichtterminierende Prozess gibt nacheinander die natürlichen Zahlen in der unären Zahlendarstellung aus — die Zahlendarstellung unserer Urahnen: für jedes erlegte Bison einen Strich! Sehen Sie dem statischen Programmtext diese Dynamik an?

Fortsetzung folgt in Abschnitt 7.3.4.

Übungen.

1. 1. Schreiben Sie eine Funktion *minimum4*, die von vier natürlichen Zahlen das Minimum bestimmt.

```
let minimum4 (a : Nat, b : Nat, c : Nat, d : Nat) : Nat
```

2. Schreiben Sie eine Funktion *sort4*, die vier natürliche Zahlen sortiert.

```
let sort4 (a : Nat, b : Nat, c : Nat, d : Nat) : Nat * Nat * Nat * Nat
```

Wieviele Vergleiche werden im schlechtesten Fall benötigt? Ist Ihre Definition optimal? Wieviele Vergleiche benötigt das beste Programm im schlechtesten Fall?

2. Der Musterabgleich wird durch die Relation $p \sim \nu \Downarrow \delta$ formalisiert. Welche Umgebungen

werden beim Abgleich der folgenden Werte und Muster jeweils erzeugt?

p	$\sim$	ν	$\Downarrow$	δ
$a \,\&\, b$	$\sim$	4711	$\Downarrow$	
$\_ \,\&\, b$	$\sim$	4711	$\Downarrow$	
$\_$	$\sim$	(4711, "Lisa")	$\Downarrow$	
$(n, \_)$	$\sim$	(4711, "Lisa")	$\Downarrow$	
(n, s)	$\sim$	(4711, "Lisa")	$\Downarrow$	
$p \,\&\, (n, s)$	$\sim$	(4711, "Lisa")	$\Downarrow$	
t	$\sim$	$(4711, (\mathit{false}, 815))$	$\Downarrow$	
$(\_, p)$	$\sim$	$(4711, (\mathit{false}, 815))$	$\Downarrow$	
$t \,\&\, (\_, p \,\&\, (\_, n))$	$\sim$	$(4711, (\mathit{false}, 815))$	$\Downarrow$	
$t \,\&\, (m, p \,\&\, (b, n))$	$\sim$	$(4711, (\mathit{false}, 815))$	$\Downarrow$	

3. Was ist der Unterschied zwischen

$$\textbf{\textit{let}}\ x = e\ \textbf{\textit{in}} \ldots x \ldots$$

und

$$\textbf{\textit{let}}\ x\ () = e\ \textbf{\textit{in}} \ldots x\ () \ldots$$

4. Nichtnegative rationale Zahlen können durch Paare von natürlichen Zahlen, den Zähler und den Nenner, repräsentiert werden.

$$\textbf{\textit{type}}\ \mathit{Ratio} = \{\, \mathit{numerator} : \mathit{Nat}; \mathit{denominator} : \mathit{Nat} \,\}$$

Implementieren Sie die üblichen arithmetischen Operationen und die üblichen Vergleichsoperationen auf diesem Typ.

$$\begin{array}{l} \textbf{\textit{let}}\ \mathit{add}\ (a : \mathit{Ratio}, b : \mathit{Ratio}) : \mathit{Ratio} \\ \ldots \\ \textbf{\textit{let}}\ \mathit{less}\ (a : \mathit{Ratio}, b : \mathit{Ratio}) : \mathit{Bool} \end{array}$$

Jede *positive* rationale Zahl hat eine eindeutige Darstellung, wenn man vereinbart, dass Nenner und Zähler teilerfremd sind. Erweitern Sie Ihre Implementierung, so dass die Rechenergebnisse stets gekürzt vorliegen. Wie kann die Null behandelt werden?

5. Ganze Zahlen können mit Hilfe des folgenden Variantentyps repräsentiert werden.

$$\begin{array}{l} \textbf{\textit{type}}\ \mathit{Int} = \\ \quad \mid \mathit{Neg}\ \textbf{\textit{of}}\ \mathit{Nat} \\ \quad \mid \mathit{Pos}\ \textbf{\textit{of}}\ \mathit{Nat} \end{array}$$

Dabei repräsentiert ein Wert der Form $\mathit{Neg}\ n$ die Zahl $-n$ und entsprechend $\mathit{Pos}\ n$ die Zahl $+n$. Implementieren Sie die üblichen arithmetischen Operationen und die üblichen Vergleichsoperationen auf diesem Typ. Wie ändern sich die Definitionen, wenn man vereinbart, dass $\mathit{Neg}\ n$ die Zahl $-(n+1)$ repräsentiert? Diese Darstellung hat den Vorteil, dass jede ganze Zahl eine *eindeutige* Repräsentation besitzt.

6. Verwenden Sie das Peano Entwurfsmuster, um eine natürliche Zahl vom Typ *Nat* in ihre Binärdarstellung vom Typ *Leibniz* zu überführen.

let *leibniz* $(n : Nat) : Leibniz$

Programmieren Sie die Funktion ein zweites Mal, diesmal mit Hilfe des Leibniz Entwurfsmusters. Vergleichen Sie die Laufzeit der beiden Versionen.

7.

1. Leiten Sie mit den Regeln der statischen Semantik ab, dass der Ausdruck *Cons* $(true, Nil)$ den Typ $List\,\langle Bool\rangle$ hat.
2. Geben Sie für die folgenden Typen jeweils fünf — möglichst vielfältige — Ausdrücke dieses Typs an.
   a) $List\,\langle Nat\rangle$
   b) $Option\,\langle Nat\rangle$
   c) $List\,\langle Option\,\langle Nat\rangle\rangle$
   d) $Option\,\langle List\,\langle Nat\rangle\rangle$

8. Eine Matrix kann durch eine Liste von Zeilenvektoren dargestellt werden, wobei ein einzelner Zeilenvektor durch eine Liste von natürlichen Zahlen repräsentiert wird. (Die folgenden Definitionen führen *Typsynonyme* ein: Der Bezeichner auf der linken Seite ist eine Abkürzung für den Typausdruck auf der rechten Seite.)

type $Vector = List\,\langle Nat\rangle$
type $Matrix = List\,\langle Vector\rangle$

Implementieren Sie die Transposition von Matrizen und die Matrizenmultiplikation auf dieser Darstellung.

let *transpose* $(m : Matrix) : Matrix$
let *mul* $(m_1 : Matrix, m_2 : Matrix) : Matrix$

9. Erfinden Sie zu jedem der folgenden Typen einen Ausdruck, der den jeweiligen Typ besitzt.

1. $'a \to 'a$
2. $'a * 'a \to 'a$
3. $'a * 'b \to 'a$
4. $'a * 'b \to 'b * 'a$
5. $List\,\langle 'a\rangle \to List\,\langle 'a\rangle$
6. $List\,\langle 'a * 'b\rangle \to List\,\langle 'b * 'a\rangle$

Wieviele *semantisch verschiedene* Ausdrücke gibt es jeweils? *Beispiel:* ein Ausdruck des Typs $List\,\langle 'a * 'b\rangle \to List\,\langle 'a\rangle * List\,\langle 'b\rangle$ ist *unzip* mit

```
let rec unzip = function
  | Nil              → (Nil, Nil)
  | Cons ((x, y), xys) → let (xs, ys) = unzip xys in (Cons (x, xs), Cons (y, ys))
```

10. Zeigen Sie, dass die Funktion *append* bzw. der Operator @ assoziativ ist:

$$(list_1 \mathbin{@} list_2) \mathbin{@} list_3 = list_1 \mathbin{@} (list_2 \mathbin{@} list_3)$$

Unterscheiden Sie in Analogie zum Struktur Entwurfsmuster für Listen zwei Fälle: $list_1 = Nil$ (*Induktionsbasis*) und $list_1 = Cons\ (x_1, xs_1)$ (*Induktionsschritt*). Im zweiten Fall dürfen Sie annehmen, dass die Aussage bereits für xs_1 gilt.

$$(xs_1 \mathbin{@} list_2) \mathbin{@} list_3 = xs_1 \mathbin{@} (list_2 \mathbin{@} list_3)$$

11. Schreiben Sie eine Funktion

$$\textbf{\textit{let}}\ sublists\ (list : List\ \langle 'a \rangle) : List\ \langle List\ \langle 'a \rangle \rangle$$

die eine Liste aller *Teillisten* erzeugt. Die Reihenfolge der Teillisten in der Ergebnisliste ist dabei irrelevant. *Hinweis:* Eine Liste mit n Elementen hat 2^n Teillisten.

12. Schreiben Sie eine Funktion

$$\textbf{\textit{let}}\ segments\ (list : List\ \langle 'a \rangle) : List\ \langle List\ \langle 'a \rangle \rangle$$

die alle *Segmente* einer Liste berechnet. Ein Segment ist eine Teilliste, die nur aufeinanderfolgende Elemente der Ursprungsliste enthält. Die Reihenfolge der Segmente in der Ergebnisliste ist irrelevant. *Hinweis:* Eine Liste mit n Elementen hat $\binom{n+1}{2} = n(n+1)/2$ nicht-leere Segmente.

13. Schreiben Sie eine Funktion

$$\textbf{\textit{let}}\ max\text{-}segment\ (list : List\ \langle Int \rangle) : List\ \langle Int \rangle$$

die das Segment einer Liste von *ganzen* Zahlen bestimmt, dessen Summe maximal ist.

14. Ändert man die Reihenfolge der Elemente einer Liste, ohne Elemente zu entfernen oder hinzuzunehmen, so erhält man eine *Permutation*. Zu einer Liste der Länge n existieren n Fakultät verschiedene Permutationen. (Anmerkung: Enthält die Ausgangsliste Duplikate, so sind nicht alle Permutationen verschieden). Schreiben Sie eine Funktion

$$\textbf{\textit{let}}\ permutations\ (list : List\ \langle 'a \rangle) : List\ \langle List\ \langle 'a \rangle \rangle$$

die eine Liste aller Permutationen der Liste *list* berechnet. Die Reihenfolge der Permutationen in der Ergebnisliste ist irrelevant.

15. Harry Hacker hat die folgende Funktion geschrieben, um die Elemente einer Liste aufzuaddieren. (Pate für die Funktionsdefinition stand eine entsprechende Funktion für Arrays, siehe Abschnitt 4.4.)

$$\begin{array}{l}
\textbf{\textit{let}}\ sum\ (a : List\ \langle Nat \rangle) : Nat = \\
\quad \textbf{\textit{let rec}}\ s\ (i : Int) : Nat = \\
\qquad \textbf{\textit{if}}\ i = length\ a\ \textbf{\textit{then}}\ 0 \\
\qquad\qquad\qquad\qquad \textbf{\textit{else}}\ \ nth\ (a, i) + s\ (i+1) \\
\quad \textbf{\textit{in}}\ s\ 0
\end{array}$$

Welche Laufzeit hat die Funktion? Was raten Sie Harry?

16. Der folgende rekursive Variantentyp eignet sich zur Repräsentation von arithmetischen Ausdrücken.

```
type Expr =
    | Const of Nat
    | Add   of Expr * Expr
    | Sub   of Expr * Expr
    | Mul   of Expr * Expr
    | Div   of Expr * Expr
```

Die Konstruktoren *Add*, *Sub*, *Mul* und *Div* modellieren die entsprechenden arithmetischen Operatoren, der Konstruktor *Const* dient zur Repräsentation einer natürlichen Zahl.

1. Schreiben Sie eine Funktion *evaluate*, die einen Ausdruck des Typs *Expr* auswertet.

   ```
   let evaluate (expr : Expr) : Nat
   ```

2. Schreiben Sie eine Funktion *pretty-print-infix*, die einen Ausdruck des Typs *Expr* in einen *String* überführt:

   ```
   let pretty-print-infix (expr : Expr) : String
   ```

   Die Operatoren sollen dabei infix notiert werden.

   ```
   Mini⟩ pretty-print-infix (Add (Const 4711, Const 815))
   "(4711 + 815)"
   ```

17. Wir erweitern den Datentyp *Expr* aus Aufgabe 4.16 um Bezeichner und Wertebindungen.

```
type Expr =
    | Const of Nat
    | Add   of Expr * Expr
    | Sub   of Expr * Expr
    | Mul   of Expr * Expr
    | Div   of Expr * Expr
    | Id    of String
    | Let   of String * Expr * Expr
```

Erweitern Sie entsprechend die Funktion *evaluate*, die einen arithmetischen Ausdruck auswertet.

```
let evaluate (expr : Expr, env : String → Nat) : Nat
```

Ein Ausdruck wird relativ zu einer Umgebung ausgewertet, in der die Werte der freien Bezeichner festgehalten werden. Wir modellieren eine Umgebung als Abbildung von Bezeichnern auf Werte.

Wie muss der Auswerter abgeändert werden, wenn zusätzlich Laufzeitfehler wie Division durch 0 abgefangen werden?

```
let evaluate (expr : Expr, env : String → Option ⟨Nat⟩) : Option ⟨Nat⟩
```

Tritt ein Fehler auf, so wird *None* zurückgegeben, anderenfalls ist das Ergebnis *Some value*, wobei *value* der Wert des Ausdrucks ist.

18. Geben Sie jeweils einen Mini–F# Ausdruck an, um die folgenden Arrays zu konstruieren:

1. das Array aller ungeraden Zahlen zwischen 0 und 100;
2. das Array aller Zahlen zwischen 0 und 100, die durch 3 teilbar sind;
3. das Array der Größe 100, dessen i-tes Element die Summe der ersten i natürlichen Zahlen ist.

5. Algorithmik ⧹ Rechnen mit System

Writing programs needs genius to save the last order or the last millisecond. It is great fun, but it is a young man's game. You start it with great enthusiasm when you first start programming, but after ten years you get a bit bored with it, and then you turn to automatic-programming languages and use them because they enable you to get to the heart of the problem that you want to do, instead of having to concentrate on the mechanics of getting the program going as fast as you possibly can, which is really nothing more than doing a sort of crossword puzzle.

— Christopher Strachey (1916–1975)

In den letzten beiden Kapiteln haben wir den funktionalen Sprachkern von Mini–F# eingeführt. Jetzt machen wir eine Pause vom Sprachentwurf und wenden uns einigen Anwendungen zu. Wir werden einen kleinen Ausflug in die Algorithmik unternehmen, der Lehre vom systematischen maschinellen Rechnen.

Wir haben in der Einleitung angesprochen, dass es mit der Existenz von Rechenmaschinen interessant wird, Aufgaben in Rechenaufgaben zu verwandeln, die es von Natur aus nicht sind und die wir mit unserem Verstand auch nie so behandeln würden. Zu einer Problemstellung gilt es, die richtigen Abstraktionen zu finden, sie durch Formeln und Rechengesetze, genannt Datenstrukturen und Algorithmen, so weitgehend zu erfassen, dass wir dem Ergebnis der Rechnung eine Lösung des Problems entnehmen können. Einige Schritte in diese Richtung haben wir bereits unternommen und dabei festgestellt, dass sich die gleiche Problemstellung oft ganz unterschiedlich lösen lässt, so dass die Frage aufkommt, wie wir verschiedene Lösungen miteinander vergleichen können. Welche ist besser, welche schneidet schlechter ab? Bei der Bewertung von Datenstrukturen und Algorithmen spielt allgemein der *Resourcenverbrauch* eine zentrale Rolle. Wie lang werden die Rechnungen — wieviele Rechenschritte werden benötigt? Wie groß werden die einzelnen Rechenausdrücke — wieviel Platz wird benötigt, um die Rechnung durchzuführen? Werden die Rechnungen auf einem batteriegetriebenen Rechenknecht ausgeführt, wird man sich vielleicht dafür interessieren, den Energieverbrauch zu minimieren. Zu diesem Themen werden Sie sehr viel mehr in der Vorlesung „Algorithmen und Datenstrukturen“ erfahren. Gemäß dem Motto „Time is money“, konzentrieren wir uns im Folgenden auf die Resource „Zeit“, auf die Analyse der Laufzeit von Algorithmen.

Im Gegensatz zum Rechnen ist das Aufstellen von Rechenregeln eine kreative gedankliche Leistung und eine bemerkenswerte dazu. Zur Kreativität gesellen sich aber, wie auch in der Kunst oder im Handwerk, weitere Qualitäten: Erfahrung, Geduld, Ausdauer und die Beherrschung der Werkzeuge. Kreativität lässt sich nur schwer vermitteln; aber wir können Werkzeuge vorstellen und deren Verwendung trainieren. Ein wichtiges

Werkzeug für das algorithmische Lösen von Problemen haben wir bereits kennengelernt: *Entwurfsmuster*! Das Peano und das Leibniz Entwurfsmuster haben wir erfolgreich eingesetzt, um Probleme über den natürlichen Zahlen zu lösen. Das Struktur Entwurfsmuster haben wir verwendet, um Funktionen über rekursiv definierten Variantentypen wie zum Beispiel dem Listentyp systematisch zu entwickeln. Zur Erinnerung:

- *Peano Entwurfsmuster:*
  Das Problem für n wird auf das Problem für $n \dot{-} 1$ zurückgeführt.
- *Leibniz Entwurfsmuster:*
  Das Problem für n wird auf das Problem für $n \div 2$ zurückgeführt.

Den Entwurfsmustern ist gemeinsam, dass sie Lösungen für Probleme aus Lösungen für „kleinere“ Probleme konstruieren. Daraus lässt sich eine allgemeine Lösungsstrategie, ein allgemeines Prinzip ableiten:

> Um ein Problem zu lösen, genügt es zu zeigen, dass sich eine Lösung für jede Probleminstanz aus Lösungen kleinerer Probleminstanzen konstruieren lässt.

Mit anderen Worten, es ist nicht nötig, jede Probleminstanz von Grund auf zu lösen. Stattdessen sollte man versuchen, eine Probleminstanz auf kleinere Probleminstanzen zu *reduzieren*. In diesem Sinne lassen sich unsere Entwurfsmuster verallgemeinern:

- *Allgemeines Peano Entwurfsmuster:*
  Ein Problem der Größe n wird auf Probleme der Größe $n \dot{-} 1$ zurückgeführt.
- *Allgemeines Leibniz Entwurfsmuster:*
  Ein Problem der Größe n wird auf Probleme der Größe $n \div 2$ zurückgeführt.

Wenn wir einen solchen Ansatz in ein Programm überführen, dann kümmert sich jeweils ein rekursiver Aufruf um die Lösung der Teilprobleme. Abbildung 5.1 illustriert die Vorgehensweise: Beim rekursiven Abstieg wird das Problem sukzessive in Teilprobleme aufgegliedert; sind die Teilprobleme hinreichend einfach, lösen wir sie ad hoc; beim rekursiven Aufstieg werden die Teillösungen zu Gesamtlösungen zusammengeführt. Sowohl die Aufgliederung in Teilprobleme als auch die Zusammenführung von Teillösungen stellt uns oft vor *neue* Aufgaben, die sodann auf die gleiche Art und Weise angegangen werden!

Wenn wir Probleminstanzen auf „kleinere“ Probleminstanzen zurückführen wollen, müssen wir natürlich überlegen, was die Größe oder die Schwere eines Problems ausmacht. Wie lässt sich der Schwierigkeitsgrad messen? Darauf gibt es keine allgemein gültige Antwort. In der Auseinandersetzung mit einem neuen Problem ist die Festlegung des „Maßes“ ein erster wichtiger Schritt.

Im Einzelnen haben wir Folgendes vor. Abschnitt 5.1 zeigt, wie sich die Entwurfsmuster auf das uns schon bekannte Sortierproblem anwenden lassen. Dabei richten wir ein besonderes Augenmerk auf die Analyse der Sortieralgorithmen und überlegen, wie schwierig das Sortierproblem prinzipiell ist. Abschnitt 5.2 widmet sich dem Suchen, einem weiteren klassischen Thema der Algorithmik. Wir vertiefen die Suche in Datenstrukturen und die Suche in „virtuellen Räumen“. Dabei richten wir ein besonderes Augenmerk

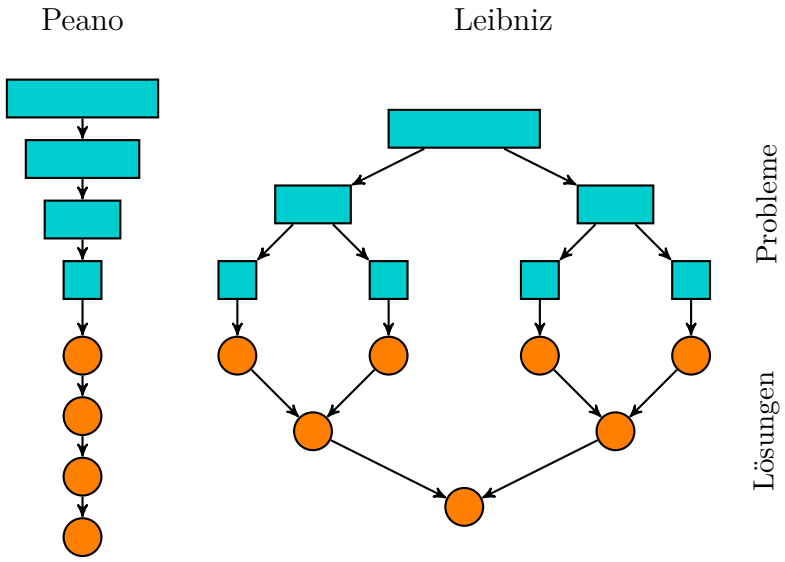


Abbildung 5.1.: Peano und Leibniz Entwurfsmuster.

auf die Korrektheit und die Terminierung der Algorithmen. Abschnitt 5.3 erweitert den Blickwinkel und diskutiert, wie man Suchstrukturen konstruiert, modifiziert und traversiert. Aus einer Datenstruktur wird ein abstrakter Datentyp. Abschnitt 5.4 stellt ein weiteres Beispiel für einen abstrakten Datentyp vor: Prioritätswarteschlangen.

5.1. Sortieren

module
Algorithmics.
Sort

> *Computer manufacturers of the 1960s estimated that more than 25% of the running time on their computers was spent on sorting, when all their customers were taking into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either*
>
> *(i) there are many important applications of sorting, or*
>
> *(ii) many people sort when they shouldn't, or*
>
> *(iii) inefficient sorting algorithms have been in common use.*
>
> *The real truth probably involves all three of these possibilities, but in any event we can see that sorting is worthy of serious study, as a practical matter.*
>
> — Donald E. Knuth (1938), *TAOCP 3*

Auch sechzig Jahre später wird das Sortieren von Daten einen nicht unerheblichen Anteil an der gesamten Rechenleistung von Computern ausmachen. Bevor wir uns mit konkreten Sortierverfahren beschäftigen, sollten wir das Sortierproblem zunächst präzise spezifizieren.

Als Eingabe erhalten wir eine Folge von Elementen des gleichen Typs:

$x_0, x_1, x_2, \ldots, x_{n-1}$

Die Problemgröße, von der wir vorher gesprochen haben, ist hier die Anzahl der Elemente, also n. Als Ausgabe ist eine Permutation der Eingabe

$$x_{\pi(0)}, x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n-1)}$$

gesucht, so dass

$$x_{\pi(0)} \leqslant x_{\pi(1)} \leqslant x_{\pi(2)} \leqslant \cdots \leqslant x_{\pi(n-1)}$$

Die Abbildung $\pi : \mathbb{N}_n \to \mathbb{N}_n$ ordnet dabei jedem Index i eineindeutig einen Index $\pi(i)$ zu. Mit anderen Worten: kein Eingabelement wird vergessen, kein Ausgabeelement erfunden. Da die Eingaben Elemente eines beliebigen Typs sein können, müssen wir weiterhin klären, was '$\leqslant$' bedeutet. Wir setzen folgende Eigenschaften voraus:[1]

1. *reflexiv:* $x \leqslant x$;
2. *transitiv:* wenn $x \leqslant y$ und $y \leqslant z$, dann $x \leqslant z$;
3. *total:* $x \leqslant y$ oder $y \leqslant x$.

Diese Eigenschaften definieren eine sogenannte *totale Quasiordnung* (engl. total preorder). (In Anhang A.1.4 erfahren sie mehr zu Ordnungen und Verbänden.) Die Ordnung auf den natürlichen Zahlen zum Beispiel erfüllt die geforderten Eigenschaften; die Teilmengenbeziehung erfüllt sie nicht: '$\subseteq$' ist zwar reflexiv und transitiv, aber nicht total. Zum Beispiel gilt weder $\{1, 2\} \subseteq \{2, 3\}$ noch umgekehrt $\{2, 3\} \subseteq \{1, 2\}$.

Wir verlangen *nicht*, dass die Quasiordnung antisymmetrisch ist: Wenn $x \leqslant y$ und $y \leqslant x$, dann $x = y$. Diese Forderung wäre zu einschränkend: Wenn wir zum Beispiel Personen nach ihrem Alter ordnen, dann gilt nicht, dass es sich bei zwei Personen des gleichen Alters auch um die gleiche Person handeln muss. Eine antisymmetrische Quasiordnung heißt übrigens schlicht *Ordnung* — deswegen spricht man auch von der „Ordnung auf den natürlichen Zahlen".

In einer totalen Quasiordnung gibt es insgesamt genau *drei* Möglichkeiten, wie zwei Elemente zueinander stehen:

1. sowohl $x \leqslant y$ als auch $y \leqslant x$ gilt (notiert durch $x \sim y$);
2. $x \leqslant y$ gilt, nicht aber $y \leqslant x$ (notiert durch $x < y$); und schließlich
3. $x \leqslant y$ gilt nicht, wohl aber $y \leqslant x$ (notiert durch $x > y$).

Der vierte Fall, weder $x \leqslant y$ noch $y \leqslant x$ gilt, kann nicht auftreten, da die Quasiordnung total ist. Die folgende Tabelle fasst die Fälle noch einmal zusammen.

		$y \leqslant x$	
		false	*true*
$x \leqslant y$	*false*	—	$x > y$
	true	$x < y$	$x \sim y$

[1] Wenn eine Relation total ist, dann ist sie automatisch auch reflexiv: $x \leqslant y$ oder $y \leqslant x$ gilt für beliebige Elemente, somit insbesondere wenn x und y identisch sind.

Um Missverständnissen vorzubeugen: Die drei Relationen werden von der Relation '$\leqslant$' abgeleitet; sie werden durch die Tabelle definiert. Die Relation '$\sim$' ist eine sogenannte Äquivalenzrelation; '$<$' und '$>$' sind sogenannte strikte Ordnungen.

5.1.1. Einfache Sortierverfahren

Wenden wir das verallgemeinerte Peano Entwurfsmuster auf das Sortierproblem an. Um die Problemgröße zu reduzieren, müssen wir ein Element der zu sortierenden Folge zur Seite legen. *Zwei* kanonische Ansätze drängen sich auf: Wir entfernen das erste Element der Eingabe *oder* wir wählen das erste Element der Ausgabe. Zwei Möglichkeiten, zwei Sortierverfahren:

Sortieren durch Einfügen:

- Lege das erste Element zur Seite,
- sortiere die restlichen Elemente,
- *füge* das erste Element in die sortierte Folge *ein*.

Sortieren durch Auswählen:

- *Wähle* das kleinste Element *aus*,
- sortiere die restlichen Elemente,
- setze das kleinste Element vor die sortierte Folge.

Die beiden Verfahren sind im gewissen Sinne „dual" zueinander. Beim Sortieren durch Einfügen ist der erste Schritt einfach und der letzte Schritt aufwändig; beim Sortieren durch Auswählen ist es genau umgekehrt.

Sortieren durch Einfügen Das Verfahren haben wir bereits in Abschnitt 4.2.2 ausführlich besprochen. Die Eingabefolge wird durch eine Liste repräsentiert. Zur Erinnerung:

```
let insertion-sort (⩽) =
  let rec insert k = function
      | []      → [k]
      | x :: xs → if k ⩽ x then k :: x :: xs else x :: insert k xs
  let rec sort = function
      | []      → []
      | x :: xs → insert x (sort xs)
  in sort
```

Die Sortierfunktion ist mit der Quasiordnung '$\leqslant$' parametrisiert. (In Abschnitt 4.2.2 haben wir *less-equal* als Bezeichner gewählt, hier verwenden wir einen symbolischen Bezeichner, der den vordefinierten Operator verschattet.)

Ist das Programm korrekt, erfüllt es seine Spezifikation? Wir müssen zeigen, dass die Ausgabe eine Permutation der Eingabe ist. Wenn man den Programmtext genau inspiziert, sieht man, dass kein Eingabeelement vergessen wird und kein Ausgabeelement erfunden wird. Ist die Ausgabe aufsteigend geordnet? Nun, gemäß des Peano Entwurfsmusters erweitert die Hilfsfunktion *insert* eine Teillösung zu einer Gesamtlösung. Mit anderen Worten, wir müssen zeigen, dass, wenn *xs* sortiert ist, dann auch *insert k xs* sortiert ist. Die Korrektheit von *insert* hängt von einer *Vorbedingung* ab. (Wenn die Liste *xs* nicht aufsteigend geordnet ist, lässt sich über das Verhalten von *insert k xs* in

der Tat wenig aussagen.) Nach diesen Vorüberlegungen kann man die Korrektheit von *insert* und *sort* mit Hilfe von Induktionsbeweisen nachweisen. Dabei fließt wesentlich die Annahme ein, dass '$\leqslant$' sowohl transitiv als auch total ist. (An welchen Stellen?)

Wenden wir uns der Analyse der Laufzeit zu. Sortieren durch Einfügen ist im gewissen Sinne *adaptiv:* „Vorsortierte" Eingaben werden schneller verarbeitet als „unsortierte". Der beste Fall liegt vor, wenn die Eingabe bereits aufsteigend sortiert ist. In diesem Fall benötigt das Einfügen lediglich einen Vergleich, so dass insgesamt $n-1$ Vergleiche durchgeführt werden. Das ist optimal — schneller kann man nicht sortieren. (Warum?) Der schlechteste Fall liegt vor, wenn das Einfügen jeweils die gesamte Liste durchlaufen muss, also wenn die Eingabe absteigend sortiert ist.[2] In diesem Fall benötigt das Einfügen $i-1$ Vergleiche, so dass insgesamt $\sum_{i=1}^{n} i-1 = \binom{n}{2} = n\cdot(n-1)/2$ Vergleiche durchgeführt werden. In der Regel interessiert man sich nicht für die präzise Anzahl der Rechenschritte, sondern nur für die Größenordnung, so dass wir festhalten: Die Laufzeit von „Sortieren durch Einfügen" ist im schlechtesten Fall quadratisch.

Sortieren durch Auswählen War vorher die „Lösungserweiterung" aufwändig, so ist es jetzt die „Problemreduktion". Wir müssen das kleinste Element einer Liste bestimmen; für den rekursiven Aufruf benötigen wir zusätzlich die Liste ohne das kleinste Element. Die Funktion *split-min* erledigt beide Aufgaben in einem Rutsch:

$$\mathit{split\text{-}min} : \mathit{List}\,\langle'a\rangle \to 'a * \mathit{List}\,\langle'a\rangle$$

Dieser Ansatz führt allerdings noch nicht ganz ans Ziel. Was machen wir im Basisfall, wenn die Eingabeliste leer ist? Es eröffnet sich eine weitere Anwendungsmöglichkeit für den Variantentyp *Option*.

$$\mathit{split\text{-}min} : \mathit{List}\,\langle'a\rangle \to \mathit{Option}\,\langle'a * \mathit{List}\,\langle'a\rangle\rangle$$

Das Ergebnis von *split-min* ist ein optionales Paar: Wir erhalten entweder *None* — dann war die Liste leer — oder *Some* (m, ys), wobei m das kleinste Element ist und ys die restlichen Elemente umfasst. Die Definition von *split-min* ergibt sich dann mit dem Struktur Entwurfsmuster für Listen.

```
let selection-sort (⩽) =
  let rec split-min = function
      | []      → None
      | x :: xs → Some (match split-min xs with
                          | None          → (x, xs)
                          | Some (m, ys) → if x ⩽ m then (x, xs) else (m, x :: ys))
  let rec sort xs =
    match split-min xs with
      | None          → []
      | Some (m, ys) → m :: sort ys
  in sort
```

[2] Wir haben oben das Adjektiv „vorsortierte" in Anführungsstriche gesetzt, da keineswegs klar ist, was damit genau gemeint ist. Man kann mit Fug und Recht behaupten, dass auch eine absteigend geordnete Liste im gewissen Sinne vorsortiert ist.

Im Gegensatz dazu folgt die Definition von *sort nicht* dem Struktur Entwurfsmuster: *ys* ist im Allgemeinen nicht die Restliste von *xs*.

Ist das Programm korrekt, erfüllt es seine Spezifikation? Aus der obigen Beschreibung von *split-min* lassen sich die Beweispflichten ableiten: Wenn *xs* nicht leer ist, dann gilt *split-min xs* = *Some* (m, ys), wobei m das kleinste Element von *xs* ist und *ys* die restliche Liste. Anderenfalls ist *split-min xs* = *None*. Sind diese Eigenschaften etabliert, folgt die Korrektheit von *sort* auf dem Fuße. (Natürlich sollten wir uns noch vergewissern, dass die Ausgabe eine Permutation der Eingabe ist. Eine kurze Inspektion des Programmtextes zeigt wiederum, dass kein Eingabeelement vergessen wird und kein Ausgabeelement erfunden wird.)

Wenden wir uns der Analyse der Laufzeit zu. Sortieren durch Auswählen ist im gewissen Sinne „vergesslich“ (engl. oblivious): Die Laufzeit hängt nicht von den konkreten Eingabeelementen ab. Das Verfahren benötigt im besten wie im schlechtesten Fall exakt die gleiche Anzahl von Vergleichen. Um jeweils das Minimum zu bestimmen, werden $i - 1$ Vergleiche benötigt. Das ist optimal. (Warum?) Insgesamt werden somit $\sum_{i=1}^{n} i - 1 = \binom{n}{2} = n \cdot (n-1)/2$ Vergleiche getätigt. Kurz: Die Laufzeit von „Sortieren durch Auswählen“ ist ebenfalls quadratisch. (Das Attribut „vergesslich“ soll andeuten, dass wir bei der Bestimmung des Minimums jeweils von vorne anfangen und keinen Nutzen aus den vorherigen Durchläufen ziehen.)

5.1.2. Sortieren durch Mischen

Wenden wir das verallgemeinerte Leibniz Entwurfsmuster auf das Sortierproblem an. Jetzt müssen wir die Problemgröße ungefähr halbieren. Zwei Ansätze bieten sich an:

Sortieren durch Mischen:

- Teile die Eingabefolge in zwei ungefähr gleich große Folgen,
- sortiere jede der Teilfolgen,
- *„mische“* die zwei sortierten Teilfolgen.

(*Sortieren durch Austauschen:*

- Wähle ein „Pivotelement“ aus und teile die Eingabeliste in kleinere und größere Elemente,
- sortiere jede der Teilfolgen,
- hänge die beiden sortierten Teilfolgen aneinander.)

Die beiden Verfahren sind im gewissen Sinne „dual“ zueinander. Beim Sortieren durch Mischen ist der erste Schritt relativ einfach und der letzte Schritt aufwändig; beim Sortieren durch Austauschen ist es genau umgekehrt. Letzteres Verfahren ist übrigens auch unter dem Namen *Quicksort* bekannt. Ob Quicksort tatsächlich quick ist, hängt wesentlich von der Wahl des „Pivotelements“ ab. Das ist eine Wissenschaft für sich, so dass wir den Ansatz hier nicht weiter verfolgen wollen. (Um die Eingabeliste ungefähr in zwei Teillisten aufzuteilen, wählt man idealerweise den *Median*. Dessen Bestimmung ist aber aufwändig, siehe Abschnitt 5.1.5, so dass man Kompromisse eingeht …)

Beim Sortieren durch Mischen müssen wir die Eingabeliste zunächst in zwei etwa gleich große Listen aufteilen. Das hört sich nach einer „neuen“ Aufgabe an. Erfreulicherweise führt das Struktur Entwurfsmuster unmittelbar zum Ziel:

```
// unzip : 'a list → 'a list * 'a list
let rec unzip = function
    | []      → ([], [])
    | x :: xs → let (xs1, xs2) = unzip xs in (x :: xs2, xs1)
```

Das Verb „unzip“ heißt im Englischen „den Reißverschluss von etwas aufmachen“ und beschreibt das Verfahren sehr schön: Die Elemente der Eingabeliste werden alternierend den Ergebnislisten zugeschlagen. Der Aufruf *unzip xs* teilt die Liste *xs* somit in die Liste *es* der Elemente an geraden und die Liste *os* der Elemente an ungeraden Positionen. Hat *xs* die Länge n, dann hat *es* die Länge $\lceil n/2 \rceil$ und *os* die Länge $\lfloor n/2 \rfloor$. Mit anderen Worten: Die Teillisten sind entweder gleich lang oder die erste ist um eins länger.

Nach diesen Vorarbeiten lässt sich das Leibniz Entwurfsmuster direkt umsetzen.

```
let merge-sort (⩽) =
  let rec merge = function
      | ([], xs) | (xs, []) → xs
      | (x :: xs, y :: ys)  → if x ⩽ y then x :: merge (xs, y :: ys)
                              else  y :: merge (x :: xs, ys)
  let rec sort = function
      | []  → []
      | [x] → [x]
      | xs  → let (xs1, xs2) = unzip xs in merge (sort xs1, sort xs2)
  in sort
```

Die Hilfsfunktion *merge* führt die Teillösungen zu einer Gesamtlösung zusammen: Zwei sortierte Listen werden zu einer sortierte Liste zusammengefügt.[3] Sie hat merkenswerte Eigenschaften: Das Mischen sortierter Listen ist assoziativ und hat die leere Liste als neutrales Element. Letzteres wird durch die Regel $([], xs) \mid (xs, []) \to xs$ sehr schön eingefangen. Die Funktion *merge* verallgemeinert *insert*, es gilt: *insert x ys* = *merge* $[x]$ *ys*. Durch die symmetrische Behandlung der Funktionsargumente ist die Implementierung von *merge* eleganter als die von *insert*, obwohl *merge* ein allgemeineres Problem löst.

Die Hilfsfunktion *sort* implementiert die Idee des „divide et impera“ („Teile und Herrsche“, engl. divide and conquer) buchstabengetreu: Die Eingabeliste wird halbiert, beide Teillisten werden sortiert, die Ergebnisse werden zur sortierten Ausgabeliste zusammengeführt. Für die Korrektheit spielt es keine Rolle, dass sich eine Liste nicht immer *exakt* halbieren lässt. Wir müssen nur sicherstellen, dass die Problemgröße tatsächlich reduziert wird: *length* xs_1 < *length xs* und *length* xs_2 < *length xs*. Mit dem Wissen über die Arbeitsweise von *unzip* muss gelten: $\lfloor n/2 \rfloor \leqslant \lceil n/2 \rceil < n$, wobei n die Länge von *xs* ist. Letztere Ungleichung ist aber nur erfüllt für $n \geqslant 2$. Aus diesem Grund behandeln wir zwei Basisfälle: die leere Liste und einelementige Listen — diese sind bereits sortiert.

[3] Die Übersetzung des englischen Begriffs „Mergesort“ mit „Sortieren durch Mischen“ ist unglücklich. Im Deutschen meint „mischen“ oft “etwas in Unordnung bringen“, etwa beim Mischen von Karten. Hier ist das genaue Gegenteil gemeint: *merge* stiftet nicht Unordnung, sondern erhält und vergrößert die Ordnung. Mergesort wird auch mit „Sortieren durch Verschmelzen“ übersetzt — das erinnert aber eher an eine innige Liebesbeziehung.

So wie „Sortieren durch Mischen“ das Verfahren „Sortieren durch Einfügen“ verallgemeinert, so verallgemeinert sich auch der Nachweis der Korrektheit. Wie im Fall von *insert* hängt die Korrektheit von *merge* von einer Vorbedingung ab:

Wenn $list_1$ und $list_2$ sortiert sind, dann ist auch $merge\ (list_1, list_2)$ sortiert.

Versuchen wir uns an einem Induktionsbeweis.

Fall $list_1 = [\,]$ **oder** $list_2 = [\,]$**:** Als Ergebnis wird die jeweils andere Liste zurückgegeben. Diese ist gemäß der Vorbedingung bereits sortiert.

Fall $list_1 = x :: xs$ **und** $list_2 = y :: ys$**:** Auf Grund der Vorbedingung wissen wir, dass x das kleinste Element von $x :: xs$ ist und entsprechend y das kleinste Element von $y :: ys$. Wenn $x \leqslant y$ gilt (im ***then***-Zweig), dann muss x auf Grund der Transitivität von ‘$\leqslant$’ das Minimum aller Elemente sein. Anderenfalls gilt $y < x$ (im ***else***-Zweig), da ‘$\leqslant$’ total ist. In diesem Fall ist y das Minimum und somit notwendigerweise das Kopfelement der Ergebnisliste. Wenn $x :: xs$ und $y :: ys$ sortiert sind, dann sind es auch xs und ys, so dass wir die Induktionsannahme anwenden können. Damit ist garantiert, dass auch die Restliste $merge\ (xs, y :: ys)$ bzw. $merge\ (x :: xs, ys)$ sortiert ist. Was zu beweisen war.

Wenden wir uns der Analyse der Laufzeit zu. Die schematische Darstellung des Leibniz Entwurfsmusters in Abbildung 5.1 legt die Vorgehensweise nah. Wir müssen uns überlegen, wie tief der „Rekursionsbaum“ ist und wie aufwändig die einzelnen Kästchen sind, die das Zerteilen der Probleme und das Zusammenführen der Lösungen symbolisieren. Wir können uns das Leben vereinfachen, wenn wir nicht jeden Schritt isoliert betrachten, sondern den Gesamtaufwand pro „*Rekursionsebene*“, das heißt alle Kästchen auf einer horizontalen Linie zusammenfassen. Der Gesamtaufwand ergibt sich dann als Produkt der Rekursionstiefe und des Aufwands pro Rekursionsebene.

Da wir die Problemgröße wiederholt halbieren, ist die Rekursionstiefe durch den binären Logarithmus gegeben: $\lg n$. Die Laufzeit von *unzip* ist proportional zur Eingabegröße; die Laufzeit von *merge* ist proportional zur Ausgabegröße. Da die Gesamtzahl der Elemente pro Rekursionsebene stets gleich ist (!), ist der Aufwand pro Ebene somit linear: n. Die Laufzeit von *merge-sort* beträgt entsprechend $n \lg n$. Das Produkt lässt sich als Fläche eines Rechtecks deuten, siehe Abbildung 5.2. Die folgende Tabelle zeigt, dass Programme mit einer linear-logarithmischen Laufzeit einen erheblichen Geschwindigkeitsvorteil gegenüber Programmen mit quadratischer Laufzeit haben.

n	$\lg n$	$n \lg n$	n^2
100	$\approx 6,6$	≈ 660	10.000
1.000	$\approx 10,0$	≈ 10.000	1.000.000
10.000	$\approx 13,3$	≈ 133.000	100.000.000
100.000	$\approx 16,6$	$\approx 1.660.000$	10.000.000.000
1.000.000	$\approx 20,0$	$\approx 20.000.000$	1.000.000.000.000

Für hinreichend große n ist Sortieren durch Mischen somit wesentlich schneller als Sortieren durch Einfügen oder Auswählen.

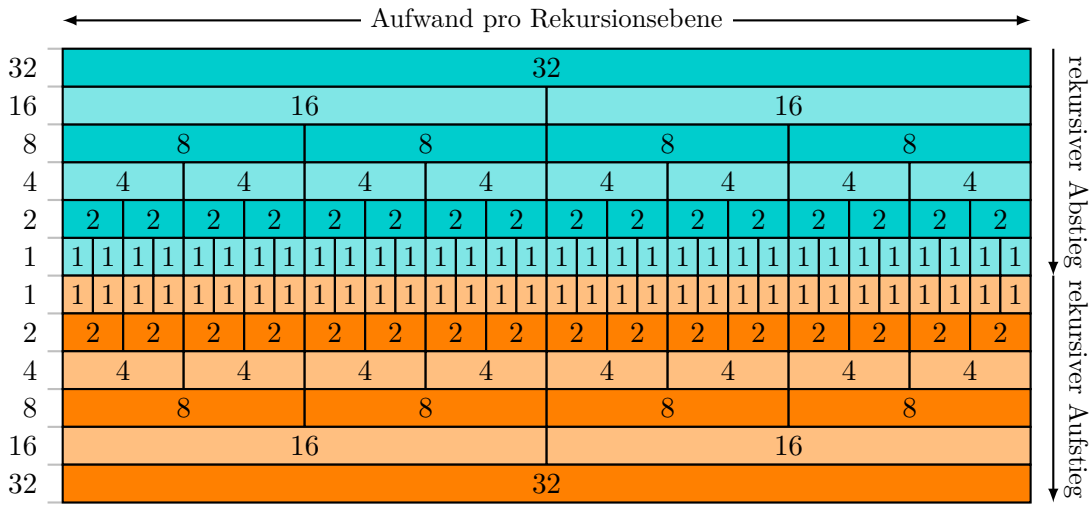


Abbildung 5.2.: Laufzeit von Mergesort.

5.1.3. Komplexität des Sortierproblems

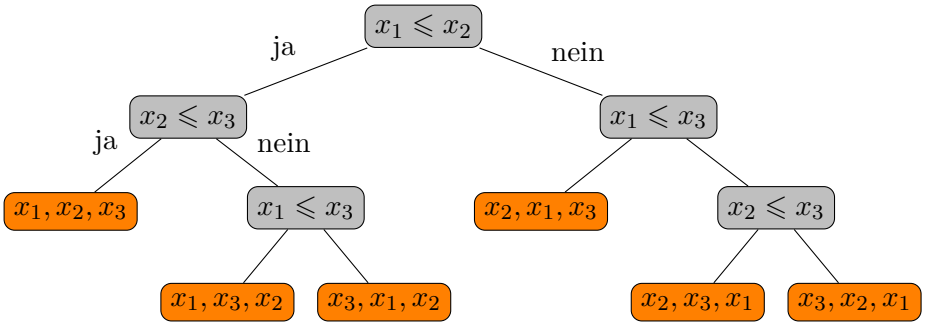
Sortieren durch Mischen hat eine linear-logarithmische Laufzeit. Können wir die Laufzeit weiter verbessern? Lässt sich eine Folge von Elementen in linearer oder gar in logarithmischer Zeit sortieren? Man überlegt sich leicht, dass eine sublineare Laufzeit unmöglich ist. Um nur das Minimum zu bestimmen, müssen wir jedes Element der Eingabe berücksichtigen. Das Sortieren der Eingabe benötigt somit mindestens eine lineare Anzahl von Schritten. Diese Beobachtung lässt sich noch verschärfen. Dazu ist es hilfreich, sich noch einmal die zentrale Annahme ins Gedächtnis zu rufen.

Wir setzen voraus, dass eine Quasiordnung gegeben ist, die der Sortierfunktion als Parameter mit auf den Weg gegeben wird.

$$sort : ('a \to 'a \to Bool) \to ('a\ list \to 'a\ list)$$

Die Quasiordnung ist für die Sortierfunktion eine „black box“, ein Orakel. Um Informationen über die relative Anordnung von Elementen zu gewinnen, kann lediglich das Orakel befragt werden, der Funktionsparameter auf zwei Elemente angewendet werden. Da die Sortierfunktion einen polymorphen Typ hat bzw. haben soll, kennen wir noch nicht einmal den konkreten Typ der zu sortierenden Elemente: Sind es natürliche Zahlen, Personendaten oder Musikstücke?

Jedes Sortierprogramm wird wiederholt das Orakel befragen und aus den jeweiligen Antworten seine Rückschlüsse ziehen. Wenn wir uns nur auf die Befragung des Orakels konzentrieren und alle sonstigen buchhalterischen Aktivitäten unter den Tisch fallen lassen, können wir ein Programm durch einen sogenannten *Entscheidungsbaum* repräsentieren. Hier ist ein Entscheidungsbaum, um die Folge x_1, x_2, x_3 zu sortieren:



Das repräsentierte Programm testet zunächst ob $x_1 \leqslant x_2$. Ist das der Fall, wird $x_2 \leqslant x_3$ geprüft. Gilt auch dies, lässt sich schließen, dass die Eingabe x_1, x_2, x_3 bereits sortiert ist. Ist hingegen $x_2 > x_3$, dann erfolgt ein weiterer Test: Ist $x_1 \leqslant x_3$, dann ist x_1, x_3, x_2 die gesuchte sortierte Ausgabefolge; anderenfalls x_3, x_1, x_2. Und so weiter …

Ein konkreter Lauf des Sortierprogramms, eine konkrete Rechnung korrespondiert zu einem Pfad von der „Wurzel“ des Entscheidungsbaums zu einem Blatt, der sortierten Permutation der Eingabe. Drei Elemente lassen sich auf $3! = 6$ Arten anordnen, entsprechend hat der Entscheidungsbaum 6 Blätter. Die maximale Laufzeit des Sortierprogramms entspricht der Länge des längsten Pfades, der sogenannten *Höhe* des Entscheidungsbaums. In unserem Beispiel ist die Höhe 3, wir müssen also das Orakel maximal dreimal befragen. Kommen wir auch mit maximal zwei Befragungen aus? Nein! Ein Entscheidungsbaum der Höhe 2 hat höchstens 4 Blätter, wir müssen aber 6 Fälle unterscheiden. Wir halten fest: Mit Hilfe von Entscheidungsbäumen können wir Aussagen über alle denkbaren Sortierprogramme treffen — das sind unendlich (!) viele; neben den naheliegenden, geschickten Programmen gibt es ja auch viele, viele ungeschickte.

Diese Beobachtungen lassen sich auf eine beliebige Anzahl von zu sortierenden Elementen verallgemeinern. Ein Entscheidungsbaum, der n Elemente sortiert, muss notwendigerweise $n!$ Blätter besitzen. Man kann zeigen, dass die Höhe eines solchen Entscheidungsbaums mindestens $n \lg n$ ist (von der Größenordnung, nicht exakt). Mit anderen Worten, jedes Sortierverfahren, *das auf dem Vergleichen von Elementen basiert*, benötigt im schlechtesten Fall mindestens $n \lg n$ Vergleiche. Da wir uns bereits ein Verfahren überlegt haben, das im schlechtesten Fall höchstens $n \lg n$ Vergleiche benötigt, haben wir die *inhärente Komplexität* des Sortierproblems ermittelt: $n \lg n$. Damit ist Sortieren durch Mischen ein *optimales* Sortierverfahren — bzw. genauer ein *asymptotisch optimales* Verfahren, da wir nicht die Anzahl von Vergleichen auf Punkt und Komma bestimmt haben, sondern nur die ungefähre Größenordnung.

Wenn wir den Typ der zu sortierenden Elemente und die zugrundeliegende Quasiordnung kennen, dann können wir tatsächlich schneller sortieren. Ist der Typ zum Beispiel *Unit*, können wir in konstanter Zeit sortieren; ist der Typ *Bool*, dann benötigen wir höchstens lineare Zeit. (Warum?)

Abbildung 5.3 fasst die Diskussion noch einmal zusammen. Um die inhärente Komplexität eines algorithmischen Problems zu bestimmen, kreisen wir das Problem von zwei Seiten ein. Wir überlegen uns Programme, die das Problem lösen und somit obere Schranken für die Laufzeit setzen. Für das Sortierproblem haben wir erst quadratische und dann ein linear-logarithmisches Verfahren entwickelt. Schwieriger ist es, untere

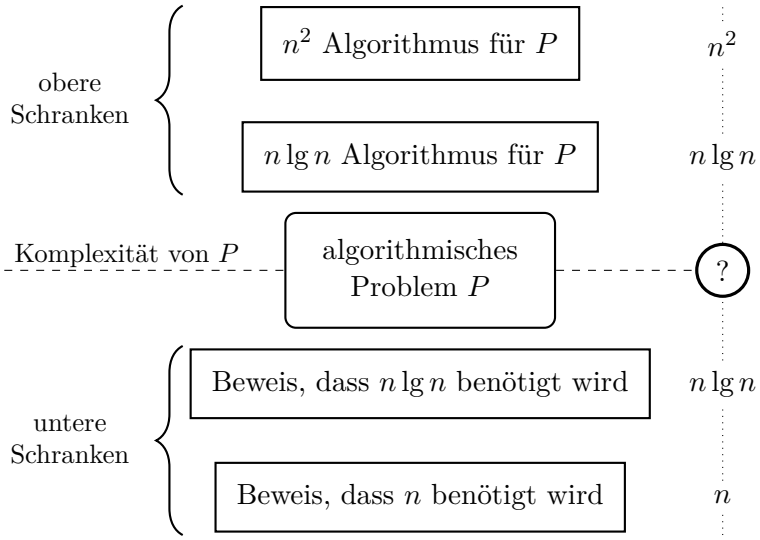


Abbildung 5.3.: Die inhärente Komplexität eines algorithmischen Problems.

Schranken zu setzen, da wir Aussagen über alle möglichen Programme machen müssen. Für das Sortierproblem lässt sich einfach argumentieren, dass eine lineare Laufzeit notwendig ist. Mit Hilfe eines informationstheoretischen Arguments haben wir eine linear-logarithmische Schranke hergeleitet. Fallen die obere und die untere Schranke zusammen, ist das Problem durchdrungen. Dann wissen wir, wie schwierig es ist, das Problem zu lösen. Das ist der Idealfall — im Laufe des Studiums werden Ihnen viele Probleme begegnen, deren Komplexität nicht bekannt ist, wo die beste untere und die beste obere Schranke weit auseinanderklaffen. (Treiben Sie die Forschung auf diesem Gebiet entscheidend voran, können Sie reich und berühmt werden, siehe http://www.claymath.org/millennium-problems/p-vs-np-problem.)

5.1.4. Anwendung: Bebaute Fläche

Abbildung 5.4 zeigt ein Luftbild des neuen Campus der Rheinland-Pfälzischen Universität Kaiserslautern-Landau. Die schematische Darstellung ist das Ergebnis eines schon mehrfach angesprochenen Abstraktionsprozesses: Gebäude und Gebäudeteile werden dabei durch an den Achsen ausgerichtete Rechtecke repräsentiert. Um zu überprüfen, ob der Campus den neuesten Bestimmungen zur „Beschränkung der Oberflächenversiegelung“ genügt, muss die gesamte bebaute bzw. überbaute Fläche berechnet werden.

module
Algorithmics.
Area

In Abschnitt 4.1.3 haben wir uns bereits mit den beiden einfachsten, nicht-trivialen Probleminstanzen beschäftigt und die Gesamtfläche von zwei bzw. drei Rechtecken bestimmt. Jetzt wollen wir das Problem in voller Allgemeinheit angehen und die Gesamtfläche von n Rechtecken ausrechnen bzw. ausrechnen lassen.

Rufen wir uns ins Gedächtnis, wie wir das Problem für $n = 2$ bzw. $n = 3$ angegangen sind. Die Programme in Abschnitt 4.1.3 fußen auf dem Prinzip der Ein- und

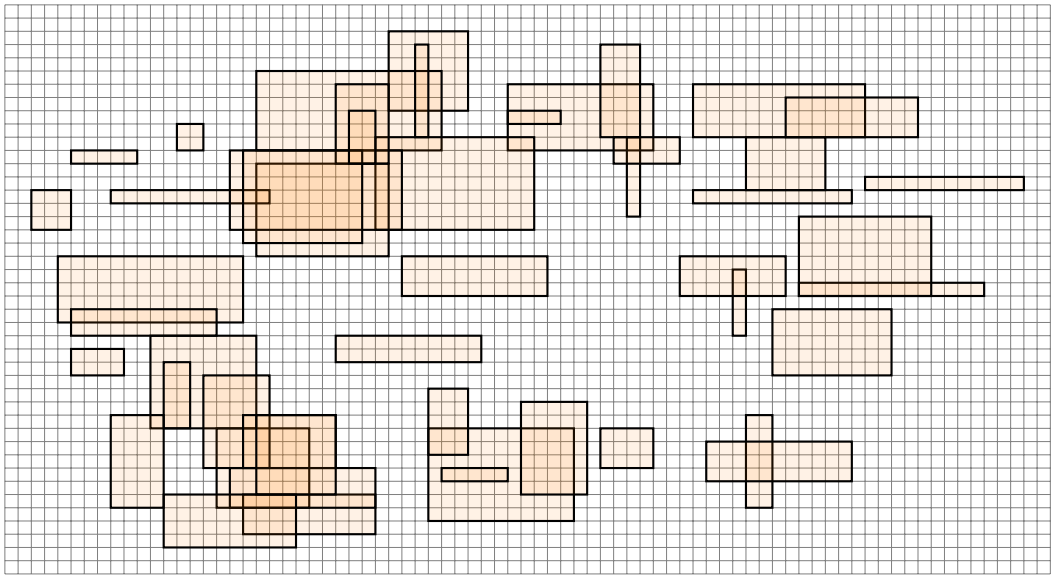


Abbildung 5.4.: Campus der Rheinland-Pfälzischen Universität Kaiserslautern-Landau.

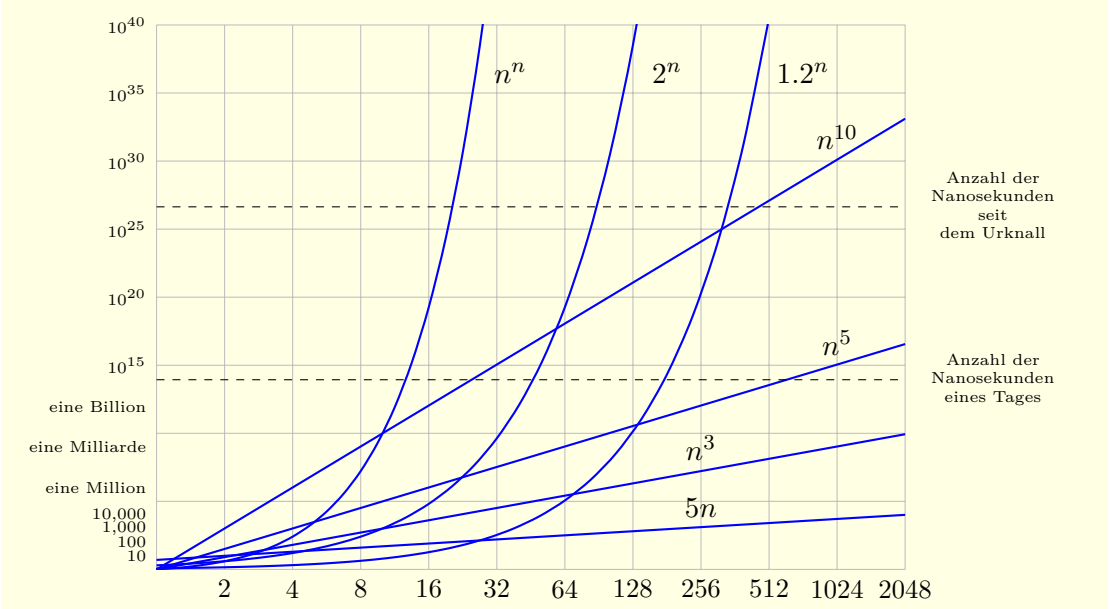


Abbildung 5.5.: Wachstum verschiedener Funktionen (logarithmische Skalen).

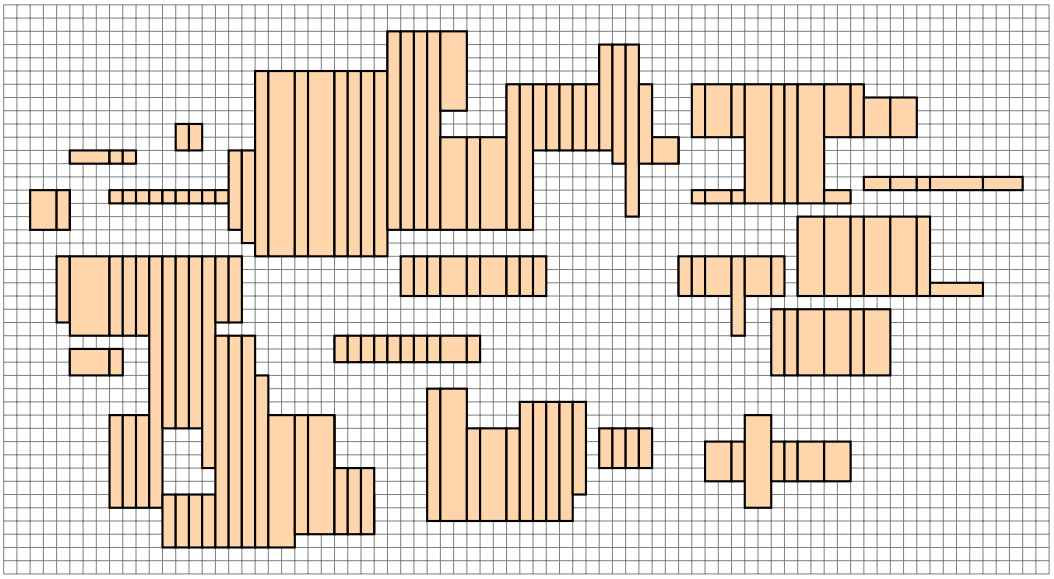


Abbildung 5.6.: Repräsentation des Campus aus Abbildung 5.4 durch vertikale Streifen.

Ausschließung, siehe Abbildung 4.2. Leider skaliert dieser Ansatz nicht: Die Formel für die Gesamtfläche benötigt die Größen sämtlicher Schnittflächen. Das sind viele, zu viele, wenn n hinreichend groß ist: Sind n Flächen gegeben, dann gibt es insgesamt 2^n mögliche Schnitte — jede Fläche kann an einem Schnitt beteiligt sein oder auch nicht. (Die Programme *area2* und *area3* enthalten tatsächlich $2^2 - 1 = 3$ bzw. $2^3 - 1 = 7$ Summanden, da der „leere Schnitt“, bei dem keine Fläche beteiligt ist, nicht benötigt wird.) Abbildung 5.5 illustriert, warum es praktisch nicht möglich ist, 2^n Teilprobleme zu lösen: Wenn wir annehmen, dass wir für die Lösung eines Teilproblems eine Nanosekunde benötigen ($1\,ns = 0,000\,000\,001\,s$), dann brauchen wir mehr als einen Tag, um 47 Flächen zu bearbeiten; verdoppeln wir deren Anzahl, dann reicht selbst die gesamt Zeit seit dem Urknall nicht!

Das Prinzip der Ein- und Ausschließung gilt für beliebige Flächen. Nun sind Rechtecke, deren Kanten parallel zu den Achsen verlaufen, sehr spezielle, sehr einfache Flächen. So ist es unmöglich, mit n Rechtecken 2^n unterschiedliche Schnittflächen zu konstruieren — Abbildung 4.2 zeigt, dass es mit 4 Rechtecken noch klappt, mit 5 Rechtecken lassen sich höchstens 28 Flächen konstruieren (das ist eine Vermutung). Allgemein ist die Anzahl der Schnittflächen durch $(2 \cdot n + 1)^2$ nach oben begrenzt. (Wenn wir die Kanten jedes Rechtecks ins Unendliche verlängern, dann wird die Ebene in maximal $2 \cdot n + 1$ horizontale Abschnitte und in ebenso viele vertikale Abschnitte unterteilt.) Es besteht also die berechtigte Hoffnung, dass sich das Problem effizient lösen lässt.

Die Gesamtfläche ließe sich einfach bestimmen, wenn wir davon ausgehen könnten, dass die Rechtecke paarweise disjunkt sind. Eine vielleicht naheliegende Idee ist, genau dafür zu sorgen, indem wir Überlappungen durch Aufteilung von Rechtecken eliminieren — natürlich ohne dabei die Gesamtfläche zu verändern. Zum Beispiel könnten wir den Cam-

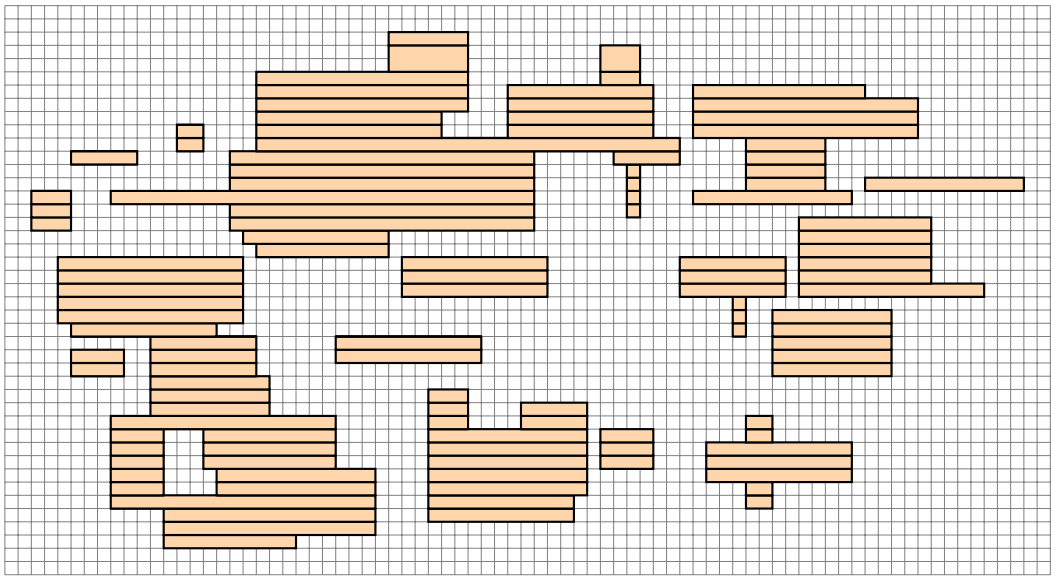


Abbildung 5.7.: Repräsentation des Campus aus Abbildung 5.4 durch horizontale Streifen

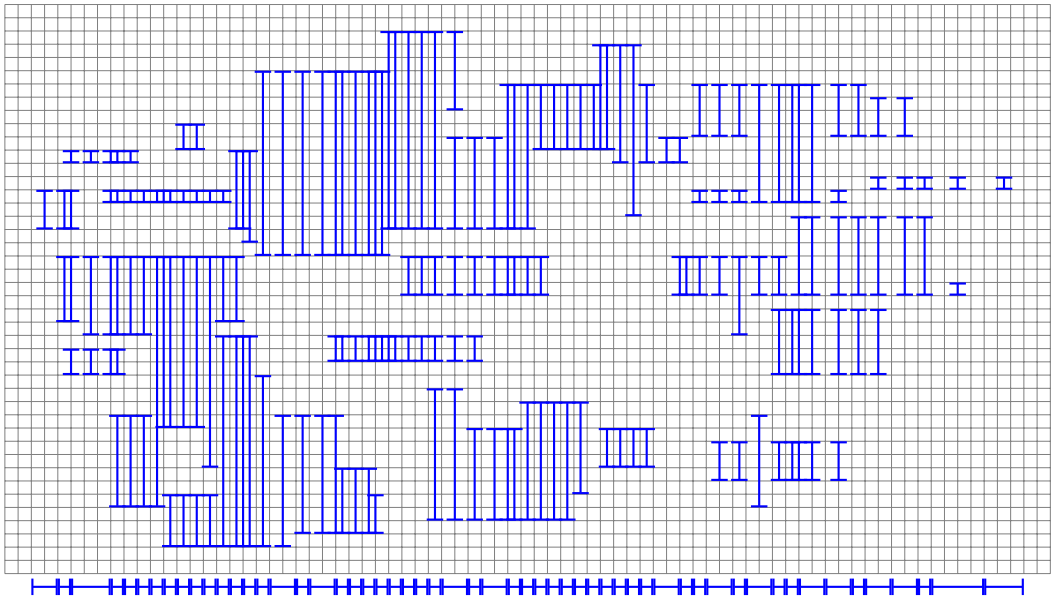


Abbildung 5.8.: x-Intervalle mit assoziierten Listen von y-Intervallen.

pus durch vertikale Streifen repräsentieren, siehe Abbildung 5.6, oder alternativ durch horizontale Streifen, siehe Abbildung 5.7. Mit anderen Worten, wir streben einen *Repräsentationswechsel* an: Eine Menge von Rechtecken wird durch eine Menge disjunkter Streifen repräsentiert, wobei ein vertikaler Streifen durch *ein* x-Intervall und *eine Menge* von disjunkten y-Intervallen gegeben ist, siehe Abbildung 5.8. Das 2-dimensionale Problem lässt sich durch den Repräsentationswechsel auf seine 1-dimensionale Variante zurückführen: die Gesamtlänge einer Menge von Intervallen zu bestimmen. Wenden wir uns dieser Aufgabe als Erstes zu.

Intervalle Wie in Abschnitt 4.1.3 repräsentieren wir ein Intervall durch die beiden Intervallgrenzen.

```
type Interval = {lo : Int; hi : Int}                    // low und high
module I =
  let length (i : Interval) = max 0 (i.hi − i.lo)       // „monus“
  let union (i : Interval, j : Interval) =
    {lo = min i.lo j.lo; hi = max i.hi j.hi}
```

Das lokale Modul I dient dem Zweck, Namenskollisionen zu vermeiden. (In Abschnitt 4.1.3 haben wir auf Tricks wie unterschiedliche Groß- und Kleinschreibung zurückgegriffen.) Die Funktion *union* (i, j) bestimmt das kleinste Intervall, das die Intervalle i und j enthält. Sie implementiert die mengentheoretische Vereinigungen der Punktmengen genau dann, wenn sich die Intervalle überlappen.

Um die Gesamtlänge einer Liste von Intervallen zu berechnen, nehmen wir wie schon angedacht einen *Repräsentationswechsel* vor. Wir überführen die gegebene Liste in eine Liste *disjunkter* Intervalle. Um diese Eigenschaft einfach sicherstellen zu können, vereinbaren wir, dass die Listen im folgenden Sinne *geordnet* sind: (1) für jedes in der Liste enthaltene Intervall i gilt $i.lo < i.hi$ — das Intervall ist nicht leer; (2) für zwei aufeinanderfolgende Intervalle i und j gilt $i.hi < j.lo$ — die Intervalle überschneiden sich nicht, sie berühren sich nicht einmal. Mit anderen Worten, die Intervallgrenzen sind von links nach rechts gelesen streng aufsteigend angeordnet.

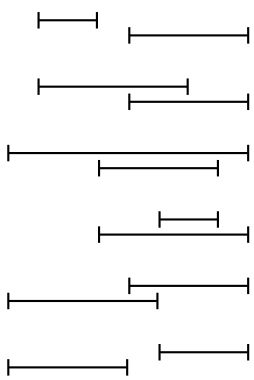
Um den Repräsentationswechsel zu implementieren, verwenden wir das Leibniz Entwurfsmuster („Teile und Herrsche“, engl. divide and conquer). Ähnlich wie beim Sortieren durch Mischen, müssen wir im Wesentlichen eine Funktion schreiben, die zwei geordnete Intervalllisten vereinigt.

```
union : Interval list * Interval list → Interval list
```

Wir nehmen an, dass die beiden Argumente geordnet sind und erwarten dies auch für das Ergebnis. Unsere Annahmen und Erwartungen werden leider nicht im Typ von *union* reflektiert. Das Typsystem stellt nur sicher, dass die Argumente und das Ergebnis Listen von Intervallen sind.

Das Rekursionsmuster von *union* entspricht im Wesentlichen dem von *merge*: wir betrachten jeweils die beiden ersten Listenelemente und rekurrieren über die Restlisten.

Lediglich die Fallunterscheidung ist etwas umfangreicher, da die Listenelemente nicht Zahlen sind, sondern Intervalle, Paare von Zahlen. Wir haben in Abschnitt 3.3 bereits diskutiert, dass zwei Intervalle, i und j, auf sechs mögliche Art und Weisen zueinander liegen können.



setzte i voran	$i :: union\ (is, j :: js)$
erweitere j um i	$union\ (is, I.union\ (i, j) :: js)$
ignoriere j	$union\ (i :: is, js)$
ignoriere i	$union\ (is, j :: js)$
erweitere i um j	$union\ (I.union\ (i, j) :: is, js)$
setzte j voran	$j :: union\ (i :: is, js)$

Zu jedem Fall erhalten wir einen symmetrischen Fall, indem wir die Rollen von i und j vertauschen. Im Kern müssen wir also drei Situationen unterscheiden: (1) ein Intervall liegt vor dem anderen; (2) die Intervalle überlappen sich teilweise; (3) ein Intervall liegt vollständig in dem anderen. Im ersten Fall setzen wir das weiter links liegende Intervall der Ergebnisliste voran; im zweiten Fall vereinigen wir die Intervalle; im dritten Fall verwerfen wir das eingeschlossene Intervall.

```
module Is =
  // union : Interval list * Interval list -> Interval list
  let rec union = function
    | ([], is) | (is, [])                          -> is
    | (i :: is, j :: js) when i.hi < j.lo          -> i :: union (is, j :: js)
    | (i :: is, j :: js) when j.hi < i.lo          -> j :: union (i :: is, js)
    | (i :: is, j :: js) when i.lo < j.lo && i.hi < j.hi -> union (is, I.union (i, j) :: js)
    | (i :: is, j :: js) when i.lo <= j.lo && i.hi >= j.hi -> union (i :: is, js)
    | (i :: is, j :: js) when i.lo >= j.lo && i.hi <= j.hi -> union (is, j :: js)
    | (i :: is, j :: js) when i.lo > j.lo && i.hi > j.hi -> union (I.union (i, j) :: is, js)
```

Das Schlüsselwort ***when*** verbindet ein Muster mit einem Booleschen Ausdruck und erlaubt so, die verschiedenen Fälle sehr direkt und übersichtlich in Programmcode zu überführen. Wie beim Musterabgleich üblich, werden die Bedingungen von oben nach unten abgearbeitet. Sind zum Beispiel die ersten drei Bedingungen nicht zutreffend, so wissen wir, dass die Listen nicht leer sind und sowohl $i.hi \geqslant j.lo$ als auch $j.hi \geqslant i.lo$ gilt.

Nach diesen Vorarbeiten ist die Implementierung des Repräsentationswechslers *interval* Routine — so wie die Funktion *union* zu *merge* korrespondiert, so korrespondiert *intervals* zu *merge-sort* (die Funktion *intervals* wird wie union in dem lokalen Modul *Is* definiert).

```
module Is =
  ⋮
  // intervals : Interval list → Interval list
  let rec intervals = function
    | []  → []
    | [i] → [i]
    | is  → let (is1, is2) = unzip is
            union (intervals is1, intervals is2)
```

Auch hier spiegelt sich der Repräsentationswechsel nicht im Typ wider; *interval* bildet tatsächlich eine beliebige Liste von Intervallen auf eine geordnete Liste ab. Da sichergestellt ist, dass die Intervalle disjunkt sind, ergibt sich sodann die gewünschte Gesamtlänge als Summe der Einzellängen.

```
module Is =
  ⋮
  let total-length = sum-by I.length
```

Die Lösung für das Problem der Berechnung der Gesamtlänge ist somit durch die Funktionskomposition $intervals \gg total\text{-}length : Interval\ list \to Int$ gegeben.

Extrahieren wir alle x-Intervalle bzw. alle y-Intervalle aus der Rechteckliste, die den Campus repräsentiert,

```
Mini⟩ Is.intervals [for r in campus → r.x]
[{ lo = 2; hi = 77 }]
Mini⟩ Is.intervals [for r in campus → r.y]
[{ lo = 2; hi = 41 }]
```

erkennen wir, dass die Gebäude „dicht“ liegen. Blicken wir in Richtung der Achsen auf das Gelände, sehen wir jeweils einen zusammenhängenden Wall von Wänden. (Letztere Aussage interpretiert die Rechenergebnisse. Aus Sicht des Rechners ist das Ergebnis jeweils eine einelementige Liste, aus Sicht des Stadtplaners vielleicht ein Hinweis auf eine zu dichte Bebauung.) Das Argument von *intervals* ist übrigens eine sogenannte *Listenbeschreibung* — eine ähnliche Syntax haben wir auch für die Konstruktion von Arrays verwendet. Ist xs die Liste $[x_1; \ldots; x_n]$ dann bezeichnet $[\mathbf{for}\ x\ \mathbf{in}\ xs \to f\ x]$ die Liste $[f\ x_1; \ldots; f\ x_n]$; die Funktion f wird auf jedes Element der Liste xs angewendet.

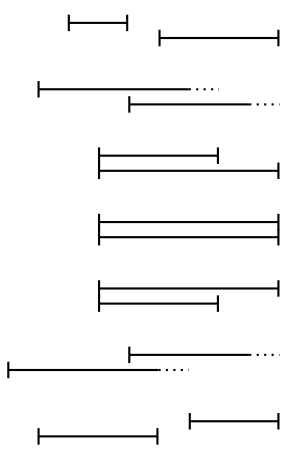
Nach diesen Vorarbeiten sind wir bestens gerüstet, um unser ursprüngliches Problem zu lösen.

Intervalle von Intervallen Erinnern wir uns an die Idee des Repräsentationswechsels: Wir wollen eine Rechteckmenge bzw. eine Liste von Rechtecken durch eine Liste von vertikalen Streifen repräsentieren, wobei ein vertikaler Streifen durch ein x-Intervall und eine Liste von y-Intervallen gegeben ist. Der Repräsentationswechsler hat somit den Typ:

$$rectangles : Rectangle\ list \to (Interval * Interval\ list)\ list$$

Um sicherzustellen, dass die resultierenden Rechtecke disjunkt sind, muss die äußere Liste bezüglich der x-Intervalle geordnet sein und zusätzlich müssen alle inneren Listen, die Listen der y-Intervallen, geordnet sein. Im Detail verlangen wir für die äußere Liste: (1) für jeden in der Liste enthaltenen Eintrag (i, a) gilt $i.lo < i.hi$ — das Intervall ist nicht leer; (2) für zwei aufeinanderfolgende Einträge (i, a) und (j, b) gilt $i.hi \leqslant j.lo$ — die Intervalle überschneiden sich nicht, sie dürfen sich aber berühren. Wir sind etwas „großzügiger“ im Vergleich zu den inneren Listen: Dort sind Berührungen nicht zugelassen (zwei sich berührende Intervalle können ja stets vereinigt werden), hier sehr wohl (eine Vereinigung wäre möglich, aber nur wenn beiden x-Intervallen exakt die gleichen y-Intervalle zugeordnet sind).

Da die Invarianten der Datenstrukturen etwas unterschiedlich sind, müssen wir bei der Vereinigung von x-Intervallen insgesamt sieben Fälle unterscheiden (statt sechs Fälle wie bei den y-Intervallen).



setzte i voran	$(i, a) :: union\ (is, (j, b) :: js)$
teile i	$(il, a) :: union\ ((ir, a) :: is, (j, b) :: js)$
vereinige Präfix	$(i, a \mathbin{@} b) :: union\ (is, (jr, b) :: js)$
vereinige i und j	$(i, a \mathbin{@} b) :: union\ (is, js)$
vereinige Präfix	$(j, a \mathbin{@} b) :: union\ ((ir, a) :: is, js)$
teile j	$(jl, b) :: union\ ((i, a) :: is, (jr, b) :: js)$
setzte j voran	$(j, b) :: union\ ((i, a) :: is, js)$

Zu jedem Fall erhalten wir einen symmetrischen Fall, indem wir die Rollen von i und j vertauschen. Da der vierte Fall symmetrisch zu sich selbst ist, kommen wir auf insgesamt sieben Fälle. Im Kern müssen wir vier Situationen unterscheiden: (1) ein Intervall liegt vor dem anderen; (2) ein Intervall ragt in das andere hinein; (3) ein Intervall ist ein Präfix des anderen; (4) die beiden Intervalle sind identisch. Im ersten Fall setzen wir das weiter links liegende Intervall der Ergebnisliste voran; im zweiten Fall teilen wir ein Intervall auf; im dritten Fall vereinigen wir zusätzlich die assoziierten y-Intervalle; im vierten Fall müssen wir nicht teilen, sondern nur die assoziierten y-Intervalle vereinigen.

Die Hilfsfunktion *split-at* trennt ein Intervall an einer gegebenen Stelle auf.

$$\textbf{\textit{let}}\ split\text{-}at\ x\ i = (\{\, lo = i.lo; hi = x\,\}, \{\, lo = x; hi = i.hi\,\})$$

Wir stellen beim Aufruf stets sicher, dass $i.lo < x < i.hi$, so dass ein nicht-leeres Intervall in zwei nicht-leere Intervalle überführt wird.

Die Funktion *union* implementiert die oben detaillierte Fallunterscheidung. Die Fälle sind etwas umgeordnet worden, um die Bedingungen einfacher formulieren zu können.

```
// union : (Interval * 'a list) list * (Interval * 'a list) list → (Interval * 'a list) list
let rec union = function
  | ([], is) | (is, [])                          → is
  | ((i, a) :: is, (j, b) :: js) when i.hi ⩽ j.lo → (i, a) :: union (is, (j, b) :: js)
  | ((i, a) :: is, (j, b) :: js) when j.hi ⩽ i.lo → (j, b) :: union ((i, a) :: is, js)
  | ((i, a) :: is, (j, b) :: js) when i.lo < j.lo → let (il, ir) = split-at j.lo i
                                                    (il, a) :: union ((ir, a) :: is, (j, b) :: js)
  | ((i, a) :: is, (j, b) :: js) when i.lo > j.lo → let (jl, jr) = split-at i.lo j
                                                    (jl, b) :: union ((i, a) :: is, (jr, b) :: js)
  | ((i, a) :: is, (j, b) :: js) when i.hi < j.hi → let (jl, jr) = split-at i.hi j
                                                    (i, a @ b) :: union (is, (jr, b) :: js)
  | ((i, a) :: is, (j, b) :: js) when i.hi = j.hi → (i, a @ b) :: union (is, js)
  | ((i, a) :: is, (j, b) :: js) when i.hi > j.hi → let (il, ir) = split-at j.hi i
                                                    (j, a @ b) :: union ((ir, a) :: is, js)
```

Der Typ von *union* ist allgemeiner als erwartet: Die den x-Intervallen zugeordneten Werte müssen lediglich Listen sein, nicht notwendigerweise Listen von y-Intervallen. Das liegt daran, dass wir im Fall von Überlappungen die assoziierten Werte einfach konkatenieren — das klappt für Listen beliebigen Grundtyps.

Der Repräsentationswechsler *rectangles* etabliert die Invarianten getrennt für jede Dimension. Im ersten Schritt, implementiert durch die Hilfsfunktion *union-all*, wird die Liste der Rechtecke in eine geordnete Liste von x-Intervallen überführt. Im zweiten Schritt wird für jedes x-Intervall die Liste der assoziierten y-Intervalle „sortiert“, implementiert durch die Funktion *intervals*.

```
let rec union-all = function
  | []  → []
  | [r] → [(r.x, [r.y])]
  | rs  → let (rs1, rs2) = unzip rs
          union (union-all rs1, union-all rs2)
// rectangles : Rectangle list → (Interval * Interval list) list
let rectangles rs = [for (i, js) in union-all rs → (i, Is.intervals js)]
```

Jetzt sind wir fast am Ziel. Da sowohl die x-Intervalle als auch die y-Intervalle disjunkt sind, ergibt sich die Gesamtfläche als Summe der Flächen der vertikalen Streifen. Die Fläche eines einzelnen Streifens entspricht dem Produkt der Länge des x-Intervalls und der Gesamtlänge der y-Intervalle.

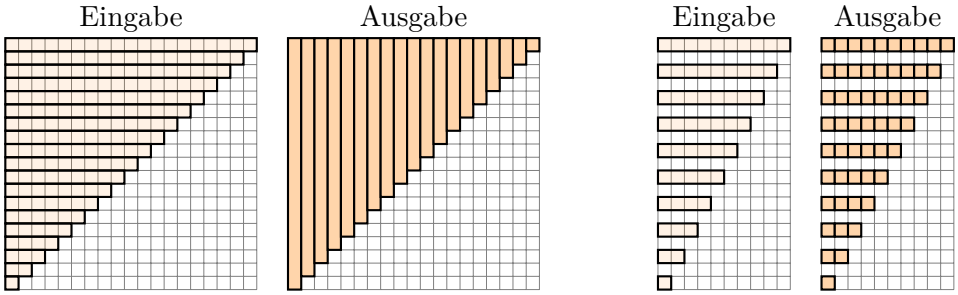
```
let total-area = sum-by (fun (i, js) → I.length i * Is.total-length js)
```

Für den Campus der Rheinland-Pfälzischen Universität Kaiserslautern-Landau ergeben sich die folgenden Werte.

```
Mini⟩ length campus
50
Mini⟩ let rs = rectangles campus
Mini⟩ total-area rs
1124
Mini⟩ length rs
56
Mini⟩ sum-by (fun (i, js) → length js) rs
156
```

Der aus 50 Rechtecken bestehende Campus wird somit durch 56 x-Intervalle und insgesamt 156 y-Intervalle repräsentiert, siehe Abbildung 5.8.

Analyse der Laufzeit Wie schnell ist unser Algorithmus zur Bestimmung der Gesamtfläche? Die Analyse ist etwas komplizierter als im Fall von *merge-sort*. Das liegt daran, dass wir beim Aufspalten der x-Intervalle Daten verdoppeln (aus (i, a) wird (il, a) und (ir, a)), so dass wir zunächst überlegen müssen, wieviele x-Intervalle und wieviele y-Intervalle maximal generiert werden. Die Zahl der x-Intervalle ist im schlechtesten Fall linear in der Zahl der Rechtecke: Es kann höchstens $2 \cdot n - 1$ x-Intervalle geben, wenn n die Anzahl der Rechtecke ist. (Warum?) Entsprechende Überlegungen gelten für die *einem* x-Intervall zugeordneten y-Intervalle. Summa summarum können somit höchstens quadratisch viele Intervalle generiert werden. Dieser Fall kann tatsächlich auftreten:



Die Eingabedaten sind jeweils lange horizontale Streifen, treppenförmig angeordnet. Im ersten Beispiel werden die horizontale Streifen von *rectangles* in vertikale Streifen überführt — jedem x-Intervall wird also genau ein y-Intervall zugeordnet. Liegen die horizontale Streifen etwas auseinander, erhalten wir eine quadratische Anzahl von y-Intervallen.

Im 1-dimensionalen Fall ist es weniger kompliziert. Um die Gesamtlänge eines Streifens zu berechnen, benötigen wir ungefähr $n \lg n$ Schritte. Da diese Berechnung im 2-dimensionalen Fall für jeden vertikalen Streifen durchgeführt wird, kommen wir auf eine Gesamtlaufzeit von $n^2 \lg n$. Im Vergleich zur exponentiellen Laufzeit eines Verfahrens, das auf dem Prinzip der Ein- und Ausschließung beruht, ist das allerdings eine gigantische Verbesserung, siehe Abbildung 5.5.

5.1.5. Ordnungsstatistik⋆

module
Algorithmics.
Median

Wir haben in Abschnitt 4.3.3 gesehen, dass wir für die Berechnung einer optimalen Trassenführung das Minimum, das Maximum und den Median einer Folge von Elementen benötigen. Alle drei Aufgaben lassen sich einfach lösen, indem man die Folge zunächst sortiert und dann auf das erste, das letzte bzw. das mittlere Element zugreift. Die Laufzeit ist damit linear-logarithmisch, da die Sortierung die Laufzeit dominiert. In diesem Abschnitt wollen wir der Frage nachgehen, ob wir die Aufgaben auch direkter und damit effizienter lösen können. Gleichzeitig verallgemeinern wir die Aufgabenstellung etwas: Es gilt, das i-kleinste Element einer Folge von Elementen zu finden, die sogenannte i-te *Ordnungsstatistik* oder *Ordnungsgröße*.

Minimum und Maximum Die folgenden Programme lassen sich etwas lesbarer aufschreiben, wenn wir Infixoperatoren für das Minimum bzw. das Maximum zweier Elemente einführen — die beiden Definitionen sind nicht spezifisch für Zahlen, sie funktionieren für beliebige totale Quasiordnungen.

$$\textbf{\textit{let}}\ (\lhd)\ a\ b = \textbf{\textit{if}}\ a \leqslant b\ \textbf{\textit{then}}\ a\ \textbf{\textit{else}}\ b$$
$$\textbf{\textit{let}}\ (\rhd)\ a\ b = \textbf{\textit{if}}\ a \leqslant b\ \textbf{\textit{then}}\ b\ \textbf{\textit{else}}\ a$$

Statt den Namen voranzustellen, *min* a b bzw. *max* a b, schreiben wir den Bezeichner zwischen die Argumente, $a \lhd b$ bzw. $a \rhd b$. Nur bei der Definition selbst ist das leider nicht möglich; dort muss der Bezeichner in Klammern gesetzt vorangestellt werden. Die Dreiecke können übrigens als Pfeile oder Pfeilspitzen gelesen werden: Wenn wir a und b der Größe nach anordnen, dann ist $a \lhd b$ das linke, d.h. das kleinere Element und $a \rhd b$ das rechte, d.h. das größere Element. Infixnotation ist immer dann vorteilhaft, wenn die Funktion wie im Fall vom Minimum und Maximum *assoziativ* ist: $(a \lhd b) \lhd c = a \lhd (b \lhd c)$ und $(a \rhd b) \rhd c = a \rhd (b \rhd c)$. Dann können wir Anwendungen des Operators aneinanderreihen, ohne Klammern setzen zu müssen, zum Beispiel, $a \rhd b \rhd c \rhd d \rhd e$.

Ordnungsstatistik für $n \leqslant 5$ Fangen wir klein an. Für eine Folge von drei Elementen lässt sich das i-kleinste Element wie folgt bestimmen.

3-kleinste	$(a \rhd b \rhd c)$
2-kleinste	$(a \rhd b) \lhd (a \rhd c) \lhd (b \rhd c)$
1-kleinste	$(a) \lhd (b) \lhd (c)$

Wie erklärt sich die Formel für den Median, das 2-kleinste Element? Nehmen wir an, wir würden die zwei kleinsten Elemente kennen. Deren Maximum ist der gesuchte Median. Weiterhin ist das Maximum aller anderen 2-elementigen Mengen größer gleich dem Median da $\rhd$ monoton ist. Somit ergibt sich das 2-kleinste Element als das Minimum der Maxima aller 2-elementigen Teilmengen.

Wenn wir das 3-kleinste Element von vier Elementen bestimmen wollen, können wir entsprechend vorgehen. Wir bestimmen die Maxima aller 3-elementigen Teilmengen; deren Minimum ist das gesuchte Element.

4-kleinste	$(a \rhd b \rhd c \rhd d)$
3-kleinste	$(a \rhd b \rhd c) \lhd (a \rhd b \rhd d) \lhd (a \rhd c \rhd d) \lhd (b \rhd c \rhd d)$
2-kleinste	$(a \rhd b) \lhd (a \rhd c) \lhd (a \rhd d) \lhd (b \rhd c) \lhd (b \rhd d) \lhd (c \rhd d)$
1-kleinste	$(a) \lhd (b) \lhd (c) \lhd (d)$

Allgemein ist das i-kleinste Element das Minimum der Maxima aller i-elementigen Teilmengen. Entsprechend ist das i-größte Element das Maximum der Minima aller i-elementigen Teilmengen. Für $n = 3$:

1-größte	$(a) \rhd (b) \rhd (c)$
2-größte	$(a \lhd b) \rhd (a \lhd c) \rhd (b \lhd c)$
3-größte	$(a \lhd b \lhd c)$

Die Klammern umfassen jeweils die Elemente der Teilmengen. Für $n = 4$:

1-größte	$(a) \rhd (b) \rhd (c) \rhd (d)$
2-größte	$(a \lhd b) \rhd (a \lhd c) \rhd (a \lhd d) \rhd (b \lhd c) \rhd (b \lhd d) \rhd (c \lhd d)$
3-größte	$(a \lhd b \lhd c) \rhd (a \lhd b \lhd d) \rhd (a \lhd c \lhd d) \rhd (b \lhd c \lhd d)$
4-größte	$(a \lhd b \lhd c \lhd d)$

Für jede Ordnungsstatistik oder Ordnungsgröße gibt es somit zwei verschiedene Formeln, je nachdem, ob das i-kleinste Element (Min-Max Formel) oder das $(n + 1 - i)$-größte Element (Max-Min Formel) betrachtet wird.

Tatsächlich erhält man viele weitere Formeln, wenn man die Min-Max oder die Max-Min Formel massiert, mit Hilfe algebraischer Eigenschaften umformt. Minimum und Maximum erfüllen zum Beispiel die *Distributivgesetze*:

$$x \lhd (y \rhd z) = (x \lhd y) \rhd (x \lhd z) \qquad (x \rhd y) \lhd z = (x \lhd z) \rhd (y \lhd z) \tag{5.1a}$$

$$x \rhd (y \lhd z) = (x \rhd y) \lhd (x \rhd z) \qquad (x \lhd y) \rhd z = (x \rhd z) \lhd (y \rhd z) \tag{5.1b}$$

Wir können die Anzahl der Operatoren in einem Ausdruck verringern, wenn wir die Gesetze von rechts nach links anwenden — *viele Programmoptimierungen basieren auf einem Distributivgesetz!*

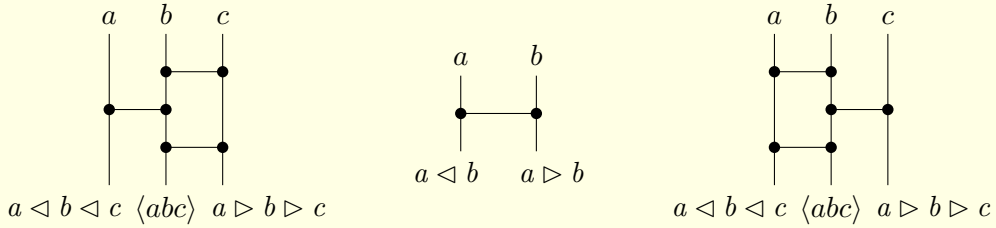
Median für $n \leqslant 5$ Schauen wir uns die Umformungen am Beispiel des Medians an.

Piano $n = 3$: Die Formel für das 2-kleinste Element von 3 Elementen lässt sich wie folgt vereinfachen.

$$\begin{aligned} & (a \rhd b) \lhd (a \rhd c) \lhd (b \rhd c) \\ = \quad & \{ \text{ Distributivgesetz (5.1b) } \} \\ & (a \rhd (b \lhd c)) \lhd (b \rhd c) \end{aligned}$$

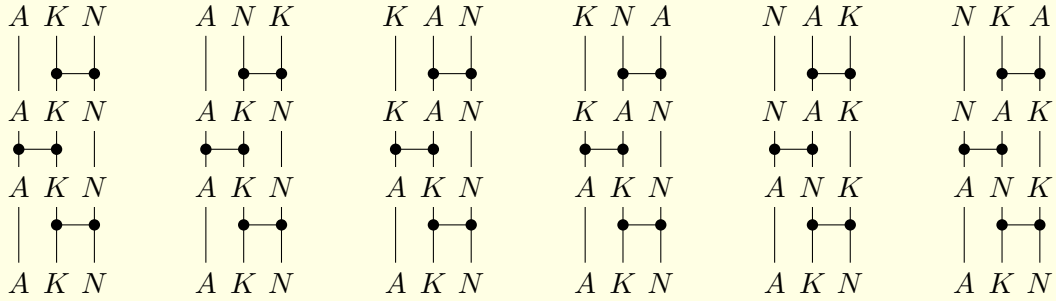
Der resultierende Ausdruck ist optimal in der Anzahl der Operatoren — mit drei Operatoren (Kombinationen von $\lhd$ und $\rhd$) lässt sich der Median nicht bestimmen. Die Formel bestimmt das kleinste der beiden größten Elemente (die Elemente *ohne* das allerkleinste Element $a \lhd b \lhd c$). Um die umgangssprachliche Deutung zu verstehen, hilft vielleicht die Metapher eines Fußball- oder Tennisturniers, siehe Abbildung 5.9.

Eine Abfolge von Spielen lässt sich durch ein *Turnierdiagramm* repräsentieren. Entsprechend der Anzahl der Teilnehmer gibt es n vertikale Linien. Eine horizontale Linie repräsentiert ein Match. Der Verlierer rückt im Spielplan nach links, der Gewinner nach rechts. Von oben nach unten gelesen ergibt sich der zeitliche Ablauf der Spiele; Spiele auf der gleichen Ebene können parallel ausgetragen werden.

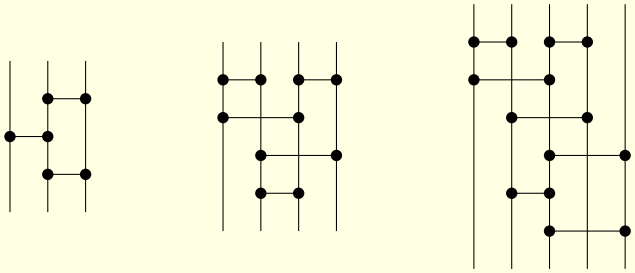


Der linke Spielplan gibt folgende Abfolge von Spielen vor: Zunächst lassen wir b und c gegeneinander antreten; der Verlierer spielt gegen a; der Gewinner dieses Spiels tritt schließlich gegen den Gewinner des ersten Spiels an. (Die Notation $\langle abc \rangle$ ist eine gebräuchliche Abkürzung für den Median von a, b und c.)

Die drei Führenden der Damenweltrangliste im Tennis sind Ashleigh Barty, Karolina Pliskova und Naomi Osaka (Women's Tennis WTA Rankings 2019). Lassen wir sie ein Turnier gemäß dem linken Spielplan austragen, ergeben sich $3! = 6$ mögliche Turnierverläufe, je nachdem in welcher Reihenfolge die Damen aufgestellt werden.



Die Median-Programme korrespondieren zu den folgenden Turnierdiagrammen.



Die ersten beiden Diagramme ermitteln tatsächlich die *vollständige* Rangliste von drei bzw. vier Teilnehmern. Mit anderen Worten, sie sortieren die Eingabe! Das ist beim dritten Diagramm nicht der Fall. (Warum? Welche zusätzlichen Spiele müssen durchgeführt werden, um die Rangliste zu ermitteln?)

Abbildung 5.9.: Turnierdiagramme (auch bekannt unter dem Namen Komparator-Netzwerke, engl. comparator networks).

$$\textbf{let}\ median3\ (a, b, c) = (a \rhd (b \lhd c)) \lhd (b \rhd c)$$

Die Medianfunktion hat viele interessante Eigenschaften. Sie ist zum Beispiel *selbst-dual*: Die Funktion ändert sich nicht, wenn wir auf der rechten Seite $\lhd$ durch $\rhd$ und umgekehrt $\rhd$ durch $\lhd$ ersetzen. Probieren Sie es aus! Oben haben wir den Median mit Hilfe vom Minimum und Maximum definiert. Umgekehrt lassen sich Minimum und Maximum auf den Median zurückführen, sofern die totale Quasiordnung über ein kleinstes bzw. ein größtes Element verfügt:

$$a \lhd b = median\ (-\infty, a, b)$$
$$a \rhd b = median\ (a, b, +\infty)$$

wobei $-\infty$ das kleinste und $+\infty$ das größte Element repräsentiert. Für die Ordnung auf den Wahrheitswerten, $false < true$, spezialisieren sich die obigen Formeln zu $a\ \texttt{\&\&}\ b = median\ (false, a, b)$ und $a\ \texttt{||}\ b = median\ (a, b, true)$. (Die Konjunktion entspricht dem Minimum und die Disjunktion dem Maximum zweier Wahrheitswerte.)

Crescendo $n = 4$*:* Die Formel für das 2-kleinste Element von 4 Elementen lässt sich wie folgt vereinfachen.

$$\begin{aligned}
& (a \rhd b) \lhd (a \rhd c) \lhd (a \rhd d) \lhd (b \rhd c) \lhd (b \rhd d) \lhd (c \rhd d) \\
= \quad & \{\ \text{Distributivgesetz (5.1b), zweimal}\ \} \\
& (a \rhd b) \lhd (a \rhd (c \lhd d)) \lhd (b \rhd (c \lhd d)) \lhd (c \rhd d) \\
= \quad & \{\ \text{Distributivgesetz (5.1b)}\ \} \\
& (a \rhd b) \lhd ((a \lhd b) \rhd (c \lhd d)) \lhd (c \rhd d)
\end{aligned}$$

Der resultierende Ausdruck für den Untermedian berechnet das kleinste der drei größten Elemente, siehe auch Abbildung 5.9.

$$\textbf{let}\ median4\ (a, b, c, d) = (a \rhd b) \lhd ((a \lhd b) \rhd (c \lhd d)) \lhd (c \rhd d)$$

Fortissimo $n = 5$*:* Wenn die Anzahl der Elemente größer wird, werden die Formeln langsam aber sicher unhandlich. Die Max-Min Formel für das 3-größte Element von 5 Elementen,

$$\begin{aligned}
& (a \lhd b \lhd c) \rhd (a \lhd b \lhd d) \rhd (a \lhd b \lhd e) \rhd (a \lhd c \lhd d) \rhd (a \lhd c \lhd e) \\
& \qquad \rhd (a \lhd d \lhd e) \rhd (b \lhd c \lhd d) \rhd (b \lhd c \lhd e) \rhd (b \lhd d \lhd e) \rhd (c \lhd d \lhd e)
\end{aligned}$$

besteht aus $\binom{5}{3} = 10$ Min-Gliedern und aus $10 \cdot 2 + 9 = 29$ Operatoren. Glücklicherweise hilft ein modularer Ansatz weiter: Der Median von 5 Elementen lässt sich auf den Median von 3 Elementen zurückführen, wenn wir zwei Elemente „rauskicken“, die für den Median nicht in Frage kommen.

$$\textbf{let}\ median5\ (a, b, c, d, e) = median3\ ((a \lhd b) \rhd (c \lhd d), (a \rhd b) \lhd (c \rhd d), e)$$

Das kleinste und das größte Element der ersten vier Elemente können ignoriert werden. Der Median aus den mittleren beiden Elementen und dem Element e ist der gesuchte

Median der 5 Elemente, siehe auch Abbildung 5.9. (Schaffen Sie es, das obige Programm aus der Max-Min Formel herzuleiten?)

Für größere n stoßen wir an die Grenzen des Ansatzes: Da es $\binom{n}{k}$ k-elementige Teilmengen einer n-elementigen Menge gibt, enthalten die Min-Max und die Max-Min Formeln insgesamt $k \cdot \binom{n}{k} - 1$ Operatoren. Mit anderen Worten, für die Bestimmung des 1-kleinsten Elements benötigen wir lineare Laufzeit (optimal), für das 2-kleinste Element quadratische Laufzeit (na ja), für das 3-kleinste Element kubische Laufzeit (oh je) usw.

Ordnungsstatistik für $n > 5$ Versuchen wir die allgemeine Lösungsstrategie auf die Ordnungsstatistik anzuwenden und überlegen, wie wir das Problem auf die Lösung kleinerer Probleme zurückführen können.

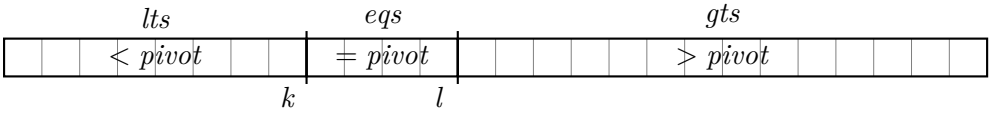
Für das Peano Entwurfsmuster müssen wir ein Element zur Seite legen. Ein beliebiges Element herauszugreifen ist nicht zielführend: Wenn wir zum Beispiel das Element 11 zur Seite legen und 47 ist das i-kleinste Element der restlichen Folge, dann können wir nur schließen, dass 47 das $(i+1)$-kleinste Element der Gesamtfolge ist — das nützt uns aber nichts. Das Entwurfsmuster führt zum Ziel, wenn wir ein extremales Element, das kleinste oder das größte, auswählen. Leider ist das reslutierende Verfahren nicht besonders effizient: Um das i-kleinste Element zu bestimmen, benötigen wir $i \cdot n$ Schritte, im Fall des Medians also quadratisch viele Schritte. Im Wesentlichen re-implementieren wir das Sortierverfahren „Sortieren durch Auswählen“. Dessen große Schwester ist das „Sortieren durch Austauschen“, das wir in Abschnitt 5.1.2 haben links liegen lassen. Vielleicht können wir die zugrundeliegende Idee adaptieren?

Im Sinne des Leibniz Entwurfsmusters müssen wir die Problemgröße ungefähr halbieren. Dazu wählen wir ein „Pivotelement“ aus und teilen die Eingabefolge in kleinere und größere Elemente. Gibt es insgesamt k kleinere Elemente und ist $i \leqslant k$, dann bestimmen wir das i-kleinste Element unter diesen Elementen, anderenfalls das $(i-k)$-kleinste Element unter den größeren Elementen. Im Unterschied zum „Sortieren durch Austauschen“ tätigen wir nur *einen* rekursiven Aufruf.

Die „1 Million €“ Frage ist natürlich: Wie wählen wir das Pivotelement? Wir drücken uns für den Moment um die Antwort und nehmen einfach an, dass wir ein *Orakel* befragen können. Wenn wir dem Orakel die Folge präsentieren, wählt es für uns ein geeignetes Folgenelement aus. Die Funktion *quickselect*, die das i-kleinste Element einer *nicht-leeren* Liste *xs* von Elementen bestimmt, ist somit zusätzlich mit dem Orakel parametrisiert. (Wir nehmen an, dass der Index i gültig ist: $1 \leqslant i \leqslant \mathit{length}\ \mathit{xs}$.)

```
let rec quickselect (oracle : 'a list → 'a) i xs =          // 1 ⩽ i ⩽ length xs
  let pivot = oracle xs
  let lts   = filter (fun x → x < pivot) xs
  let eqs   = filter (fun x → x = pivot) xs
  let gts   = filter (fun x → x > pivot) xs
  let k     = length lts
  let l     = k + length eqs
  if   i ⩽ k then quickselect oracle i       lts
  elif i ⩽ l then pivot
             else quickselect oracle (i − l) gts
```

Die gegebene Liste *xs* wird tatsächlich in *drei* Teillisten partioniert: in echt kleinere Elemente, gleiche Element und echt größere Elemente.



Wenn $k < i \leqslant l$ gilt, ist das Pivotelement das gesuchte Element; in den beiden anderen Fällen erfolgt ein rekursiver Aufruf.

Die Partitionierung erledigt die vordefinierte Funktion *filter*, die aus einer gegebenen Liste alle „guten“ Elemente aussiebt — frei nach Aschenputtel „die guten ins Töpfchen, die schlechten ins Kröpfchen.“

```
let rec filter (good : 'a → Bool) : 'a list → 'a list = function
  | []      → []
  | x :: xs → if good x then x :: filter good xs
                        else      filter good xs
```

Die Funktion *quickselect* folgt einem etwas merkwürdigen Rekursionsschema. Zum Beispiel ist nicht unmittelbar klar, wo der Basisfall behandelt wird. Nun, wenn die Eingabeliste einelementig ist, dann wählt *oracle* dieses eine Element aus. Die Listen *lts* und *gts* sind dann leer und wir landen im zweiten Zweig der Fallunterscheidung, in dem das Pivotelement zurückgegeben wird. Für längere Listen ist die Terminierung gewährleistet, da *lts* und *gts* stets weniger Elemente als *xs* enthalten. (Es sei denn das Orakel mogelt: Der Aufruf *quickselect* (*fun* _ → 0) 7 [1 . . 9] terminiert nicht, da das Orakel ein Element zurückgibt, das gar nicht in der Liste enthalten ist.)

Um die Terminierung zu gewährleisten, partitionieren wir übrigens die ursprüngliche Liste in drei Teillisten (<, = und >) und nicht nur in zwei ($\leqslant$ und > oder < und $\geqslant$). Die binäre Partitionierung ist problematisch, wenn die Eingabeliste Elemente mehrfach enthält. Sind im Extremfall alle Elemente der Eingabe identisch, dann ist eine Teilliste stets leer und die andere zur Eingabeliste identisch — Nichtterminierung ist die Folge.

Kommen wir zur Analyse der Laufzeit. Nehmen wir zunächst an, dass das Orakel uns nicht wohlgesonnen ist. Im schlechtesten Fall wählt das Orakel ein extremales Element

Sei $T(n)$ die Zeit (engl. time), die ein Algorithmus für eine Eingabe der Größe n benötigt. Die Zeitfunktion der binären Suche erfüllt im Rekursionsschritt die Gleichung $T(n) = 1+T(n/2)$. Wir berechnen eine Zeiteinheit für die Unterteilung des Suchraums und weitere administrative Aufgaben. Der binäre Loagrithmus „löst“ die Gleichung: $T(n) \sim \log_2 n$. Die binäre Suche hat somit eine logarithmische Laufzeit.

Im Fall von *quickselect* ist der Aufwand im Rekursionsschritt nicht mehr konstant, sondern linear: $T(n) = n+T(n/2)$. Die Gesamtlaufzeit ist in diesem Fall ebenfalls linear: $T(n) \sim 2{\cdot}n$. Der Faktor 2 ergibt sich als Wert der geometrischen Reihe $\sum_{k=0}^{\infty}(1/2)^k$.

Zur Erinnerung: Eine *geometrische Folge* a_n hat die Eigenschaft, dass der Quotient aufeinanderfolgender Glieder konstant ist: $a_{n+1}/a_n = q$. Im einfachsten Fall ist $a_n = q^n$. Die dazugehörige Folge der Partialsummen heißt *geometrische Reihe*: $s_n = \sum_{k=0}^{n-1} a_k$. Eine geschlossene Formel für $s_n = \sum_{k=0}^{n-1} q^k$ lässt sich wie folgt herleiten. Wir betrachten die Summe s_{n+1}; diese lässt sich auf zwei Arten aufschreiben, indem man entweder die ersten n oder die letzten n Summanden zusammenfasst:

$$s_n + q^n = 1 + q + q^2 + q^3 + \cdots + q^{n-1} + q^n = 1 + q \cdot s_n$$

Die resultierende Gleichung lösen wir sodann nach s_n auf:

$$s_n + q^n = 1 + q \cdot s_n \quad \Longleftrightarrow \quad s_n - q \cdot s_n = 1 - q^n \quad \Longleftrightarrow \quad s_n = (1 - q^n)/(1 - q)$$

Ist $|q| < 1$, dann strebt der Zähler für $n \to \infty$ gegen eins: $\lim_{n\to\infty} s_n = 1/(1 - q)$.

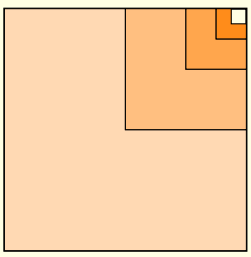
Den Grenzwert einer geometrischen Reihe kann man auch direkt herleiten:

$$\begin{aligned}
& s = 1 + q + q^2 + q^3 + \cdots \\
\Longleftrightarrow \quad & \{ \text{ Distributivgesetz } \} \\
& s = 1 + q \cdot (1 + q + q^2 + \cdots) \\
\Longleftrightarrow \quad & \{ \text{ Definition von } s \} \\
& s = 1 + q \cdot s \quad \Longleftrightarrow \quad s = 1/(1 - q)
\end{aligned}$$

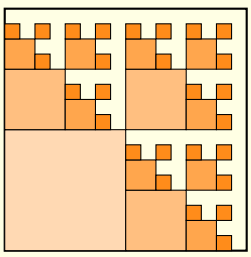
(Eine ähnliche Herleitung haben wir schon einmal gesehen. Erinnern Sie sich wo?)

Die Grenzwerte geometrischer Reihen lassen sich übrigens geometrisch sehr ansprechend als Parkettierungen des Einheitsquadrats deuten.

$\frac{3}{4} \cdot \sum_{k=0}^{\infty} \left(\frac{1}{4}\right)^k = 1$



$\frac{1}{4} \cdot \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k = 1$



$\frac{3}{4} + \frac{3}{4} \cdot \frac{1}{4} + \frac{3}{4} \cdot \frac{1}{16} + \frac{3}{4} \cdot \frac{1}{64} + \cdots = 1$

$\frac{1}{4} + \frac{1}{4} \cdot \frac{3}{4} + \frac{1}{4} \cdot \frac{9}{16} + \frac{1}{4} \cdot \frac{27}{64} + \cdots = 1$

Abbildung 5.10.: Laufzeitanalyse und geometrische Reihen.

aus, so dass der rekursive Aufruf eine Liste erhält, die nur um ein Element kleiner ist. In diesem Fall ergibt sich eine quadratische Laufzeit.

Wenn uns das Orakel wohlgesonnen ist, dann beschleunigt sich die Rechnung. Im besten Fall wählt das Orakel den Median der Eingabefolge, so dass in jedem Schritt die Problemgröße ungefähr halbiert wird. Dann benötigt der Algorithmus insgesamt

$$\frac{1}{1}n + \frac{1}{2}n + \frac{1}{4}n + \frac{1}{8}n + \cdots < 2 \cdot n$$

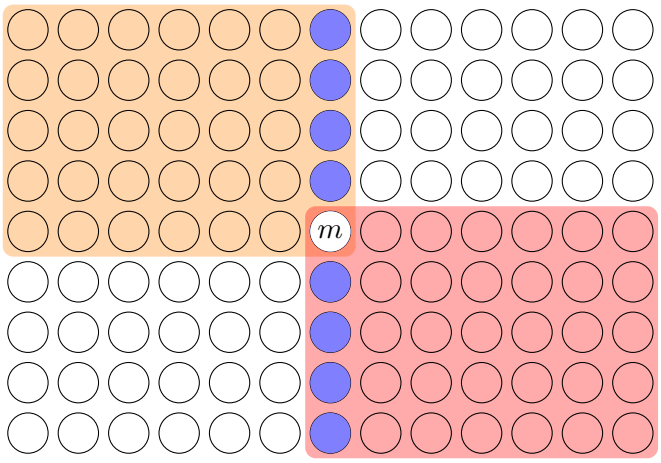
Schritte, hat also eine lineare Laufzeit, siehe Abbildung 5.10. Das ist asymptotisch optimal — in sublinearer Zeit können wir das i-kleinste Element nicht bestimmen, da wir uns zumindest jedes Element anschauen müssen. Halten wir fest: *Wenn* wir den Median in linearer Zeit berechnen können, *dann* lässt sich jede beliebige Ordnungsgröße in linearer Zeit berechnen. (Der Median ist gewissermaßen der schwierigste Spezialfall.) Wenn das Wörtchen „wenn“ nicht wäre …

Median der Medianе Zurück zum Ausgangspunkt: Die allgemeine Lösungsstrategie empfiehlt, Probleme auf kleinere Probleme zurückzuführen. Dabei ist es nicht zwingend, die Problemgröße zu halbieren. Wenn wir zum Beispiel in jedem Schritt die Größe um den Faktor $\frac{3}{4}$ verringern, dann hat *quickselect* weiterhin eine lineare Laufzeit,

$$\frac{1}{1}n + \frac{3}{4}n + \frac{9}{16}n + \frac{27}{64}n + \cdots < 4 \cdot n$$

da die *geometrische Reihe* $\sum_{k=0}^{\infty} q^k$ für $|q| < 1$ konvergiert: $\sum_{k=0}^{\infty} q^k = 1/(1-q)$. Die Laufzeit verdoppelt sich lediglich: Wir benötigen jetzt $4 \cdot n$ statt wie vorher $2 \cdot n$ Schritte. Auf unser Problem angewendet heißt das, es genügt ein Pivotelement zu finden, so dass mindestens $\frac{1}{4}$ der Elemente kleiner und mindestens $\frac{1}{4}$ der Elemente größer sind — die Lage der restlichen Elemente zum Pivotelement ist uns egal. Damit wäre garantiert, dass der rekursive Aufruf von *quickselect* auf höchstens $\frac{3}{4}$ der ursprünglichen Elemente angewendet wird. Was uns jetzt noch fehlt ist eine zündende Idee, wie wir ein solches Pivotelement bestimmen.

Das folgende Gedankenexperiment weist den Weg. Wir ordnen die Elemente gedanklich 2-dimensional in einem Rechteck an und bestimmen zunächst den Median jeder einzelnen Zeile und anschließend den Median der Zeilenmediane.



Der Median der Zeilenmediane m besitzt die gewünschte Eigenschaft. Um uns davon zu überzeugen, rearrangieren wir die Elemente: In jeder Zeile plazieren wir die kleineren Elemente links und die größeren rechts vom Zeilenmedian, der blau eingefärbt jeweils in der Mitte liegt. Weiterhin plazieren wir die Zeilen, deren Median kleiner ist als m, oberhalb von m und die Zeilen, deren Median größer ist, unterhalb. Auf Grund der Transitivität der Quasiordnung folgt, dass m größer gleich den Elementen in der orange hinterlegten Fläche ist und kleiner gleich den Elementen in der roten Fläche — über die Beziehung der restlichen Element zu m lässt sich allerdings nichts aussagen.

Noch sind wir nicht am Ziel — wir müssen ja noch klären, wie wir die Zeilenmediane und den Median der Zeilenmediane bestimmen. Welche Optionen haben wir? Zunächst einmal spielt die Form des Rechtecks für das obige Argument keine Rolle. Insbesondere können wir eine *konstante*, vorher festgelegte Breite verwenden, wie zum Beispiel 13 in der obigen Grafik. Für die Breite wählen wir geschickterweise ein ungerade Zahl, anderenfalls müssten wir den Unter- oder den Obermedian bestimmen. Aber wie klein können wir sie wählen? Klappt es mit 5 oder gar mit 3 Elementen? Für diese Spezialfälle haben wir ja bereits Lösungen programmiert, die wir für die Berechnung der Zeilenmediane verwenden könnten. Und wie berechnen wir den Median der Zeilenmediane? Nun, dafür tätigen wir einen weiteren rekursiven Aufruf!

Die obige Analyse der Laufzeit ist damit nicht länger gültig, da wir ja ursprünglich von einem, nicht zwei rekursiven Aufrufen ausgegangen sind. Versuchen wir die Laufzeit für die Breite 3 *grob* abzuschätzen. Dazu sei $T(n)$ die Anzahl der Schritte (engl. time), die der Algorithmus für eine Folge von n Elementen benötigt. Im Rekursionsfall erfüllt T die Gleichung $T(n) = n + T\left(\frac{1}{3} \cdot n\right) + T\left(\left(1 - \frac{2}{3} \cdot \frac{1}{2}\right) \cdot n\right)$. Wir tätigen wie gesagt zwei rekursive Aufrufe. Mit dem ersten bestimmen wir den Median von $\frac{1}{3} \cdot n$ Zeilenmedianen. Mit diesem partitionieren wir dann die ursprüngliche Liste. Die Teilliste *lts* enthält somit mindestens $\frac{2}{3} \cdot \frac{1}{2} \cdot n$ Elemente (die orangene Fläche); gleiches gilt für *gts* (die rote Fläche). Die jeweils andere Teilliste hat also höchstens $\left(1 - \frac{2}{3} \cdot \frac{1}{2}\right) \cdot n$ Elemente. (Wir gehen wie immer von dem schlechtesten Fall aus.) Für alle restlichen Arbeiten veranschlagen wir n Schritte.[4] Nun kann man zeigen, dass eine Funktion T mit dieser Eigenschaft leider

[4]Das ist natürlich etwas niedrig gegriffen. Es spielt aber tatsächlich keine Rolle, ob wir n, $2n$, $47n$

linear-logarithmisch ist. Das ist die schlechte Nachricht; die gute Nachricht ist, dass der Ansatz für die Breite 5 funktioniert: T mit $T(n) = n + T\left(\frac{1}{5} \cdot n\right) + T\left(\left(1 - \frac{3}{5} \cdot \frac{1}{2}\right) \cdot n\right)$ ist tatsächlich linear. Die folgende Übersicht zeigt, dass je größer wir die Breite wählen, desto näher die Laufzeit an den Ausgangspunkt $4 \cdot n$ (nur ein rekursiver Aufruf) heranrückt.

$$
\begin{aligned}
T(n) &= n + T(n/3) + T((1 - 2/6) \cdot n) && T(n) \sim 3n \cdot \log_{6.75} n \\
T(n) &= n + T(n/5) + T((1 - 3/10) \cdot n) && T(n) \sim 10n \\
T(n) &= n + T(n/7) + T((1 - 4/14) \cdot n) && T(n) \sim 7n \\
T(n) &= n + T(n/9) + T((1 - 5/18) \cdot n) && T(n) \sim 6n \\
&\vdots && \vdots \\
T(n) &= n + T(n/99) + T((1 - 50/198) \cdot n) && T(n) \sim 4.125n
\end{aligned}
$$

Die Notation $f(n) \sim g(n)$ bedeutet, dass der Quotient der Funktionswerte für hinreichend große n gegen 1 strebt: $\lim_{n\to\infty} f(n)/g(n) = 1$. Ab einer Breite von 5 fällt der zusätzliche rekursive Aufruf nicht signifikant ins Gewicht, so dass wir endlich, endlich zur Implementierung des Verfahrens schreiten können.

Für die Berechnung der Zeilenmediane teilen wir die Eingabeliste in Gruppen zu je 5 Elementen auf.

```
let rec medians-of-5 = function
  | []                          → []
  | [x1]                        → [x1]
  | [x1; x2]                    → [x1 ◁ x2]
  | [x1; x2; x3]                → [median3 (x1, x2, x3)]
  | [x1; x2; x3; x4]            → [median4 (x1, x2, x3, x4)]
  | x1 :: x2 :: x3 :: x4 :: x5 :: xs → median5 (x1, x2, x3, x4, x5) :: medians-of-5 xs
```

Da die Division durch 5 nicht immer glatt aufgeht, müssen wir insgesamt 5 Basisfälle behandeln. Dazu machen wir ausgiebig Gebrauch von Listenmustern, wobei zum Beispiel $[x_1; x_2; x_3]$ das geschachtelte Muster $x_1 :: x_2 :: x_3 :: [\,]$ abkürzt.

Kommen wir zum großen Finale: Die Funktion *ordselect* berechnet die Ordnungsstatistik in linearer Zeit. Sie spezialisiert *quickselect* mit dem Median der Zeilenmediane als Orakel. Die Medianfunktion selbst wird rekursiv mit Hilfe von *ordselect* definiert.

```
let rec ordselect i xs = quickselect (median ≪ medians-of-5) i xs
and median = function
  | [x1]                        → x1
  | [x1; x2]                    → x1 ◁ x2
  | [x1; x2; x3]                → median3 (x1, x2, x3)
  | [x1; x2; x3; x4]            → median4 (x1, x2, x3, x4)
  | [x1; x2; x3; x4; x5]        → median5 (x1, x2, x3, x4, x5)
  | xs                          → ordselect (length xs ÷ 2) xs
```

oder $c \cdot n$ Schritte benötigen — es geht uns ja um Größenordnungen: linear, linear-logarithmisch, quadratrisch etc. Solange der Faktor c konstant ist, haben die folgenden Ergebnisse Bestand.

Das Rekursionsmuster ist jetzt noch merkwürdiger: *ordselect* verwendet *median* und *median* verwendet umkehrt *ordselect*. Im Fachjargon sagt man, die beiden Funktion sind *verschränkt rekursiv* definiert. Verschränkt rekursive Funktionsdefinitionen müssen mit dem Schlüsselwort ***and*** verbunden werden, damit jede Funktion jede andere sieht: ***let rec*** $f_1\ (x_1) = e_1$ ***and*** $f_2\ (x_2) = e_2$. Die Bezeichner f_1 und f_2 sind dann sowohl in e_1 als auch in e_2 sichtbar. Im Gegensatz dazu ist in ***let rec*** $f_1\ (x_1) = e_1$ ***let rec*** $f_2\ (x_2) = e_2$ der Bezeichner f_2 nur in e_2 sichtbar, nicht aber in e_1.

Die Funktion *median* verwendet die weiter oben definierten Spezialfunktionen, um den Median für Listen der Länge $n \leqslant 5$ zu bestimmen, behandelt also insgesamt 5 Basisfälle. Somit führt das Orakel nur dann einen rekursiven Aufruf von *ordselect* aus, wenn die ursprüngliche Liste mehr als $5{\cdot}5 = 25$ Element umfasst. (Der Basisfall der einelementigen Liste muss zwingend behandelt werden, um die Terminierung zu gewährleisten.)

Fazit: Mit einer großen Portion Hartnäckigkeit sind wir ans Ziel gekommen. Beim Entwurf von Algorithmen hilft wie bei vielen anderen Tätigkeiten Erfahrung. So haben wir uns wiederholt gefragt, wieviele zusätzliche Rechnungen wir uns erlauben können, ohne das große Ziel, eine lineare Gesamtlaufzeit, aus den Augen zu verlieren bzw. zu gefährden (Verkleinerung der Problemgröße um $\frac{3}{4}$ statt $\frac{1}{2}$; zwei rekursive Aufrufe statt einem). Darüber hinaus gibt es oft einen kreativen Moment, den oft zitierten Geistesblitz. In unserem Beispiel ist das sicherlich die Idee, die Elemente 2-dimensional in einem Rechteck anzuordnen und den Median der Mediane als Privotelement zu verwenden.

5.2. Suchen

module
Algorithmics.
Search

Im täglichen Leben wie in der Informatik gehört Suchen zu den häufigen, wenn auch nicht immer beliebten Tätigkeiten. Wir suchen nach Schlüsseln, Unterlagen, Waren, Personen …. Beim Schachspielen suchen wir nach einem Gewinnzug, bei unserem Ratespiel aus Abschnitt 3.6 nach einer Zahl.

Bei der ersten Gruppe von Beispielen liegen die zu durchsuchenden Daten konkret vor, zerstreut in einer Wohnung oder wohlorganisiert in einer Datenstruktur auf einem Rechner. In den Abschnitten 5.2.1–5.2.3 schauen wir uns drei mögliche Organisationsformen an: Listen, Suchlisten und Suchbäume.

Bei der zweiten Kategorie von Beispielen sind die Daten virtuell: Nicht alle möglichen Stellungen eines Schachspiels sind explizit repräsentiert; sie ergeben sich implizit mittels der Regeln des Schachspiels. Wir beschreiben den Suchraum in geeigneter Weise und nutzen die Beschreibung bei der Suche. Abschnitt 5.2.4 beschäftigt sich mit dem Suchen in „virtuellen Räumen“ und richtet dabei ein besonderes Augenmerk auf Korrektheit und Terminierung.

5.2.1. Listen

Nehmen wir an, wir wollen für die Personalabteilung eines Unternehmens Personaldaten verwalten. Um Personen gleichen Namens einfach auseinanderhalten zu können, erhält jede Mitarbeiterin und jeder Mitarbeiter bei Amtsantritt eine eindeutige Personalnummer. Den Personalstamm können wir dann durch eine Liste von Einträgen der Typs

```
type Entry = {key : Nat; person : Person}
```

repräsentieren. Jeder Eintrag besteht aus einer Personalnummer und den eigentlichen Personaldaten. Vereinbarungsgemäß enthält die Liste, die den Personalstamm repräsentiert, keine zwei Einträge mit der gleichen Personalnummer. Zum Beispiel:

```
let team = [{key = 7;    person = ralf    };
            {key = 815;  person = melanie};
            {key = 4711; person = julia   };
            {key = 4712; person = andres  }]
```

Eine wiederkehrende Aufgabe ist es, zu einer gegebenen Personalnummer die zugehörigen Personaldaten herauszusuchen. Um die Funktion für beliebige Personaldaten verwenden zu können, parametrisieren wir sie mit der Liste der Einträge.

```
lookup (key : Nat, staff : List ⟨Entry⟩) : Person
```

Was machen wir, wenn kein passender Eintrag existiert? Wir stellen die beiden möglichen Resultate einer Suche, erfolglos und erfolgreich, mit Hilfe des Typs *Option* dar.

```
lookup (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩
```

Der Typ drückt aus, dass die Suche *möglicherweise* eine Person zurückgibt. Schlägt die Suche fehl, wird der Wert *None* zurückgegeben, sonst der Wert *Some p*, wobei *p* die gesuchte Person ist.

Nach diesen Vorarbeiten kommen wir mit dem Struktur Entwurfsmuster für *List* unmittelbar zum Ziel.

```
let rec lookup (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []             → None
  | entry :: entries → if key = entry.key then Some entry.person
                       else  lookup (key, entries)
```

Die Implementierung von *lookup* erinnert an die lineare Suche aus Abschnitt 3.6: Die Liste wird von vorne nach hinten durchsucht, bis ein passender Listeneintrag gefunden wird oder das Listenende erreicht ist. Verbessern lässt sich das Verfahren nicht ohne weiteres, denn Listen sind der Natur nach linear. (Eine Liste ähnelt wie gesagt einem Akten- oder Tellerstapel: Auf das oberste Objekt kann man bequem zugreifen, auf die darunterliegenden Objekte nicht.)

5.2.2. Suchlisten

Wir können die Suche, zumindest die erfolglose Suche, etwas beschleunigen, wenn wir die Liste nach der Personalnummer ordnen. Eine geordnete oder sortierte Liste nennen wir auch *Suchliste* — dies ist allerdings kein etablierter Begriff. Wenn wir von einer

Sortierung ausgehen — die Liste *team* ist in der Tat nach der Personalnummer geordnet —, lässt sich *lookup* wie folgt verbessern.

```
let rec lookup (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []                → None
  | entry :: entries → if   key < entry.key   then None
                       elif key = entry.key   then Some entry.person
                       (* key > entry.key *) else  lookup (key, entries)
```

Aus dem 2-Wege Vergleich wird ein 3-Wege Vergleich. Ist die gesuchte Nummer echt kleiner als die des ersten Eintrags, so kann die Suche unmittelbar abgebrochen werden. Die erfolglose Suche wird schneller, die erfolgreiche nicht. Suchen wir zum Beispiel nacheinander nach allen Personalnummern, so benötigen wir in beiden Fällen $1 + 2 + \cdots + n - 1 + n = \binom{n+1}{2} = n \cdot (n+1)/2$ (rekursive) Aufrufe.

3-Wege Vergleiche sind übrigens das Konstrukt der Wahl, wenn man mit totalen Quasiordnungen arbeitet, da zwei Elemente auf genau drei Arten zueinander in Beziehung stehen können: $a < b$, $a \sim b$ oder $a > b$. Da die Ordnung auf den natürlichen Zahlen antisymmetrisch ist, ist im obigen Programm '$\sim$' durch '$=$' gegeben.

5.2.3. Binäre Suchbäume

module
Algorithmics.
Tree

Können wir die binäre Suche aus Abschnitt 3.6 adaptieren? Nein, nicht ohne den Geschwindigkeitsvorteil zu verlieren. Im Gegensatz zur Halbierung des Suchintervalls kostet die Halbierung einer Liste viele Rechenschritte. (Wieviele?)

Wenn wir schnell auf das mittlere Element zugreifen wollen, müssen wir die Struktur der Liste ändern. Wir brauchen einen anderen Containertyp!

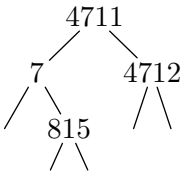
```
type Tree ⟨'a⟩ =
  | Leaf
  | Node of Tree ⟨'a⟩ * 'a * Tree ⟨'a⟩
```

Der Konstruktor *Leaf* tritt an die Stelle von *Nil* bzw. []; wie *Nil* repräsentiert auch *Leaf* die leere Folge. Der Konstruktor *Node* (l, x, r) tritt an die Stelle von *Cons* (x, xs) bzw. $x :: xs$ und repräsentiert eine mindestens einelementige Folge, bestehend aus der „linken" Teilfolge l, dem Element x und der „rechten" Teilfolge r. Die Folge 7 815 4711 4712 kann zum Beispiel durch

```
Node (Node (Leaf, 7, Node (Leaf, 815, Leaf)), 4711, Node (Leaf, 4712, Leaf))
```

repräsentiert werden. Neben dieser Möglichkeit gibt es noch 13 andere. (Nachrechnen!)

Elemente des neuen Variantentyps nennen wir auch *Binärbaume*; Baum wegen der hierarchischen Struktur; *Binär*baum, da jeder nicht-leere Baum in zwei Teilbäume verzweigt. Sind die Elemente von links nach rechts geordnet, so spricht man weiterhin von einem *Suchbaum*. Klar, *Tree* soll ja die Suche besser unterstützen als *List*. (Binärbäume haben aber noch viele andere Anwendungen, deswegen lohnt sich ein spezieller Begriff.) Grafisch dargestellt sieht der obige Ausdruck so aus:



Projiziert man die Elemente auf die x-Achse, so erhält man die ursprüngliche Folge 7 815 4711 4712. Da diese geordnet ist, handelt es sich bei dem Binärbaum um einen Suchbaum.

Mit Hilfe des Struktur Entwurfsmusters für den Datentyp *Tree* können wir die letzte Definition von *lookup* leicht auf Binärbäume adaptieren — wir gehen davon aus, dass der Personalstamm nicht länger als Liste vorliegt, sondern als Binärbaum.

```
let rec lookup (key : Nat, staff : Tree ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | Leaf                       → None
  | Node (left, entry, right) → if   key < entry.key   then lookup (key, left)
                                elif key = entry.key   then Some entry.person
                                (* key > entry.key *)  else  lookup (key, right)
```

In einem Suchbaum dient das Element in der *Wurzel*, dem obersten Knoten (engl. node), als Wegweiser. In unserem Anwendungsfall sind die Einträge nach dem Schlüssel geordnet. Ist der gesuchte Schlüssel kleiner als der Schlüssel des Wegweisers, suchen wir im linken Teilbaum weiter. (Da die Einträge im rechten Teilbaum größer sind als der gesuchte Schlüssel, ist eine Suche im rechten Teilbaum zwar möglich, aber unnötig — sie wäre stets erfolglos.) Der symmetrische Fall, der gesuchte Schlüssel ist größer als der Schlüssel des Wegweisers, wird entsprechend symmetrisch behandelt.

Welche Laufzeit hat die neue Version von *lookup*? Nun, das kommt ganz auf die Form des Suchbaums an. Sind die Elemente links und rechts jeweils gleichmäßig verteilt, ist die Laufzeit logarithmisch. Ein solcher Suchbaum heißt auch *ausgeglichen* oder *balanciert*. Ist hingegen einer der Teilbäume jeweils leer — der Baum degeneriert zur Liste —, dann ist die Laufzeit linear.

Im Allgemeinen ist die Laufzeit von *lookup* proportional zur *Höhe* des Binärbaums.

```
let rec height = function
  | Leaf          → 0
  | Node (l, x, r) → 1 + max (height l) (height r)
```

Die Höhe entspricht der Länge des längsten Pfades von der Wurzel zu einem *Blatt* (engl. leaf), siehe auch Abschnitt 5.1.3.

Die Funktion *lookup* geht davon aus, dass die Stammdaten als Binärbaum vorliegen. Wie aber konstruieren wir einen Binärbaum? Diesem und anderen Themen wenden wir uns in Abschnitt 5.3 ausführlich zu.

5.2.4. Binäre Suche: Korrektheit und Terminierung

module
Algorithmics.
Neighbours

Schlag die Nachbarn! Nehmen Sie sich etwas Zeit und versuchen Sie das folgende Problem zu lösen, *bevor* sie weiterlesen.

> Sie sind in der populären Spielshow „Schlag die Nachbarn!“ ins Finale gekommen und müssen die letzte Aufgabe meistern. Ihnen wird eine nicht-leere Folge von Schachteln präsentiert, die jeweils eine für Sie nicht sichtbare Zahl enthalten. Sie müssen eine Schachtel finden, deren Zahl größer ist als die ihrer Nachbarn. Eine Schachtel zu öffnen kostet 100€. Wenn Sie weniger Geld als Ihre Konkurrent*innen ausgeben, gewinnen Sie das Finale!

Schauen wir uns eine konkrete Schachtelfolge an (die uns als laufendes Beispiel dienen wird). Aus Gründen der Übersichtlichkeit beschränken wir uns auf zehn Schachteln — in der Spielshow ist die Gesamtzahl der Schachteln tatsächlich wesentlich größer.

0	1	2	3	4	5	6	7	8	9
0	4	2	7	6	5	3	9	8	1

(5.2)

Die zehn Schachteln sind von 0 bis 9 durchnummeriert, damit wir uns einfach auf sie beziehen können. Eine der Schachteln mit den Hausnummern 1, 3 oder 7 ist gesucht — jede von ihnen schlägt ihre Nachbarn.

Über den Inhalt der Schachteln wissen wir a priori nichts: Die versteckten Zahlen können beliebig klein sein, beliebig groß sein; die Zahlen können alle verschieden sein oder alle gleich. Die letzte Möglichkeit zwingt uns die umgangssprachliche Formulierung „deren Zahl größer ist“ zu präzisieren. Um zu garantieren, dass immer eine Lösung existiert, interpretieren wir „größer“ als ‘$\geqslant$’ und *nicht* als ‘$>$’. Die Schachtel i schlägt somit ihre Nachbarn genau dann, wenn

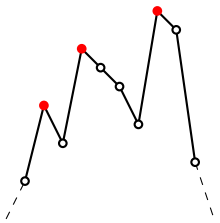
für

	$i-1$	i	$i+1$	
$\cdots$	a	m	b	$\cdots$

gilt: $\quad a \leqslant m \geqslant b$.

Ein zweites Detail bedarf der Klärung: Wann schlagen die Schachteln an den Rändern ihre Nachbarn? Um Spezialfälle zu vermeiden, behelfen wir uns eines Tricks: Wir nehmen an, dass es am linken und am rechten Rand jeweils eine „virtuelle“ Schachtel gibt, die $-\infty$ enthält.

-1	0	1	2	3	4	5	6	7	8	9	10
$-\infty$	0	4	2	7	6	5	3	9	8	1	$-\infty$

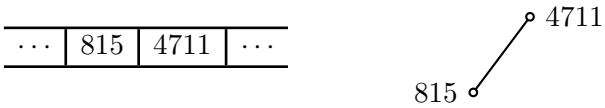


Der Wert $-\infty$ ist keine gültige Zahl *in* Mini–F# (in der Objektsprache); wir verwenden $-\infty$ nur, wenn wir *über* Mini–F# Programme reden (in der Metasprache). Dabei gilt vereinbarungsgemäß $-\infty < n$ für alle ganzen Zahlen n. (Ähnliche Betrachtungen lassen sich für die neu hinzugekommene Hausnummer -1 anstellen. Da wir später im

Programm die Schachteln mit den natürlichen Zahlen durchnummerieren, ist -1 keine gültige Hausnummer. Wir verwenden -1 nur, wenn wir über das Programm reden.)

Die Zuordnung von Hausnummern zu Inhalten ist der Natur nach eine endliche Abbildung. Wenn wir ihren Graph plotten (siehe oben; der Übersichtlichkeit halber haben wir die Funktionswerte miteinander verbunden), sehen wir etwas plastischer, wonach wir eigentlich suchen. Der Graph erinnert an eine Berglandschaft; wir wollen einen Berggipfel besteigen; irgendeinen, nicht notwendigerweise den höchsten Gipfel. Etwas profaner ausgedrückt: Wir suchen ein *lokales Maximum*, *nicht* ein globales. In unserem Beispiel gibt es insgesamt drei lokale Maxima (und zwei lokale Minima). Um das globale Maximum zu finden, müssen wir notwendigerweise alle Schachteln öffnen. Kommen wir mit weniger Versuchen aus, wenn wir nur ein lokales Maximum suchen?

Das globale Maximum bestimmen wir mit einer linearen Suche; kommen wir für ein lokales Maximum mit der binären Suche ans Ziel? Eine gute Idee, aber nicht unmittelbar zielführend. Wir öffnen die mittlere Schachtel und sehen, sagen wir, 4711. Was können wir aus dieser Beobachtung schließen? Nichts! Da die Schachteln beliebige Werte enthalten können, benötigen wir einen Referenzwert. Eine vielleicht naheliegende Idee ist, zwei benachbarte Schachteln zu öffnen und aus dem Vergleich ihrer Werte Rückschlüsse über das weitere Vorgehen zu ziehen. Wir öffnen die beiden mittleren Schachteln:



Die positive Steigung verrät, dass wir uns auf einer Flanke links von einem Berggipfel befinden, so dass es naheliegt, rechts von der Mitte weiterzusuchen. Wäre die Steigung negativ, würden wir entsprechend links fortfahren. Das folgende Programm setzt die Idee um.

```
let beat-your-neighbours (box : Nat → Nat) (lower : Nat, upper : Nat) =
  let rec search (l, u) =
    if l = u then u
             else let m = (l + u) ÷ 2
                  if box m ⩽ box (m + 1) then search (m + 1, u)
                                         else search (l,     m)
  in search (lower, upper)
```

Die Schachteln werden durch die Funktion *box* repräsentiert, die Hausnummern im angegebenen Intervall auf die Inhalte abbildet. (Auf diese Weise wird eine endliche Abbildung repräsentiert. Das Intervall gibt den Definitionsbereich an, die Funktion die Zuordnung.)

Die Vorgehensweise ähnelt der binären Suche, ist aber von der „Grundstimmung“ eine andere. Bei der binären Suche, so wie bei der Suche in einem binären Suchbaum haben wir in jedem Schritt eine Hälfte des Suchraums ausgeschlossen, da wir garantieren konnten, dass das gesuchte Objekt dort *nicht* zu finden ist (negative Stimmung). Hier suchen wir in der Hälfte weiter, in der die Existenz eines lokalen Maximums garantiert ist (positive Stimmung). In der anderen Hälfte können ebenfalls lokale Maxima existieren, nur garantieren können wir dies eben nicht.

Die Metapher des Bergsteigens lässt plausibel erscheinen, dass das Programm ein lokales Maximum ermittelt. Aber tut es das wirklich? Versuchen wir uns an einem Korrektheitsbeweis. Wir konzentrieren uns auf die Hilfsfunktion *search*, die die eigentliche Arbeit verrichtet.[5] Die Beschreibung des Problems legt nahe, dass

$$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1) \quad \text{wobei} \quad i = search\ (l, u)$$

Leider erfüllt das Programm diese Spezifikation nicht: $search\ (4, 6)$ gibt für unser laufendes Beispiel 4 zurück; der Inhalt der entsprechenden Schachtel, $box\ 4 = 6$, ist aber kein lokales Maximum. Was läuft schief? Im Allgemeinen gibt es drei Möglichkeiten:

1. das Programm ist falsch oder
2. die Spezifikation ist falsch oder
3. beide sind falsch.

In unserem Fall ist die Spezifikation nicht korrekt. Das gewünschte Ergebnis ist an eine Vorbedingung geknüpft: Wir setzen voraus, dass wir uns zwischen einer linken Bergflanke / und einer rechten Bergflanke \ befinden.

$$box\ (l-1) \leqslant box\ l \ \wedge \ box\ u \geqslant box\ (u+1)$$

Nur wenn diese *Vorbedingung* erfüllt ist, dann gilt die obige *Nachbedingung*. Eine Vorbedingung knüpft Erwartungen an den Parameter einer Funktion. Die Aufrufer*in der Funktion muss diese Bedingung sicherstellen. Eine Nachbedingung beschreibt Eigenschaften des Funktionsresultats — die Funktion selbst muss diese garantieren.

Im Fall einer rekursiven Funktion muss die Vorbedingung für alle rekursiven Aufrufe gelten. Die Parameter mögen sich ändern; die Bedingung bleibt immer erhalten. Aus diesem Grund spricht man auch von einer *Invariante*.

Invariante von $search\ (l, u)$: $\quad box\ (l-1) \leqslant box\ l \ \wedge \ box\ u \geqslant box\ (u+1)$

Das „Leben“ einer Invariante unterteilt sich in drei Abschnitte:

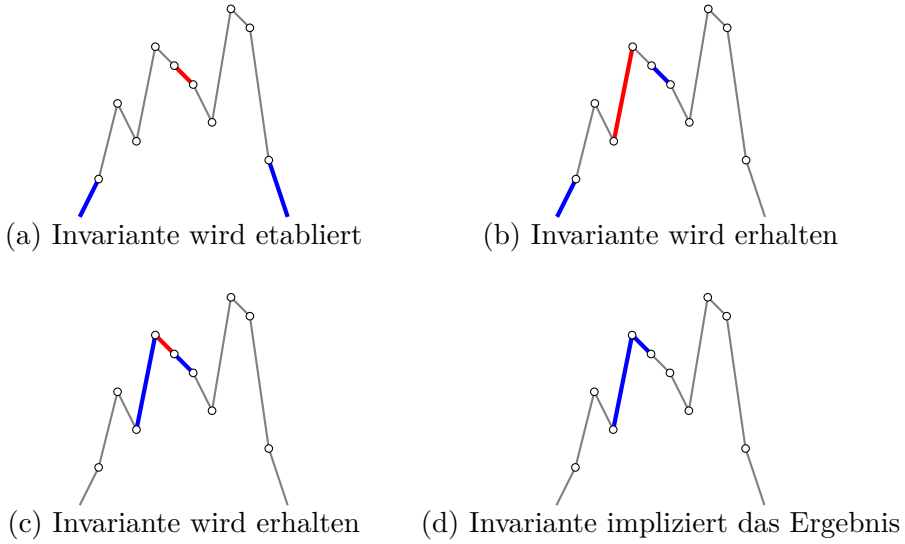
1. die Invariante wird etabliert (initialer Aufruf);
2. die Invariante wird erhalten (Rekursionsschritt);
3. aus der Invariante folgt das gewünschte Ergebnis (Rekursionsbasis).

Für unser Beispiel ergeben sich die folgenden Überlegungen:

Schritt 1: Wir müssen zeigen, dass beim ersten Aufruf der Hilfsfunktion die Invariante etabliert wird, $search\ (lower, upper)$.

$$box\ (lower-1) = -\infty < box\ lower \ \wedge \ box\ upper > -\infty = box\ (upper+1)$$

[5] Das Verhältnis zwischen Hauptfunktion und Hilfsfunktionen entspricht dem Verhältnis zwischen Theorem und Lemmata. Letztere machen die ganze Arbeit, während ersteres die Lorbeeren einheimst.



(a) Invariante wird etabliert (b) Invariante wird erhalten

(c) Invariante wird erhalten (d) Invariante impliziert das Ergebnis

Abbildung 5.11.: Schlag die Nachbarn: Programmablauf von *beat-your-neighbours* für die Eingabe (5.2) — Abfrage jeweils in rot, Invariante in blau.

Hier fließt die Annahme über die virtuellen Schachteln an den Rändern ein.

Schritt 2: Wir müssen sicherstellen, dass die zwei rekursiven Aufrufe, *search* $(m{+}1, u)$ und *search* (l, m), die Invariante erhalten. Die entsprechenden Ungleichungen

$$box\ m \leqslant box\ (m+1) \quad \wedge \quad box\ u \geqslant box\ (u+1)$$
$$box\ (l-1) \leqslant box\ l \quad \wedge \quad box\ m \geqslant box\ (m+1)$$

folgen aus der Invariante für *search* (l, u) und der Abfrage $box\ m \leqslant box\ (m+1)$.

Schritt 3: Schließlich, und das ist der wichtigste Schritt, müssen wir nachweisen, dass im Basisfall die Invariante die gewünschte Nachbedingung impliziert:

$$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1)$$

folgt aus der Invariante für *search* (l, u) und $l = i = u$.

Abbildung 5.11 stellt die verschiedenen Phasen im Leben unserer Invariante grafisch dar. Man sieht sehr schön, dass die Abfrage $box\ m \leqslant box\ (m+1)$ im nächsten Schritt zu einem Teil der Invariante wird. Bildlich gesprochen werden die linke und die rechte Flanke aufeinander zubewegt, bis sie sich im gesuchten Berggipfel berühren.

Partielle und totale Korrektheit Ist damit die Korrektheit von *beat-your-neighbours* nachgewiesen? Nicht ganz, wir haben lediglich die sogenannte *partielle Korrektheit* gezeigt. Für alle zulässigen Eingaben gilt: *Wenn* das Programm terminiert, *dann* produziert es die gewünschte Ausgabe.

Wir werden später sehen, dass *beat-your-neighbours* tatsächlich für alle Eingaben terminiert. Es ist aber ganz lehrreich, sich zu überlegen, was schieflaufen kann. Wenn wir

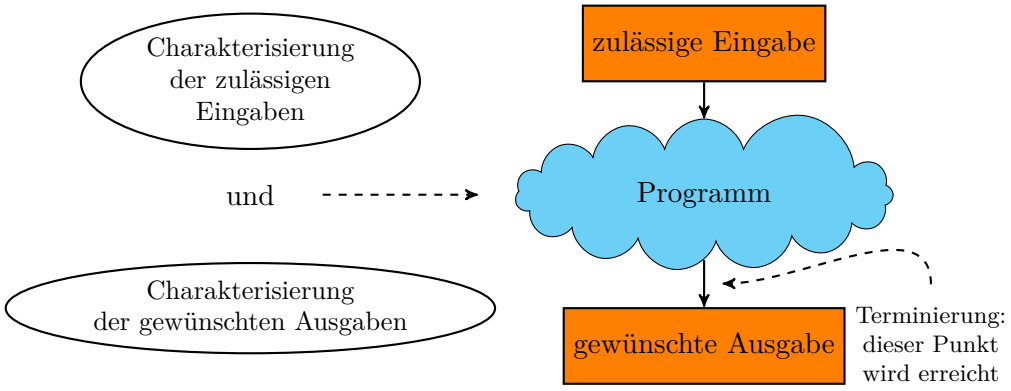


Abbildung 5.12.: Partielle und totale Korrektheit.

den obigen Beweis Revue passieren lassen, stellen wir fest, dass wir gar nicht von der Tatsache Gebrauch machen, dass der Index m ungefähr in der Mitte zwischen l und u liegt. Ersetzen wir zum Beispiel die lokale Definition ***let*** $m = (l+u) \div 2$ durch ***let*** $m = u$, dann hat der Beweis der partiellen Korrektheit unverändert Bestand. Aber das Programm terminiert nicht mehr für alle Eingaben. Nun ist es vielleicht klar, dass ***let*** $m = u$ keine gute Wahl ist. Terminierungsprobleme können aber auch subtiler daherkommen: Die natürliche Division $a \div b$ rundet nach unten ab, $a \div b = \lfloor a/b \rfloor$; würde sie nach oben runden, $a \div b = \lceil a/b \rceil$, wäre die Termierung ebenfalls nicht mehr in allen Fällen gewährleistet. (Können Sie ein Beispiel konstruieren?)

Im Allgemeinen sind wir an der *totalen Korrektheit* interessiert. Für alle zulässigen Eingaben gilt: Das Programm terminiert *und* es produziert die gewünschte Ausgabe. Um die totale Korrektheit zu zeigen, können wir zum Beispiel einen Induktionsbeweis führen, so wie wir das im Fall von „Sortieren durch Mischen“ angedeutet haben. Oder wir teilen uns die Arbeit auf. Wir zeigen in einem ersten Schritt die partielle Korrektheit, etwa mit Hilfe von Invarianten, und kümmern uns dann in einem zweiten Schritt um die Terminierung.

Abbildung 5.12 stellt die Zusammenhänge noch einmal grafisch dar. Es ist wichtig zu betonen, dass die Frage „Ist Programm P korrekt?“ für sich isoliert *keinen Sinn* ergibt! Um Aussagen über die Korrektheit treffen zu können, benötigen wir einen Referenzpunkt, eine *Spezifikation*, die festlegt, *was* das Programm leisten soll, die unsere Erwartungen an das Programm präzisiert. Ein Programm, das detailliert, *wie* die gewünschten Ergebnisse erzielt werden, ist dann korrekt (oder auch nicht) bezüglich dieser Spezifikation.

Die Spezifikation charakterisiert zum einen die zulässigen Eingaben: statische Eigenschaften mit Hilfe von Typen, dynamische Eigenschaften mit Hilfe von Vorbedingungen. Zum anderen charakterisiert sie die gewünschten Ausgaben: statische Eigenschaften wiederum mit Hilfe von Typen, dynamische Eigenschaften mit Hilfe von Nachbedingungen.

Exerzieren wir die verschiedenen Beweisschritte noch einmal mit einem alten Bekannten durch.

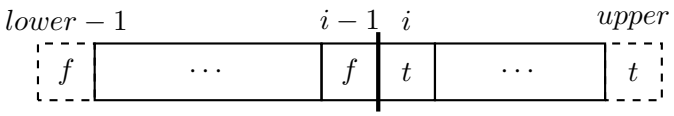
module
Algorithmics.
BinarySearch

Partielle Korrektheit der binären Suche In Abschnitt 3.6 haben wir uns intensiv mit der Modellierung und Implementierung eines Ratespiels beschäftigt. Sozusagen als Nebenprodukt dieser Auseinandersetzung haben wir eine nützliche Bibliotheksfunktion erhalten: die binäre Suche.

```
let binary-search (oracle : Nat → Bool) (lower : Nat, upper : Nat) : Nat =
  let rec search (l, u) =
    if l ⩾ u then u
             else let m = (l + u) ÷ 2
                  if oracle m then search (l,     m)
                              else search (m + 1, u)
  in search (lower, upper)
```

Der erste Parameter, das sogenannte Orakel, gab ursprünglich zu einem gegebenen n Auskunft, ob die gesuchte Zahl gleich oder kleiner als n ist. Wir haben aber gesehen, dass sich die gesuchte Zahl auch durch eine Formel beschreiben lässt, etwa um die natürliche Quadratwurzel einer Zahl zu berechnen. Im allgemeinen Fall bestimmt die binäre Suche das *kleinste* Argument i, für das *oracle* i wahr ist. Wie können wir diese informelle Spezifikation präzisieren?

Zunächst einmal müssen wir formalisieren, dass das Orakel nicht mogelt. Wenn i das gesuchte Argument ist, dann erwarten wir, dass *oracle* für kleinere Argumente stets *false* und für größere stets *true* zurückgibt. Mit anderen Worten, *oracle* springt nicht willkürlich zwischen den beiden möglichen Funktionswerten *true* und *false* hin und her.



Mathematisch gesehen ist *oracle* eine *monotone* Funktion — dabei nehmen wir wie üblich an, dass ‘falsch’ echt kleiner ist als ‘wahr’: *false* $<$ *true*. Ähnlich wie bei „Schlag die Nachbarn!“ machen wir zusätzliche Annahmen über die Ränder: Links ist *oracle* falsch, rechts wahr. Insgesamt stellen wir drei *Vorbedingungen:*

1. $i \leqslant j \implies$ *oracle* $i \leqslant$ *oracle* j — das Orakel mogelt nicht (Monotonie);
2. *lower* $\leqslant$ *upper* — der Suchraum ist nicht leer;
3. *oracle* $(\mathit{lower} - 1) <$ *oracle upper* — eine Lösung existiert stets.

Wenn a und b Boolesche Werte sind, dann bedeutet die Annahme $a < b$ schlicht und einfach, dass a falsch und b wahr ist.

Sind die Vorbedingungen erfüllt, dann garantieren wir die *Nachbedingung:*

$$\mathit{oracle}\ (i-1) < \mathit{oracle}\ i \quad \text{wobei} \quad i = \mathit{binary\text{-}search}\ \mathit{oracle}\ (\mathit{lower}, \mathit{upper})$$

Die binäre Suche bestimmt somit die Stelle, an der der Funktionswert von *oracle* umspringt, von *false* auf *true*.

Die Nachbedingung stellt sicher, dass die Aufrufer*in der Bibliotheksfunktion das gewünschte Ergebnis erhält. Die Aufrufer*in hegt aber noch eine weitere Erwartung, die wir noch nicht thematisiert haben. Die Funktion *oracle* zusammen mit dem Intervall $(lower, upper)$ repräsentiert eine endliche Abbildung. Die Aufrufer*in erwartet, dass diese Abbildung nur auf Elemente ihres Definitionsbereichs angewendet wird. Wird die binäre Suche zum Beispiel verwendet, um ein Array zu durchforsten,

$$binary\text{-}search\ (\textbf{\textit{fun}}\ k \to 4711 \leqslant a.[k])\ (0, a.Length - 1)$$

ist die Abbildung tatsächlich für Werte außerhalb des Definitionsbereichs nicht definiert.

Die eigentliche Arbeit erledigt wie so oft eine Hilfsfunktion. Es gilt die algorithmische Idee dieser Funktion, Intervallschachtelung, mit Hilfe einer Invariante einzufangen. Nun, wir müssen bei jedem rekursiven Aufruf, bei jeder Schachtelung sicherstellen, dass der gesuchte Index weiterhin im Suchintervall enthalten ist. Das können wir garantieren, wenn *oracle* am linken Rand falsch und am rechten Rand wahr ist.

1. Invariante von $search\ (l, u)$: $\quad oracle\ (l - 1) < oracle\ u$

Die Invariante ähnelt verdächtig der Nachbedingung, was natürlich kein Zufall ist.

Wir müssen zusätzlich garantieren, dass die Funktion *oracle* nur mit Werten aus ihrem Definitionsbereich aufgerufen wird. Wir fordern, dass die lokalen Intervallgrenzen l und u stets innerhalb der vorgegebenen, globalen Grenzen *lower* und *upper* liegen.

2. Invariante von $search\ (l, u)$: $\quad lower \leqslant l \leqslant u \leqslant upper$

Mit ähnliche Rechnungen wie für „Schlag die Nachbarn!“ lassen sich die beiden Invarianten nachweisen, siehe Abbildungen 5.13 und 5.14.

Terminierung der binären Suche Um die totale Korrektheit der binären Suche zu etablieren, steht noch der Nachweis der Terminierung aus. Wir müssen zeigen, dass bei jedem rekursiven Aufruf die Argumente „echt kleiner“ werden. Als „Problemgröße“ legen wir fest: Die Größe des Intervalls (l, u) ist $u \dot{-} l$.

Um die Terminierung von *search* zu garantieren, müssen wir somit sicherstellen, dass das Intervall (l, u) bei jedem rekursiven Aufruf echt kleiner wird. Im Rekursionsschritt gilt $l < u$; unter dieser Voraussetzung ergeben sich für die Intervallgrößen:

$$\begin{aligned} u \dot{-} (m + 1) < u \dot{-} l \quad &\iff \quad l < m + 1 \\ m \dot{-} l < u \dot{-} l \quad &\iff \quad m < u \end{aligned}$$

Die rechten Seiten folgen jeweils aus der Eigenschaft des „Mittelwerts“ m, siehe (5.3).

Verträge: Anforderungen und Garantien Nachdem alle wichtigen Detailfragen geklärt sind, blicken wir noch einmal auf das Zusammenspiel zwischen der Bibliotheksfunktion und ihrer Benutzer*in. Dabei ist die Metapher eines juristischen Vertrags hilfreich. Wir haben zwei Parteien, die ihre Anforderungen (engl. requirements) und ihre Garantien (engl. guarantees) durch einen Vertrag (engl. contract) regeln.

Schritt 1: Wir müssen zeigen, dass der initiale Aufruf *search* $(lower, upper)$ die Invariante etabliert:

$$oracle\ (lower - 1) < oracle\ upper$$

Dies folgt unmittelbar aus der 3. Vorbedingung.

Schritt 2: Wir müssen sicherstellen, dass die zwei rekursiven Aufrufe, *search* $(m+1, u)$ und *search* (l, m), die Invariante erhalten. Die entsprechenden Ungleichungen

$$oracle\ m < oracle\ u$$
$$oracle\ (l - 1) < oracle\ m$$

folgen aus der 1. Invariante für *search* (l, u) und der Abfrage *oracle* m.

Schritt 3: Wir müssen zeigen, dass die Invariante das gewünschte Ergebnis, die Nachbedingung impliziert. Die Ungleichung

$$oracle\ (i - 1) < oracle\ i$$

folgt aus der 1. Invariante für *search* (l, u) und $l = i = u$. Letzteres folgt aus der Bedingung $l \geqslant u$ und der 2. Invariante.

Abbildung 5.13.: Binäre Suche: Nachweis der 1. Invariante.

Schritt 1: Aus der 2. Vorbedingung folgt, dass der initiale Aufruf *search* $(lower, upper)$ die Invariante etabliert. (Zusätzlich verwenden wir, dass '$\leqslant$' reflexiv ist.)

Schritt 2: Die rekursiven Aufrufe *search* $(m + 1, u)$ und *search* (l, m) erhalten die Invariante. Die Ungleichungen folgen aus

$$l < u \implies l \leqslant m < m + 1 \leqslant u \quad \text{wobei} \quad m = (l + u) \div 2 \tag{5.3}$$

In den Nachweis fließen Eigenschaften der natürlichen Division ein. Zur Erinnerung: Für den Mittelwert gilt $m = \lfloor (l + u)/2 \rfloor$. (Weiterhin nutzen wir aus, dass '$\leqslant$' transitiv ist.)

Schritt 3: Die Invariante in Verbindung mit (5.3) stellt sicher, dass *oracle* m stets definiert ist:

$$lower \leqslant m < upper$$

Wir machen eine interessante Beobachtung: *oracle* wird nie auf die rechte Intervallgrenze angewendet. Dies ist auch nicht notwendig — wir setzen ja voraus, dass *oracle upper* wahr ist.

Abbildung 5.14.: Binäre Suche: Nachweis der 2. Invariante.

- Die Benutzer*in stellt Anforderungen an das Ergebnis von *binary-search* und
- an die Argumente von *oracle*;
- die Bibliotheksfunktion *binary-search* stellt Anforderungen an ihr zweites Argument und
- an die Ergebnisse von *oracle*, ihrem ersten Argument.

Die Gegenseite muss die Anforderungen jeweils garantieren. Im Allgemeinen stellt die Benutzer*in Anforderungen an das Ergebnis; die Bibliotheksfunktion stellt Anforderungen an die Argumente. Für funktionale Argumente wie *oracle* kehren sich Anforderungen und Garantien um! Warum? Nun, die Autor*in einer Funktion formuliert Anforderungen an die Argumente und gibt Garantien für das Ergebnis. Die Autor*in eines funktionalen Arguments wie *oracle* ist aber die Benutzer*in der Bibliotheksfunktion, nicht die Bibliotheksfunktion selbst.

Die Metapher des Vertrags lässt sich noch weiterspinnen. Im Fall einer Vertragsverletzung (engl. contract violation) lässt sich die oder der Schuldige schnell ausmachen: Sind die Anforderungen verletzt, trifft die Benutzer*in die Schuld (engl. blame); werden die Garantien nicht erfüllt, ist die Bibliotheksfunktion zur Rechenschaft zu ziehen. In unserem Beispiel haben wir bewiesen, dass letzteres Problem nicht auftreten kann. Nicht immer lässt sich ein solcher Beweis führen oder mit vertretbarem Aufwand führen. Alternativ kann man Vor- und Nachbedingungen dynamisch beim Rechnen (man sagt auch während der Laufzeit) überprüfen, indem man die Formeln der Metasprache, in der wir die Bedingungen formuliert haben, zu Ausdrücken der Objektsprache macht. Im Fall von Vertragsverletzungen vulgo Programmierfehlern lässt sich dann der Ort der Fehler genau einkreisen. Verträge werden so zu einem integralen Bestandteil beim Entwurf von Programmen, griffig als „design by contract“ bezeichnet. Zu diesem Thema erfahren Sie im weiteren Verlauf des Studiums mehr aus der Abteilung „Software Engineering“.

Schlag die Nachbarn, da capo! Beim Beweisen wie beim Programmieren sollte man ein gesundes Misstrauen an den Tag legen. In Beweise als auch in Programme können sich Fehler einschleichen: offensichtliche, subtile, unentdeckte. Eine einfache Plausibilitätsprüfung beim Beweisen besteht darin, zu überprüfen, ob man alle Vorausetzungen, alle Annahmen in dem Beweis auch tatsächlich verwendet hat. Führen wir diese Prüfung für den obigen Korrektheitsbeweis durch, stellen wir (vielleicht) erstaunt fest, dass wir die 1. Vorbedingung, die Annahme, dass das Orakel nicht mogelt, gar nicht benutzt haben!

Ein derartiger Befund kann mehrere Ursachen haben: Der Beweis ist falsch, wir haben ein Detail übersehen; oder das Theorem, das der Beweis zeigt, ist allgemeiner als vermutet. In unserem Beispiel ist letzteres der Fall: Die binäre Suche funktioniert auch wunderbar, wenn das Orakel mogelt!

Wenn das Orakel *nicht* mogelt, die Funktion *oracle* also monoton ist, dann können wir schlussfolgern, dass das Ergebnis i das *kleinste* Argument ist, für das *oracle* i wahr ist. Die Tatsache, dass die binäre Suche den Wert minimiert, ist zum Beispiel für die

Berechnung der natürlichen Quadratwurzel essentiell: Das kleinste i mit $n \leqslant i^2$ ist gesucht. *Lisa:* Tatsächlich haben wir das größte i mit $i^2 \leqslant n$ gesucht! Wirklich? Dann müssen wir die Bedingung umschreiben:[6] Wir suchen das kleinste i mit $n < (i+1)^2$.

$$\textit{let } \textit{square-root } n = \textit{binary-search } (\textit{fun } k \to n < \textit{square } (k+1))\ (0, n)$$

Wir sollten uns noch vergewissern, dass das Orakel tatsächlich monoton ist:

$$i \leqslant j \implies (n < \textit{square } (i+1)) \leqslant (n < \textit{square } (j+1))$$

Die Ordnung auf den Booleschen Werten ist durch die Implikation gegeben — die Symbole '$\Longrightarrow$' und '$\leqslant$' bezeichnen tatsächlich die gleiche Boolesche Operation. Wir müssen somit zeigen: Wenn $i \leqslant j$ und $n < \textit{square } (i+1)$ gelten, dann gilt auch $n < \textit{square } (j+1)$. Dies ist der Fall, da *square* auf den natürlichen Zahlen monoton ist.

Wenn das Orakel hingegen mogelt, dann gibt die binäre Suche *irgendein* Argument i mit $\textit{oracle } (i-1) < \textit{oracle } i$ zurück, nicht notwendigerweise das kleinste. Auch für diese Variante haben wir schon einen Anwendungsfall kennengelernt: Schlag die Nachbarn! Und in der Tat: Wenn wir die Arbeitsfunktionen gleichen Namens miteinander vergleichen, stellen wir eine frappierende Ähnlichkeit fest.

```
let rec search (l, u) =
  if l ⩾ u then u
  else
    let m = (l + u) ÷ 2
    if oracle m
      then search (l,     m)
      else search (m + 1, u)
```

```
let rec search (l, u) =
  if l = u then u
  else
    let m = (l + u) ÷ 2
    if box m ⩽ box (m + 1)
      then search (m + 1, u)
      else search (l,     m)
```

Da wir als Vorbedingung $l \leqslant u$ annehmen, sind die beiden Tests $l \geqslant u$ und $l = u$ gleichwertig. Die Zweige der zweiten Alternative lassen sich in Übereinstimmung bringen, indem wir die Bedingung $\textit{box } m \leqslant \textit{box } (m+1)$ negieren. Heureka! Unser Problem „Schlag die Nachbarn!“ lässt sich mit Hilfe der binären Suche lösen:

```
let beat-your-neighbours (box : Nat → Nat) =
  binary-search (fun i → box i > box (i + 1))
```

Die Leser*in sollte sich überzeugen, dass die Vorbedingungen für die binäre Suche tatsächlich erfüllt sind — jede Vorbedingungen sollte sich aus den Annahmen für „Schlag die Nachbarn!“ herleiten lassen.

5.3. Endliche Abbildungen

module Algorithmics.Map.Map

Wenden wir uns wieder dem Mini-Projekt aus Abschnitt 5.2.1 zu, der Verwaltung des Personalstamms eines Unternehmens. Zur Erinnerung: Die Stammdaten haben wir durch eine Liste bzw. einen Baum von Einträgen des Typs

[6] Dabei nutzen wir aus, dass $\textit{max } \{\, n \in \mathbb{Z} \mid P(n) \,\} = \textit{min } \{\, n \in \mathbb{Z} \mid \neg P(n+1) \,\}$.

$$\textbf{\textit{type}}\ \mathit{Entry} = \{\mathit{key} : \mathit{Nat}; \mathit{person} : \mathit{Person}\}$$

repräsentiert. Wir haben bisher diskutiert, wie man zu einer Personalnummer die entsprechenden Personaldaten heraus*suchen* kann. Neben dem Suchen von Einträgen gibt es eine Reihe weiterer Aufgaben: Zum Beispiel müssen neue Mitarbeiter*innen zum Personalstamm hinzugefügt werden; umgekehrt wird man einen Eintrag aus dem Personalstamm entfernen, wenn die entsprechende Mitarbeiter*in das Unternehmen verlässt.

Die Zuordnung von Personaldaten zu Personalnummern ist ihrer Natur nach eine endliche Abbildung. Ihr Definitionsbereich wird durch die Menge der aktiven Personalnummern festgelegt; die eigentliche Zuordnung ist implizit durch die jeweils verwendete Datenstruktur gegeben bzw. explizit durch die Funktion *lookup*. Diese Erkenntnis legt nahe, sich von der speziellen Anwendung zu lösen und ganz allgemein endliche Abbildungen zu implementieren. Die Verwaltung von Personaldaten wird durch den Abstraktionsschritt zu einem biederen Spezialfall. Endliche Abbildungen haben vielfältige Anwendungen: Sie lassen sich verwenden, um Bücher nach ISB-Nummern[7], Studierende nach Matrikelnummern oder Telefonnummern nach Spitznamen zu verwalten.

In Abschnitt 2.1 haben wir endliche Abbildungen als Teil des Vokabulars eingeführt, mit dem wir über unsere Programmiersprache reden. Jetzt wollen wir endliche Abbildungen in Mini–F# selbst realisieren. Ähnlich zur Vorgehensweise in Abschnitt 2.1 müssen wir uns zunächst über die *Schnittstelle* (engl. interface) Gedanken machen: Welche Operationen wollen wir auf endlichen Abbildungen unterstützen? Inspiriert von unserer ursprünglichen Anwendung legen wir die folgende Schnittstelle fest.

$$\begin{aligned}
&\textbf{\textit{type}}\ \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\ \textbf{\textit{when}}\ '\mathit{key} : \mathit{comparison}\rangle\\
&\mathit{empty} &&: \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle\\
&\mathit{add} &&: '\mathit{key} * '\mathit{value} \to \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle \to \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle\\
&\mathit{remove} &&: '\mathit{key} \to \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle \to \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle\\
&\mathit{is\text{-}empty} &&: \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle \to \mathit{Bool}\\
&\mathit{lookup} &&: '\mathit{key} \to \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle \to '\mathit{value}\ \mathit{option}\\
&\mathit{from\text{-}list} &&: ('\mathit{key} * '\mathit{value})\ \mathit{list} \to \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle\\
&\mathit{to\text{-}list} &&: \mathit{Map}\,\langle'\mathit{key}, '\mathit{value}\rangle \to ('\mathit{key} * '\mathit{value})\ \mathit{list}
\end{aligned}$$

Da wir von konkreten Anwendungen abstrahieren, werden aus Personalnummern, ISBN-Angaben oder Matrikelnummern ganz allgemein Schlüssel (engl. keys); aus Personaldaten, Büchern und Studierenden werden noch allgemeiner Werte (engl. values). Eine endliche Abbildung bildet somit Schlüssel auf Werte ab. Da wir uns nicht auf konkrete Typen festlegen wollen, ist Map, der Typ der endlichen Abbildungen, sowohl mit dem Typ der Schlüssel $'\mathit{key}$ als auch mit dem Typ der Werte $'\mathit{value}$ parametrisiert. Die konkreten Typen für Schlüssel und Werte können je nach Anwendung frei gewählt werden. Mit einer kleinen Einschränkung: Wir setzen voraus, dass sich Schlüssel vergleichen lassen — diese Annahme wird durch den Zusatz $\textbf{\textit{when}}\ '\mathit{key} : \mathit{comparison}$ ausgedrückt.

[7] Die Internationale Standardbuchnummer, kurz ISBN für engl. International Standard Book Number, dient der eindeutigen Kennzeichnung von Büchern und anderen Veröffentlichungen.

Schauen wir uns die Operationen im Detail an. Die Konstante *empty* repräsentiert die leere Abbildung. Mit Hilfe von *add* fügen wir ein Schlüssel-Wert Paar zu einer Abbildung hinzu; *remove* entfernt den Eintrag mit dem angegebenen Schlüssel. Die Funktion *is-empty* überprüft, ob eine endliche Abbildung leer ist. Die Funktion *lookup* verallgemeinert die gleichnamige Operation aus Abschnitt 5.2.1: *lookup* sucht einen Eintrag mit dem angegebenen Schlüssel; ist die Suche erfolgreich, wird der Wert *Some v* zurückgegeben, wobei *v* der mit dem Schlüssel assoziierte Wert ist; schlägt die Suche fehl, wird *None* zurückgegeben. Schließlich stellen wir noch Funktionen zur Verfügung, um eine Liste von Schlüssel-Wert Paaren in eine endliche Abbildung zu konvertieren und umgekehrt. Mit der Funktion *from-list* lässt sich zum Beispiel der Personalstamm aus Abschnitt 5.2.1 einfach in eine endliche Abbildung überführen.

$$
\begin{aligned}
\textbf{\textit{let}}\ \mathit{team} = \mathit{from\text{-}list}\ [\{\ & \mathit{key} = 7; && \mathit{person} = \mathit{ralf}\ \ \ \ \ \};\\
\{\ & \mathit{key} = 815; && \mathit{person} = \mathit{melanie}\};\\
\{\ & \mathit{key} = 4711; && \mathit{person} = \mathit{julia}\ \ \ \ \ \};\\
\{\ & \mathit{key} = 4712; && \mathit{person} = \mathit{andres}\ \ \};]
\end{aligned}
$$

Alternativ können wir den Personalstamm auch peu à peu mit Hilfe der Operationen *empty* und *add* aufbauen.

$$
\begin{aligned}
\textbf{\textit{let}}\ \mathit{team} = \mathit{empty}\ &|> \mathit{add}\ (7, \ \ \mathit{ralf}\ \ \ \ )\\
&|> \mathit{add}\ (815, \ \mathit{melanie})\\
&|> \mathit{add}\ (4711, \mathit{julia}\ \ \ \ )\\
&|> \mathit{add}\ (4712, \mathit{andres}\ \ )
\end{aligned}
$$

Die Definition verwendet den vordefinierten Operator '|>' für die Postfixapplikation: $x\ |>\ f$ ist alternative Syntax für $f\ x$. Also: $\mathit{add}\ (7, \mathit{ralf})$ wird auf *empty* angewendet; auf das Ergebnis wird $\mathit{add}\ (815, \mathit{melanie})$ angewendet und so weiter.

Ist damit die Bedeutung der Operationen geklärt? Nicht ganz, welche endliche Abbildung bezeichnet zum Beispiel $\mathit{empty}\ |>\ \mathit{add}\ (\texttt{"Lisa"}, 4711)\ |>\ \mathit{add}\ (\texttt{"Lisa"}, 815)$? Da Elemente des Typs *Map* endliche Abbildungen repräsentieren, legt es nahe, die Semantik der Operationen präzise mit Hilfe der Konstrukte aus Abschnitt 2.1 zu beschreiben:

Die leere Abbildung $\varnothing$ wird durch *empty* repräsentiert. Repräsentiert *fm* die endliche Abbildung φ, dann repräsentiert $\mathit{add}\ (k, v)\ \mathit{fm}$ die Erweiterung von φ um $\{k \mapsto v\}$. Wir erinnern uns: Erweiterungen notieren wir mit dem Kommaoperator: $\varphi, \{k \mapsto v\}$. Mit anderen Worten, im Fall von Überschneidungen wird dem „neuen" Eintrag (k, v) Vorrang eingeräumt. Umgekehrt repräsentiert $\mathit{remove}\ k\ \mathit{fm}$ die Einschränkung von φ auf $\mathit{dom}\ \varphi \setminus \{k\}$, notiert $\varphi \setminus \{k\}$. Repräsentiert *fm* die endliche Abbildung φ, dann testet $\mathit{is\text{-}empty}\ \mathit{fm}$, ob $\varphi = \varnothing$. (Wann sind zwei endliche Abbildungen gleich?) Die Anwendung einer endlichen Abbildung wird durch *lookup* realisiert: $\mathit{lookup}\ k\ \mathit{fm}$ ergibt $\mathit{Some}\ (\varphi(k))$, wenn $k \in \mathit{dom}\ \varphi$ und *None* anderenfalls. Unsere Schnittstelle stellt übrigens keine Operation zur Verfügung, um den Definitionsbereich einer endlichen Abbildung

direkt zu bestimmen; diesen erhält man nur etwas indirekt mit Hilfe von *to-list*.

Syntax	Semantik
fm	φ
empty	$\varnothing$
add (k, v) *fm*	$\varphi, \{k \mapsto v\}$
remove k *fm*	$\varphi \setminus \{k\}$
is-empty fm	$\varphi = \varnothing$

Die obige Schnittstelle beschreibt, *was* wir mit Elementen des Typs *Map* machen können. Wir haben noch nicht diskutiert, *wie* wir den Typ *Map* und die zugehörigen Operationen konkret realisieren. Wir werden im Folgenden drei Implementierungen der Schnittstelle vorstellen. Bevor wir uns den Details zuwenden, lohnt es sich, die Idee einer Schnittstelle noch einmal genauer zu beleuchten.

Abstrakte Datentypen Der Typ *Map* ist ein Beispiel für einen sogenannten *abstrakten Datentyp* (ADT). Zu einem abstrakten Datentyp gehört

- eine *Schnittstelle* (engl. interface), die die verfügbaren Typen und Operationen beschreibt („was“), und
- eine oder mehrere *Implementierungen* (engl. implementations), die die Typen und Operationen realisieren („wie“).

Wohldefinierte Schnittstellen sind in der Softwareentwicklung wie im täglichen Leben nahezu unverzichtbar. Eins der Paradebeispiele für gutes Schnittstellendesign ist *SchuKo* (kurz für Schutz-Kontakt), unser System von Steckern und Steckdosen für die Stromversorgung. Über die genormte Schnittstelle lassen sich verschiedenste elektrische Geräte in weiten Teilen Europas mit Strom versorgen. Solange sich Stromverbraucher und Stromerzeuger an die Schnittstellenvereinbarung halten, spielt es keine Rolle, welches Gerät mit Strom versorgt wird (Kühlschrank, Rasenmäher oder Rechner) oder wie der Strom erzeugt wird (Dieselaggregat, Batterie oder Windkraftanlage).

In der Softwareentwicklung werden aus Stromerzeugern *Bibliotheken*, die eine Schnittstelle implementieren, und aus Stromverbrauchern *Anwendungsprogramme*, die auf die Dienste einer Bibliothek zurückgreifen. Die Vorteile einer Schnittstelle bleiben erhalten: Viele verschiedene Anwendungsprogramme können die gleiche Schnittstelle verwenden; diese kann durch unterschiedliche Bibliotheken realisiert werden, siehe Abbildung 5.15. Die Trennung in Bibliotheken und Anwendungsprogramme (die tatsächlich nicht zu eng zu verstehen ist) ist mit einem beträchtlichen Gewinn an *Modularität* verbunden: Beide können unabhängig voneinander entwickelt werden; Bibliotheken lassen sich leicht austauschen; von den Verbesserungen einer Bibliothek profitieren viele Anwendungsprogramme. Die Vorteile kommen auch zum Tragen, wenn es nur eine Anwendung und nur eine Bibliothek gibt. Es ist stets eine gute Idee, Implementierungsdetails hinter einer Schnittstelle zu verbergen.

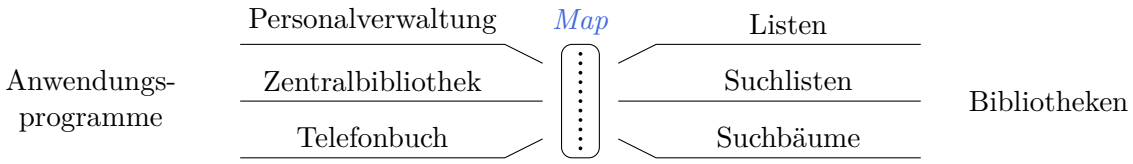


Abbildung 5.15.: Anwendungsprogramme und Bibliotheken „kommunizieren“ über eine genormte Schnittstelle.

In den letzten Kapiteln haben wir sowohl konkrete als auch abstrakte Datentypen kennengelernt. Beide „Arten“ von Typen verfolgen grundverschiedene Ansätze:

- ein konkreter Datentyp wird durch seine Elemente definiert;
- ein abstrakter Datentyp wird durch seine Operationen definiert.

Ein konkreter Datentyp ist die Summe seiner Elemente. Record- und Variantentypen führen konkrete Typen ein. Die jeweilige Typdefinition beschreibt präzise, welche Elemente der definierte Typ enthält.

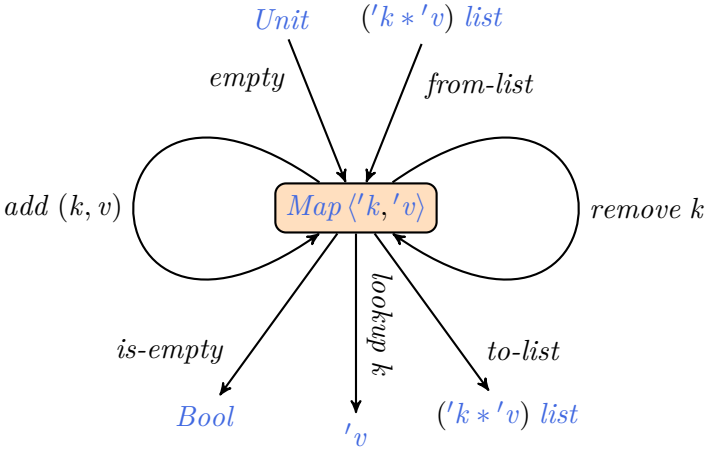
Ein abstrakter Datentyp ist die Summe seines Verhaltens. Die Elemente des ADTs werden nicht offenbart — sie sind abstrakt. Die ersten Typen, die wir eingeführt haben, *Bool* und *Nat*, sind der Natur nach abstrakte Datentypen. Wir haben Operationen festgelegt und präzise beschrieben, mit deren Hilfe wir Wahrheitswerte und natürliche Zahlen konstruieren und analysieren können. Wie diese tatsächlich implementiert werden, verschließt sich uns — die Details sind hinter der jeweiligen Schnittstelle verborgen.

Ein abstrakter Datentyp wird in der Regel durch einen konkreten Datentyp realisiert: Wahrheitswerte können durch einen binären Variantentyp implementiert werden, siehe Abschnitt 4.2.1; für natürliche Zahlen haben wir zwei mögliche Implementierungen diskutiert, eine die auf der unären Zahlendarstellung (*Peano*) und eine die auf der binären Zahlendarstellung (*Leibniz*) basiert, siehe Abschnitt 4.2.2.

Schnittstellen nehmen eine sehr zentrale Rolle bei der Entwicklung von Software ein. Aus diesem Grund sollte ihr Design wohlüberlegt sein. Nachträgliche Änderungen einer Schnittstelle oder unterschiedliche, inkompatible Schnittstellen (andere Steckersysteme für die Stromversorgung im Ausland) sind im besten Fall nur ärgerlich. In der Regel kosten die daraus resultierenden Anpassungen von Anwendungsprogrammen und Bibliotheken Zeit und Geld. Es empfiehlt sich, zumindest eine einfache *Plausibilitätsprüfung* durchzuführen: Die Schnittstelle eines abstrakten Datentyps sollte Operationen bereitstellen

- um Elemente des abstrakten Datentyps zu *konstruieren*;
- um Elemente des abstrakten Datentyps in andere Elemente zu *transformieren*;
- um Elemente des abstrakten Datentyps zu *analysieren*.

Alle drei Aspekte finden sich in der Schnittstelle für endliche Abbildungen wieder. Die Operationen *empty* und *from-list* konstruieren; *add* und *remove* transformieren; und *is-empty*, *lookup* und *to-list* analysieren.



Genug der Vorüberlegungen — wenden wir uns der Implementierung der Schnittstelle zu. In den Abschnitten 5.2.1–5.2.3 haben wir uns mögliche Organisationsformen für endliche Abbildungen angeschaut: Listen, Suchlisten und Suchbäume. Diese greifen wir in den folgenden Abschnitten wieder auf und erweitern sie zu Implementierungen des abstrakten Datentyps *Map*.

module
Algorithmics.
Map.
List

5.3.1. Listen

Die einfachste Darstellung von endlichen Abbildungen ist eine ungeordnete Liste von sogenannten *Schlüssel-Wert Paaren* (engl. key-value pairs).

```
type Entry ⟨'key, 'value⟩ =
  {key : 'key; value : 'value}
type Map ⟨'key, 'value when 'key : comparison⟩ =
  | Rep of List ⟨Entry ⟨'key, 'value⟩⟩
```

Der Recordtyp *Entry* verallgemeinert den gleichnamigen Typ aus Abschnitt 5.2.1: Das anwendungsspezifische Label *person* haben wir in *value* umbenannt; der Typ ist jetzt sowohl mit dem Schlüsseltyp als auch mit dem Typ der assoziierten Werte parametrisiert.

Die zweite Typdefinition gibt an, wie der abstrakte Typ *Map* konkret implementiert wird. Wir verwenden einen 1-Variantentyp, um den abstrakten Typ und den konkreten Typ einfach auseinander halten zu können (siehe Abschnitt 4.2.1). Der einzige Konstruktor des Typs, *Rep* (engl. für representation), überführt die konkrete Repräsentation, eine ungeordnete Liste von Schlüssel-Wert Paaren, in ein Element des abstrakten Typs, eine endliche Abbildung. *Kurz: list* ist konkret, *Rep list* ist abstrakt. Die endliche Abbildung $\{Anja \mapsto Spagetti, Ralf \mapsto Pizza\}$ aus Abschnitt 2.1 wird zum Beispiel durch [{*key* = `"Anja"`; *value* = `"Spaghetti"`}; {*key* = `"Ralf"`; *value* = `"Pizza"`}] repräsentiert — wenn wir Personen und Gerichte durch Zeichenketten darstellen.

Die leere Abbildung wird entsprechend durch die leere Liste implementiert; das Hinzufügen durch '::' — wir müssen lediglich das Paar in ein Record verwandeln.

```
let empty = Rep []
let is-empty (Rep list) = List.is-empty list
let add (key, value) (Rep list) = Rep ({key = key; value = value} :: list)
let lookup key (Rep list) =
  let rec find = function
  | []                -> None
  | entry :: entries -> if key = entry.key then Some entry.value
                                           else  find entries
  in find list
```

Viele listenverarbeitende Funktionen wie zum Beispiel *is-empty* sind in dem vordefinierten Modul *List* zusammengefasst; mit Hilfe der Punktnotation können die Bibliotheksfunktionen aufgerufen werden. Ein weiteres technisches Detail: Das Muster *Rep list* ist unwiderlegbar; aus diesem Grund können wir es ohne Bedenken als formalen Parameter verwenden.

Die Funktion *lookup*, die wir schon aus Abschnitt 5.2.1 kennen, gibt den Wert des *ersten* Eintrags mit dem passenden Schlüssel zurück. Da *add* vorne, nicht hinten einfügt, werden im Fall von Überschneidungen auf diese Weise den „neueren" Einträgen Vorrang eingeräumt: *lookup* 4711 (*add* (4711, `"yes"`) (*add* (4711, `"no"`) *empty*)) ist *Some* `"yes"`, nicht *Some* `"no"`. Mit anderen Worten, der erste Eintrag (4711, `"no"`) wird durch den zweiten Eintrag (4711, `"yes"`) verschattet. Semantisch gesehen entspricht *add* (*key*, *value*) *fm* der Anwendung des Kommaoperators: $\varphi, \{key \mapsto value\}$. So soll es sein, so haben wir es bei der Beschreibung der Schnittstelle festgelegt — jede Implementierung muss diese Vorgaben buchstabengetreu umsetzen.

Beim Entfernen von Einträgen müssen wir aufpassen: Da mehrere Einträge mit dem gleichen Schlüssel auftreten können, müssen wir stets die gesamte Liste durchlaufen. Das ist sozusagen der Preis für die einfache Implementierung von *add*.

```
let remove key (Rep list) =
  let rec del = function
  | []                -> []
  | entry :: entries -> if key = entry.key then del entries
                                           else  entry :: del entries
  in Rep (del list)
```

Wie im Fall von *lookup*, delegiert die Funktion *remove* die eigentliche Arbeit an eine „Arbeiterfunktion" (engl. worker function): *del* arbeitet auf den konkreten Werten, auf der Repräsentation einer endlichen Abbildung; *remove* arbeitet auf den abstrakten Werten.

Die Operationen *from-list* und *to-list* müssen Paare in Einträge verwandeln und umgekehrt.

```
let from-list list =
  Rep (List.map (fun (key, value) → {key = key; value = value}) list)
let to-list (Rep list) =
  List.map (fun entry → (entry.key, entry.value)) list
```

Die vordefinierte Funktion *map* wendet die angegebene Funktion, erstes Argument, auf jedes Element der Liste, zweites Argument, an: *map f* bildet $[x_1; x_2; \ldots; x_n]$ auf $[f\ x_1; f\ x_2; \ldots; f\ x_n]$ ab.

module
Algorithmics.
Map.
OrdList

5.3.2. Suchlisten

Alternativ können wir eine endliche Abbildung durch eine Liste von Schlüssel-Wert Paaren darstellen, die nach dem Schlüssel geordnet ist und die keine zwei Einträge mit dem gleichen Schlüssel enthält. (Die eindeutige Identifizierbarkeit definiert in der Theorie der relationalen Datenbanken gerade einen „Schlüssel“.)

```
type Map ⟨'key, 'value when 'key : comparison⟩ =    // geordnet, ohne Duplikate
  | Rep of List ⟨Entry ⟨'key, 'value⟩⟩
```

Die Typdefinition ändert sich nicht, da wir die zusätzlichen Eigenschaften leider nicht in Mini–F# ausdrücken können. Die geforderten Eigenschaften werden zu einer *Invariante* des Typs: Alle Funktionen, die Argumente vom Typ $Map\ \langle 'k, 'v\rangle$ entgegennehmen, dürfen annehmen, dass die Invariante gilt; alle Funktionen, die Elemente vom Typ $Map\ \langle 'k, 'v\rangle$ zurückgeben, müssen garantieren, dass die Invariante erfüllt ist — daher der Begriff „Invariante“.

Die leere Abbildung wird weiterhin durch die leere Liste implementiert; das Hinzufügen ist jetzt ein Einfügen, wie beim „Sortieren durch Einfügen“.

```
let empty = Rep []
let is-empty (Rep list) = List.is-empty list
let add (key, value) (Rep list) =
  let new-entry = {key = key; value = value}
  let rec ins = function
  | []                → [new-entry]
  | entry :: entries → if   key < entry.key   then new-entry :: entry :: entries
                       elif key = entry.key   then new-entry :: entries
                       (* key > entry.key *) else  entry :: ins entries
  in Rep (ins list)
let lookup key (Rep list) =
  let rec find = function
  | []                → None
  | entry :: entries → if key < entry.key     then None
                       elif key = entry.key   then Some entry.value
                       (* key > entry.key *) else  find entries
  in find list
```

Der 2-Wege Vergleich aus dem letzten Abschnitt ist jeweils einem 3-Wege Vergleich gewichen. Letzter ist typisch für die Verarbeitung von sortierten Listen *ohne* Duplikate: Beim Einfügen müssen wir Duplikate eliminieren (der neue Eintrag ersetzt den alten); beim Suchen können wir frühzeitig Misserfolg signalisieren. Das Löschen von Einträgen verwendet das exakt gleiche Rekursionsmuster.

```
let remove key (Rep list) =
  let rec del = function
  | []                  → []
  | entry :: entries → if   key < entry.key   then entry :: entries
                       elif key = entry.key   then entries
                       (* key > entry.key *) else  entry :: del entries
  in Rep (del list)
```

Die Konvertierungsfunktion *from-list* hat jetzt einen Haufen Arbeit: Sie muss sicherstellen, dass die Liste nach dem Schlüssel geordnet ist und keine Duplikate enthält.

```
let from-list list =
  Rep (List.map (fun (key, value) → {key = key; value = value})
        (List.distinct-by fst
          (List.sort-by fst list)))
let to-list (Rep list) =
  List.map (fun entry → (entry.key, entry.value)) list
```

Die Liste von Schlüssel-Wert Paaren wird zunächst nach dem Schlüssel sortiert, dann werden Duplikate entfernt, und schließlich werden die Paare in Einträge überführt. Die Bibliotheksfunktionen *sort-by* und *distinct-by* erwarten keine Quasiordnung als Argument, sondern eine „Projektionsfunktion“. Die zugrundeliegende Quasiordnung ‘$\preceq$’ wird von der Projektionsfunktion f abgeleitet: $x \preceq y \iff f\ x \leqslant f\ y$.

Eine stilistische Anmerkung: Die Definition von *from-list* verwendet auf der rechten Seite vierfach geschachtelte Funktionsaufrufe. Wenn man die resultierenden Klammergebirge vermeiden möchte, kann man alternativ den Operator ‘|>’ für die Postfixapplikation verwenden. Zur Erinnerung: Statt $f\ x$ schreibt man $x |> f$; das Argument wird vor die Funktion gesetzt. Der Vorteil erschließt sich, wenn man Applikationen schachtelt: Statt $h\ (g\ (f\ x))$ formuliert man $x |> f |> g |> h$. Im Fall von *from-list* erhalten wir:

```
let from-list list =
  list |> List.sort-by      fst
       |> List.distinct-by fst
       |> List.map          (fun (key, value) → {key = key; value = value})
       |> Rep
```

Wir haben eine Verarbeitungspipeline vor uns: *list* wird in die Pipeline gepumpt und dann in vier Schritten zu einer endlichen Abbildung verarbeitet.

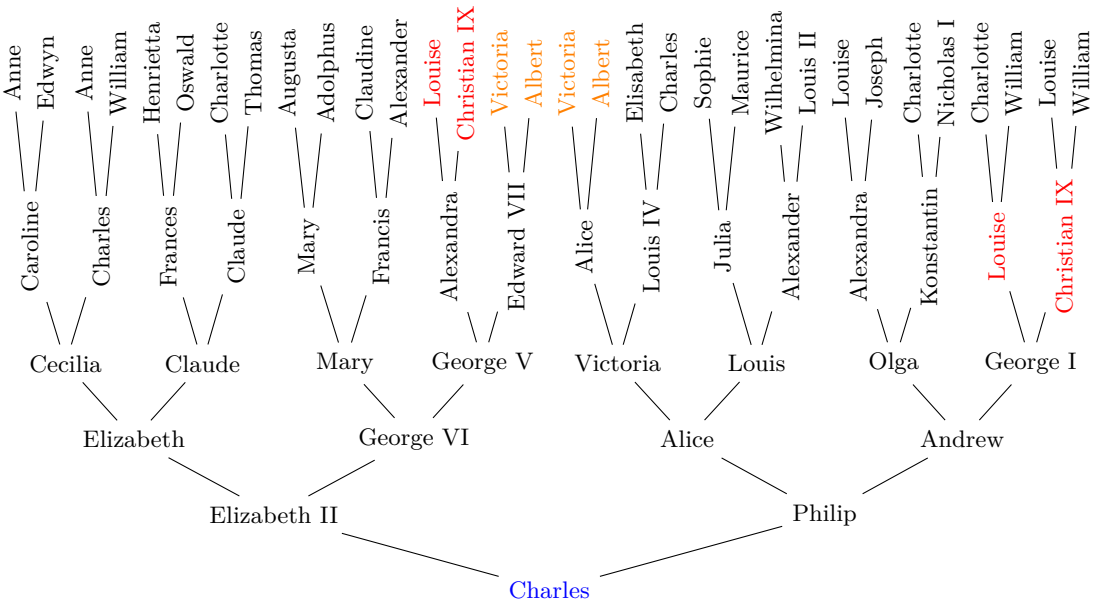


Abbildung 5.16.: Stammbaum von Charles Philip Arthur George, Prince of Wales (Quelle [Knu97, Fig.18(a)]).

Fassen wir zusammen: Die Konstruktoren eines abstrakten Datentyps müssen die Invariante etablieren; die Transformationen müsssen die Invariante erhalten; die Nutznießerinnen sind die Funktionen, die Elemente des Datentyps analysieren. Im Zusammenspiel dieser drei Parteien entscheidet sich, ob eine Invariante gewinnbringend ist. In unserem Fall ist der Gewinn eher bescheiden: Die (erfolglose) Suche wird beschleunigt, das Einfügen hingegen entschleunigt.

module
Algorithmics.
Tree

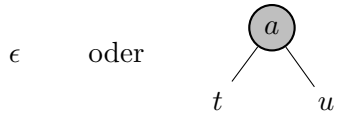
module
Algorithmics.
Map.
SearchTree

5.3.3. Binäre Suchbäume

Neben Listen und Suchlisten haben wir uns Suchbäume als Organisationsform für endliche Abbildungen angeschaut. Bäume im Allgemeinen sind uns in der Vorlesung schon wiederholt begegnet: als abstrakte Syntaxbäume, Beweisbäume, Entscheidungsbäume und Rekursionsbäume. Binäre Bäume lassen sich als fleischgewordene Rekursionsbäume deuten, genauer als die Rekursionsbäume der binären Suche. (Die Rekursionsbäume der linearen Suche sind entsprechend lineare Listen.) Eine Kontrollstruktur wird sozusagen in eine Datenstruktur verwandelt. Neben der naheliegenden Verwendung als Suchstruktur haben Binärbäume noch viele weitere Anwendungen — man verwendet sie zum Beispiel als Stammbäume (siehe Abbildung 5.16), Kodierungsbäume oder Turnierbäume —, so dass wir uns die Datenstruktur noch einmal in Ruhe anschauen wollen.

Binärbäume Binärbäume sind induktiv definiert: Ein *binärer Baum* ist entweder ein leeres Blatt oder ein Knoten, der aus einem linken Baum, einem Element und einem

rechten Baum besteht. Wenn wir Binärbäume zeichnen, verwenden wir den griechischen Buchstaben ϵ, unser Symbol für die leere Sequenz, für Blätter und Kreise für Knoten.



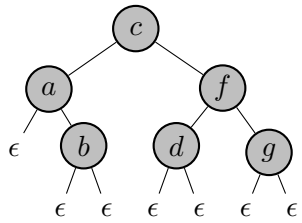
Das Element wird dabei üblicherweise in den Kreis geschrieben — deswegen spricht man auch von einer Knotenmarkierung. In der grafischen Darstellung lässt man die Blätter aus Gründen der Lesbarkeit oft auch weg.

Die induktive Definition entspricht exakt der rekursiven Variantentypdefinition, die wir bereits eingeführt haben.

```
type Tree⟨'elem⟩ =
    | Leaf
    | Node of Tree⟨'elem⟩ * 'elem * Tree⟨'elem⟩
```

Wie Listen sind auch Binärbäume Container: Sie enthalten Elemente. Je nach Typparameter haben wir Bäume von natürlichen Zahlen, $Tree\,\langle Nat\rangle$, Bäume von Zeichenketten, $Tree\,\langle String\rangle$, oder hochkomplexe Gebilde vor uns, etwa $Tree\,\langle List\,\langle Tree\,\langle String\rangle\rangle\rangle$.

Der Binärbaum auf der linken Seite wird durch den Ausdruck bzw. Wert auf der rechten Seite repräsentiert.




```
Node (Node (Leaf,
                "a",
                Node (Leaf, "b", Leaf)),
      "c",
      Node (Node (Leaf, "d", Leaf),
                "f",
                Node (Leaf, "g", Leaf)))
```

Die hierarchische Baumstruktur wird in der linearen Notation etwas unvollkommen durch Einrückung angedeutet. (Das kennen wir von Mini–F# Programmen selbst: Die hierarchische Struktur der abstrakten Syntax wird in der konkreten Syntax idealerweise durch Einrückung reflektiert.) In der Informatik wachsen Bäume typischerweise von oben nach unten, das heißt Bäume werden mit der Wurzel nach oben gezeichnet. Eine Ausnahme haben wir schon kennengelernt: Beweisbäume. Auch Stammbäume werden biologisch korrekt gemalt, siehe Abbildung 5.16.

Nicht zuletzt auf Grund ihrer Popularität gibt es eine Vielzahl von Begriffen um über Binärbäume zu reden. Nehmen wir den obigen Baum als Beispiel:

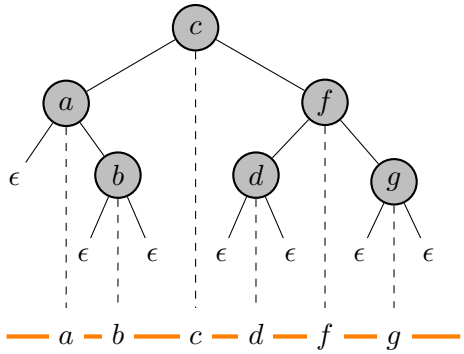
- Der Knoten c ist die *Wurzel* des Baums; die leeren Teilbäume sind die *Blätter*.
- Alle Knoten, mit Ausnahme der Wurzel, haben einen *Vorgänger*[8]: Der Vorgänger von d ist f; der Vorgänger von f wiederum ist die Wurzel c.

[8] In älteren Informatiktexten wird auch der Begriff Vaterknoten verwendet.

- Knoten können *Nachfolger* oder *Kinder* haben: f hat die Kinder d und g; der Knoten d hat keine Kinder; a hat ein Kind.
- Die Knoten a und f sind *Geschwister*; d und g sind ebenfalls Geschwister; b hat keine Geschwister.

Man sieht, die Terminologie ist eine krude Mischung aus Verwandtschaftsbeziehungen (Kind, Geschwister), biologischen Begriffen (Wurzel, Blatt) und Begriffen aus der Graphentheorie (Knoten, Vorgänger, Nachfolger). Leider ist die Terminologie nicht einheitlich: Zum Beispiel werden Knoten, die keine Kinder haben (in unserem laufenden Beispiel b, d und g) auch oft als Blätter bezeichnet. (Gelegentlich sind die Bezeichnungen auch paradox oder irreführend: In Abbildung 5.16 hat der Wurzelknoten „Charles“ zwei Kinder: „Elizabeth II“ und „Philip“.)

Binäre Suchbäume Sind die Elemente eines Binärbaums von links nach rechts geordnet, spricht man von einem *binären Suchbaum*. Unser laufendes Beispiel ist ein solcher binärer Suchbaum:




```
Node (Node (Leaf,
            "a",
            Node (Leaf, "b", Leaf)),
      "c",
      Node (Node (Leaf, "d", Leaf),
            "f",
            Node (Leaf, "g", Leaf)))
```

In der linearen Notation lässt sich die Suchbaumeigenschaft besonders einfach überprüfen. In der grafischen Darstellung projiziert man die Elemente auf eine horizontale Linie. Im Fall eines Suchbaums erhalten wir eine aufsteigend geordnete Sequenz.[9]

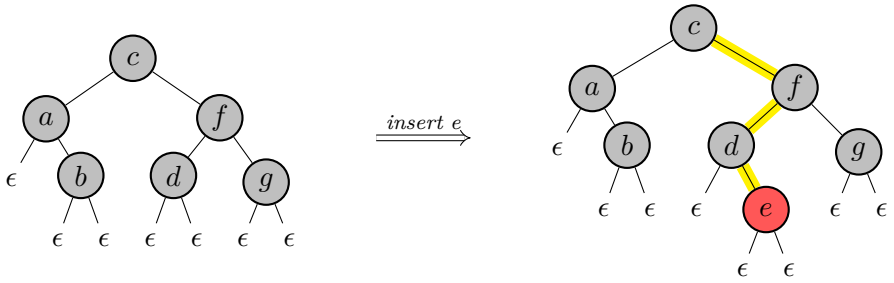
Die Suchbaumeigenschaft lässt sich alternativ auch direkt an Hand der Struktur eines Binärbaums festmachen. Für alle Knoten muss gelten, dass die Elemente im linken Teilbaum kleiner sind als das Element in der Wurzel und, in symmetrischer Weise, dass die Elemente im rechten Teilbaum größer sind als das Element in der Wurzel. (Je nach Anwendung meint „kleiner“ tatächlich entweder ‘$<$’ oder ‘$\leqslant$’.) Auf diese Weise dient das Wurzelelement als *Wegweiser* bei der Suche.

```
// contains : 'key → Tree ('key) → Bool when 'key : comparison
let contains key = function
  | Leaf          → false
  | Node (l, x, r) → if   key < x  then contains key l
                     elif key = x  then true
                     (* key > x *) else  contains key r
```

[9] Ob das tatsächlich klappt, hängt von der Zeichnung ab: Schieben wir in unserem Beispiel b ganz nah an d heran, so wechseln b und c möglicherweise die Positionen.

Die Funktion *contains* überprüft, ob ein Element in einem Binärbaum enthalten ist. Sie ist die Mutter aller Algorithmen auf Suchbäumen. Fast alle anderen Algorithmen orientieren sich an ihrem Rekursionsmuster.

Wie Listen können wir auch Bäume ohne allzu großen Aufwand wachsen oder schrumpfen lassen. So wie wir ein Element in eine Suchliste einfügen, so können wir ein Element in einen Suchbaum einfügen. Dabei folgen wir dem Suchpfad bis zu einem Blatt und ersetzen dieses durch einen Knoten. Wenn wir zum Beispiel e in unseren Binärbaum einfügen, gehen wir erst nach rechts, dann nach links und schließlich wieder nach rechts.

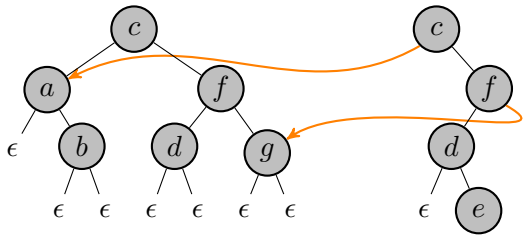


Die algorithmische Idee lässt sich direkt in eine Rechenregel überführen.

```
// insert : 'key → Tree ⟨'key⟩ → Tree ⟨'key⟩ when 'key : comparison
let rec insert key = function
  | Leaf           → Node (Leaf, key, Leaf)
  | Node (l, x, r) → if   key < x  then Node (insert key l, x, r)
                     elif key = x  then Node (l, key, r)
                     (* key > x *) else  Node (l, x, insert key r)
```

Ein Blatt wird auf einen Minibaum abgebildet, einen Knoten mit leeren Teilbäumen und dem einzufügenden Element als Markierung. Wenn der Schlüssel kleiner ist als der Wegweiser, wird rekursiv in den linken Teilbaum eingefügt. Genauer: Es wird ein Knoten zurückgegeben, der den gleichen rechten Teilbaum und die gleiche Markierung enthält und dessen linker Teilbaum das Ergebnis des rekursiven Aufrufs ist.

Für jeden Knoten entlang des Suchpfades wird somit ein neuer Knoten angelegt. Die Teilbäume, die nicht traversiert werden, sind sowohl Teilbäume im ursprünglichen Baum (der Eingabe) als auch Teilbäume des erweiterten Baums (der Ausgabe). Im Englischen spricht man auch von „sharing“. Es ergibt sich das folgende, interessante Bild:



Das „Teilen“ von Datenstrukturen ist bedeutsam, wenn man genauer untersucht, wieviel Speicherplatz für einen Wald benötigt wird. Der Bedarf hängt nicht nur von der

Größe der einzelnen Bäume ab, sondern auch vom Grad des „Teilens“. So lässt sich unter Umständen ein Baum von exponentieller Größe in linearem Platz unterbringen, siehe Aufgabe 5.8.

Implementierung der Schnittstelle Wenden wir uns nach diesen Vorüberlegungen der Implementierungsarbeit zu. Als Repräsentation wählen wir jetzt Bäume von Einträgen.

```
type Map ⟨'key, 'value when 'key : comparison⟩ =
  | Rep of Tree ⟨Entry ⟨'key, 'value⟩⟩
let empty = Rep Leaf
let is-empty (Rep tree) =
  match tree with
  | Leaf            → true
  | Node (_, _, _) → false
```

Die Funktion *lookup* leitet sich mehr oder weniger direkt von der Funktion *contains* ab, der Mutter aller Algorithmen auf Suchbäumen.

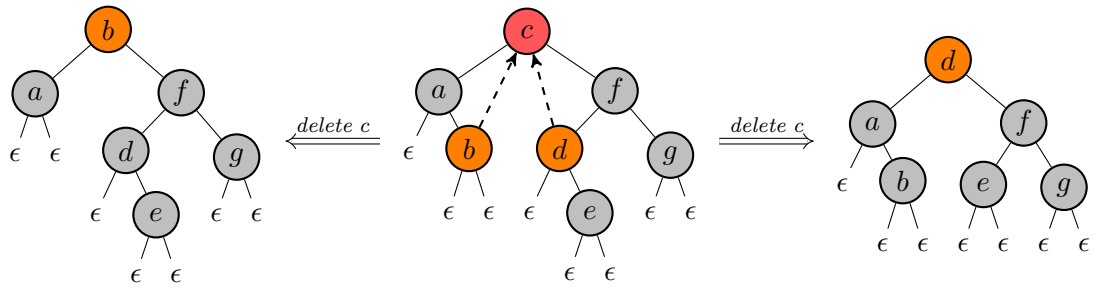
```
let lookup key (Rep tree) =
  let rec find = function
  | Leaf                      → None
  | Node (left, entry, right) → if    key < entry.key then find left
                                elif key = entry.key then Some entry.value
                                (*key > entry.key*)  else  find right
  in find tree
```

An die Stelle des Ergebnistyps *Bool* ist der Typ *Option* getreten. Wenn man möchte, kann man optionale Werte als konstruktive Varianten der Wahrheitswerte ansehen. Anstatt die Frage „Kennen Sie den Weg zum Audimax?“ etwas schroff mit „Ja!“ oder „Nein!“ zu beantworten, sagt man im positiven Fall „Ja! Betreten Sie das Gebäude durch den Haupteingang, dann halten Sie sich links …“. An die Stelle des Wahrheitswerts *false* tritt *None*, an die Stelle von *true* tritt *Some v*, wobei *v* das erfragte Objekt ist, sozusagen der „Beweis“ für die positive Antwort.

Die Schnittstellenfunktion *add* delegiert die Arbeit an eine Arbeitsfunktion, die das eigentliche Einfügen übernimmt. Wie immer achten wir darauf, dass keine zwei Einträge mit dem gleichen Schlüssel vorkommen. Im Fall von Überschneidungen wird wie gefordert dem neuen Eintrag Vorrang eingeräumt.

```
let add (key, value) (Rep tree) =
  let new-entry = {key = key; value = value}
  let rec ins = function
  | Leaf                      → Node (Leaf, new-entry, Leaf)
  | Node (left, entry, right) → if    key < entry.key then Node (ins left, entry, right)
                                elif key = entry.key then Node (left, new-entry, right)
                                (*key > entry.key*)  else  Node (left, entry, ins right)
  in Rep (ins tree)
```

Wenden wir uns der nächsten Aufgabe zu, dem Entfernen eines Elements. Entfernen ist im Allgemeinen aufwändiger als Einfügen. Dies ist der Tatsache geschuldet, dass das zu entfernende Element möglicherweise mitten im Baum liegt. (Beim Einfügen operieren wir immer an den Rändern.) Zwei Fälle sind zu unterscheiden: Wenn der Knoten, der das Element enthält, einen leeren Teilbaum besitzt, dann können wir den Knoten einfach durch den anderen Teilbaum ersetzen. Also, *Node* $(Leaf, x, r)$ wird zu r und *Node* $(l, x, Leaf)$ entsprechend zu l, wobei x das zu löschende Element ist. Schwieriger wird es, wenn keiner der Teilbäume leer ist. Dieser Fall liegt zum Beispiel vor, wenn wir das Element c in unserem laufenden Beispiel entfernen.



Eine Idee ist, das zu löschende Element durch ein anderes zu ersetzen. Da wir die Suchbaumeigenschaft nicht kompromittieren dürfen, gibt es genau zwei Wahlmöglichkeiten: das größte Element im linken Teilbaum *oder* das kleinste Element im rechten Teilbaum. (Wenn wir die Elemente auf eine horizontale Linie projizieren, sind dies die Elemente direkt vor bzw. direkt hinter dem zu löschenden Element.) Wir entscheiden uns willkürlich für die erste Variante.

Die Funktion *split-max* bestimmt das größte Element in dem angegeben Binärbaum und gibt zusätzlich den Baum ohne dieses Element zurück. Wie *split-min* vom „Sortieren durch Auswählen“, returniert *split-max* ein optionales Paar. Wenn der Baum leer ist, das heißt ein Blatt vorliegt, dann existiert kein Maximum.

```
// split-max : Tree ⟨'elem⟩ → (Tree ⟨'elem⟩ * 'elem) option
let rec split-max = function
  | Leaf          → None
  | Node (l, a, r) → Some (match split-max r with
                             | None          → (l, a) // einfacher Fall: r ist leer
                             | Some (r', max) → (Node (l, a, r'), max))
// join : Tree ⟨'elem⟩ → Tree ⟨'elem⟩ → Tree ⟨'elem⟩
let join l r =
  match split-max l with
  | None          → r // einfacher Fall: l ist leer
  | Some (l', max) → Node (l', max, r)
```

Die Funktion *join* konkatentiert zwei Binärbäume; was *append* bzw. ‘@’ für Listen ist, ist *join* für Binärbäume. Keine der beiden Funktionen setzt übrigens voraus, dass die

Elemente geordnet sind. Die Funktionen arbeiten problemlos mit Nicht-Suchbäumen; *split-max* bestimmt in diesem Fall das letzte, nicht das größte Element.

Nach diesen Vorarbeiten geht die Implementierung von *remove* leicht von der Hand.

```
let remove key (Rep tree) =
  let rec del = function
  | Leaf                      → Leaf
  | Node (left, entry, right) → if   key < entry.key then Node (del left, entry, right)
                                elif key = entry.key then join left right
                                (*key > entry.key*)  else  Node (left, entry, del right)
  in Rep (del tree)
```

Wir entfernen den zu löschenden Eintrag und konkatenieren die beiden Teilbäume.

Die Konvertierungsfunktionen sind aufwändiger denn je. Die Funktion *from-list* überführt die Liste von Schlüssel-Wert Paaren zunächst in eine Suchliste, die von der noch zu definierenden Funktion *balanced-tree* in einen Suchbaum überführt wird. Umgekehrt überführt *inorder* einen Suchbaum in eine Suchliste, die sodann in eine Liste von Schlüssel-Wert Paaren überführt wird.

```
let from-list list =
  list |> List.sort-by      fst
       |> List.distinct-by fst
       |> List.map          (fun (key, value) → {key = key; value = value})
       |> balanced-tree
       |> Rep
let to-list (Rep tree) =
  tree |> inorder
       |> List.map (fun entry → (entry.key, entry.value))
```

Mit der Implementierung der Funktionen *balanced-tree* und *inorder* beschäftigen wir uns im nächsten Abschnitt.

Kommen wir zur Analyse der Laufzeit. Wir haben in Abschnitt 5.2.3 gesehen, dass die Laufzeit von *lookup* proportional zur Höhe des Binärbaums ist. Da *add* und *remove* dem Rekursionsmuster von *lookup* folgen, erben sie auch ihr Laufzeitverhalten. (Was lässt sich über die Laufzeit von *split-min* und *join* aussagen?) Das ist eine gute und eine schlechte Nachricht.

Die Klient*in einer Bibliothek interessiert sich zunächst einmal dafür, wie die Laufzeit von der Gesamtzahl der Einträge abhängt, also von der *Größe* der jeweiligen Binärbäume, nicht von der Höhe. (Die Höhe ist eine spezifische Eigenschaft von Suchbäumen, keine, die für alle denkbaren Implementierungen von endlichen Abbildungen Sinn ergibt.)

Die gute Nachricht ist, dass sich die Höhe im besten Fall logarithmisch zur Größe verhält: In einem Binärbaum der Höhe h lassen sich $2^h - 1$ Einträge unterbringen.

Die schlechte Nachricht ist, dass die Höhe im schlechtesten Fall linear zur Größe ist. Ist einer der Teilbäume jeweils leer, dann degeneriert der Suchbaum zu einer Suchliste mit den entsprechenden Konsequenzen für die Laufzeit.

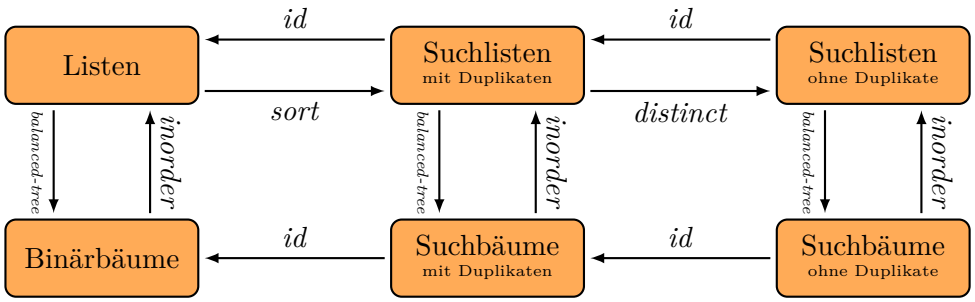


Abbildung 5.17.: Wechsel der Repräsentation.

Die wirklich schlechte Nachricht ist, dass der schlechteste Fall nicht selten auftritt. Mit den Operationen der Schnittstelle lässt sich leicht ein degenerierter Baum konstruieren, zum Beispiel indem nacheinander Einträge mit aufsteigendem Schlüssel eingefügt werden:

$$\mathit{empty} \mid\!> \mathit{add}\ (0, \texttt{"n"}) \mid\!> \mathit{add}\ (1, \texttt{"e"}) \mid\!> \mathit{add}\ (2, \texttt{"z"}) \mid\!> \mathit{add}\ (3, \texttt{"d"}) \mid\!> \mathit{add}\ (4, \texttt{"v"})$$

Wie lässt sich dieses Problem beheben? Man kann beim Einfügen und beim Löschen Sorge tragen, dass die Binärbäume nicht zu sehr in Schieflage geraten. Mit entsprechenden Balancierungsschemata beschäftigt sich die Vorlesung „Algorithmen und Datenstrukturen“. Ein kleiner Lichtblick: Verwendet die Anwendung *from-list* für die initiale Konstruktion einer endlichen Abbildung und anschließend *add* und *remove* nur sehr sporadisch, dann sind Suchbäume tatsächlich sehr viel effizienter als Suchlisten.

5.3.4. Wechsel der Repräsentation

Endliche Abbildungen lassen sich auf vielfältige Weisen repräsentieren: unter anderem mit Hilfe von Listen, Suchlisten oder Suchbäumen. Wechselt man im laufenden Betrieb die Repräsentation, müssen die Daten entsprechend konvertiert werden, siehe Abbildung 5.17. Die Überführung von Listen in Binärbäume und umgekehrt ist tatsächlich ein algorithmisches Problem von allgemeinem Interesse, da, wie bereits angedeutet, Binärbäume viele weitere Anwendungen haben.

Linearisierung von Binärbäumen Wie können wir einen Baum in eine Liste überführen? Das Struktur Entwurfsmuster für *Tree* führt fast direkt zum Ziel.

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{inorder}\ (\mathit{tree} : \mathit{Tree}\,\langle'\mathit{elem}\rangle) : \mathit{List}\,\langle'\mathit{elem}\rangle = \\ \quad \textbf{\textit{match}}\ \mathit{tree}\ \textbf{\textit{with}} \\ \quad \mid \mathit{Leaf} \qquad\qquad\quad \to \ldots \\ \quad \mid \mathit{Node}\ (\mathit{left}, x, \mathit{right}) \to \ldots\ \mathit{inorder}\ \mathit{left} \ldots \mathit{inorder}\ \mathit{right} \ldots\end{array}$$

Im Rekursionsfall müssen wir zwei Listen, die Ergebnisse der rekursiven Aufrufe, und ein Element zu einer Liste zusammenfügen. Das Konkatenieren zweier Listen besorgt *append* bzw. ‘@’, so dass wir formulieren können.

```
let rec inorder (tree : Tree ⟨'elem⟩) : List ⟨'elem⟩ =
  match tree with
  | Leaf                → []
  | Node (left, x, right) → inorder left @ [x] @ inorder right
```

Die relative Reihenfolge der Elemente — erst die Elemente aus dem linken Teilbaum, dann das Wurzelelement, dann die Elemente aus dem rechten Teilbaum — bleibt bei der Transformation erhalten. Auf diese Weise wird ein Suchbaum in eine Suchliste überführt. Weil das Wurzelelement *zwischen* die Elemente des linken und des rechten Teilbaums wandert, heißt die Funktion entsprechend *inorder*. Zwei weitere Alternativen sind geläufig: das Wurzelelement voranzustellen (*preorder*) oder hinten anzuhängen (*postorder*), siehe Aufgabe 5.5.

Wie schnell ist *inorder*? Das hängt wie so oft von der Form des Binärbaums ab. Zunächst müssen wir kurz überlegen, wieviele Schritte *append* benötigt, um zwei Listen zu konkatenieren. Rufen wir uns die Definition von *append* ins Gedächtnis.

```
let rec append list y =
  match list with
  | []      → y
  | x :: xs → x :: append xs y
```

Die Funktion rekurriert über das erste Argument, das zweite Argument wird gar nicht inspiziert. Die Laufzeit von *append* ist also proportional zur Länge der ersten Liste.

Also, wenn der linke Teilbaum immer leer ist, dann ist *append* jedesmal in einem Schritt fertig. Insgesamt erhalten wir für *inorder* eine lineare Laufzeit. Das ist optimal; schneller geht es nicht, da wir jedes Element der Ergebnisliste einmal anfassen müssen. Umgekehrt, wenn der rechte Teilbaum immer leer ist, dann muss *append* ackern: Nacheinander werden Listen der Längen 1, 2, …, $n-2$, $n-1$ durchlaufen. Insgesamt erhalten wir eine Laufzeit von $1+2+\cdots+n-2+n-1 = \binom{n}{2} = n\cdot(n-1)/2$ Schritten — grob gesprochen eine *quadratische* Laufzeit.

Das ist vielleicht unerwartet, auf jeden Fall ist es unbefriedigend. Um etwa eine Liste mit zehntausend Elementen zu produzieren — Bäume dieser Größenordnung sind nicht ungewöhnlich —, werden hundertmillionen Schritte benötigt. Den Aufwand kann man sehen — sogar hören, falls der Rechner luftgekühlt wird —, wenn man den F#-Interpreter entsprechende Testbeispiele rechnen lässt. Zu diesem Zweck schreiben wir zwei Funktionen, eine, die linksentartete, und eine, die rechtsentartete Binärbäume generiert.

```
let rec left-skewed (n : Nat, x : 'elem) : Tree ⟨'elem⟩ =
  if n = 0 then Leaf else Node (left-skewed (n − 1, x), x, Leaf)
let rec right-skewed (n : Nat, x : 'elem) : Tree ⟨'elem⟩ =
  if n = 0 then Leaf else Node (Leaf, x, right-skewed (n − 1, x))
```

Um Tipparbeit zu sparen, definieren wir noch eine Funktion, die nacheinander einen Baum generiert, den Baum mit *inorder* in eine Liste überführt und anschließend die Länge der Liste ermittelt.

let *test* (*generate*, *n*) = *length* (*inorder* (*generate* (*n*, `"Hello, world!"`)))

Die Funktion *test* abstrahiert von dem Generator und von der Anzahl der Elemente. Wenn wir alles richtig gemacht haben, sollte *test* (*generator*, *n*) zu *n* auswerten.

```
Mini⟩ test (left-skewed, 50.000)
50.000
Mini⟩ test (right-skewed, 50.000)
50.000
```

Auf dem Rechner des Dozenten benötigt der F#-Interpreter mehr als 2 Minuten, bis das erste Ergebnis ausgerechnet ist; der zweite Aufruf wird hingegen in Nullkommanichts erledigt, in genau 0,15 Sekunden. Je weiter man den zweiten Parameter erhöht, desto deutlicher wird der Laufzeitunterschied.

Was ist zu tun? Versuchen wir *inorder* zu verbessern oder im Fachjargon zu *optimieren*. (Der Begriff „optimieren“ wird in der Programmierung tatsächlich synonym zu verbessern verwendet: Optimal ist das resultierende Programm nicht notwendigerweise. Das ist tatsächlich auch ein schwieriges Feld. Optimierung im eigentlichen Sinn erfordert zu zeigen, dass es ein besseres Programm nicht geben kann. Man muss also die *Problemkomplexität* kennen.)

Verbessern oder optimieren sollte man nicht blindlings. Zunächst sollte man den oder die Verbrecher identifizieren, die für die mangelhafte Laufzeit verantwortlich sind (Laufzeit-Cluedo). In unserem Fall ist der Verbrecher leicht ausgemacht: Die Funktion *append* treibt die Laufzeit in die Höhe. Die Funktion *append* selbst lässt sich allerdings nicht verbessern, das liegt an der Natur des Typs *List*. Zur Erinnerung: Listen sind asymmetrisch, auf das erste Element einer Liste können wir unmittelbar zugreifen, auf das letzte Element nicht.

Wenn wir *append* selbst nicht verbessern können, müssen wir versuchen, ohne *append* auszukommen. Das hört sich schwierig an. Der Schlüssel zum Erfolg liegt wie so oft in einer geschickten Verallgemeinerung der Aufgabenstellung. Die grundlegende Idee ist, eine Funktion zu programmieren, die gleichzeitig linearisiert *und* konkateniert, also zwei Aufgaben auf einen Schlag erledigt. Wir spezifizieren:

$$\mathit{inorder\text{-}append}\ (\mathit{tree}, \mathit{list}) = \mathit{inorder}\ \mathit{tree} \mathbin{@} \mathit{list} \tag{5.4}$$

Die Spezifikation formuliert, dass *inorder-append* das erste Argument, einen Baum, in eine Liste überführt, und *zusätzlich* mit dem zweiten Argument, einer Liste, konkateniert. Wir können die Spezifikation von *inorder-append* benutzen, um die Funktionsdefinition auszurechnen — aus der Spezifikation wird die Implementierung *hergeleitet*! Das Struktur Entwurfsmuster für *Tree* gibt die folgende Fallunterscheidung vor.

Basisfall $tree = Leaf$**:**

$$
\begin{aligned}
& \mathit{inorder\text{-}append}\ (\mathit{Leaf}, \mathit{list}) \\
= \quad & \{\ \text{Spezifikation von } \mathit{inorder\text{-}append}\ (5.4)\ \} \\
& \mathit{inorder}\ \mathit{Leaf}\ @\ \mathit{list} \\
= \quad & \{\ \text{Definition von } \mathit{inorder}\ \} \\
& [\,]\ @\ \mathit{list} \\
= \quad & \{\ \text{Definition von } \mathit{append} \text{ — '}[\,]\text{' ist das neutrale Element von '@'}\ \} \\
& \mathit{list}
\end{aligned}
$$

Rekursionsfall $tree = Node\ (left, x, right)$**:**

$$
\begin{aligned}
& \mathit{inorder\text{-}append}\ (\mathit{Node}\ (\mathit{left}, x, \mathit{right}), \mathit{list}) \\
= \quad & \{\ \text{Spezifikation von } \mathit{inorder\text{-}append}\ (5.4)\ \} \\
& \mathit{inorder}\ (\mathit{Node}\ (\mathit{left}, x, \mathit{right}))\ @\ \mathit{list} \\
= \quad & \{\ \text{Definition von } \mathit{inorder}\ \} \\
& (\mathit{inorder}\ \mathit{left}\ @\ [x]\ @\ \mathit{inorder}\ \mathit{right})\ @\ \mathit{list} \\
= \quad & \{\ \mathit{append} \text{ ist assoziativ}\ \} \\
& \mathit{inorder}\ \mathit{left}\ @\ ([x]\ @\ \mathit{inorder}\ \mathit{right}\ @\ \mathit{list}) \\
= \quad & \{\ \text{Definition von } \mathit{append}\ \} \\
& \mathit{inorder}\ \mathit{left}\ @\ (x :: \mathit{inorder}\ \mathit{right}\ @\ \mathit{list}) \\
= \quad & \{\ \text{Spezifikation von } \mathit{inorder\text{-}append}\ (5.4)\ \} \\
& \mathit{inorder\text{-}append}\ (\mathit{left}, x :: \mathit{inorder}\ \mathit{right}\ @\ \mathit{list}) \\
= \quad & \{\ \text{Spezifikation von } \mathit{inorder\text{-}append}\ (5.4)\ \} \\
& \mathit{inorder\text{-}append}\ (\mathit{left}, x :: \mathit{inorder\text{-}append}\ (\mathit{right}, \mathit{list}))
\end{aligned}
$$

Die Herleitung macht wesentlichen Gebrauch von der Tatsache, dass die Konkatenation von Listen assoziativ ist, siehe Aufgabe 4.10.

Insgesamt erhalten wir das folgende Programm.

```
let rec inorder-append (tree : Tree ⟨'elem⟩, list : List ⟨'elem⟩) : List ⟨'elem⟩ =
  match tree with
  | Leaf                  → list
  | Node (left, x, right) → inorder-append (left, x :: inorder-append (right, list))
```

Das ursprüngliche Problem ist jetzt ein biederer Spezialfall:

```
let inorder (tree : Tree ⟨'elem⟩) : List ⟨'elem⟩ = inorder-append (tree, [])
```

Hier nutzen wir aus, dass die leere Liste das neutrale Element der Konkatenation ist: $xs\ @\ [\,] = xs$.

Wie hat sich die Rechenzeit verbessert? Beide Aufrufe, *test* (*left-skewed*, 50.000) und *test* (*right-skewed*, 50.000), benötigen jetzt 0,15 Sekunden. Die Laufzeit ist also erfreulicherweise unabhängig von der Struktur des zu linearisierenden Binärbaums — sie ist linear zur *Größe* des Binärbaums.

Konstruktion von Binärbäumen Wie können wir einen balancierten Suchbaum aus einer geordneten Liste konstruieren? Oder etwas allgemeiner: Wie können wir einen Binärbaum aus einer Liste konstruieren, so dass der Inorder-Durchlauf wieder die ursprüngliche Liste ergibt?

$$\mathit{inorder} \ll \mathit{balanced\text{-}tree} = \mathit{id}$$

Mathematisch gesehen, suchen wir die Umkehrfunktion von *inorder* bzw. genauer die *Rechtsinverse*. Die gespiegelte Beziehung $\mathit{balanced\text{-}tree} \ll \mathit{inorder} = \mathit{id}$ lässt sich nicht erfüllen, da es viele Bäume mit dem gleichen Inorder-Durchlauf gibt. Kurz: *inorder* ist surjektiv, aber nicht injektiv. Die Spezifikation macht auch deutlich, dass die ursprüngliche Aufgabe ein Spezialfall ist: *balanced-tree* überführt eine Suchliste in einen Suchbaum.

Das Konstruktionsverfahren orientiert sich an der Struktur eines Binärbaums: Ist die Liste leer, so haben wir einen leeren Baum. Eine mindestens einelementige Liste teilen wir in drei Teile auf, den linken Teil, das Element und den rechten Teil. Das Verfahren wenden wir rekursiv auf die beiden Teillisten an. Um die Ausgeglichenheit zu gewährleisten, sollten die beiden Teillisten natürlich möglichst gleich lang sein. Aber wie können wir eine Liste halbieren? Wir sehen ja einer Liste die Anzahl der Elemente nicht an. (Die Funktion *unzip* taugt übrigens nicht für diesen Zweck. Warum?)

Statt eine Funktion zu programmieren, die eine Liste genau halbiert, ist es geschickter, die Aufgabe etwas zu verallgemeinern: Wir schreiben eine Funktion, die eine Liste *list* in zwei Teillisten der Längen n und $\mathit{length}\ \mathit{list} - n$ zerteilt falls $n \leqslant \mathit{length}\ \mathit{list}$. Halbierung ist dann ein Spezialfall mit $n = \mathit{length}\ \mathit{list} \div 2$.

$$\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{split}\ (n : \mathit{Nat}, \mathit{list} : \mathit{List}\ \langle'a\rangle) : \mathit{List}\ \langle'a\rangle * \mathit{List}\ \langle'a\rangle = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ ([\,], \mathit{list}) \\
\qquad\quad \textbf{\textit{else}}\ \ \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\
\qquad\qquad\quad \mid [\,] \quad\ \ \to ([\,], [\,]) \\
\qquad\qquad\quad \mid x :: xs \to \textbf{\textit{let}}\ (xs_1, xs_2) = \mathit{split}\ (n \dot{-} 1, xs)\ \textbf{\textit{in}}\ (x :: xs_1, xs_2)
\end{array}$$

Die beiden Parameter n und *list* werden im Tandem verringert. Der Programmcode behandelt zusätzlich den Fall, dass $n > \mathit{length}\ \mathit{list}$. Für diesen speziellen Fall legen wir $\mathit{split}\ (n, \mathit{list}) = (\mathit{list}, [\,])$ fest.

Nach diesen Vorarbeiten können wir uns wieder der ursprünglichen Aufgabe zuwenden.

$$\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{balanced\text{-}tree}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle) : \mathit{Tree}\ \langle'a\rangle = \\
\quad \textbf{\textit{let}}\ n = \mathit{length}\ \mathit{list} \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ \mathit{Leaf} \\
\qquad\quad \textbf{\textit{else}}\ \ \textbf{\textit{let}}\ (xs_1, x :: xs_2) = \mathit{split}\ (n \div 2, \mathit{list}) \\
\qquad\qquad\quad \mathit{Node}\ (\mathit{balanced\text{-}tree}\ xs_1, x, \mathit{balanced\text{-}tree}\ xs_2)
\end{array}$$

Die Liste wird im Rekursionsfall halbiert, das Element für die Wurzel wird der zweiten Teilliste entnommen. Man beachte, dass die Fallunterscheidung unvollständig ist: Das Muster $(xs_1, [\,])$ fehlt. Schiefgehen kann aber nichts: Die Länge n der gesamten Liste ist mindestens eins, die zweite Teilliste hat somit die Länge $n - n \div 2 = (n + 1) \div 2$, also umfasst sie ebenfalls mindestens ein Element.

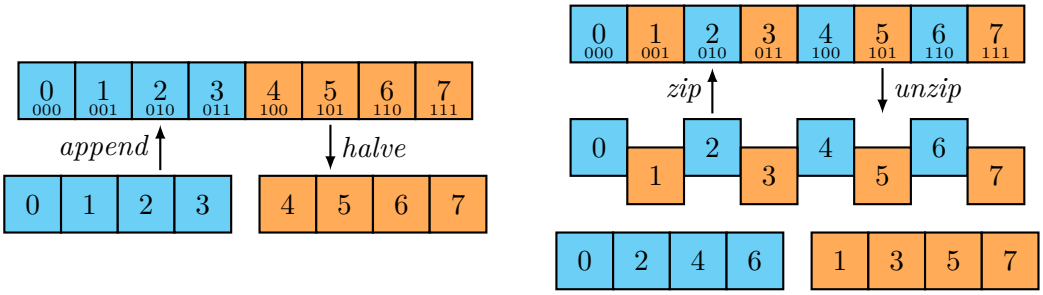
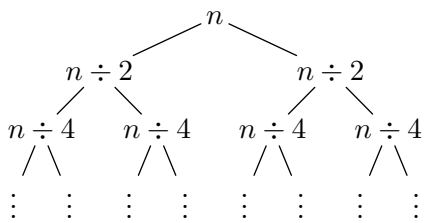


Abbildung 5.18.: Teilen einer Liste: werden die Elemente mit 0 beginnend durchnummeriert, dann teilt *halve* nach dem höchstwertigsten Bit, während *unzip* nach dem niederwertigsten Bit aufteilt (vorausgesetzt, die Länge der Liste ist eine exakte Zweiterpotenz).

Welche Laufzeit hat *balanced-tree*? Die Länge der Liste spielt sicherlich eine zentrale Rolle. Schauen wir uns also die rekursive Aufrufstruktur von *balanced-tree* in Abhängigkeit von der Listenlänge an (die Längen stimmen nicht ganz).



Der *Rekursionsbaum* ist — genau wie der erzeugte Suchbaum — ausgeglichen: Der initiale Aufruf führt zu zwei rekursiven Aufrufen, jeder dieser Aufrufe resultiert in zwei weiteren rekursiven Aufrufen usw. Die Höhe des Rekursionsbaumes ist $\lg n$, wenn n die Länge der ursprünglichen Liste ist. Vor jedem rekursiven Aufruf wird das aktuelle Listenargument $1\frac{1}{2}$ mal durchlaufen (1 mal von *length* und $\frac{1}{2}$ mal von *split*). Alle Aufrufe auf einer horizontalen Ebene haben somit *zusammen* eine lineare Laufzeit. Multipliziert mit der Anzahl der Ebenen, der Höhe des Rekursionsbaumes, ergibt sich insgesamt eine Laufzeit von $n \lg n$.

Blättern wir im Skript zurück, merken wir, dass die Laufzeitanalyse im Wesentlichen der Analyse von Mergesort entspricht. Dies überrascht vielleicht nicht, da beide Funktionen nach dem Leibniz Entwurfsmuster gestrickt sind. Sie illustrieren zwei duale Ansätze, die Problemreduktion (in der „Teile“-Phase) zu realisieren: Mergesort trennt die Listen reißverschlussartig mit Hilfe von *unzip* auf, *balanced-tree* trennt in der Mitte mit Hilfe von *halve*, siehe Abbildung 5.18. Je nach Problemstellung wird man dem einen oder dem anderen Verfahren den Vorzug geben.[10] Mergesort funktioniert prinzipiell mit beiden Ansätzen; tatsächlich ist aber vorteilhafter *halve* zu verwenden, siehe Aufgabe 5.2.

[10] Die Trennung in der Mitte erscheint auf den ersten Blick natürlicher oder offensichtlicher. Tatsächlich aber verwenden wir oft das Reißverschlussverfahren, da die Mitte manchmal nur schwer abzuschätzen ist: etwa beim Austeilen von Karten, die alternierend an die Mitspieler*innen vergeben werden.

Zurück zum ursprünglichen Problem: Lässt sich die Laufzeit von *balanced-tree* verbessern? Vielleicht können wir wiederum die Aufgabenstellung geschickt verallgemeinern? Im Fall von *inorder* haben wir die Linearisierung mit der Konkatenation von Listen kombiniert. Jetzt lösen wir das duale Problem, so dass es naheliegt, die Konstruktion mit dem Aufspalten einer Liste zu kombinieren. Wir spezifizieren:

$$\mathit{balanced\text{-}tree\text{-}of\text{-}size}\ n\ \mathit{list} = (\mathit{balanced\text{-}tree}\ \mathit{xs}, \mathit{ys}) \quad \text{wobei} \quad (\mathit{xs}, \mathit{ys}) = \mathit{split}\ n\ \mathit{list}$$

Der erste Parameter von *balanced-tree-of-size* gibt sozusagen den „erlaubten Verbrauch“ an Listenelementen an; nicht benötigte Elemente müssen zurückgegeben werden. Beim initialen Aufruf wird n die Länge der gesamten Liste sein, so dass wir als Vorbedingung annehmen, dass $n \leqslant \mathit{length}\ \mathit{list}$ — es sind stets genügend Elemente vorhanden.

Die ursprüngliche, umgangssprachliche Beschreibung des Konstruktionsverfahrens lässt sich jetzt umittelbar in Rechenregeln überführen. Ist $n = 0$, so geben wir ein Blatt und alle Listenelemente zurück. Anderenfalls konstruieren wir aus den ersten $m = (n-1) \div 2$ Elementen den linken Teilbaum, entnehmen ein Element für die Wurzel und konstruieren aus den restlichen Elementen den rechten Teilbaum.

$$\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{balanced\text{-}tree\text{-}of\text{-}size}\ n\ \mathit{xs} = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ (\mathit{Leaf}, \mathit{xs}) \\
\qquad\quad \textbf{\textit{else}}\ \ \textbf{\textit{let}}\ m = (n \dot{-} 1) \div 2 \\
\qquad\qquad\qquad \textbf{\textit{let}}\ (l,\ x :: \mathit{ys}) = \mathit{balanced\text{-}tree\text{-}of\text{-}size}\ m \qquad\qquad\quad \mathit{xs} \\
\qquad\qquad\qquad \textbf{\textit{let}}\ (r, \mathit{zs}) \qquad\ = \mathit{balanced\text{-}tree\text{-}of\text{-}size}\ (n \dot{-} 1 \dot{-} m)\ \mathit{ys} \\
\qquad\qquad\qquad (\mathit{Node}\ (l, x, r), \mathit{zs})
\end{array}$$

Im Rekursionsfall verbrauchen wir wie vorgegeben insgesamt $m + 1 + (n - 1 - m) = n$ Elemente. Auf Grund der Vorbedingung $n \leqslant \mathit{length}\ \mathit{xs}$ kann das widerlegbare Muster $x :: \mathit{ys}$, mit dem wir das Wurzelelement x entnehmen, nicht scheitern.

Das ursprüngliche Problem ist jetzt ein biederer Spezialfall: Wir setzen den „erlaubten Verbrauch“ auf den maximalen Wert, die Listenlänge.

$$\textbf{\textit{let}}\ \mathit{balanced\text{-}tree}\ x = \mathit{fst}\ (\mathit{balanced\text{-}tree\text{-}of\text{-}size}\ (\mathit{length}\ x)\ x)$$

Die Laufzeit von *balanced-tree-of-size* n *list* ist proportional zu n. Somit hat die Funktion *balanced-tree* eine lineare Laufzeit — das ist optimal, da wir jedes Listenelement verarbeiten müssen. Übrigens würde auch *merge-sort* von einem ähnlichen Ansatz profitieren, siehe Aufgabe 5.2. Wir können zwar nicht die asymptotische Laufzeit verbessern (an der Größenordnung ändert sich nichts), wohl aber die konkrete Zahl der Rechenschritte.

Fazit: Ein schwierigeres Problem muss nicht schwieriger zu lösen sein. Die Ursache für diese scheinbar paradoxe Tatsache liegt in der Rekursion begründet: Im Rekursionsschritt können wir auf Teillösungen zurückgreifen; die rekursiven Aufrufe lösen aber bereits schwierigere Teilprobleme, so dass der Schritt zur Gesamtlösung oft einfacher ist. Im Fall von *inorder-append* zum Beispiel erledigt der rekursive Aufruf zusätzlich das Aneinanderhängen der Teillisten.

Das *Rekursionsparadoxon* stellt eine allgemeine Programmiertechnik dar und firmiert unter verschiedensten Namen: Verallgemeinerung der Aufgabenstellung, akkumulierende Parameter (*inorder*), Tupeltransformation (*balanced-tree*); beim Beweisen, einer dem Programmieren eng verwandten Tätigkeit, spricht man von der Verstärkung der Induktionsannahme, im Englischen auch bekannt unter dem Namen „Inventor's Paradox".

5.3.5. Laufzeitverhalten der Implementierungen

Wir haben gesehen, dass sich endliche Abbildungen auf mindestens drei verschiedene Arten implementieren lassen (tatsächlich gibt es eine fast unüberschaubare Anzahl von Möglichkeiten). Bezüglich der funktionalen Eigenschaften, des Ein- und Ausgabeverhaltens, sind die Implementierungen austauschbar — die Spezifikation der Semantik legt das Verhalten präzise fest; die Implementierungen setzen die Anforderungen buchstabengetreu um. Unterschiede treten bezüglich der nichtfunktionalen Eigenschaften auf, bezüglich des Resourcenverbrauchs. Die folgende Tabelle trägt die einzelnen Ergebnisse zusammen (dabei steht n für die Gesamtzahl der Einträge, h mit $\lceil \lg(n+1) \rceil \leqslant h \leqslant n$ für die Höhe der Binärbäume).

	Listen	Suchlisten	Suchbäume
empty	1	1	1
add	1	n	h
remove	n	n	h
is-empty	1	1	1
lookup	n	n	h
from-list	n	$n \lg n$	$n \lg n$
to-list	n	n	n

Die linear-logarithmische Laufzeit von *from-list* im Fall von Suchlisten und Suchbäumen erklärt sich durch die Sortierung der Eingabe. (Lässt sich *from-list* auf lineare Laufzeit beschleunigen?)

Qual der Wahl? Nicht wirklich. Ungeachtet der Tatsache, dass Binärbäume nicht ihr volles Potential ausschöpfen (Stichwort: Balancierung), sind sie unter den vorgestellten Implementierungen die Bibliothek der Wahl.

5.4. Prioritätswarteschlangen

In Abschnitt 5.1.5 haben wir uns überlegt, wie man das i-kleinste Element einer Folge bestimmen kann, die i-te Ordnungsstatistik. Die Folge ist dabei fest vorgegeben; wenn sich die Folge ändert, dann muss die i-te Ordnungsstatistik neu berechnet werden. Die Algorithmen ziehen dabei keinen Nutzen aus der oder den vorausgegangenen Berechnungen. In diesem Abschnitt wenden wir uns der „dynamischen Variante" dieser Aufgabe zu. Aus einem algorithmischen Problem, der Bestimmung der Ordnungsstatistik, wird ein abstrakter Datentyp, die *Prioritätswarteschlange* oder Vorrangwarteschlange (engl. priority queue).

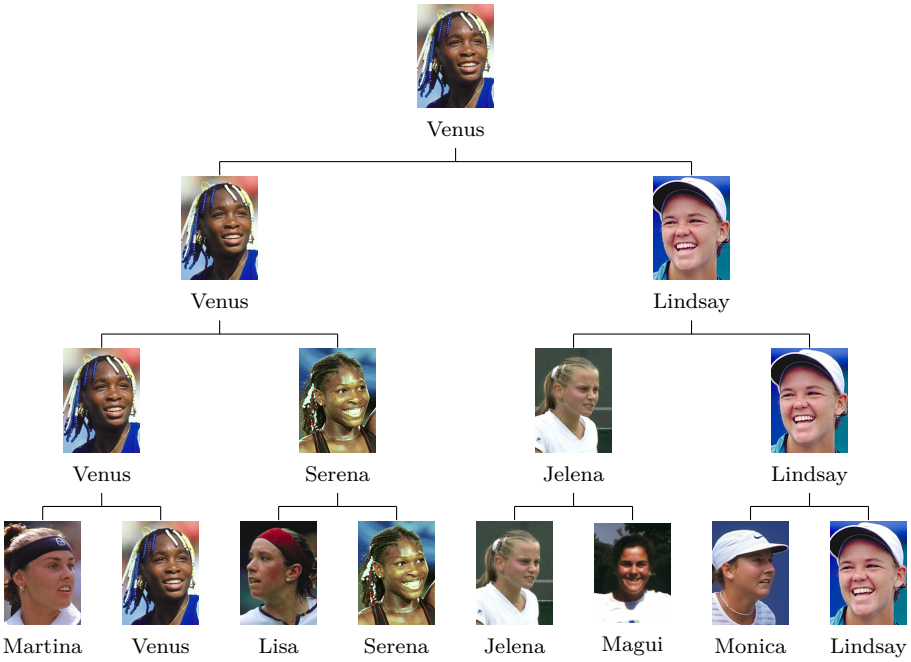


Abbildung 5.19.: Dameneinzel (Wimbledon Championships 2000).

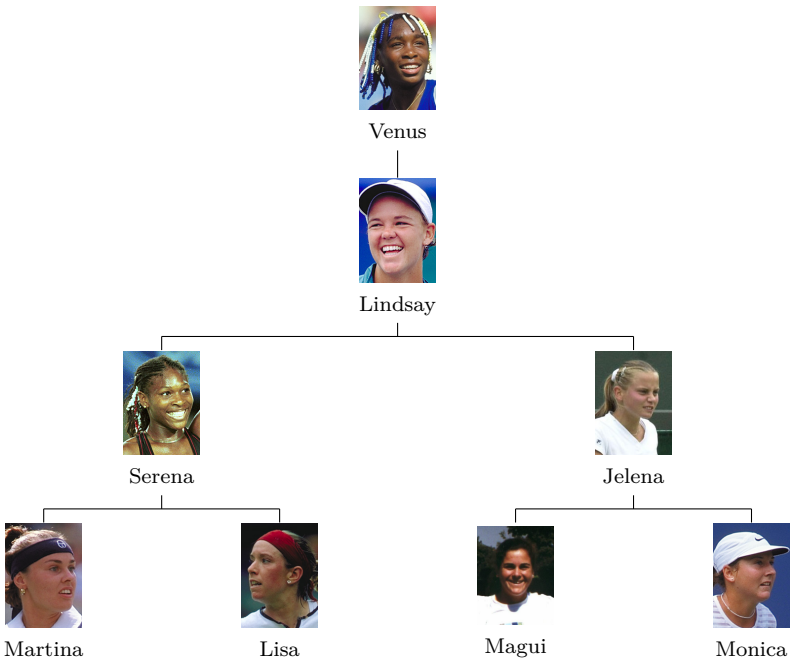


Abbildung 5.20.: Der zu dem Turnierbaum aus Abbildung 5.19 korrespondierende Baum der Verliererinnen mit dem Champion an der Spitze.

Warteschlangen kennen wir aus dem täglichen Leben. Oft werden sie im „first-in first-out“ Modus betrieben: Bei Behördengängen entscheidet in der Regel die Ankunftszeit („Ziehen Sie bitte eine Wartemarke …“), wann eine Person an der Reihe ist. Bei einer Prioritätswarteschlange ist hingegen nicht die Ankunftszeit, sondern eine vorher festgelegte Priorität ausschlaggebend, zum Beispiel die Wichtigkeit oder die Hilfsbedürftigkeit einer Person. Dieser „Modus Operandi“ begegnet einem zum Beispiel beim Boarding eines Flugzeugs („Wir bitten zunächst Senatoren und Familien mit kleinen Kindern …“). Auch das Betriebssystem eines Rechners verwendet eine Prioritätswarteschlange, um anstehende Rechenaufgaben zu verwalten und diese nach Dringlichkeit abzuarbeiten. Die berühmt-berüchtigten Todo-Listen sind von ähnlicher Natur; die Priorität einer Aufgabe spiegelt deren Dringlichkeit oder die noch verbleibende Zeit bis zum avisierten Fertigstellungstermin wider.

Schnittstelle Prioritätswarteschlangen unterstützen die folgenden Operationen.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{\textit{type}}\ \mathit{PQueue}\,\langle'\mathit{elem}\ \textbf{\textit{when}}\ '\mathit{elem} : \mathit{comparison}\rangle} \\
\mathit{empty} & : \mathit{PQueue}\,\langle'\mathit{elem}\rangle \\
\mathit{single} & : 'elem \to \mathit{PQueue}\,\langle'\mathit{elem}\rangle \\
\mathit{insert} & : 'elem * \mathit{PQueue}\,\langle'\mathit{elem}\rangle \to \mathit{PQueue}\,\langle'\mathit{elem}\rangle \\
\mathit{meld} & : \mathit{PQueue}\,\langle'\mathit{elem}\rangle * \mathit{PQueue}\,\langle'\mathit{elem}\rangle \to \mathit{PQueue}\,\langle'\mathit{elem}\rangle \\
\mathit{split\text{-}min} & : \mathit{PQueue}\,\langle'\mathit{elem}\rangle \to ('\mathit{elem} * \mathit{PQueue}\,\langle'\mathit{elem}\rangle)\ \mathit{option} \\
\mathit{from\text{-}list} & : '\mathit{elem}\ \mathit{list} \to \mathit{PQueue}\,\langle'\mathit{elem}\rangle \\
\mathit{to\text{-}ord\text{-}list} & : \mathit{PQueue}\,\langle'\mathit{elem}\rangle \to '\mathit{elem}\ \mathit{list}
\end{array}
$$

Der abstrakte Datentyp *PQueue* ist mit dem Typ der Elemente parametrisiert; der Zusatz ***when*** $'elem : comparison$ legt fest, dass sich Elemente vergleichen lassen müssen. Die Priorität eines Elements wird also nicht indirekt durch eine Zahl angegeben, sondern direkt durch die zugrundeliegende Ordnung auf den Elementen — wie immer setzen wir eine totale Quasiordnung voraus.

Abstrakt gesehen repräsentieren Warteschlangen Multimengen (engl. bags) bzw. *geordnete Sequenzen*. Insbesondere können Elemente in einer Warteschlange mehrfach auftreten. Mit Hilfe der ersten vier Operationen werden Warteschlangen konstruiert bzw. transformiert. Die *meld* Operation verdient besondere Beachtung: Im Unterschied zu endlichen Abbildungen können zwei Warteschlangen „verschmolzen“, zu einer Schlange vereinigt werden. (Unsere endlichen Abbildungen bieten eine analoge Operation, sprich den „Kommaoperator“, nicht an, da geignete Implementierungen wie zum Beispiel Suchbäume die Operation nicht effizient unterstützen.) Um eine Warteschlange zu „dekonstruieren“, steht lediglich die Operation *split-min* zur Verfügung, die eine Schlange in das kleinste Element und die restliche Schlange aufteilt. Die Operation stellt das Herzstück der Signatur dar: Prioritätswarteschlangen sollen ja den Zugriff auf das jeweils kleinste Element effizient unterstützen. Zusätzlich gibt es noch zwei Bulk-Operationen[11], mit denen eine Liste in eine Schlange und umgekehrt eine Schlange in eine *geordnete Liste* überführt werden kann.

[11] So werden Operationen bezeichnet, die eine „große“ Menge von Daten (engl. bulk) verarbeiten.

Interdefinierbarkeit Die verschiedenen Operationen der Signatur sind nicht unabhängig voneinander; einige lassen sich mit Hilfe der anderen Operationen implementieren. Im Fachjargon sagt man, die Operationen sind *interdefinierbar*. Zum Beispiel lässt sich *insert* auf *single* und *meld* zurückführen.

```
let insert (x, q) = meld (single x, q)
```

Können Sie umgekehrt *meld* mit den anderen Operationen der Signatur realisieren?

Warum ist das eine bedeutsame Erkenntnis? Nun, Anwendungsprogrammierer*innen wünschen sich eine möglichst reichhaltige Schnittstelle; Implementierer*innen eine möglichst schmale. Um zwischen den gegenläufigen Interessen zu vermitteln, sind *Default-Implementierungen* wie die von *insert* nützlich. Sie reduzieren den Implementierungsaufwand, da sie unabhängig von einer speziellen Implementierung arbeiten, zumindest den initialen Implementierungsaufwand — oft lässt sich die betreffende Operation für eine spezielle Implementierung direkter und damit effizienter umsetzen.

Insbesondere lassen sich für die beiden Bulk-Operationen Default-Implementierungen angeben. Um eine Wartschlange in eine geordnete Liste zu überführen, rufen wir *split-min* solange auf, bis die Warteschlange leer ist.

```
let rec to-ord-list q =
  match split-min q with
  | None          → []
  | Some (x, q′) → x :: to-ord-list q′
```

Der Verfahren erinnert an „Sortieren durch Auswählen“ mit dem Unterschied, dass die *Eingabe*daten nicht konkret sind, eine Liste von Elementen, sondern abstrakt, eine Prioritätswarteschlange. Dementsprechend führen wir nicht direkt eine Fallunterscheidung über die Eingabe durch, mit den Mustern [ ] und $x :: xs$, sondern diskriminieren die Rückgabe von *split-min*, mit den Mustern *None* und *Some* (x, q'). Von der Syntax abgesehen entsprechen sich die Muster — mit Hilfe von *split-min* kann eine Prioritätswarteschlange peu à peu linearisiert werden.

Umgekehrt können wir eine Liste in eine Prioritätswarteschlange überführen, indem wir nacheinander die Elemente in die anfangs leere Warteschlange einfügen. Der Verfahren erinnert an „Sortieren durch Einfügen“ mit dem Unterschied, dass die *Ausgabe*daten nicht konkret sind, sondern abstrakt. Statt das Peano Entwurfsmuster zu verwenden, können wir auch das Leibniz Entwurfsmuster einsetzen und erhalten dann eine abstrakte Variante des „Sortieren durch Mischens“. Im Folgenden wollen wir uns eine weitere Alternative anschauen, die von Tennisturnieren inspiriert ist.

Um die beste Tennisspieler*in zu ermitteln, kann man ein Turnier im K.-o.-System austragen (engl. knock-out tournament). Die Gewinner*in eines Spiels zieht in die nächste Runde ein, die Verlierer*in scheidet aus. Der Turnierverlauf lässt sich durch einen *Turnierbaum*, einen binären Baum, repräsentieren, siehe Abbildung 5.19. Aus Gründen der Fairness und um möglichst viele Spiele gleichzeitig austragen zu können (Parallelverarbeitung wird auch in der Informatik immer wichtiger), ist der Turnierbaum in der Regel vollständig ausgeglichen. Dies setzt voraus, dass die Gesamtzahl der Spieler*innen eine exakte Zweierpotenz ist.

Die folgenden Funktionen übertragen die Idee des Tennisturniers auf das Problem der Konstruktion einer Prioritätswarteschlange.

```
// play-round : (PQueue ⟨'elem⟩) list → (PQueue ⟨'elem⟩) list when 'elem : comparison
let rec play-round = function
    | []          → []
    | [q1]        → [q1]
    | q1 :: q2 :: qs → meld (q1, q2) :: play-round qs
// tournament : (PQueue ⟨'elem⟩) list → PQueue ⟨'elem⟩ when 'elem : comparison
let rec tournament = function
    | []  → empty
    | [q] → q
    | qs  → tournament (play-round qs)
let from-list (xs : 'elem list) = tournament [for x in xs → single x]
```

) Wir überführen die Listenelemente zunächst in einelementige Warteschlangen. In einer Turnierrunde werden jeweils zwei benachbarte Warteschlangen verschmolzen. Wenn die Eingabe von *play-round* die Länge n hat, dann umfasst die Ausgabeliste $\lceil n/2 \rceil$ Warteschlangen. Wir spielen soviele Runden wie nötig; die Funktion *play-round* wird solange aufgerufen, bis nur noch eine einzige Warteschlange übrigbleibt. Ein gewichtiger Unterschied zu einem Tennisturnier fällt vielleicht ins Auge: Wir verwalten eine Liste von Warteschlangen, nicht von Elementen, da wir nicht nur den Champion bestimmen wollen. Mit anderen Worten, Verlierer*innen scheiden nicht sofort aus — später mehr dazu. Da der Turnierbaum von unten nach oben aufgebaut wird, spricht man auch von einem „bottom-up“ Verfahren. Im Unterschied dazu ist „Sortieren durch Mischen“ ein „top-down“ Verfahren — Aufgabe 5.3 fragt nach einer alternativen „bottom-up“ Implementierung.

Wie effizient ist *from-list*? *Wenn* wir davon ausgehen, dass *meld* in konstanter Zeit arbeitet, *dann* ergibt sich das folgende Bild. Die Hilfsfunktion *play-round* hat eine lineare Laufzeit; da sie in jedem Schritt die Anzahl der Warteschlangen ungefähr halbiert, ist die Gesamtlaufzeit ebenfalls linear:

$$\frac{1}{1}n + \frac{1}{2}n + \frac{1}{4}n + \frac{1}{8}n + \cdots < 2 \cdot n$$

Ähnliche Überlegungen haben wir bei Analyse der Laufzeit von *quickselect* angestellt, siehe auch Abbildung 5.10. Aber ist die Annahme, dass *meld* lediglich konstante Zeit benötigt, realistisch? Schauen wir uns ein paar Implementierungsideen an, um ein Gefühl für die Komplexität der Operationen zu bekommen.

Implementierungsideen Vielleicht lassen sich die Implementierungen von endlichen Abbildungen aus Abschnitt 5.3 adaptieren?

- *ungeordnete Listen:* Das Einfügen ist schnell, aber das Auswählen des Minimums ist langsam (siehe „Sortieren durch Auswählen“).

- *Suchlisten:* Vor- und Nachteile kehren sich um; das Einfügen ist langsam (siehe „Sortieren durch Einfügen“), während das Auswählen schnell ist.
- *Suchbäume:* Das kleinste Element in einem Suchbaum ist das linkeste Element. (Wenn wir uns die Größe der Teilbäume merken, dann unterstützen Suchbäume sogar Ordnungsstatistiken.) Einfügen und Auswählen arbeiten somit proportional zur Höhe der Bäume.

Leider unterstützt keine der obigen Datenstrukturen eine effiziente Implementierung von *meld*. Die Laufzeit ist jeweils linear zur Zahl der Elemente.

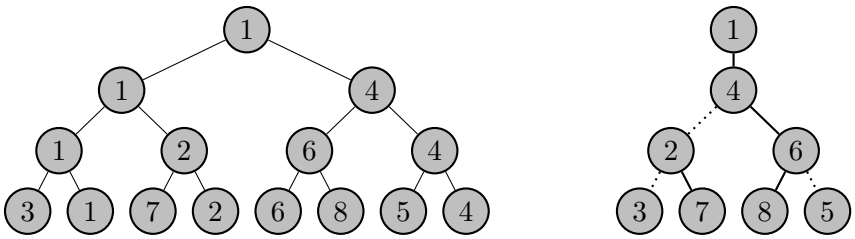
Was können wir erwarten? Lässt sich eine geniale Implementierung finden, so dass alle Operationen (mit Ausnahme der Bulk-Operationen) in konstanter Zeit arbeiten? Die Antwort ist ein klares und entschiedenes „Nein!“ Warum? Wenn wir die Bulk-Operationen komponieren,

let heap-sort = from-list ≫ to-ord-list

erhalten wir ein abstraktes Sortierverfahren[12], „Sortieren mit Prioritätswarteschlangen“. Hätten sowohl *insert* als auch *split-min* eine konstante Laufzeit (oder *merge* und *split-min*), dann könnten wir in linearer Zeit sortieren. In Abschnitt 5.1.3 haben wir gezeigt, dass eben dies nicht möglich ist. Wir können messerscharf folgern, dass entweder das Einfügen (*insert* und *meld*) oder das Auswählen (*split-min*) mindestens eine logarithmische Laufzeit haben muss. Schade!

5.4.1. Turnierbäume

Wir brauchen eine zündende Idee. Vielleicht können wir Turnierbäume als Datenstruktur für die Implementierung von Prioritätswarteschlangen verwenden? Der linke untere Binärbaum korrespondiert zu dem Turnierbaum aus Abbildung 5.19 — wir haben lediglich die Spielerinnen durch die vermutete Spielstärke ersetzt.



Ein Defekt fällt sofort ins Auge. Der Turnierbaum enthält Elemente mehrfach: Je besser eine Spielerin, je kleiner der numerische Wert, desto öfter tritt sie bzw. er auf. Können wir die Redundanz eliminieren? In einem K.-o.-Turnier verliert jede Spielerin genau ein Match — mit Ausnahme des späteren Champions —, so dass die Idee naheliegt den Gewinnerbaum durch einen Verliererbaum zu ersetzen, so wie in der Graphik oben rechts, siehe auch Abbildung 5.20.

[12] Alle bisher eingeführten Sortierverfahren lassen sich als konkrete Instanzen dieses Verfahrens deuten, je nachdem ob man Warteschlangen durch ungeordnete Listen, Suchlisten oder Suchbäume realisiert.

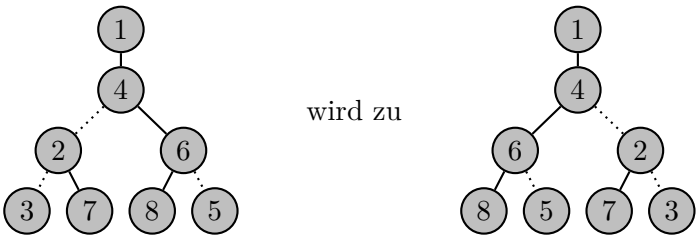
Der Champion wird zusätzlich oben auf den Verliererbaum gesetzt. Die durchgezogenen Linien deuten jeweils an, aus welcher Turnierhälfte die Verliererin kommt. Diese Information ist bedeutsam. So wissen wir zum Beispiel, dass 4 kleiner ist als alle Werte im rechten Teilbaum — Lindsay hat die rechte Turnierhälfte gewonnen. Über das Verhältnis zu den Werten aus dem linken Teilbaum ist hingegen nichts bekannt — mit den Teilnehmerinnen aus der linken Turnierhälfte hat Lindsay keine Spiele ausgetragen. Den „getoppten“ Verliererbaum können wir durch ein Paar bestehend aus einem Element und einem Binärbaum repräsentieren. Diese Kombination nennt man auf Grund der Form auch *Wimpel* (engl. pennant). Ein Wimpel enspricht einem nicht-leeren Binärbaum mit einem leeren linken oder rechten Teilbaum.

```
type Pennant ⟨'elem⟩ = 'elem * Tree ⟨'elem⟩
```

Die Funktion *versus* führt ein Spiel durch.

```
// versus : Pennant ⟨'elem⟩ * Pennant ⟨'elem⟩ → Pennant ⟨'elem⟩ when 'a : comparison
let versus ((a, t), (b, u)) =
  if a ⩽ b then (a, Node (u, b, t))      // b dominiert u
           else (b, Node (t, a, u))      // a dominiert t
```

Im Unterschied zu einem K.-o.-Spiel scheidet die Verliererin nicht unmittelbar aus. Ganz im Gegenteil: Mit Hilfe der Verliererbäume protokollieren wir den Turnierverlauf. Ein Detail ist bedeutsam: Wir arrangieren den Verliererbaum jeweils so, dass die Wurzel den *linken* Teilbaum dominiert. Mit anderen Worten, für jeden Knoten gilt, dass die Knotenmarkierung kleiner gleich den Elementen im linken Teilbaum ist. Binarbäume mit dieser Eigenschaft heißen auch *Semi-Heaps*. (In einem *Heap* dominiert die Knotenmarkierung jeweils beide Teilbäume; in einem Semi-Heap nur einen.)



Die Elemente auf einem Pfad entlang der *durchgezogenen* Linien sind jeweils aufsteigend geordnet (von oben nach unten). Während die Elemente in einem Suchbaum *horizontal* angeordnet sind, sind die Elemente in einem Semi-Heap bzw. in einem Heap *vertikal* angeordnet.

Anders als in der Tenniswelt können unsere Turniere auch leer sein: die Operation *empty* soll ja eine leere Warteschlange repräsentieren, so dass wir einen nicht-rekursiven Variantentyp als Implementierungstyp verwenden.

```
type PQueue ⟨'elem when 'elem : comparison⟩ =
  | Inf
  | Min of Pennant ⟨'elem⟩
```

Die leere Warteschlange wird durch *Inf* repräsentiert; eine nicht-leere Warteschlange durch *Min* (a, t), wobei a das kleinste Element ist — a ist der Champion, t der Verliererbaum. (Alternativ hätten wir als Implementierungstyp *Pennant* $\langle'elem\rangle$ *option* verwenden können. Die Konstruktornamen *Inf* und *Min* machen den Code aber etwas lesbarer.) Die Graphiken und die Typdefinition verdeutlichen, dass unsere Warteschlangen Zwitterwesen sind: Sie fangen als Liste an und hören als Binärbaum auf.

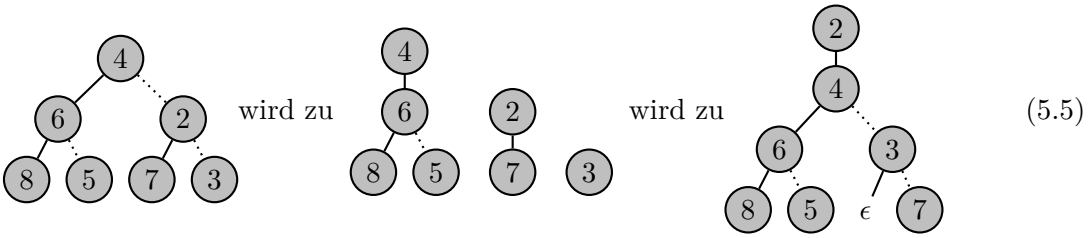
Nach diesen Vorarbeiten können wir zur Implementierung der Operationen schreiten.

```
let empty = Inf
let single x = Min (x, Leaf)
let meld = function
    | (Inf, q) | (q, Inf)          → q
    | (Min (a, t), Min (b, u)) → Min (versus ((a, t), (b, u)))
```

Bei der Implementierung von *meld* nutzen wir aus, dass *empty* das neutrale Element der Vereinigung ist. Wenn beide Warteschlangen nicht-leer sind, tragen wir ein Spiel aus. Vielleicht überraschend: *meld* arbeitet in konstanter Zeit. Mit einer einzigen Vergleichsoperation können wir zwei Warteschlangen verschmelzen.

Da *meld* und somit *insert* eine konstante Laufzeit haben, muss *split-min* notwendigerweise mindestens eine logarithmische Laufzeit an den Tag legen. Bei einem Tennisturnier erhält die Verliererin des Endspiels den zweiten Preis. Das Preisgeld ist vielleicht verdient, aber nicht gerechtfertigt: Die Verliererin dominiert ja nur eine Turnierhälfte; damit ist sie im ungünstigsten Fall lediglich besser als die Hälfte der Teilnehmerinnen.[13] Dieser Fall tritt ein, wenn die besten Spielerinnen „zufällig“ in der gleichen Turnierhälfte gesetzt sind — eine Situation, die man mit Hilfe von Ranglisten zu vermeiden versucht.

In unserem Beispiel ist die Verliererin lediglich die 4-beste Spielerin. Welche Spielerinnen kommen für den 2. Preis überhaupt in Frage? Das sind diejenigen, die gegen den späteren Champion verloren haben! In dem Verliererbaum lassen sich die Elemente leicht lokalisieren; sie befinden sich auf dem Pfad von der Wurzel zu dem *rechtesten* Blatt, da vereinbarungsgemäß jede Knotenmarkierung den *linken* Teilbaum dominiert. Zwischen diesen Teilnehmerinnen muss der 2. Platz ausgespielt werden.



(5.5)

Wir machen eine interessante Beobachtung: Ein Binärbaum korrespondiert zu einer Liste von Wimpeln! In der Funktion *second-best* werden diese von rechts nach links mit-

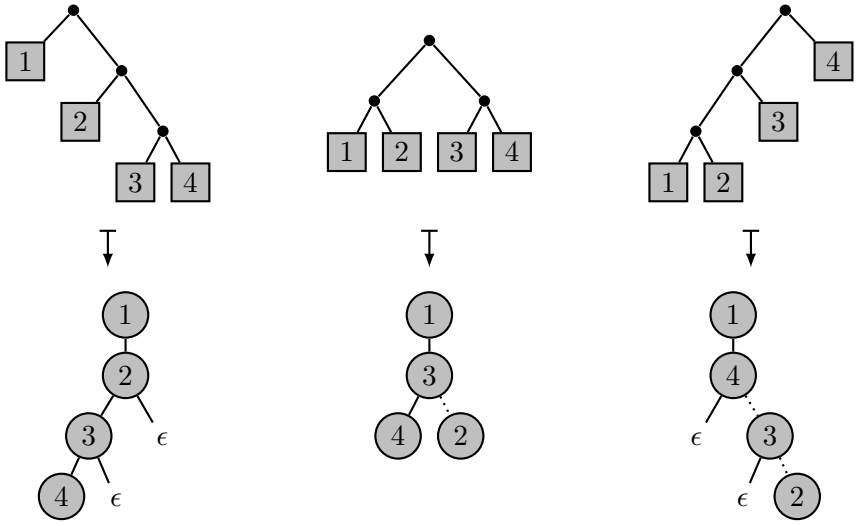
[13] Charles Lutwidge Dodgson besser bekannt als Lewis Carroll hat früh auf dieses Problem hingewiesen und in der Abhandlung „Lawn Tennis Tournaments, The True Method of Assigning Prizes with a Proof of the Fallacy of the Present Method.“ eine alternative Turnierplanung entwickelt.

einander verschmolzen (streng nach dem Strukturentwurfsmuster für Listen). Die resultierende Warteschlange enthält sieben Elemente und kann somit nicht mehr vollständig ausgeglichen sein: 3 ist gleich in der 1. Runde gegen den Champion ausgeschieden und dominiert damit einen leeren linken Teilbaum (durch ϵ repräsentiert).

```
// second-best : Tree ⟨'a⟩ → PQueue ⟨'a⟩ when 'a : comparison
let rec second-best = function
  | Leaf           → Inf
  | Node (t, a, u) → meld (Min (a, t), second-best u)
let split-min = function
  | Inf        → None
  | Min (a, t) → Some (a, second-best t)
```

Der Typ von *second-best* verdeutlicht, dass die Funktion einen Verliererbaum in eine Warteschlange überführt. Da der Baum von der Wurzel bis zum rechtesten Blatt traversiert wird, ist die Laufzeit von *second-best* und somit von *split-min* proportional zur Höhe des Verliererbaums. Wie bei Suchbäumen wird die Laufzeit nicht nur von der Gesamtzahl der Knoten beeinflusst, sondern auch von der Gestalt des Binärbaums.

Ist der zugrundeliegende Turnierbaum ausgeglichen, dann ist die Laufzeit logarithmisch — das ist asymptotisch optimal. Erfreulicherweise konstruiert zum Beispiel die Default-Implementierung von *from-list* einen balancierten Turnierbaum. Daraus folgt insbesondere, dass *heap-sort* eine linear-logarithmische Laufzeit hat — das ist ebenfalls asymptotisch optimal. Wo Licht ist, da fällt auch Schatten. Ist der Turnierbaum degeneriert — ist er zum Beispiel links- oder rechtsentartet —, dann ist auch die resultierende Warteschlange degeneriert. Im Unterschied zu Suchbäumen gilt die Gleichung „degenerierter Baum = schlechte Laufzeit“ aber *nicht*. Die folgenden Beispiele illustrieren warum — die schwarzen, inneren Knoten entsprechen einem Aufruf von *meld*, die quadratischen, äußeren Knoten einem Aufruf von *single*.



Die drei resultierenden Warteschlangen enthalten jeweils die gleichen Elemente. Im ersten Fall ist die Warteschlange zwar *links*entartet, aber das ist tatsächlich vorteilhaft, da

second-best stets nach *rechts* läuft! Die durchgezogenen Linien deuten an, dass wir die vollständige Ordnung der Elemente kennen. Der ausgeglichene Baum in der Mitte ist somit weniger vorteilhaft. Am schlechtesten schneidet die *rechts*entartete Warteschlange ab. Die gestrichelten Linien deuten an, dass wir über die relative Ordnung der Elemente im Verliererbaum überhaupt nichts wissen. Je mehr durchgezogene Linien, desto besser!

Nun mag die Leser*in einwenden, dass man in der Praxis niemals einen unbalancierten Turnierplan aufstellen würde. Das mag für Tennis- oder Fußballturniere gelten, aber nicht für unsere Bibliothek, da wir schlicht und einfach keinen Einfluss darauf haben, wie die Schnittstelle verwendet wird. Zum Beispiel könnte die Anwendungsprogrammierer*in ihre eigene Variante von *from-list* definieren und dabei streng nach dem Struktur Entwurfsmuster für Listen vorgehen.

```
let rec from-list = function
  | []      → empty
  | x :: xs → meld (single x, from-list xs)      // oder: meld (from-list xs, single x)
```

Wenden wir *from-list* auf eine aufsteigend geordnete Liste an, erhalten wir einen linksentarteten Verliererbaum; für eine absteigend geordnete Liste einen rechtsentarteten Baum. (Warum?)

```
Mini⟩ from-list [1..5]
Min (1, Node (Node (Node (Node (Leaf, 5, Leaf), 4, Leaf), 3, Leaf), 2, Leaf))
Mini⟩ from-list [5..-1..1]
Min (1, Node (Leaf, 5, Node (Leaf, 4, Node (Leaf, 3, Node (Leaf, 2, Leaf)))))
```

Komponieren wird diese Version von *from-list* mit *to-ord-list* — wir wollen Listen sortieren —, dann ergeben sich erhebliche Laufzeitschwankungen: Für aufsteigend geordnete Listen ist die Gesamtlaufzeit linear (*heap-sort* verhält sich wie „Sortieren durch Einfügen“), für absteigend geordnete Listen hingegen quadratisch (*heap-sort* verhält sich wie „Sortieren durch Auswählen“).

Wie können wir die Laufzeit von *second-best* verbessern? Wir diskutieren zwei Ansätze; der erste ist einfach zu implementieren, aber schwierig zu analysieren; die Implementierung des zweiten ist aufwändiger, aber dafür leichter zu analysieren.

5.4.2. Pairing-Heaps⋆

Wie lässt sich die Laufzeit von *second-best* verbessern? Was haben wir für Optionen? Die Funktion geht konzeptionell in zwei Schritten vor (5.5): Im ersten Schritt wird der Verliererbaum in eine Liste von Warteschlangen überführt, die im zweiten Schritt von rechts nach links zu einer einzigen Warteschlange zusammengefasst werden.

```
second-best = spine ≫ right-to-left
```

Die Funktion *spine*, die den ersten Schritt implementiert, nimmt in Prinzip einen Repräsentationswechsel vor. Auf Grund der Invariante — die Verlierer dominieren die jeweils linke Turnierhälfte — ist ihre Definition allerdings in Stein gemeißelt.

```
// spine : Tree ⟨'a⟩ → (PQueue ⟨'a⟩) list
let rec spine = function
  | Leaf          → []
  | Node (t, a, u) → Min (a, t) :: spine u
```

Der Pfad von der Wurzel bis zu dem rechtesten Blatt wird auch bildhaft das rechte *Rückgrat* des Baums genannt (engl. right spine).

```
// right-to-left : (PQueue ⟨'a⟩) list → PQueue ⟨'a⟩ when 'a : comparison
let rec right-to-left = function
  | []      → empty
  | q :: qs → meld (q, right-to-left qs)
```

Die Funktion *right-to-left* ist die große Schwester von *from-list*: Statt einer Folge von Elementen verarbeitet sie eine Folge von Warteschlangen. (Die Implementierung von *from-list* auf Seite 237 lässt sich auf *right-to-left* zurückführen: *from-list* = *map single* ≫ *right-to-left*.) Die Definition von *right-to-left* ist alles andere als in Stein gemeißelt, hier können wir mit unseren Optimierungsbemühungen ansetzen. Eine Alternative haben wir bereits implementiert: die Funktion *tournament*, die wiederholt Warteschlangen paarweise kombiniert.

Wäre *tournament* die bessere Wahl? Vielleicht. Die Frage ist tatsächlich schwierig zu beantworten, da wir die Gestalt der zugrundeliegenden Verliererbäume nicht kennen. Sind alle Bäume gleich groß, dann ist *tournament* eine gute Wahl; sind die Bäume absteigend der Höhe nach geordnet, dann sollten wir *right-to-left* den Vorzug geben. Da wir aber im Dunkeln stochern, liegt die Idee nahe, die beiden Strategien zu kombinieren — wie so oft im Leben macht es die gesunde Mischung. (Die Form der Binärbäume zunächst zu analysieren ist übrigens keine Option — eine Analyse würde schlicht und einfach zu viel Zeit kosten.)

```
second-best = spine ≫ play-round ≫ right-to-left
```

Wir fügen einen zusätzlichen Transformationsschritt ein, in dem die Warteschlangen in einer Runde paarweise verschmolzen werden. Dieser zusätzliche Schritt gibt der Datenstruktur ihren Namen: engl. pairing heaps. Die Länge des Pfades von der Wurzel bis zu dem rechtesten Blatt bestimmt die Laufzeit von *second-best*; der Aufruf von *play-round* sorgt dafür, dass sich die Länge des Rückgrats mit jedem Aufruf ungefähr halbiert.

Die drei Schritte lassen sich zu einem zusammenfassen.

```
let rec second-best = function
  | Leaf                    → Inf
  | Node (t, a, Leaf)       → Min (a, t)
  | Node (t, a, Node (u, b, v)) → meld (meld (Min (a, t), Min (b, u)), second-best v)
```

Im Fachjargon sagt man auch, die drei Funktionen werden *fusioniert*. Fusion ist eine wichtige Programmoptimierung; in unserem Fall sparen wir uns die Erzeugung und Verarbeitung von zwei intermediären Listen. Damit ist die Implementierung von Pairing-Heaps abgeschlossen; alle anderen Operationen übernehmen wir unverändert.

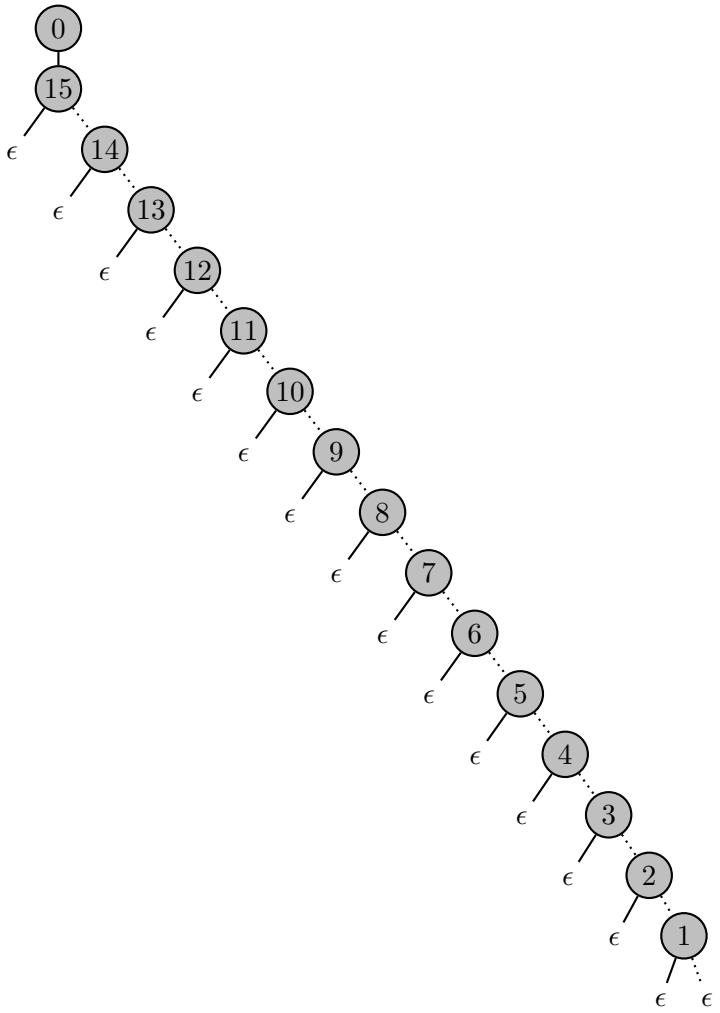


Abbildung 5.21.: Ein rechtsentarteter Pairing-Heap (*from-list* $[15\,..\,{-1}\,..\,0]$).

Es ist instruktiv, die neue Definition von *second-best* in Aktion zu sehen. Der Pairing-Heap in Abbildung 5.21 dient dabei als Ausgangspunkt. Alle Elemente im Verliererbaum befinden sich auf dem rechten Rückgrat — es liegt somit der schlechteste Fall vor. Wenn wir jetzt mit Hilfe von *split-min* die drei kleinsten Elemente entfernen, erhalten wir die in Abbildung 5.22 dargestellte Abfolge von Pairing-Heaps. Jeder Aufruf von *split-min* halbiert die Wegstrecke; nach der Entnahme von drei Elementen ist der zugrundeliegende Verliererbaum fast vollständig ausgeglichen. Man ist fast versucht zu sagen „auf wundersame Weise“, aber natürlich basiert das „Wunder“ ausschließlich auf unseren Rechenregeln. Zum Vergleich: Die Implementierung von *split-min* aus Abschnitt 5.4.1 verkürzt die Wegstrecke jeweils nur um einen Knoten; insbesondere wird die rechtsentartete Form nicht verändert.

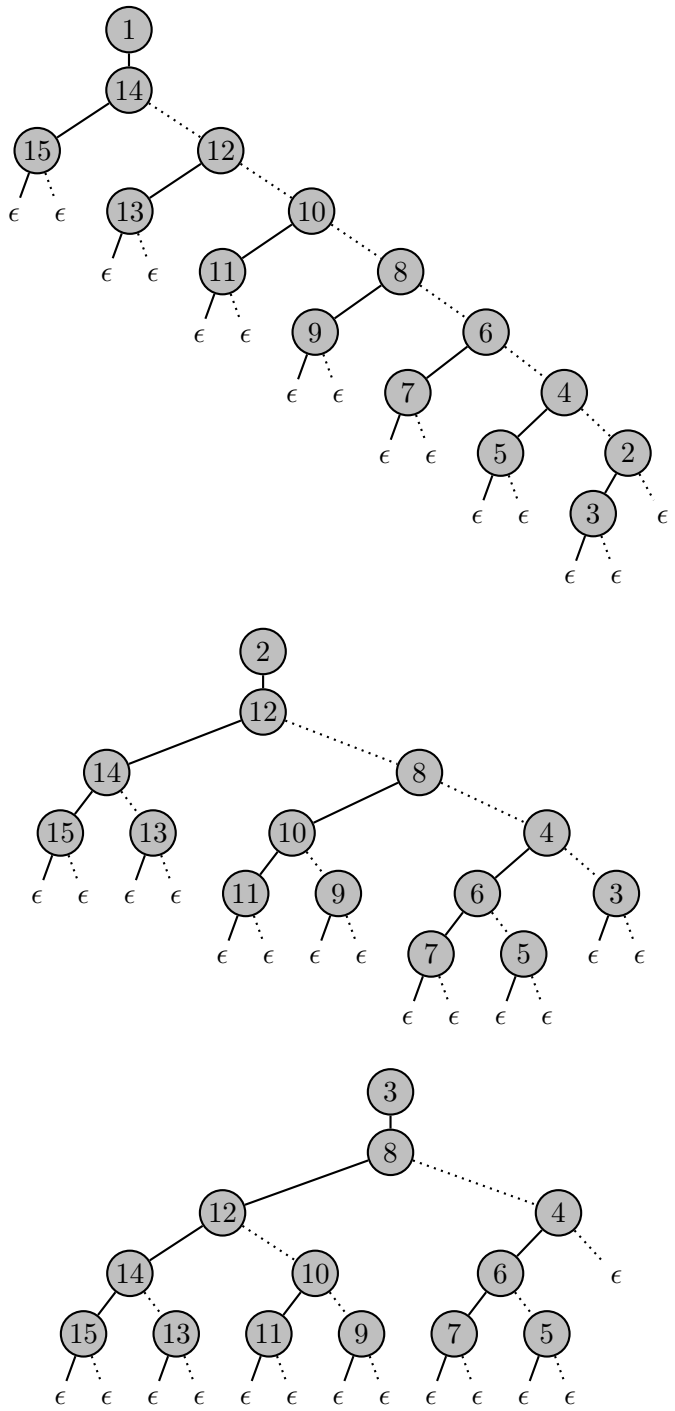


Abbildung 5.22.: Aus dem Pairing-Heap, siehe Abbildung 5.21, werden die drei kleinsten Elemente nacheinander entfernt.

Was haben wir erreicht? Zunächst einmal gilt es zu betonen, dass wir die Laufzeit von *split-min*, die im *schlechtesten* Fall auftritt, *nicht* verbessert haben. *Aber*, der schlechteste Fall wird nicht mehr so häufig auftreten, da die Bäume sich re-organisieren. Betrachtet man jetzt nicht isoliert die Laufzeit einer einzelnen Operation, sondern bestimmt die Laufzeit einer Folge von Operationen, so kann man zeigen, dass die über diese Folge *amortisierte* Laufzeit im Fall von *insert* und *meld* konstant und im Fall von *split-min* logarithmisch ist.[14] Die Analyse ist schwierig, da die Operationen beliebig geschachtelt werden können; zu schwierig für das erste Semester. Allerdings ist die Performanz von Pairing-Heaps zu gut und die Implementierung zu einfach, als dass wir sie hätten ignorieren wollen.

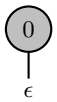
5.4.3. Binomial-Heaps⋆

Zurück zum Ausgangspunkt. Die Metapher des Tennisturniers hat sich als hilfreich erwiesen, aber tatsächlich hinkt sie etwas. Die Aufstellung eines Turnierbaums ist zumindest konzeptionell einfacher als die Verwaltung einer Prioritätswarteschlange. Die Teilnehmer eines Turniers stehen vor dem Turnierstart fest — in der Regel gibt es eine vorher festgelegte Zahl an Startplätzen, typischerweise eine exakte Potenz von 2, zum Beispiel 2^k. Ein Turnier ist also was die Struktur anbelangt eine eher *statische* Angelegenheit. Im Gegensatz dazu ist eine Prioritätswarteschlange ein *dynamisches* Gebilde: Elemente können peu à peu hinzugefügt werden; Minima können entfernt werden; Einfüge- und Auswahloperationen dürfen beliebig verschränkt werden.

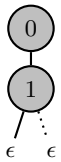
Das folgende Gedankenexperiment spinnt die Metapher etwas weiter mit dem Ziel, den dynamischen Aspekt einzufangen. Stellen wir uns vor, die Teilnehmer eines Turniers treffen zeitversetzt nacheinander ein. Wie können wir das Turnier trotz dieser widrigen Umstände organisieren? Insbesondere sollen die gleichen Spiele wie im statischen Fall ausgetragen werden, sobald 2^k Teilnehmer angekommen sind. Eine vielleicht naheliegende Idee ist, bei jeder Ankunft eines Teilnehmers, soviele Spiele wie möglich auszutragen. Es ergibt sich die folgende Abfolge von Spielen (um die Spiele einfach nachvollziehen zu können, entspricht die Spielstärke jeweils der Ankunftszeit):

Das Turnier beginnt.

Der erste Spieler kommt.

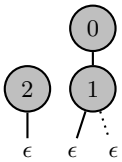


Der zweite Spieler kommt. Ein Spiel wird ausgetragen.

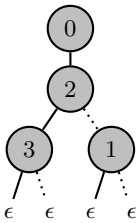


[14] In Wirklichkeit ist es noch komplizierter. Diese Aussage gilt nicht, wenn die *gleiche* Warteschlange als Eingabe für *mehrere* nachfolgende Operationen verwendet wird. Oder positiv ausgedrückt: Die Aussagen zur Laufzeit setzen voraus, dass jedes Zwischenergebnis höchstens einmal weiterverwendet wird. Man spricht in diesem Fall auch von einer *ephemeralen* Verwendung, im Gegensatz zu einer *persistenten* Verwendung.

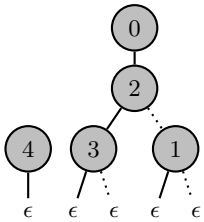
Der dritte Spieler kommt.



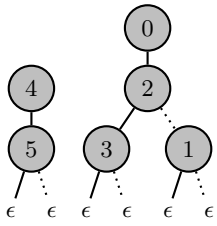
Der vierte Spieler kommt. Zwei Spiele werden ausgetragen.



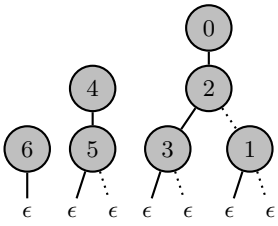
Der fünfte Spieler kommt.



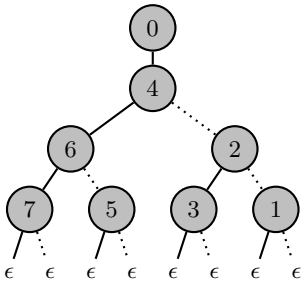
Der sechste Spieler kommt. Ein Spiel wird ausgetragen.



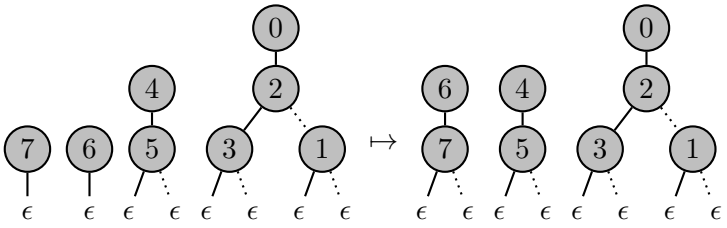
Der siebte Spieler kommt.

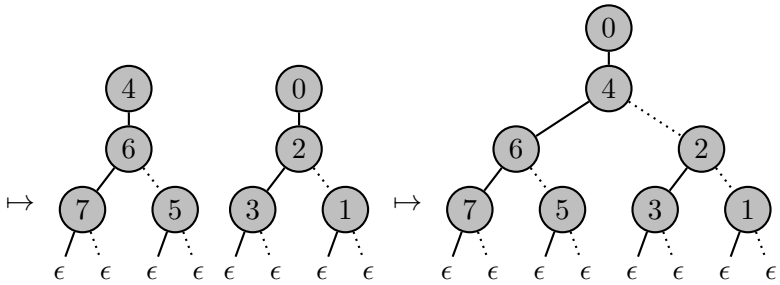


Der achte Spieler kommt. Drei Spiele werden ausgetragen.



Ein Spiel wird nur dann ausgetragen, wenn die beiden beteiligten Wimpel exakt die gleiche Form besitzen und damit auch die gleiche Anzahl von Elementen enthalten. Jedes Spiel verdoppelt somit die Zahl der Elemente, so dass die Größe jedes Wimpels durch eine exakte Potenz von 2 gegeben ist. Nachdem 7 Teilnehmer eingetroffen sind, erhalten wir zum Beispiel eine Folge von Wimpeln der Größen 2^0, 2^1 und 2^2, da $7 = 1 + 2 + 4 = 2^0 + 2^1 + 2^2$. Trifft der 8. Teilnehmer ein, ergibt sich die folgende Abfolge von Spielen:





Wenn n die Anzahl der Teilnehmer ist, dann wird die Anzahl der Wimpel und deren Größe durch die Binärrepräsentation von n festgelegt. Leibniz lässt grüßen!

Das Gedankenexperiment legt nahe, eine Warteschlange durch eine Folge von Wimpeln darzustellen. Mehr noch: Wir modellieren die Repräsentation und die dazugehörigen Operationen nach dem Binärsystem! Was für eine wundervolle Idee: Ein Zahlensystem dient als Blaupause für einen Containertyp.

```
type Bit ⟨'a⟩ =
  | Zero
  | One of Pennant ⟨'a⟩
type PQueue ⟨'a⟩ =
  | Rep of List ⟨Bit ⟨'a⟩⟩
```

Eine Warteschlange wird durch eine Liste von Binärziffern repräsentiert. Diese Datenstruktur hört auf den Namen *Binomial-Heap* (engl. binomial heap) — aus Gründen, auf die wir hier nicht näher eingehen wollen.

Das Binärsystem ist ebenso wie das uns vertraute Dezimalsystem ein *Stellenwertsystem*: Jede Ziffer hat abhängig von ihrer Position in einer Zahl eine festgelegte Wertigkeit. In einem Binomial-Heap gesellen sich zu den Ziffern Datenstrukturen: Hat die Ziffer d die Wertigkeit 2^i, dann wird der Ziffer ein Wimpel der Größe $d \cdot 2^i$ zur Seite gestellt. Fügen wir ein Element zu einem Binomial-Heap hinzu, dann wird die Binärzahl entsprechend inkrementiert (aus Gründen der Lesbarkeit kürzen wir *Leaf* und *Node* durch den ersten Buchstaben ab):

```
Mini⟩  from-list [4..-1..0]
Rep [One (4, L); Zero; One (0, N (N (L, 3, L), 2, N (L, 1, L)))]
Mini⟩  insert (5, it)
Rep [Zero; One (4, N (L, 5, L)); One (0, N (N (L, 3, L), 2, N (L, 1, L)))]
Mini⟩  insert (6, it)
Rep [One (6, L); One (4, N (L, 5, L)); One (0, N (N (L, 3, L), 2, N (L, 1, L)))]
Mini⟩  insert (7, it)
Rep [Zero; Zero; Zero; One (0, N (N (N (L, 7, L), 6, N (L, 5, L)), 4,
                                 N (N (L, 3, L), 2, N (L, 1, L))))]
```

Im letzten Schritt inkrementieren wir `111` und erhalten die Binärzahl `0001` — die niederwertigste Ziffer steht also links (nicht rechts wie im Dezimalsystem). Wir werden sehen, dass jeder Übertrag, `1 + 1 = 01`, der Austragung eines Spiels entspricht — aber wir greifen vor.

Ein Binomial-Heap enthält gewissermaßen die Repräsentation ihrer Größe. Diese lässt sich ausrechnen, indem wir die zugrundeliegende Binärzahl in eine natürliche Zahl umwandeln.

```
let rec size = function
  | []          → 0
  | Zero :: ps  → 0 + 2 * size ps
  | One _ :: ps → 1 + 2 * size ps
```

Wenden wir uns den Operationen auf Warteschlangen zu. Wie schon angedeutet, verfolgen wir die grundlegende Idee, die korrespondierenden Operationen auf Binärzahlen als Blaupause zu verwenden.

Zahlensystem	*Containertyp*
0	leerer Container
1	einelementiger Container
Nachfolgerfunktion$\setminus$Inkrement	Einfügen eines Elements
Addition	Vereinigung zweier Container

Die leere Warteschlange wird durch die leere Ziffernfolge repräsentiert; die einelementige Warteschlange entprechend durch die einelementige Ziffernfolge. Der Ziffer `1` bzw. *One* wird dabei der Wimpel (a, Leaf) zur Seite gestellt.

```
let empty = Rep []
let single a = Rep [One (a, Leaf)]
```

Die Nachfolgerfunktion erhöht ihr Argument um `1`. Auch dieser Eins muss ein Wimpel der entsprechenden Größe beigefügt werden. Aus diesem Grund erhält *succ* zwei Argumente: einen Wimpel für den impliziten Übertrag und einen Binomial-Heap.

```
let rec succ = function
  | (p, [])           → [One p]
  | (p, Zero :: ps)   → One p :: ps
  | (p, One q :: ps) → Zero :: succ (versus (p, q), ps)      // 1 + 1 = 01
let insert (a, Rep q) = Rep (succ ((a, Leaf), q))
```

Beim initialen Aufruf wird der Nachfolgerfunktion der Wimpel (a, Leaf) mitgegeben — der initiale Übertrag hat das Gewicht $2^0 = 1$. Dieser Wimpel wird in den ersten beiden Fällen der Ziffer `1` bzw. *One* zur Seite gestellt. Ergibt sich eine Übertrag, so tragen wir ein Spiel aus: *versus* (p, q). Dabei ist sichergestellt, dass die Ziffern das gleiche Gewicht besitzen und sich somit die Größe des Wimpels verdoppelt.

Der Schulalgorithmus für die Addition führt die Addition von Zahlen auf die Addition von Ziffer zurück. Dabei wird sichergestellt, dass die addierten Ziffern jeweils das gleiche Gewicht besitzen. Entsprechend führen wir die Vereinigung von Warteschlangen auf die Vereinigung von Wimpeln zurück.

```
let rec add = function
  | ([], ps) | (ps, [])          → ps
  | (Zero :: ps, Zero :: qs)     → Zero :: add (ps, qs)
  | (Zero :: ps, One p :: qs)
  | (One p :: ps, Zero :: qs)    → One p :: add (ps, qs)
  | (One p :: ps, One q :: qs) → Zero :: succ (versus (p, q), add (ps, qs))   // 1 + 1 = 01
let meld (Rep q1, Rep q2) = Rep (add (q1, q2))
```

Wenn beide Zahlen mit der Ziffer `1` anfangen, ergibt sich ein Übertrag, `1 + 1 = 01`, den wir mit Hilfe von *succ* hinzuaddieren.

```
let empty = Rep []
let single a = Rep [One (a, Leaf)]
let rec succ = function
    | (p, [])            → [One p]
    | (p, Zero :: ps)    → One p :: ps
    | (p, One q :: ps)   → Zero :: succ (versus (p, q), ps)
let insert (a, Rep q) = Rep (succ ((a, Leaf), q))
let rec add = function
    | ([], ps) | (ps, [])          →  ps
    | (Zero :: ps, Zero :: qs)     →  Zero :: add (ps, qs)
    | (Zero :: ps, One p :: qs)
    | (One p :: ps, Zero :: qs)    →  One p :: add (ps, qs)
    | (One p :: ps, One q :: qs)   →  Zero :: succ (versus (p, q), add (ps, qs))
let meld (Rep q1, Rep q2) = Rep (add (q1, q2))
let rec split-min-pennant = function
    | []           → None
    | Zero :: ps   → match split-min-pennant ps with
                       | None          → None
                       | Some (q, [])  → Some (q, [])
                       | Some (q, qs)  → Some (q, Zero :: qs)
    | One p :: ps  → match split-min-pennant ps with
                       | None          → Some (p, [])
                       | Some (q, qs)  → if fst p ⩽ fst q then Some (p, Zero :: ps)
                                         else  Some (q, One p :: qs)
let rec spine = function
    | Leaf           → []
    | Node (t, a, u) → (a, t) :: spine u
let split-min (Rep ps) =
  match split-min-pennant ps with
    | None               → None
    | Some ((a, t), qs)  → Some (a, Rep (add (map One (rev (spine t)), qs)))
```

Abbildung 5.23.: Implementierung von Binomial-Heaps.

Die Implementierung von *split-min* ist am aufwändigsten, siehe Abbildung 5.23. Da ein Binomial-Heap aus einer Liste von Wimpeln besteht, lässt sich das kleinste Element, der Turniersieger, nicht länger in konstanter Zeit bestimmen. Die Wimpelliste repräsentiert ja ein noch nicht vollständig ausgetragenes Turnier. Die Operation *split-min* geht konzeptionell in drei Schritten vor:

1. Der Binomial-Heap wird in den Wimpel mit der kleinsten Wurzel und dem restlichen Heap aufgespalten (*split-min-pennant*). Der extrahierte Wimpel wird dabei durch die Ziffer `0` ersetzt. Die Wurzel des extrahierten Wimpels ist das gesuchte minimale Element.

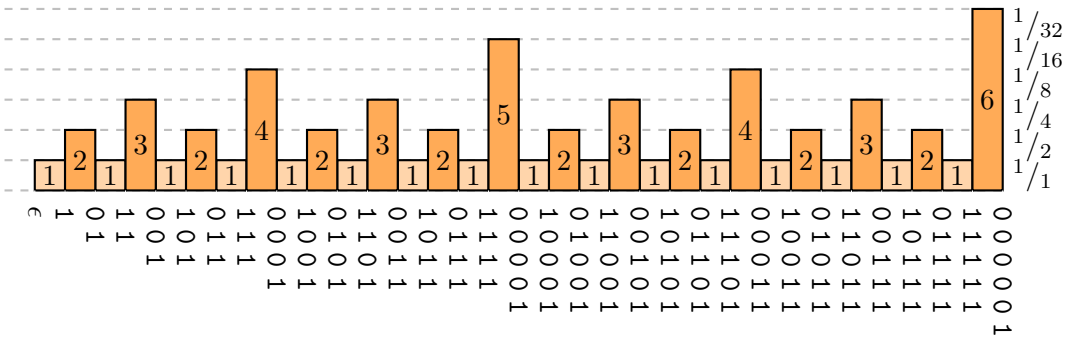
2. Der Verliererbaum des extrahierten Wimpels wird in eine Liste von Wimpeln überführt (*spine*). Dabei werden die Wimpel der Größe nach *absteigend* angeordnet. Wir erhalten einen Binomial-Heap, indem wir die Liste spiegeln (*rev*) und jeden Wimpel mit der Ziffer `1` „taggen“ (*map One*).

3. Die Binomial-Heaps aus den ersten beiden Schritten werden anschließend miteinander verschmolzen (*add*).

Ähnlich wie bei Pairing-Heaps wird *split-min* summa summarum auf *meld* zurückgeführt. Im Unterschied zu Pairing-Heaps wird aber eine Warteschlange nicht mehr durch einen einzigen Wimpel, sondern durch eine Liste von Wimpeln repräsentiert.

Amortisierte Analyse der Laufzeit Kommen wir zur Analyse der Laufzeit. Da Binomial-Heaps auf dem Binärsystem basieren, „vererben“ sich die Eigenschaften des Zahlensystems und seiner Operationen. Zur Erinnerung: Die Zahl n wird im Binärsystem durch $\lceil \lg(n+1) \rceil$ Binärziffern repräsentiert. Entsprechend besteht ein Binomial-Heap der Größe n aus maximal $\lceil \lg(n+1) \rceil$ Wimpeln. Damit ist klar, dass die Operationen *insert*, *meld* und *split-min* im schlechtesten Fall eine logarithmische Laufzeit besitzen. Gegenüber Pairing-Heaps hat sich die Laufzeit von *split-min* verbessert, allerdings auf Kosten der Laufzeit von *meld*.

Vielleicht überraschend ist die Tatsache, dass *insert* in *amortisiert konstanter* Zeit arbeitet. Im schlechtesten Fall ist die Laufzeit von *insert* logarithmisch, nämlich dann, wenn die Binärzahl aus einer Folge von Einsen besteht. In diesem Fall kaskadiert der Übertrag bis an das Ende der Ziffernfolge und hinterlässt dabei eine Folge von Nullen, gefolgt von einer Eins. Damit kann der schlechteste Fall nicht unmittelbar wieder auftreten. Ganz im Gegenteil: Die nächsten Einfügeoperationen sind sehr günstig. Mit anderen Worten, wenn wir die Laufzeit von n Einfügeoperationen mit $n \lg n$ abschätzen, ist das viel zu pessimistisch, da der schlechteste Fall nicht in jedem Schritt auftreten kann. Tatsächlich ergibt sich das folgende Bild:

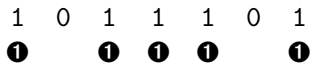


Wir zählen binär von 0 bis $n = 32$. Auf der y-Achse ist jeweils die Laufzeit der Nachfolgerfunktion aufgetragen. Die Gesamtlaufzeit von n Inkrements entspricht der Gesamtfläche aller Balken. Addieren wir die Flächen von unten nach oben auf, erhalten wir eine uns wohlbekannte Formel, siehe auch Abbildung 5.10.

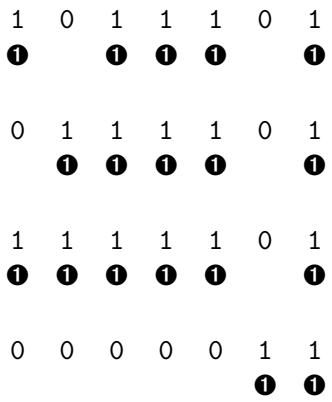
$$\frac{1}{1}n + \frac{1}{2}n + \frac{1}{4}n + \frac{1}{8}n + \cdots < 2 \cdot n$$

Amortisiert über eine Folge von Operationen ist die Laufzeit der Nachfolgerfunktion somit konstant — die Überlegungen gelten gleichermaßen für die Einfügeoperation *insert*. Einzelne „teure“ Operationen werden durch viele „günstige“ Operationen ausgeglichen.

Die Vorstellung, dass günstige für teure Operationen bezahlen, lässt sich präzisieren, indem wir Ziffern *gedanklich* mit Geldmünzen dekorieren:



Die Münzen repräsentieren Zeitguthaben (Momo und die Grauen Herren lassen grüßen), mit denen Rechenschritte bezahlt werden können. In unserem Fall ist die Ziffer `1` jeweils mit einer 1€ Münze dekoriert. Die Nachfolgerfunktion erhält diese *Invariante* und muss dafür 2€ aufwenden. Im Detail: Die Nachfolgerfunktion wandelt eine Folge von `1`en in `0`en um und eine `0` in eine `1`. Die erste Schritte (`1` → `0`) bezahlen wir mit den hinterlegten Münzen; für den letzten Schritt (`0` → `1`) müssen wir 1€ in Rechnung stellen. Zusätzlich müssen wir eine 1€ Münze hinterlegen, mit der zukünftige Operationen bezahlt werden.



Summa summarum enstehen für jeden Aufruf der Nachfolgerfunktion Kosten von 2€. Wir können die Operationen der Schnittstelle auch bunt mischen, zum Beispiel *insert* und *split-min* alternieren — die Analyse hat weiterhin Bestand.

Die vorgestellte Technik zur amortisierten Laufzeitanalyse heißt im Fachjargon *Bankkonto-Methode* (engl. banker's method). Die Methode basiert auf einer zentralen Annahme, die wir im letzten Abschnitt bereits zart angedeutet haben. Es wird vorausgesetzt, dass jedes Zwischenergebnis im weiteren Verlauf der Rechnung *höchstens einmal* weiterverwendet wird. Der Grund für diese Einschränkung ist vielleicht klar: Man kann jede Münze nur einmal ausgeben! Besteht eine Warteschlange aus einer Folge von Einsen und wir fügen in diese Warteschlange zwei verschiedene Elemente ein,

$$\begin{array}{l}\textbf{\textit{let}}\ q = \mathit{from\text{-}list}\ [1\,..\,1023] \\ \textbf{\textit{in}}\ (\mathit{insert}\ (815, q), \mathit{insert}\ (4711, q))\end{array}$$

dann können wir die Kosten der ersten Einfügeoperation aus den hinterlegten Geldmünzen begleichen, nicht aber die der zweiten Operation, oder gegebenenfalls einer dritten und vierten.

5.4.4. Laufzeitverhalten der Implementierungen

Ähnlich wie endliche Abbildungen lassen sich Prioritätswarteschlangen auf vielfältige Art und Weise implementieren. Die folgende Tabelle fasst das Laufzeitverhalten der vorgestellten Implementierungen zusammen.

	Wimpel	Pairing-Heaps	Binomial-Heaps
empty	1	1	1
single	1	1	1
insert	1	1	$1^\star$
meld	1	1	$\lg n$
split-min	n	$\lg n^\star$	$\lg n$
from-list	n	n	n
to-ord-list	$n \lg n$	$n \lg n$	$n \lg n$

* amortisierte Laufzeit

Pairing-Heaps unterscheiden sich von Wimpeln, getoppten Verliererbäumen, nur in der Implementierung der Operation *split-min*. Deren Laufzeit ist für beide Varianten im schlechtesten Fall linear; Pairing-Heaps garantieren allerdings eine *amortisiert* logarithmische Laufzeit. *Merkenswert:* Im Unterschied zu Suchbäumen lassen sich Prioritätswarteschlangen in linearer Zeit konstruieren.

Aber, wie schlagen sich die verschiedenen Implementierungen in der Praxis? Der folgende *Micro-Benchmark* vermittelt einen etwas oberflächlichen Eindruck. Wir verwenden Prioritätswarteschlangen, um verschiedene Listen der Länge 100000 zu sortieren.

Prioritätswarteschlange	aufsteigend	absteigend	zufällig	*from-list*
geordnete Listen	0.034	28:34.430	9:42.330	right-to-left
	0.290	0.270	0.430	bottom-up
Wimpel		stack overflow		right-to-left
	1.509	1.418	2.624	bottom-up
Pairing-Heaps	0.146	0.464	2.504	right-to-left
	0.413	0.371	2.556	bottom-up
Binomial-Heaps	1.310	1.329	4.164	right-to-left
	1.511	1.467	4.596	bottom-up
List.sortWith compare	0.018	0.019	0.029	

Die zu sortierenenden Listen sind entweder aufsteigend geordnet ($[1\,..\,100000]$), absteigend geordnet ($[100000\,..\,-1\,..\,1]$), oder bestehen aus zufällig ausgewählten Elementen. (Mit ***let*** *random* = *System.Random* (4711) erhält man einen Generator für sogenannte *Pseudozufallszahlen*. Der Ausdruck [***for*** *i* ***in*** $1\,..\,100000 \to$ *random.Next* ()] generiert die verwendete Liste von zufälligen Elementen; *random.Next* () erzeugt dabei bei jedem Aufruf eine neue Pseudozufallszahl, stellt also keine Funktion im mathematischen Sinne dar). Die Laufzeit wird in der obigen Tabelle jeweils in Sekunden angegeben.

Man spricht übrigens von einem Micro-Benchmark, da ein einigermaßen künstliches Anwendungsszenario nachgestellt wird, das nur einen bestimmten Aspekt beleuchtet. Prioritätswarteschlangen unterstützen ja beliebige Abfolgen von Einfüge- und Auswahloperationen; beim Sortieren fügen wir zunächst wiederholt ein, um dann wiederholt auszuwählen — wir mischen die Operationen also nicht. Pairing-Heaps zählen zu den besten Implementierungen von Prioritätswarteschlangen in der Praxis; die Spitzenposition wird auch von der Tabelle bestätigt. Geordnete Listen schneiden allerdings auch prima ab, zumindest wenn die Bottom-up Variante von *from-list* verwendet wird (Seite 232). Das ist auch nicht weiter verwunderlich; in diesem Fall entspricht das Sortierverfahren dem „Sortieren durch Mischen“, einem guten Sortieralgorithmus. Daraus lässt sich nicht folgern, dass geordnete Listen die Datenstruktur der Wahl für Prioritätswarteschlangen sind. Ganz im Gegenteil: Wird die right-to-left Implementierung von *from-list* eingesetzt (Seite 237), dann erhalten wir „Sortieren durch Einfügen“ — die Tabelleneinträge belegen die schlechte, da quadratische Laufzeit. Man sieht: Ein einzelner Micro-Benchmark kann in die Irre führen.

Übungen.

1. Die Definition von *split-min* ist nicht ganz zwingend. Die Alternative

$$\textbf{\textit{if}}\ x \leqslant m\ \textbf{\textit{then}}\ (x, xs)\ \textbf{\textit{else}}\ (m, x :: ys)$$

kann auch durch die symmetrische Formulierung

$$\textbf{\textit{if}}\ x \leqslant m\ \textbf{\textit{then}}\ (x, m :: ys)\ \textbf{\textit{else}}\ (m, x :: ys)$$

ersetzt werden. (Das resultierende Sortierverfahren hört auf den Namen „Bubble Sort“.) Ändert sich durch diese Modifikation die Laufzeit des Verfahrens?

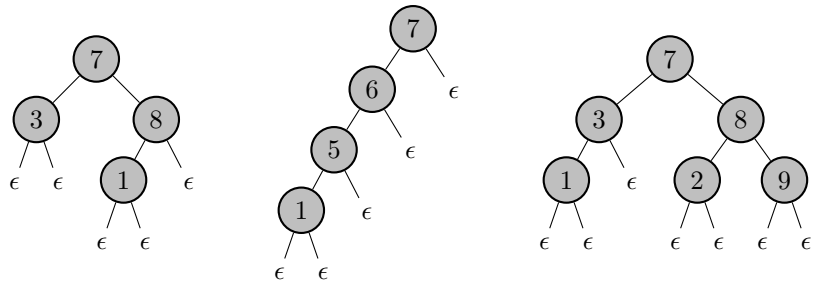
2. Ein Sortierverfahren heißt *stabil*, wenn es die relative Reihenfolge von äquivalenten Elemente nicht verändert: Wenn a in der Eingabe vor b auftritt und $a \sim b$ gilt, dann wird auch in der Ausgabe a vor b aufgeführt. (Stabilität ist eine sehr wünschenswerte Eigenschaft, da sie Vorsortierungen erhält: Wenn wir zum Beispiel eine Liste von Personen nach dem Nachnamen sortieren, die Liste aber bereits nach dem Vornamen sortiert ist, dann werden in der Ausgabe Personen mit gleichem Nachnamen nach dem Vornamen angeordnet.) Welche Sortierverfahren sind stabil, welche nicht? Lassen sich letztere gegebenenfalls „stabilisieren“?

3. Ein *Lauf* ist eine sortierte Teilliste. Schreiben Sie eine Sortierfunktion *bottom-up-merge-sort*, die auf dem wiederholten Mischen von Läufen basiert. Gehen Sie dabei in vier Schritten vor:

1. Schreiben Sie eine Funktion *pair-merge*, die eine Liste von n Läufen in eine Liste von $\lceil n/2 \rceil$ Läufen überführt, indem sie jeweils zwei benachbarte Läufe mischt: den 1. Lauf mit dem 2., den 3. mit dem 4. und so weiter.
2. Definieren Sie eine Funktion *merge-runs*, die eine Liste von Läufen in einen einzigen Lauf überführt, indem sie *pair-merge* so oft aufruft, bis die Liste von Läufen nur noch einen einzigen Lauf enthält.
3. Implementieren Sie eine Funktion *runs*, die eine ungeordnete Liste in eine Liste von Läufen überführt. Versuchen Sie in der Eingabeliste vorhandene Läufe so weit wie möglich zu verwenden.
4. Definieren sie *bottom-up-merge-sort* als Komposition von *runs* und *merge-runs*.

Analysieren Sie die Laufzeit des Sortierverfahrens. Wann ist es dem eng verwandten Verfahren aus Abschnitt 5.1.2 vorzuziehen?

4. 1. Geben Sie zu den folgenden grafischen Darstellungen von Bäumen die zugehörigen Ausdrücke an:



2. Zeichnen Sie zu den folgenden Ausdrücken die zugehörigen Bäume:

   $\mathit{Node}\ (\mathit{Leaf}, 5, \mathit{Leaf})$
   $\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}, 3, \mathit{Leaf}), 9, \mathit{Node}\ (\mathit{Leaf}, 8, \mathit{Leaf}))$
   $\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}, 1, \mathit{Node}\ (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}, 2, \mathit{Leaf}), 3, \mathit{Leaf}), 4, \mathit{Leaf})), 5, \mathit{Leaf})$

5. Programmieren Sie in Analogie zur Funktion *inorder* Funktionen, die einen Baum in Preorder- bzw. Postorder-Reihenfolge in eine Liste überführen. *Zur Erinnerung:* Inorder-Reihenfolge bedeutet, dass das Wurzelelement *zwischen* die Elemente des linken und des rechten Teilbaums wandert. Beim Preorder-Durchlauf wird das Wurzelelement vorangestellt und beim Postorder-Durchlauf hinten angehängt.

$\mathbf{let}\ \mathit{preorder}\ \ (\mathit{tree} : \mathit{Tree}\ \langle' a\rangle) : \mathit{List}\ \langle' a\rangle$
$\mathbf{let}\ \mathit{postorder}\ (\mathit{tree} : \mathit{Tree}\ \langle' a\rangle) : \mathit{List}\ \langle' a\rangle$

Lassen sich die ursprünglichen Bäume aus den resultierenden Listen rekonstruieren? Mit anderen Worten, sind die Funktionen injektiv?

6. Programmieren Sie eine Funktion

$$\textbf{\textit{let}}\ \mathit{tree\text{-}sort}\ (\mathit{list} : \mathit{List}\ \langle \mathit{Nat} \rangle) : \mathit{List}\ \langle \mathit{Nat} \rangle$$

die eine Liste sortiert. Gehen Sie dabei in drei Schritten vor:

1. Definieren Sie zunächst eine Funktion

   $$\textbf{\textit{let}}\ \mathit{insert}\ (n : \mathit{Nat}, \mathit{tree} : \mathit{Tree}\ \langle \mathit{Nat} \rangle) : \mathit{Tree}\ \langle \mathit{Nat} \rangle$$

   die ein Element in einen *Suchbaum* einfügt, so dass die Suchbaumeigenschaft erhalten bleibt.
2. Schreiben Sie mit Hilfe von *insert* eine Funktion *search-tree*, die aus einer Liste einen Suchbaum konstruiert.
3. Definieren sie *tree-sort* als Komposition von *build* und *inorder*. *Zur Erinnerung:* Die Funktion *inorder* überführt einen Suchbaum in eine geordnete Liste.

7. Ein Heap ist ein Binärbaum, der folgende Bedingung erfüllt: Das Element jedes Knotens ist kleiner oder gleich der Elemente seiner Teilbäume. Somit enthält die Wurzel des Heaps stets das kleinste Element. (Die folgende Definition führt ein Typsynonym ein: Der Bezeichner auf der linken Seite ist eine Abkürzung für den Typausdruck auf der rechten Seite.)

$$\textbf{\textit{type}}\ \mathit{Heap} = \mathit{Tree}\ \langle \mathit{Nat} \rangle$$

Schreiben Sie eine Funktion

$$\textbf{\textit{let}}\ \mathit{insert}\ (n : \mathit{Nat}, \mathit{heap} : \mathit{Heap}) : \mathit{Heap}$$

die ein Element in einen Heap einfügt, so dass die Heapeigenschaft erhalten bleibt.

8. Harry Hacker behauptet eine Funktion definiert zu haben, die einen Binärbaum der Größe n in logarithmischer (!) Zeit konstruiert. Seine Funktion erfüllt die Eigenschaft

$$\mathit{size}\ (\mathit{create}\ n) = n$$

wobei *size* wie folgt definiert ist:

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{size} = \textbf{\textit{function}} \\ \quad \mid \mathit{Leaf} \qquad\quad \rightarrow 0 \\ \quad \mid \mathit{Node}\ (l, a, r) \rightarrow \mathit{size}\ l + 1 + \mathit{size}\ r\end{array}$$

Genie oder Scharlatan?

9. Weihnachten steht vor der Tür und Lisa Lista und Harry Hacker haben beschlossen, gemeinsam einen Weihnachtsbaum zu schmücken. Für die Repräsentation des Baumes haben sie sich folgende Typdeklaration überlegt:

$$\textbf{\textit{type}}\ \mathit{Christmas\text{-}Tree} = \mid \mathit{Twig} \mid \mathit{Bauble} \mid \mathit{Fork}\ \textbf{\textit{of}}\ \mathit{Christmas\text{-}Tree} * \mathit{Christmas\text{-}Tree}$$

Der Konstruktor *Twig* steht für einen leeren Ast des Weihnachtsbaumes; der Konstruktor *Bauble* beschreibt einen Ast des Baumes, der mit einer Christbaumkugel geschmückt ist. Schreiben Sie eine Funktion *decorate*, die einen beliebigen leeren Ast mit einer Christbaumkugel schmückt.

$$\textbf{\textit{let}}\ \mathit{decorate}\ (\mathit{tree} : \mathit{Christmas\text{-}Tree}) : \mathit{Christmas\text{-}Tree}$$

Wird der gesamte Baum geschmückt, wenn die Funktion wiederholt aufgerufen wird?

10. Erweitern Sie die Schnittstelle für endliche Abbildungen um eine Funktion, die den Kommaoperator implementiert. Realisieren Sie die Operation für Listen, Suchlisten und Suchbäume und analysieren Sie jeweils die Laufzeit.

11. Eine Menge von natürlichen Zahlen kann durch eine geordnete Liste repräsentiert werden, in der kein Element doppelt vorkommt (Suchliste ohne Duplikate).

$$\textbf{\textit{type}}\ \mathit{Set} = \mathit{List}\,\langle \mathit{Nat} \rangle$$

Definieren Sie auf dieser Repräsentation die Vereinigung, den Durchschnitt und die Differenz von Mengen.

$$\begin{array}{ll}
\textbf{\textit{let}}\ \mathit{union} & (\mathit{set}_1 : \mathit{Set}, \mathit{set}_2 : \mathit{Set}) : \mathit{Set} \\
\textbf{\textit{let}}\ \mathit{intersection} & (\mathit{set}_1 : \mathit{Set}, \mathit{set}_2 : \mathit{Set}) : \mathit{Set} \\
\textbf{\textit{let}}\ \mathit{difference} & (\mathit{set}_1 : \mathit{Set}, \mathit{set}_2 : \mathit{Set}) : \mathit{Set}
\end{array}$$

Wie ändert sich die Definition der Funktionen, wenn wir statt Mengen von natürlichen Zahlen Mengen von ganzen Zahlen betrachten (vergleiche Aufgabe 4.5)?

6. Grammatiken ⧹ Konkrete Syntax

In den letzten drei Kapiteln haben wir uns angeschaut, wie man mit der Sprache Mini–F# Probleme löst: Wie man Probleme in Rechenaufgaben verwandelt und wie man Rechenregeln in Mini–F# formuliert, um diese Rechenaufgaben zu lösen. Aber wir haben nicht nur *mit* Mini–F# programmiert, wir haben auch *über* die Sprache gesprochen; wir haben die abstrakte Syntax und die Bedeutung zahlreicher Konstrukte präzise festgelegt.

In diesem Kapitel wenden wir uns der *konkreten Syntax* zu. Wir werden Formalismen kennenlernen, mit denen man die konkrete Syntax einer Sprache — und nicht nur die einer Programmiersprache — ebenso präzise wie die abstrakte Syntax definieren kann. Unsere Programmiersprache wird uns dabei als fortlaufendes Beispiel dienen.

Zur Erinnerung: Die abstrakte Syntax ist durch eine Baumsprache gegeben. Sie beschreibt den hierarchischen Aufbau eines Programms, etwa dass der Mini–F# Ausdruck für die Alternative aus drei Teilausdrücken besteht. Die konkrete Syntax beschreibt im Gegensatz dazu, welche Folgen von Zeichen ein Lexem und welche Folgen von Lexemen ein gültiges Programm darstellen, etwa dass `i` gefolgt von `f` ein Schlüsselwort ist und dass das Schlüsselwort ***if*** gefolgt von einem Ausdruck, gefolgt von dem Schlüsselwort ***then***, gefolgt von einem Ausdruck, gefolgt von dem Schlüsselwort ***else***, gefolgt von einem Ausdruck ebenfalls ein Ausdruck ist. Die konkrete Syntax legt also fest, wie ein Programm, das wir einem Rechner vorlegen wollen, konkret aussehen muss. Kurz: Die abstrakte Syntax beschreibt Bäume, die konkrete Syntax Folgen von Zeichen bzw. Lexemen.

Es ist absehbar, dass die Bedeutung der konkreten Syntax zurückgeht. Entwicklungen wie Strukturediteren, die direkt die Struktur eines Programms erfassen, integrierte Entwicklungsumgebungen (engl. IDE) oder visuelle Programmiersprachen machen ihr den Garaus. Nichtdestotrotz sind die Formalismen, mit denen man die konkrete Syntax beschreibt, bedeutsam. Neben vielfältigen Anwendungsmöglichkeiten jenseits der Programmiersprachen erzählen sie auch eine der großen Erfolgsgeschichten der Informatik: Wie man automatisch von einer deskriptiven Beschreibung zu einem ausführbaren Programm kommt, das diese Beschreibung umsetzt. Der Traum der Informatik: Ich beschreibe „was“ ein Programm leisten soll, das „wie“ erledigt der Rechner, dazu später mehr in Abschnitt 6.2.

Im Einzelnen haben wir folgendes vor: Abschnitt 6.1 beschäftigt sich mit der Beschreibung der lexikalischen Syntax einer Programmiersprache. Abschnitt 6.2 detailliert den Weg von der Deskription zur Präskription, von der Beschreibung zur Rechenvorschrift. Abschnitt 6.3 erweitert den Formalismus, so dass die konkrete Syntax von Programmiersprachen beschrieben werden kann. Fast alle Programmiersprachen haben Infixoperatoren im Angebot, Operatoren, die zwischen ihre Argumente geschrieben werden. Mit dem Sinn und Zweck von Operatoren und den damit verbundenen Problemen beschäftigt sich

ebenfalls Abschnitt 6.3. Abschnitt 6.4 zeigt schließlich, wie konkrete Syntax automatisch in abstrakte Syntax überführt werden kann.

6.1. Reguläre Ausdrücke⁄lexikalische Syntax

Eine natürliche Zahl wird in Mini–F# durch ein *Numeral* repräsentiert; umgekehrt bezeichnet ein Numeral eine natürliche Zahl. Kurz: Numeral ist Syntax, Zahl ist Semantik. Ein Numeral in Mini–F# besteht aus einer nicht-leeren Folge von Dezimalziffern. Ein Bezeichner in Mini–F# beginnt mit einem Buchstaben, gefolgt von weiteren Buchstaben, Ziffern und Sonderzeichen, wie dem Unterstrich oder dem Apostroph.

Wie können wir die lexikalische Syntax von Numeralen und Bezeichnern und den anderen syntaktischen Bestandteilen von Mini–F# formal beschreiben? Nun, wir müssen uns eine Sprache ausdenken — eine Sprache, um Sprachen zu beschreiben. Bevor wir uns dieser Aufgabe zuwenden, klären wir zunächst, was eine Sprache überhaupt ist.

Ist A eine Menge von Zeichen, ein sogenanntes *Alphabet*, dann ist eine *Sprache* eine Teilmenge von A^*, der Menge der Sequenzen über A — statt von Sequenzen spricht man in diesem Zusammenhang auch oft von *Wörtern*. Die Menge aller Sprachen ist $\mathbb{P}(A^*)$, die *Potenzmenge* von A^*. Ist A zum Beispiel das ASCII-Alphabet[1], dann kann eine Sprache über A die lexikalische Syntax beschreiben, die Menge aller Lexeme. Ist A hingegen die Menge aller Lexeme, dann kann eine Sprache die kontextfreie Syntax einer Programmiersprache festlegen (den Zusatz „kontextfrei“ erklären wir in Abschnitt 6.3).

Kommen wir zur Sprachbeschreibungssprache. In Abschnitt 2.1 haben wir einige Operationen auf Sequenzen eingeführt: die leere Sequenz, die Konkatenation und die Wiederholung. Ziehen wir diese Konstrukte heran, dann können wir bereits Numerale beschreiben. Ist *digit* eine Abkürzung für die Sprache der Ziffern, dann beschreibt

digit digit*

die Syntax von Numeralen. Lies: Ein Numeral ist eine Ziffer gefolgt von einer beliebigen Wiederholung von Ziffern, notiert *digit**. Wie können wir *digit* definieren? Eine Ziffer ist entweder `0`, oder `1`, oder `2`, oder …. Erfinden wir eine Notation für die Alternative, zum Beispiel ‘|’, dann ist

`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

die gesuchte Definition von *digit*. Die lexikalische Syntax von Bezeichnern lässt sich auf ähnliche Art und Weise festlegen:

letter (*letter* | `'` | `_`)*

wobei *letter* durch `A` | ··· | `Z` | `a` | ··· | `z` gegeben ist.

[1] ASCII steht für American Standard Code for Information Interchange — ist also eigentlich nichts für Europäer — und stellt eine Zeichenkodierung dar, eine Zuordnung von Zeichen zu Zahlen. Die ASCII Zeichenkodierung definiert 33 nicht-druckbare, sowie die folgenden 95 druckbaren Zeichen, beginnend mit dem Leerzeichen: ␣`!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~`.

Jetzt da wir eine erste Vorstellung von der Sprachbeschreibungssprache haben, können wir unser Routineprogramm abspulen. Welches Routineprogramm? Nun, wir definieren die abstrakte Syntax und die Semantik der Sprachbeschreibungssprache. Das mag etwas merkwürdig anmuten — wir wollen ja die neue Sprache gerade dazu verwenden, die Syntax anderer Sprachen zu beschreiben, und jetzt beabsichtigen wir die Syntax der Sprache selbst festzulegen. Aber die Sprachbeschreibungssprache ist nun einmal eine Sprache und als solche müssen wir uns um deren Syntax und Semantik kümmern.

6.1.1. Syntax regulärer Ausdrücke

Abstrakte Syntax Sei A ein beliebiges *Alphabet*, eine Menge von Zeichen, Symbolen oder Buchstaben. (Die Begriffe sind sehr allgemein zu verstehen: Ein Alphabet ist einfach eine Menge, zum Beispiel $\{\mathtt{0}, \mathtt{1}\}$; ein Buchstabe ist einfach ein Element der Menge.) Die abstrakte Syntax der sogenannten *regulären Ausdrücke* über A ist durch die folgende Baumsprache gegeben.

$a \in A$	// *Alphabet*
$r \in \mathsf{Reg} ::=$	// *reguläre Ausdrücke:*
$\mid \epsilon$	// das leere Wort
$\mid a$	// einzelnes Zeichen∖Terminalsymbol
$\mid r_1\ r_2$	// Konkatenation∖Sequenz
$\mid \emptyset$	// die leere Sprache
$\mid r_1 \mid r_2$	// Alternative
$\mid r^*$	// Wiederholung

Die Elemente des Alphabets, auch *Terminalsymbole* genannt, sind sozusagen die Konstanten der Sprache. Die Notation regulärer Ausdrücke ist eng an die Notation von Sequenzen angelehnt. Man sollte sich aber klarmachen, dass die obige Baumsprache die *Syntax* regulärer Ausdrücke beschreibt. Zum Beispiel ist $\emptyset$ nicht die leere Menge, sondern ein regulärer Ausdruck. Das Symbol $\emptyset$ ist *überladen*: Je nach Kontext bezeichnet es die leere Menge oder einen regulären Ausdruck, vergleiche die Diskussion am Ende von Abschnitt 2.4. Ebenso sind die Terminalsymbole überladen: a ist entweder ein einzelner Buchstabe oder ein regulärer Ausdruck. (Das Phänomen kennen wir von Booleschen und arithmetischen Ausdrücken: *true* ist sowohl ein Boolescher Wert als auch ein Boolescher Ausdruck; 4711 ist sowohl eine natürliche Zahl als auch ein arithmetischer Ausdruck.) Die Konkatenation oder Sequenz von regulären Ausdrücken wird sozusagen „ohne Syntax“ notiert, indem die Ausdrücke einfach hintereinander geschrieben werden. Damit folgen wir einer guten (?), alten mathematischen Tradition: Multiplikative Operatoren werden oft ausgelassen; aus $2 \cdot x + 1$ wird $2x + 1$; aus $a \wedge b \vee c$ wird $ab \vee c$.

Konkrete Syntax Auch reguläre Ausdrücke haben eine konkrete Syntax. Wir entwerfen hier keine eigene, sondern schauen uns zwei praktische Beispiele für Syntaxen an, siehe Tabelle 6.1. Kurz zum Hintergrund: Die Programme `grep` und `egrep` durchsuchen Sequenzen (typischerweise Textdateien) nach zusammenhängenden Teilsequenzen, die

	Abstrakte Syntax	`grep`	`egrep`
das leere Wort	ϵ	–	–
einzelnes Zeichen	a	a	a
Konkatenation	$r_1\ r_2$	$r_1\ r_2$	$r_1\ r_2$
die leere Sprache	$\emptyset$	–	–
Alternative	$r_1 \mid r_2$	r_1 `|` r_2	r_1 `|` r_2
Wiederholung	r^*	r`\*`	r`*`
Gruppierung	–	`\(` r `\)`	`(` r `)`
beliebiges Zeichen (außer Zeilenende)	$a_1 \mid \cdots \mid a_n$	`.`	`.`
optionales Vorkommen	$\epsilon \mid r$	r`\?`	r`?`
mindestens einmalige Wiederholung	$r\ r^*$	r`\+`	r`+`

Abbildung 6.1.: Konkrete Syntax regulärer Ausdrücke.

auf ein bestimmtes, vom Benutzer vorgegebenes Muster passen. Das Muster ist dabei durch einen regulären Ausdruck gegeben. Eine Teilsequenz passt auf ein Muster, wenn die Sequenz in der durch das Muster beschriebenen Sprache enthalten ist.

Die prinzipielle Schwierigkeit, mit der man sich beim Entwurf einer konkreten Syntax konfrontiert sieht, liegt darin, Terminalsymbole von Metasymbolen (`|` oder `*`) zu unterscheiden. Beide Symbole entstammen ja dem gleichen Alphabet, da die lexikalische Syntax regulärer Ausdrücke festgelegt wird. Ein einfacher Trick, dieser Schwierigkeit zu begegnen, ist, ein sogenanntes *Fluchtsymbol* (engl. escape symbol) einzuführen, das die jeweilige Kategorie, Terminal- oder Metasymbol, signalisiert. In den Beispielen, Tabelle 6.1, ist `\` (engl. backslash) das Fluchtsymbol. In `grep` kennzeichnet er die Metasymbole: `*` ist ein Terminalsymbol, `\*` ist ein Metasymbol. In `egrep` ist es genau umgekehrt: `\*` ist das Terminal- und `*` das Metasymbol.

Das Programm `egrep` bietet weiterhin einige bequeme Abkürzungen an: Die lexikalische Syntax von Numeralen kann zum Beispiel durch `[0-9]+` festgelegt werden; entsprechend beschreibt `[A-Za-z][A-Za-z'_]*` die Syntax von Bezeichnern. Der folgende Aufruf von `egrep` durchsucht das Skript nach gültigen Email-Adressen.[2]

```
> egrep "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}" Skript.lhs
"\nPlease report it as a bug to support@harry-hacker.org.")
```

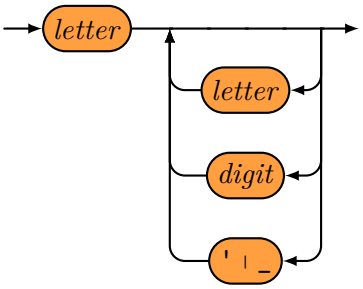
Das Programm wird mit zwei Argumenten aufgerufen, dem in Anführungsstriche gesetzten regulären Ausdruck und dem Namen einer Datei. Der reguläre Ausdruck enthält drei Punkte, die jeweils für das Terminalsymbol, das Zeichen „.“, stehen. Da der Punkt von Haus aus ein Metasymbol für ein beliebiges Zeichen ist, muss das Fluchtsymbol vorangestellt werden. Innerhalb der eckigen Klammern ist das nicht nötig. Aber: Dort ist

[2] Die Ausgabe bezieht sich auf die Version des Skripts *bevor* die Ausgabe zum Text hinzugefügt wurde.

„-“ ein Metasymbol; meint man das Terminalsymbol, den Bindestrich, muss dieser als letztes, direkt vor der schließenden Klammer aufgeführt werden.

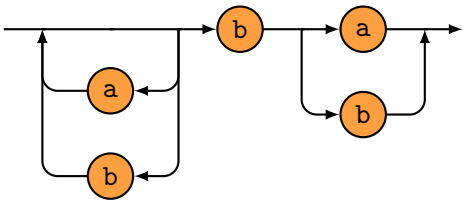
Man sieht, die konkrete Syntax zu entwerfen, ist keine ganz leichte Angelegenheit. Deswegen ist unser Ausgangspunkt allermeistens die abstrakte Syntax. Entgegen vorherrschender (Lehr-) Meinung ist ein abstraktes Konzept nicht schwieriger als ein konkretes. Abstraktion bedeutet ja Vernachlässigung irrelevanter, uninteressanter oder zufälliger Details. Deswegen abstrahieren Informatiker*innen und Mathematiker*innen auch so gerne; es macht das Leben leichter.

Syntaxdiagramme Reguläre Ausdrücke lassen sich wunderbar mit Hilfe sogenannter *Syntaxdiagramme* (engl. syntax diagrams) bzw. *Eisenbahndiagramme* (engl. railroad diagrams) darstellen. Die lexikalische Syntax von Bezeichnern wird zum Beispiel durch das folgende Diagramm eingefangen.



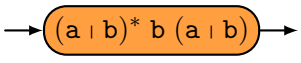
Das Diagramm hat einen „Eingang“ auf der linken Seite, den Startpunkt, und einen „Ausgang“ auf der rechten Seite, den Zielpunkt. Indem man mit dem Finger von Start- zum Zielpunkt fährt, lassen sich systematisch alle Wörter der Sprache generieren.

Die Konkatenation wird dabei durch hintereinander geschaltete Schienenstränge visualisiert, die Alternative durch parallel geschaltete Schienenstränge. Wiederholungen werden durch „Loopings“ oder Schleifen dargestellt, wie im folgenden Diagramm.



Das Syntaxdiagramm generiert die Sprache aller Wörter über dem Alphabet $\{\texttt{a}, \texttt{b}\}$, die an der vorletzten Position ein `b` haben. Um zum Beispiel das Wort `ababb` zu generieren, drehen wir uns dreimal im Kreis, fahren dann nach rechts und dann nach rechts unten.

Abbildung 6.2 illustriert, wie man für einen gegebenen regulären Ausdruck systematisch das korrespondierende Syntaxdiagramm konstruiert. Ausgangspunkt ist das Diagramm, dessen einziger „Bahnhof“ den regulären Ausdruck enthält.



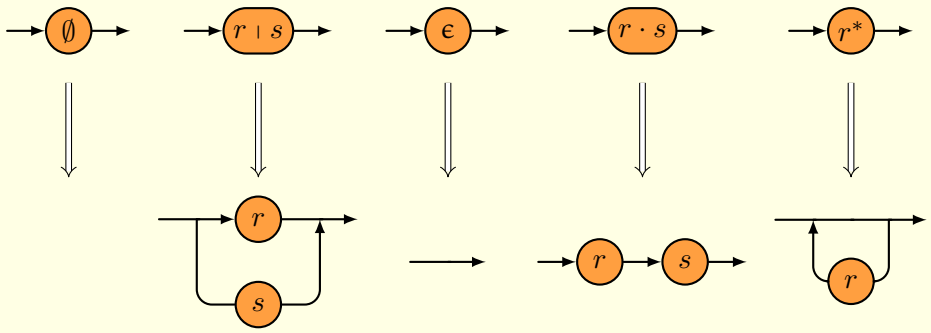
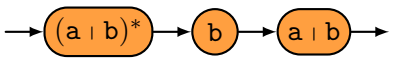
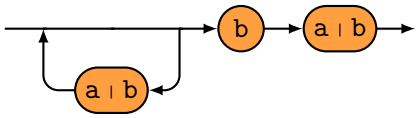


Abbildung 6.2.: Konstruktion von Syntaxdiagrammen.

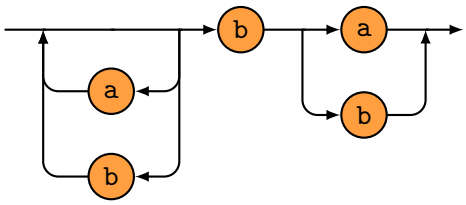
Die Transformationen werden von oben nach unten angewendet; ein Bahnhof wird dabei jeweils durch ein kleines Diagramm ersetzt. In unserem Beispiel expandieren wir gemäß des Aufbaus des regulären Ausdrucks zunächst die Konkatenation.



Anschließend wird die Wiederholung durch einen Looping ersetzt.



Wenn wir schließlich die Alternativen durch parallele Schienenstränge visualisieren, erhalten wir das ursprüngliche Diagramm.



Man kann die Transformationen solange anwenden, bis nur noch mit Buchstaben markierte Bahnhöfe übrig sind. Manchmal hört man aus Gründen der Übersichtlichkeit etwas früher auf, so geschehen im ersten Beispiel, dem Diagramm für Bezeichner.

6.1.2. Semantik regulärer Ausdrücke

In den Kapiteln 3 und 4 haben wir zwischen statischer und dynamischer Semantik unterschieden. Das ist hier nicht notwendig: Reguläre Ausdrücke sind beliebig, ohne jedwede Einschränkungen kombinierbar.

Ein regulärer Ausdruck bezeichnet eine Sprache, eine Teilmenge von A^*. Wir werden zwei ganz verschiedene Ansätze vorstellen, die Semantik regulärer Ausdrücke festzulegen. Der erste Ansatz ordnet einem regulären Ausdruck direkt eine Sprache zu. Der zweite zeigt, wie sich aus einem regulären Ausdruck ein Wort ableiten lässt; die Menge aller ableitbaren Worte ist dann die bezeichnete Sprache.

Denotationelle Semantik Kommen wir zum ersten Ansatz. Dieser heißt *mathematische* oder *denotationelle Semantik*, da einem syntaktischen Objekt, in unserem Fall ein regulärer Ausdruck, ein mathematisches Objekt, in unserem Fall eine Sprache, zugeordnet wird. Das mathematische Objekt ist die Bedeutung oder im Fachjargon die *Denotation* des syntaktischen Objekts.

Was ist die Bedeutung von ϵ oder eines Terminalsymbols? Nun, der reguläre Ausdruck ϵ bezeichnet die Sprache $\{\epsilon\}$ und a entsprechend $\{a\}$, wobei a die einelementige Sequenz $\{0 \mapsto a\}$ abkürzt. (Wer überrascht ist, dass die Bedeutung von ϵ die Menge $\{\epsilon\}$ und nicht etwa ϵ selbst ist, der sei daran erinnert, dass die Bedeutung eines regulären Ausdrucks eine Sprache ist, kein Wort.) Das nächste Konstrukt ist die Konkatenation $r_1\ r_2$. Wir haben die Konkatenation von Wörtern, sprich Sequenzen, bereits eingeführt. Jetzt müssen wir aber zwei Mengen konkatenieren: r_1 und r_2 bezeichnen zwei Sprachen, sagen wir L_1 und L_2, diese müssen verknüpft werden. Die Konkatenation von Sprachen lässt sich einfach auf die Konkatenation von Wörtern zurückführen, indem man jedes Wort der einen mit jedem Wort der anderen Menge konkateniert.

$$L_1 \cdot L_2 = \{\, w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2 \,\}$$

Das Symbol '$\cdot$' ist jetzt *überladen* (genau wie $\emptyset$ und 0 und + und …): Es wird sowohl für die Konkatenation von Wörtern als auch für die Konkatenation von Sprachen verwendet. Der Zusammenhang ist allerdings ein direkter: Man sagt, die Operation oder Funktion '$\cdot$' wird auf Mengen *fortgesetzt* und drückt damit aus, dass die obige Definition die naheliegende oder im Fachjargon die *kanonische* Wahl ist.

Die n-fache Konkatenation einer Sprache mit sich selbst wird, wie bei Wörtern mit L^n notiert, wobei $L^0 = \{\epsilon\}$ und $L^{n+1} = L \cdot L^n$. Damit lässt sich auch die beliebige Wiederholung von Sprachen definieren.

$$L^* = \bigcup\{\, L^n \mid n \in \mathbb{N} \}$$

Der Operator $\bigcup$ vereinigt eine Menge von Mengen. Enthält L ein nicht-leeres Wort, dann ist L^* eine unendliche Menge.

Jetzt haben wir alle Zutaten beisammen, um die Bedeutung regulärer Ausdrücke zu definieren. Die Semantik wird durch die unten definierte Bedeutungsfunktion angegeben, die einen regulären Ausdruck, ein Element von Reg, auf eine Sprache, ein Element von $\mathbb{P}(A^*)$, abbildet.

$$
\begin{aligned}
[\![a]\!] &= \{a\} \\
[\![\epsilon]\!] &= \{\epsilon\} \\
[\![r_1\ r_2]\!] &= [\![r_1]\!] \cdot [\![r_2]\!] \\
[\![\emptyset]\!] &= \emptyset \\
[\![r_1 \mid r_2]\!] &= [\![r_1]\!] \cup [\![r_2]\!] \\
[\![r^*]\!] &= [\![r]\!]^*
\end{aligned}
$$

Die doppelten Klammern sind die sogenannten *Oxford* oder *Strachey Klammern*[3]; sie schließen stets das syntaktische Objekt ein und trennen so Syntax von Semantik.

Jedem regulären Ausdruck r wird mit $[\![r]\!]$ eine Sprache, die Bedeutung des regulären Ausdrucks, zugeordnet. Die Bedeutungsfunktion $[\![-]\!]$ ist *kompositional*: Die Bedeutung eines Ausdrucks wird auf die Bedeutung seiner Teilausdrücke zurückgeführt. Mit Hilfe der Bedeutungsfunktion können wir die Bedeutung von regulären Ausdrücken ausrechnen. Für die folgenden Beispiele verwenden wir das Alphabet $A = \{\mathtt{a}, \mathtt{b}\}$.

$$
\begin{aligned}
& [\![\mathtt{a}\ \mathtt{b} \mid \mathtt{b}\ \mathtt{a}]\!] \\
= \ & [\![\mathtt{a}\ \mathtt{b}]\!] \cup [\![\mathtt{b}\ \mathtt{a}]\!] \\
= \ & ([\![\mathtt{a}]\!] \cdot [\![\mathtt{b}]\!]) \cup ([\![\mathtt{b}]\!] \cdot [\![\mathtt{a}]\!]) \\
= \ & (\{\mathtt{a}\} \cdot \{\mathtt{b}\}) \cup (\{\mathtt{b}\} \cdot \{\mathtt{a}\}) \\
= \ & \{\mathtt{ab}\} \cup \{\mathtt{ba}\} \\
= \ & \{\mathtt{ab}, \mathtt{ba}\}
\end{aligned}
$$

Der reguläre Ausdruck $[\![\mathtt{a}\ \mathtt{b} \mid \mathtt{b}\ \mathtt{a}]\!]$ bezeichnet somit die Sprache $\{\mathtt{ab}, \mathtt{ba}\}$.

Vertauschen wir Konkatenation und Alternative erhalten wir eine andere Sprache:

$$
\begin{aligned}
& [\![(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})]\!] \\
= \ & [\![\mathtt{a} \mid \mathtt{b}]\!] \cdot [\![\mathtt{b} \mid \mathtt{a}]\!] \\
= \ & ([\![\mathtt{a}]\!] \cup [\![\mathtt{b}]\!]) \cdot ([\![\mathtt{b}]\!] \cup [\![\mathtt{a}]\!]) \\
= \ & (\{\mathtt{a}\} \cup \{\mathtt{b}\}) \cdot (\{\mathtt{b}\} \cup \{\mathtt{a}\}) \\
= \ & \{\mathtt{a}, \mathtt{b}\} \cdot \{\mathtt{b}, \mathtt{a}\} \\
= \ & \{\mathtt{ab}, \mathtt{aa}, \mathtt{bb}, \mathtt{ba}\}
\end{aligned}
$$

Der reguläre Ausdruck $(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})$ bezeichnet die Sprache der Wörter der Länge 2. Klar: Ist das Alphabet $A = \{\mathtt{a}, \mathtt{b}\}$, dann ist $\mathtt{a} \mid \mathtt{b}$ ein einzelner beliebiger Buchstabe und $(\mathtt{a} \mid \mathtt{b})\ (\mathtt{a} \mid \mathtt{b}) = (\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})$ sind zwei beliebige Buchstaben hintereinander. Ein letztes Beispiel:

$$
\begin{aligned}
& [\![((\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}))^*]\!] \\
= \ & [\![(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})]\!]^* \\
= \ & \{\mathtt{ab}, \mathtt{aa}, \mathtt{bb}, \mathtt{ba}\}^*
\end{aligned}
$$

[3] Christopher Strachey (1916–1975), ein britischer Informatiker, war einer der Begründer der denotationellen Semantik.

Die bezeichnete Sprache umfasst alle Wörter gerader Länge.

Fassen wir zusammen: In der denotationellen Semantik wird einem syntaktischen Objekt, hier einem regulären Ausdruck, ein mathematisches Objekt, hier eine Sprache, zugeordnet. Die denotationelle Semantik ist nicht auf reguläre Ausdrücke eingeschränkt, sondern kann ganz allgemein verwendet werden, um die Bedeutung einer Sprache, insbesondere die einer Programmiersprache, zu beschreiben. Die Sprache Scheme [KCR98] verwendet zum Beispiel eine denotationelle Semantik, um die Bedeutung der Sprachkonstrukte präzise festzulegen.

Reduktionssemantik Kommen wir zum zweiten Ansatz, der *Reduktionssemantik*. Während die denotationelle Semantik ein globales Bild vermittelt („Welche Sprache bezeichnet der reguläre Ausdruck?“), bietet die Reduktionssemantik ein lokales Bild („Wie kann ich ein einzelnes Wort aus dem regulären Ausdruck ableiten?“). Die Reduktionssemantik legt fest, wie ein regulärer Ausdruck schrittweise zu einem Wort reduziert wird. Für die denotationelle Semantik sind Wörter Elemente von A^*; für die Reduktionssemantik sind Wörter besonders einfache reguläre Ausdrücke — so wie Werte besonders einfache Programme sind.

$$
\begin{array}{lll}
w ::= \epsilon & & \texttt{//} \text{ das leere Wort} \\
\quad \mid \; a & & \texttt{//} \text{ einzelnes Zeichen} \diagdown \text{Terminalsymbol} \\
\quad \mid \; w_1 \; w_2 & & \texttt{//} \text{ Konkatenation} \diagdown \text{Sequenz}
\end{array}
$$

Ein Wort zeichnet sich dadurch aus, dass seine Denotation eine einelementige Menge ist.

Die schrittweise Reduktion $r \longrightarrow r'$ (lies: r kann in *einem Schritt* zu r' reduziert werden) wird durch ein Beweissystem formalisiert. Dieses umfasst zunächst einmal die folgenden sogenannten *Rechenregeln*:

$$
\frac{}{r\ \epsilon \longrightarrow r} \qquad \frac{}{\epsilon\ r \longrightarrow r}
$$

$$
\frac{}{r_1 \mid r_2 \longrightarrow r_1} \qquad \frac{}{r_1 \mid r_2 \longrightarrow r_2}
$$

$$
\frac{}{r^* \longrightarrow \epsilon} \qquad \frac{}{r^* \longrightarrow r\ r^*}
$$

Die ersten beiden Regeln formalisieren, dass ϵ das neutrale Element der Konkatenation ist. Die interessanten Regeln sind die letzten vier; sie haben mit Wahlmöglichkeiten zu tun. Die Alternative $r_1 \mid r_2$ kann entweder zu r_1 oder zu r_2 reduziert werden. Für die Wiederholung r^* gibt es ebenfalls zwei Alternativen: ϵ oder $r\ r^*$.

Für den regulären Ausdruck `a b | b a` existieren somit zwei mögliche Reduktionen.

$$
\texttt{a b} \mid \texttt{b a} \longrightarrow \texttt{a b} \qquad \text{und} \qquad \texttt{a b} \mid \texttt{b a} \longrightarrow \texttt{b a}
$$

In beiden Fällen ist das Ergebnis direkt ein Wort, somit ist die bezeichnete Sprache $\{\texttt{ab}, \texttt{ba}\}$. Um den Ausdruck `(a | b) (b | a)` zu vereinfachen, benötigen wir weitere Regeln. Die obigen Beweisregeln erlauben nur den *gesamten* Ausdruck umzuformen; wir brauchen zusätzlich Regeln, die es uns ermöglichen, einen Ausdruck „mittendrin“ zu manipulieren,

um etwa (a | b) (b | a) zu a (b | a) zu reduzieren. Diese Regeln nennt man auch *Kongruenz-* oder *Kontextregeln*.[4]

$$\frac{r_1 \longrightarrow r_1'}{r_1\ r_2 \longrightarrow r_1'\ r_2} \qquad \frac{r_2 \longrightarrow r_2'}{r_1\ r_2 \longrightarrow r_1\ r_2'}$$

Die Kontextregeln sind rein bürokratische Regeln. Ihr einziger Zweck ist es, die Rechenregeln auf Teilausdrücke eines größeren Ausdrucks anwenden zu können. Kommen wir zu unserem Beispiel: Für (a | b) (b | a) gibt es insgesamt vier mögliche *Reduktionsfolgen*.

$$\begin{aligned}
&(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}) \longrightarrow \mathtt{a}\ (\mathtt{b} \mid \mathtt{a}) \longrightarrow \mathtt{a\ b}\\
&(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}) \longrightarrow \mathtt{a}\ (\mathtt{b} \mid \mathtt{a}) \longrightarrow \mathtt{a\ a}\\
&(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}) \longrightarrow \mathtt{b}\ (\mathtt{b} \mid \mathtt{a}) \longrightarrow \mathtt{b\ b}\\
&(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}) \longrightarrow \mathtt{b}\ (\mathtt{b} \mid \mathtt{a}) \longrightarrow \mathtt{b\ a}
\end{aligned}$$

In zwei Schritten lässt sich jeweils ein Wort ableiten; in jedem Schritt wird eine Rechenregel angewendet. Die bezeichnete Sprache ist somit $\{\mathtt{ab}, \mathtt{aa}, \mathtt{bb}, \mathtt{ba}\}$. Wesentlich mehr Möglichkeiten haben wir im dritten Beispiel, nämlich unendlich viele.

$$\begin{aligned}
((\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}))^* &\longrightarrow \epsilon\\
((\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}))^* &\longrightarrow (\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})\ ((\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}))^*\\
&\longrightarrow \mathtt{a}\ (\mathtt{b} \mid \mathtt{a})\ ((\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}))^*\\
&\longrightarrow \mathtt{a\ b}\ ((\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}))^*\\
&\longrightarrow \mathtt{a\ b}\ \epsilon\\
&\longrightarrow \mathtt{a\ b}\\
&\dots
\end{aligned}$$

Das Wort w ist aus dem regulären Ausdruck r *ableitbar*, wenn es eine Reduktionsfolge der Form

$$r \longrightarrow r_1 \longrightarrow \cdots \longrightarrow r_n \longrightarrow w$$

gibt. Die Sprache, die durch einen regulären Ausdruck beschrieben wird, ist die Menge aller *Worte*, die aus dem Ausdruck ableitbar sind. Die Betonung liegt auf Worte, denn eine Reduktionsfolge muss nicht zwangsläufig in einem Wort enden.

$$\emptyset^* \longrightarrow \emptyset\ \emptyset^* \longrightarrow \emptyset\ \emptyset\ \emptyset^* \longrightarrow \cdots$$

Da es keine Regel gibt, $\emptyset$ zu reduzieren, lässt sich mit dieser Reduktionsfolge kein Staat machen. Die einzige mögliche Reduktion ist $\emptyset^* \longrightarrow \epsilon$. Somit ist die Semantik von $\emptyset^*$ die Sprache $\{\epsilon\}$. (Auch mit der denotationellen Semantik erhalten wir $[\![\emptyset^*]\!] = \{\epsilon\}$.)

Wie auch die denotationelle Semantik wird die Reduktionssemantik verwendet, um die Bedeutung von Programmiersprachen festzulegen. Die Reduktionssemantik ist der

[4] Kontextregeln verwenden wir routinemäßig, ohne uns groß darüber bewusst zu werden: die Gleichungen $1 \cdot x = x = x \cdot 1$ werden in der Umformung $(x+1) \cdot (x-1) = x \cdot x + 1 \cdot x - x \cdot 1 - 1 \cdot 1 = x^2 - 1$ im Kontext einer Summe angewendet.

Semantik von Mini–F#, der sogenannten *Auswertungssemantik*, ähnlich. Beide werden durch Beweissysteme spezifiziert; beide beschreiben, wie ein Programm ausgerechnet wird. Die Reduktionssemantik beschreibt einen *einzelnen Schritt*; die Auswertungssemantik eine *vollständige Rechnung*. Für reguläre Ausdrücke lässt sich übrigens auch eine Auswertungssemantik angeben, siehe Aufgabe 6.6.

Zwei Semantiken, zwei Antworten auf die Frage „Was bedeutet ein regulärer Ausdruck?“. Ist die Antwort in beiden Fällen die gleiche? Ja! Da uns die nötigen Hilfsmittel für den Nachweis dieser Tatsache fehlen, nehmen wir die gute Nachricht so hin.

6.1.3. Vertiefung

Schauen wir uns weitere Beispiele für reguläre Ausdrücke an. Das zugrundeliegende Alphabet ist im Folgenden weiterhin $A = \{\mathtt{a}, \mathtt{b}\}$. Der reguläre Ausdruck

$$(\mathtt{b} \mid \mathtt{a}\ \mathtt{b}^*\ \mathtt{a})^*$$

bezeichnet die Sprache aller Wörter mit einer geraden Anzahl von as und beliebig vielen bs. Ein einzelnes b erfüllt diese Eigenschaft und ein a, gefolgt von bs, gefolgt von einem weiteren a. Von diesen Wörtern, b oder a b* a, können wir beliebig viele hintereinandersetzen, ohne die Eigenschaft zu verletzen (die Summe zweier gerader Zahlen ist wiederum gerade).

Die gleiche Sprache kann durch viele reguläre Ausdrücke beschrieben werden. Zum Beispiel bezeichnet $(\mathtt{a}\ \mathtt{b}^*\ \mathtt{a} \mid \mathtt{b})^*$ die gleiche Sprache wie $(\mathtt{b} \mid \mathtt{a}\ \mathtt{b}^*\ \mathtt{a})^*$. Das ist eine recht offensichtliche Umformung, da die Alternative semantisch eine Vereinigung ist und somit kommutativ. Weniger offensichtlich ist, dass wir die gleiche Sprache auch *ohne* Alternative beschreiben können:

$$\mathtt{b}^*\ (\mathtt{a}\ \mathtt{b}^*\ \mathtt{a}\ \mathtt{b}^*)^*$$

An die Stelle der iterierten Alternative tritt eine geschachtelte Wiederholung. Allgemein gilt der folgende Zusammenhang:

$$(r_1 \mid r_2)^* = r_1^*\ (r_2\ r_1^*)^*$$

Wie zeigt man, dass zwei reguläre Ausdrücke gleichwertig sind? Mit Hilfe der Semantik. Unter Rückgriff auf die denotationelle Semantik kann man versuchen $[\![r_1]\!] = [\![r_2]\!]$ zu beweisen. Alternativ kann man zeigen, dass es zu jeder Reduktionsfolge $r_1 \longrightarrow \cdots \longrightarrow w$ eine korrespondierende Reduktionsfolge $r_2 \longrightarrow \cdots \longrightarrow w$ gibt und umgekehrt.

Zurück zu den Beispielen: Die Sprache aller Wörter, die eine gerade Anzahl von as *oder* eine ungerade Anzahl von bs haben, lässt sich unter Rückgriff auf das vorherige Beispiel leicht definieren.

$$(\mathtt{b} \mid \mathtt{a}\ \mathtt{b}^*\ \mathtt{a})^* \mid \mathtt{a}^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b}\ \mathtt{a}^*\ \mathtt{b})^*$$

Aber wie beschreibt man die Sprache aller Wörter, die eine gerade Anzahl von as *und* eine ungerade Anzahl von bs haben? Die Sprache für reguläre Ausdrücke hat zwar eine

Operation für die Vereinigung, nicht aber für den Durchschnitt. Stände ein Pendant zum Durchschnitt bereit, sagen wir &, könnten wir formulieren:

$$(\mathtt{a}\ \mathtt{b}^*\ \mathtt{a} \mid \mathtt{b})^* \,\&\, \mathtt{a}^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b}\ \mathtt{a}^*\ \mathtt{b})^* \tag{6.1}$$

In der Tat spricht nichts dagegen, den Durchschnitt zu der Sprache der regulären Ausdrücke hinzuzufügen, siehe Aufgabe 6.5. Interessanterweise ist die Erweiterung zwar bequem, aber nicht notwendig: Sie erhöht nicht die Ausdruckskraft. Mehr dazu später aus der Abteilung der Theoretischen Informatik. Wir versuchen an dieser Stelle einfach, die Sprache mit den bisherigen Bordmitteln zu definieren. Dazu sind zwei Abkürzungen nützlich: $g = (\mathtt{aa})^*$ und $u = \mathtt{a}\ (\mathtt{aa})^*$. Starten wir bescheiden und definieren eine Sprache, die genau ein `b` und eine gerade Anzahl von `a`s enthält.

$$g\ \mathtt{b}\ g \mid u\ \mathtt{b}\ u$$

Entweder kommen vor und hinter dem `b` eine gerade Anzahl von `a`s vor oder an beiden Positionen eine ungerade Anzahl. Nächster Schritt: Wir erhöhen die Anzahl der `b`s auf zwei:

$$g\ \mathtt{b}\ g\ \mathtt{b}\ g \mid g\ \mathtt{b}\ u\ \mathtt{b}\ u \mid u\ \mathtt{b}\ g\ \mathtt{b}\ u \mid u\ \mathtt{b}\ u\ \mathtt{b}\ g$$

Jetzt werden die `a`s über drei Positionen verteilt; entweder befinden sich an allen Stellen eine gerade Anzahl von `a`s oder an exakt zwei Positionen eine ungerade Anzahl. Jetzt sind wir fast am Ziel. Eine ungerade Zahl hat die Form $2n + 1$, entsprechend definieren wir als Alternative zu (6.1)

$$(g\ \mathtt{b}\ g \mid u\ \mathtt{b}\ u)\ (g\ \mathtt{b}\ g\ \mathtt{b}\ g \mid g\ \mathtt{b}\ u\ \mathtt{b}\ u \mid u\ \mathtt{b}\ g\ \mathtt{b}\ u \mid u\ \mathtt{b}\ u\ \mathtt{b}\ g)^*$$

Man sieht, ohne den Durchschnitt müssen wir bei der Formulierung von Sprachen sehr viel mehr Grips investieren.

Das bringt uns zu der Frage, ob wir mit regulären Ausdrücken überhaupt alle Sprachen beschreiben können bzw. wenn das nicht möglich ist, ob wir wenigstens die Syntax von Programmiersprachen festlegen können. Die Antwort auf beide Fragen ist negativ, reguläre Ausdrücke sind in ihren Möglichkeiten stark eingeschränkt — deswegen werden sie tatsächlich nur für die lexikalische Syntax herangezogen. (Die Antwort auf die erste Frage ist übrigens immer negativ, da es sehr viel mehr Sprachen gibt als Ausdrücke, mit denen Sprachen beschrieben werden können.) Mit Hilfe regulärer Ausdrücke lassen sich zum Beispiel einfache Hygienevorschriften nicht fassen, etwa, dass es in Mini–F# Ausdrücken zu jeder „Klammer auf“ eine korrespondierende „Klammer zu“ geben muss. Selbst die Sprache, die nur aus ordentlich geschachtelten Klammerpaaren besteht,

$$\{\mathtt{a}^n\ \mathtt{b}^n \mid n \in \mathbb{N}\}$$

ist keine reguläre Sprache — lies `a` als „`(`“ und `b` als „`)`“. Warum das so ist, werden wir im nächsten Abschnitt sehen.

Kommen wir zur lexikalischen Syntax von Mini–F#. Abbildung 6.3 fasst die Lexeme der Sprache zusammen. Die dort aufgeführten regulären Ausdrücke verwenden zwei Er-

Numeral	*digit digit**
Stringliteral	`"` (⁀(?* (`"` ∣ `\`) ?*) ∣ `\"` ∣ `\\` ∣ `\t` ∣ `\n` ∣ `\v` ∣ `\f` ∣ `\r`)* `"`
Bezeichner klein	*lower* (*lower* ∣ *upper* ∣ *digit* ∣ `'` ∣ `_`)*
Bezeichner groß	*upper* (*lower* ∣ *upper* ∣ *digit* ∣ `'` ∣ `_`)*
Typparameter	`'` (*lower* ∣ *upper* ∣ *digit* ∣ `_`)*
Schlüsselwörter	`abstract` ∣ `and` ∣ `as` ∣ `assert` ∣ `base` ∣ `begin` ∣ `class` ∣ `default` ∣ `delegate` ∣ `do` ∣ `done` ∣ `downcast` ∣ `downto` ∣ `elif` ∣ `else` ∣ `end` ∣ `exception` ∣ `extern` ∣ `false` ∣ `finally` ∣ `for` ∣ `fun` ∣ `function` ∣ `global` ∣ `if` ∣ `in` ∣ `inherit` ∣ `inline` ∣ `interface` ∣ `internal` ∣ `lazy` ∣ `let` ∣ `match` ∣ `member` ∣ `module` ∣ `mutable` ∣ `namespace` ∣ `new` ∣ `null` ∣ `of` ∣ `open` ∣ `or` ∣ `override` ∣ `private` ∣ `public` ∣ `rec` ∣ `return` ∣ `sig` ∣ `static` ∣ `struct` ∣ `then` ∣ `to` ∣ `true` ∣ `try` ∣ `type` ∣ `upcast` ∣ `use` ∣ `val` ∣ `void` ∣ `when` ∣ `while` ∣ `with` ∣ `yield`
Separatoren	`(` ∣ `)` ∣ `,` ∣ `{` ∣ `}`
Symbole	`!` ∣ `%` ∣ `&` ∣ `&&` ∣ `*` ∣ `+` ∣ `,` ∣ `-` ∣ `->` ∣ `.` ∣ `..` ∣ `/` ∣ `:` ∣ `:=` ∣ `<` ∣ `<=` ∣ `<>` ∣ `=` ∣ `>` ∣ `>=` ∣ `[` ∣ `[|` ∣ `]` ∣ `^` ∣ `_` ∣ `|` ∣ `|]` ∣ `||`
Leerzeichen	∣ `\t` ∣ `\n` ∣ `\v` ∣ `\f` ∣ `\r`
Zeilenkommentar	`//` ⁀(?* `\n` ?*) `\n`
Kommentar	`(*` ⁀(?* `*)` ?*) `*)`

Abbildung 6.3.: Lexikalische Syntax von Mini–F#.

weiterungen: '?' steht für ein beliebiges Zeichen und $\frown r$ steht für das Komplement von r. Wie auch die Konjunktion erhöhen die neuen Konstrukte die Bequemlichkeit, nicht aber die prinzipielle Ausdruckskraft regulärer Ausdrücke — wir kommen in Abschnitt 6.2 noch einmal darauf zu sprechen. Ist das Alphabet $A = \{a_1, \ldots, a_n\}$, dann ist '?' eine Abkürzung für $a_1 \mid \cdots \mid a_n$. Im Komplement $\frown r$ sind alle Wörter enthalten, die in r *nicht* enthalten sind. Was Disjunktion (`||`), Konjunktion (`&&`) und Negation (*not*) für Boolesche Werte sind, sind Alternative ('∣'), Durchschnitt ('&') und Komplement ('⁀') für Sprachen.

Gehen wir die Lexeme in Abbildung 6.3 einmal durch. Ein Numeral ist eine Konstante vom Typ *Nat*. Ein Stringliteral ist eine Konstante vom Typ *String* und bezeichnet einen Text, eine Sequenz von Zeichen. Strings werden in Anführungsstriche (oben) eingeschlossen. Innerhalb der Anführungsstriche darf das Zeichen '`"`' selbst nicht vorkommen. Soll '`"`' ein Zeichen des Textes sein, so muss dem doppelten Anführungsstrich das Fluchtsymbol '\' vorangestellt werden. Das Literal `"\"Zitat\""` bezeichnet somit einen Text, der aus sieben Zeichen besteht. Mini–F# kennt noch weitere Fluchtsequenzen, etwa `\n` für den Zeilenvorschub, ein nicht druckbares Zeichen des ASCII-Alphabets.

Wir unterscheiden zwischen drei Arten von Bezeichnern, solchen, die mit einem kleinen Buchstaben anfangen, solchen, die mit einem großen Buchstaben anfangen, und solchen,

die mit einem Apostroph anfangen. Kleine Bezeichner werden für Werte (*size*, *area*) und Labels (*day*, *month*) verwendet; große Bezeichner für Konstruktoren (*Nil*, *Cons*) und Typen (*Person*, *List*); Apostrophbezeichner ausschließlich für Typparameter (*'a*, *'soln*).[5] Bestimmte Folgen von kleinen Buchstaben (***let***, ***in*** usw.), sogenannte Schlüsselwörter, sind reserviert und dienen der kontextfreien Syntax als Interpunktionszeichen. Bei der Unterteilung eines Programmtextes werden Lexeme so lang wie möglich gewählt (engl. longest match): `letter` ist ein Bezeichner, `let ter` aber das Schlüsselwort `let` gefolgt von dem Bezeichner `ter`.

Bestimmte Symbole wie etwa die Klammerpaare `(` und `)`, `{` und `}` werden als Interpunktionszeichen verwendet, andere Symbole dienen als Bezeichner für Operatoren, wie zum Beispiel `+` und `*`. Die als Separatoren gekennzeichneten Symbole stehen stets für ein einzelnes Lexem, alle anderen Symbole können auch zu größeren Einheiten kombiniert werden: etwa `>` und `=` zu `>=`. Wie auch bei den Schlüsselwörtern gilt, dass Lexeme so lang wie möglich gewählt werden: `>=` ist ein Lexem, `> =` sind zwei Lexeme.

Einzelne Lexeme können durch Leerzeichen oder Kommentare getrennt werden. Mini–F# unterscheidet zwischen Zeilenkommentaren, die mit `//` anfangen und sich bis zum Zeilenende erstrecken, und normalen Kommentaren, die von den Kommentarklammern `(*` und `*)` eingeschlossen werden. (Entgegen der Spezifikation in Abbildung 6.3 dürfen Kommentare auch geschachtelt werden. Ordentlich geschachtelte Klammerpaare lassen sich aber, wie wir bereits wissen, mit regulären Ausdrücken nicht beschreiben.)

6.2. Scanner⁄Akzeptoren

6.2.1. Akzeptoren

Ein regulärer Ausdruck r beschreibt eine Sprache. Wie können wir feststellen, ob ein gegebenes Wort w in der Sprache enthalten ist? Dieses Problem, $w \in [\![r]\!]$?, nennt man auch kurz und prägnant *Wortproblem*. (Bei der Aufteilung eines Programmtextes in einzelne Lexeme ist eine ähnliche Operation gefragt.) Oder besser: Können wir ein Mini–F# Programm schreiben, das uns diese Aufgabe abnimmt? Die Antwort ist diesmal positiv. Ein solches Programm nennt man auch *Akzeptor*. (Das Programm, das einen Programmtext in Lexeme aufteilt, hört auf den Namen *Lexer* oder *Scanner*.) Im Folgenden gehen wir die einzelnen Schritte exemplarisch für den regulären Ausdruck $r_{00} = (\texttt{a} \mid \texttt{b})^*\ \texttt{b}\ (\texttt{a} \mid \texttt{b})$ durch — der Ausdruck beschreibt die Sprache aller Wörter, die an der vorletzten Position ein `b` haben.

module *Grammar.* *Alphabet*

module *Grammar.* *Compiled*

Überlegen wir uns zunächst die Schnittstelle des Programms. Wir gehen davon aus, dass das Alphabet durch eine Variantentypdefinition gegeben ist. Zum Beispiel:

```
type Alphabet = | A | B
let ord-Alphabet (a : Alphabet) : Int =
  match a with | A → 0 | B → 1
```

[5] Diese Konvention zusammen mit der Farbcodierung erlaubt es, jeden Bezeichner eindeutig einer Kategorie zuzuordnen. (Wir mogeln tatsächlich etwas: in F# können kleine Bezeichner auch für Typen (*int*) und große Bezeichner für Werte und Labels (*Length*) benutzt werden.)

Zu einem Alphabet gehört eine injektive Funktion, die sogenannte Codierungsfunktion, die jedem Zeichen eine ganze Zahl zuordnet, den sogenannten *Zeichencode*.

Ein Akzeptor ist eine Funktion des folgenden Typs.

$$\mathit{accept}\text{-}r_{00} : \mathit{List}\,\langle \mathit{Alphabet}\rangle \to \mathit{Bool}$$

Die Funktion $\mathit{accept}\text{-}r_{00}$ testet, ob die Eingabe w ein Element der durch r_{00} bezeichneten Sprache ist: $w \in [\![r_{00}]\!]$. Das Struktur Entwurfsmuster für *List* motiviert einen ersten Ansatz:

$$\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept}\text{-}r_{00}\ (\mathit{input} : \mathit{List}\,\langle \mathit{Alphabet}\rangle) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ \mathit{input}\ \textbf{\textit{with}} \\
\quad \mid [\,] \qquad \to \ldots \\
\quad \mid A :: \mathit{rest} \to \ldots\, \mathit{accept}\text{-}r_{00}\ \mathit{rest} \ldots \\
\quad \mid B :: \mathit{rest} \to \ldots\, \mathit{accept}\text{-}r_{00}\ \mathit{rest} \ldots
\end{array}$$

Der rekursive Aufruf von $\mathit{accept}\text{-}r_{00}$ hilft uns an dieser Stelle leider nicht weiter. Aus $w \in [\![r_{00}]\!]$ können wir nur bedingt Rückschlüsse auf $a\ w \in [\![r_{00}]\!]$ ziehen.

Gehen wir die Zweige des ***match***-Ausdrucks der Reihe nach durch. Im Fall $\mathit{input} = [\,]$ müssen wir überlegen, ob das leere Wort in der Sprache enthalten ist: $\epsilon \in [\![r]\!]$. Können wir dem regulären Ausdruck das ansehen? Können wir eine semantische Eigenschaft — ist ϵ Element der möglicherweise unendlichen Menge $[\![r]\!]$ — an Hand der Form, das heißt der Syntax von r überprüfen? Ja, so geht's (*nullable* ist eine mathematische Funktion, kein Mini–F# Programm):

$$\begin{array}{ll}
\mathit{nullable}(a) & = \mathit{false} \\
\mathit{nullable}(\epsilon) & = \mathit{true} \\
\mathit{nullable}(r_1\ r_2) & = \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2) \\
\mathit{nullable}(\emptyset) & = \mathit{false} \\
\mathit{nullable}(r_1 \mid r_2) & = \mathit{nullable}(r_1) \vee \mathit{nullable}(r_2) \\
\mathit{nullable}(r^*) & = \mathit{true}
\end{array}$$

Die Konkatenation $r_1\ r_2$ enthält ϵ, wenn sowohl r_1 als auch r_2 das leere Wort ϵ enthalten; die Alternative $r_1 \mid r_2$ enthält ϵ, wenn entweder r_1 oder r_2 das leere Wort ϵ enthalten. Die Wiederholung r^* enthält ϵ auf jeden Fall. Somit können wir den ersten Zweig von $\mathit{accept}\text{-}r_{00}$ mit Leben füllen: $\mathit{nullable}((\mathtt{a} \mid \mathtt{b})^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b})) = \mathit{false}$, also

$$\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept}\text{-}r_{00}\ (\mathit{input} : \mathit{List}\,\langle \mathit{Alphabet}\rangle) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ \mathit{input}\ \textbf{\textit{with}} \\
\quad \mid [\,] \qquad \to \mathit{false} \\
\quad \mid A :: \mathit{rest} \to \ldots\, \mathit{accept}\text{-}r_{00}\ \mathit{rest} \ldots \\
\quad \mid B :: \mathit{rest} \to \ldots\, \mathit{accept}\text{-}r_{00}\ \mathit{rest} \ldots
\end{array}$$

Kommen wir zum Fall $\mathit{input} = A :: \mathit{rest}$. An dieser Stelle wäre es nützlich zu wissen, was von der Sprache „übrigbleibt“, jetzt da wir `a` bereits gesehen haben. Dann könnten wir die Arbeit an eine weitere Funktion delegieren, den Akzeptor für die „Restsprache“.

Diese „Restsprache“ heißt im Fachjargon *Rechtsfaktor* und ist für ein Wort w wie folgt definiert:

$$L \,/\, w = \{\, x \mid w\ x \in L \,\}$$

Der Rechtsfaktor $L \,/\, w$ (lies: L durch w) ist die Menge aller Restworte x, so dass $w\ x$ in L enthalten ist. Nun ist L eine reguläre Sprache und wir müssen überlegen, ob $L \,/\, w$ ebenfalls regulär ist. Zunächst einmal gilt

$$\begin{aligned} L \,/\, \epsilon &= L \\ L \,/\, w_1\ w_2 &= (L \,/\, w_1) \,/\, w_2 \end{aligned}$$

Das heißt, wir können uns auf die Definition von $L \,/\, a$ konzentrieren, wobei a ein Terminalsymbol ist. Der schwierigste Fall ist die Konkatenation: $(r_1\ r_2) \,/\, a$. Entweder wir entfernen a aus r_1 oder aus r_2; letzteres aber nur, wenn r_1 „nullable“ ist. Zu diesem Zweck definieren wir eine Operation auf regulären Ausdrücken:

$$\Delta(r) \quad = \quad \begin{cases} \epsilon & \text{falls } \mathit{nullable}(r) \\ \emptyset & \text{sonst} \end{cases}$$

Damit erhalten wir

$$\begin{aligned} a \,/\, x &= \begin{cases} \epsilon & \text{falls } a = x \\ \emptyset & \text{sonst} \end{cases} \\ \epsilon \,/\, x &= \emptyset \\ r_1\ r_2 \,/\, x &= (r_1 \,/\, x)\ r_2 \mid \Delta(r_1)\ (r_2 \,/\, x) \\ \emptyset \,/\, x &= \emptyset \\ r_1 \mid r_2 \,/\, x &= (r_1 \,/\, x) \mid (r_2 \,/\, x) \\ r^* \,/\, x &= (r \,/\, x)\ r^* \end{aligned}$$

Die Definition des Rechtsfaktors ist der *Ableitung von Funktionen* sehr ähnlich (lies ϵ als 1, $\emptyset$ als 0, die Alternative als Summe und die Konkatenation als Produkt). Die Regel für die Alternative entspricht exakt der Summenregel; die Regel für die Konkatenation entspricht *fast* der Produktregel — die Produktregel der Analysis ist symmetrisch: $r_1\ r_2 \,/\, x = (r_1 \,/\, x)\ r_2 \mid r_1\ (r_2 \,/\, x)$.

Wie sehen die Rechtsfaktoren für unser Beispiel aus?

$$\begin{aligned} r_{00} \,/\, \texttt{a} &= r_{00} \\ r_{00} \,/\, \texttt{b} &= r_{00} \mid (\texttt{a} \mid \texttt{b}) = r_{10} \end{aligned}$$

Wir erhalten r_{00} selbst und einen neuen regulären Ausdruck, den wir auf den Namen r_{10} taufen. Damit können wir endlich die Definition von *accept-*r_{00} vervollständigen.

```
let rec accept-r00 (input : List ⟨Alphabet⟩) : Bool =
  match input with
  | []          → false
  | A :: rest → accept-r00 rest
  | B :: rest → accept-r10 rest
```

Ist das erste Zeichen ein A, dann erfolgt ein rekursiver Aufruf; ist das Zeichen ein B, dann wird die weitere Arbeit an *accept*-r_{10} delegiert.

Es verbleibt einen Akzeptor für r_{10} zu schreiben. Diese Aufgabe gehen wir nach dem gleichen Schema wie für r_{00} an.

$$
\begin{aligned}
&\mathit{nullable}(r_{10}) = \mathit{false}\\
&r_{10} \,/\, \mathtt{a} = r_{00} \mid \epsilon \qquad\quad = r_{01}\\
&r_{10} \,/\, \mathtt{b} = r_{00} \mid (\mathtt{a} \mid \mathtt{b}) \mid \epsilon = r_{11}
\end{aligned}
$$

Wir erhalten zwei neue reguläre Ausdrücke und rechnen weiter.

$$
\begin{aligned}
&\mathit{nullable}(r_{01}) = \mathit{true}\\
&r_{01} \,/\, \mathtt{a} = r_{00}\\
&r_{01} \,/\, \mathtt{b} = r_{10}\\
&\mathit{nullable}(r_{11}) = \mathit{true}\\
&r_{11} \,/\, \mathtt{a} = r_{01}\\
&r_{11} \,/\, \mathtt{b} = r_{11}
\end{aligned}
$$

In der letzten Runde sind keine neuen regulären Ausdrücke hinzugekommen. Somit können wir unser Programm zusammenpuzzeln. Die Aufrufstruktur ist recht chaotisch: Die Funktion *accept*-r_{00} ruft *accept*-r_{10} auf, diese ruft *accept*-r_{11} auf, diese ruft *accept*-r_{01} auf und diese wiederum *accept*-r_{00}, siehe Abbildung 6.4. Mit anderen Worten, die vier

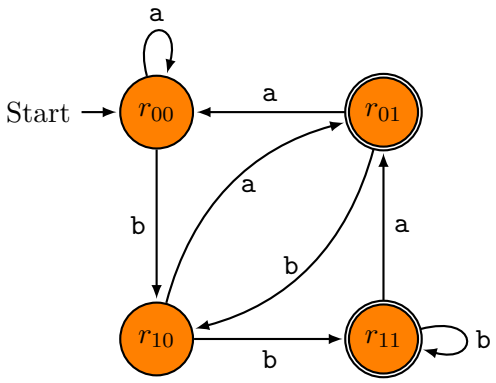


Abbildung 6.4.: Aufrufgraph des Akzeptors.

Akzeptoren sind *verschränkt rekursiv* definiert. Zur Erinnerung: Verschränkt rekursive Funktionsdefinitionen müssen mit dem Schlüsselwort ***and*** verbunden werden, damit jede Funktion jede andere sieht: ***let rec*** $f_1\ (x_1) = e_1$ ***and*** $f_2\ (x_2) = e_2$. Die Bezeichner f_1 und f_2 sind sowohl in e_1 als auch in e_2 sichtbar. Das vollständige Programm für *accept*-r_{00} ist in Abbildung 6.5 aufgeführt.

Der reguläre Ausdruck $(\mathtt{a} \mid \mathtt{b})^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b})$ hat vier verschiedene Rechtsfaktoren, der Ausdruck $(\mathtt{b} \mid \mathtt{a}\ \mathtt{b}^*\ \mathtt{a})^*$ hat zwei (welche?). Allgemein kann man zeigen, dass eine Sprache, die durch einen regulären Ausdruck beschrieben wird, nur *endlich* viele verschiedene Rechtsfaktoren besitzt. (Die Umkehrung gilt übrigens auch.) Diese Eigenschaft

```
let rec accept-r00 (input : List ⟨Alphabet⟩) : Bool =
  match input with
  | []        → false                  // nullable(r00) = false
  | A :: rest → accept-r00 rest        // r00 / a = r00
  | B :: rest → accept-r10 rest        // r00 / b = r10
and accept-r10 (input : List ⟨Alphabet⟩) : Bool =
  match input with
  | []        → false                  // nullable(r10) = false
  | A :: rest → accept-r01 rest        // r10 / a = r01
  | B :: rest → accept-r11 rest        // r10 / b = r11
and accept-r01 (input : List ⟨Alphabet⟩) : Bool =
  match input with
  | []        → true                   // nullable(r01) = true
  | A :: rest → accept-r00 rest        // r01 / a = r00
  | B :: rest → accept-r10 rest        // r01 / b = r10
and accept-r11 (input : List ⟨Alphabet⟩) : Bool =
  match input with
  | []        → true                   // nullable(r11) = true
  | A :: rest → accept-r01 rest        // r11 / a = r01
  | B :: rest → accept-r11 rest        // r11 / b = r11
```

Abbildung 6.5.: Akzeptor für $(\mathtt{a} \mid \mathtt{b})^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b})$.

können wir auch ausnutzen, um zu zeigen, dass eine Sprache *nicht* durch einen regulären Ausdruck beschrieben werden kann. Wir haben im letzten Abschnitt erwähnt, dass $L = \{\mathtt{a}^n\ \mathtt{b}^n \mid n \in \mathbb{N}\}$ eine solche Sprache ist. Die Rechtsfaktoren für Wörter der Form $\mathtt{a}^k$ sind gegeben durch:

$$L \,/\, \mathtt{a}^k = \{\mathtt{a}^n\ \mathtt{b}^{n+k} \mid n \in \mathbb{N}\}$$

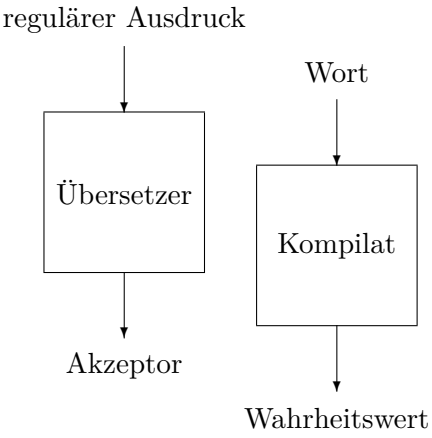
Alle diese Sprachen sind verschieden, also ist L keine reguläre Sprache.

Die Definition des Rechtsfaktors $r \,/\, x$ lässt sich übrigens leicht auf die Operationen '&' und '$\widehat{\ }$', Durchschnitt und Komplement regulärer Ausdrücke, erweitern.

$$\begin{array}{lcl} r_1 \,\&\, r_2 \,/\, x & = & (r_1 \,/\, x) \,\&\, (r_2 \,/\, x) \\ \widehat{\ } r \,/\, x & = & \widehat{\ }(r \,/\, x) \end{array}$$

Auch für diese Erweiterungen gilt, dass die Menge aller Rechtsfaktoren endlich ist. Wenn man gezeigt hat, dass eine Sprache mit endlich vielen Rechtsfaktoren regulär ist, dann ist damit auch klar, dass Durchschnitt und Komplement die Ausdruckskraft regulärer

Ausdrücke nicht erhöhen (die erweiterten regulären Ausdrücke haben ja nur endlich viele Rechtsfaktoren).



Kommen wir zurück zu unserer ursprünglichen Aufgabe, der Generierung eines Akzeptors. Da die Anzahl aller Rechtsfaktoren endlich ist, kann man das obige Verfahren verwenden, um aus einem regulären Ausdruck ein Programm zu generieren, das testet, ob ein Wort in der von dem Ausdruck bezeichneten Sprache enthalten ist. Mehr noch: Wir können das Verfahren auch automatisieren, sprich wir können ein Mini–F# Programm schreiben, das uns diese Aufgabe abnimmt! Dieses Mini–F# Programm würde als Eingabe einen regulären Ausdruck verarbeiten (in konkreter oder abstrakter Syntax) und als Ausgabe ein Mini–F# Programm, einen Akzeptor für den regulären Ausdruck, erzeugen (in konkreter Syntax). Ein Mini–F# Programm, das ein anderes Mini–F# Programm erzeugt! Ein solches Programm nennt man *Übersetzer* (engl. compiler). In diesem Fall würde man genauer von einem *Scanner-Generator* sprechen. Einen Scanner-Generator tatsächlich zu programmieren, würde den Umfang dieser Vorlesung sprengen, so dass wir an dieser Stelle keinen Programmcode angeben. (Am Ende des Kapitels haben wir aber durchaus das nötige Know-how, um ein solches Projekt anzugehen.) Nichtdestotrotz wollen wir das Thema noch etwas vertiefen. Der Übersetzer und der generierte Akzeptor sind zwei getrennte Programme, siehe obige Grafik. Der Übersetzer erzeugt ein Programm, das in einem zweiten Schritt ausgeführt wird. Die beiden Programme können alternativ auch enger miteinander verzahnt werden:

module
Grammar.
ByteCode

Schauen wir uns das Programm in Abbildung 6.5 noch einmal an. Die Struktur der einzelnen Funktionen ist identisch — das ist nicht weiter verwunderlich, wir haben sie ja nach dem gleichen Schema konstruiert. Eine naheliegende Frage ist, ob sich die vier Funktionen nicht zu einer zusammenfassen lassen? Wir könnten die vier Funktionen zum Beispiel durchnummerieren und die Hausnummer zu einem zusätzlichen Parameter machen.

$$accept\ (k : Nat, input : List\ \langle Alphabet \rangle) : Bool$$

Diese allgemeine Funktion müssen wir dann noch mit Informationen ausstatten, welcher Wahrheitswert im '[]'-Zweig zurückgegeben wird und welche Hausnummer als nächstes an der Reihe ist, wenn ein `a` bzw. ein `b` gesehen wird. Kurzum: Wir müssen die rekursive Aufrufstruktur des Programms aus Abbildung 6.5 in einer Datenstruktur ablegen — eine Kontrollstruktur wird zu einer Datenstruktur. Wie kann die Datenstruktur aussehen? Für den '[]'-Zweig müssen wir jeder Hausnummer einen Wahrheitswert zuordnen; im '::'-Zweig müssen wir in Abhängigkeit von dem gelesenen Zeichen und der aktuellen Hausnummer eine neue Hausnummer ermitteln. Beide Zuordnungen können wir durch Arrays beschreiben, ein 1-dimensionales für den '[]'-Zweig und ein 2-dimensionales für den '::'-Zweig.

```
type Control = { nullable : Array ⟨Bool⟩;
                 next     : Array ⟨Array ⟨Nat⟩⟩ }
let generic-accept (control : Control) : List ⟨Alphabet⟩ → Bool =
  let rec accept (k : Nat, input : List ⟨Alphabet⟩) : Bool =
    match input with
    | []       → control.nullable.[k]
    | a :: rest → accept (control.next.[ord-Alphabet a].[k], rest)
  in fun input → accept (0, input)
(* Anwendung *)
let control-r00 = { nullable = [| false; true; false; true |];
                    next     = [| [| 0; 0; 1; 1 |];       (* A *)
                                  [| 2; 2; 3; 3 |] |] |}   (* B *)
let accept-r00 = generic-accept control-r00
```

Abbildung 6.6.: Generischer Akzeptor (Byte-code Interpreter).

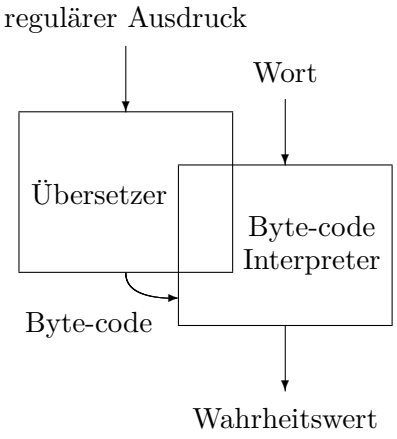
```
type Control = { nullable : Array ⟨Bool⟩;
                 next     : Array ⟨Array ⟨Nat⟩⟩ }
```

Das 2-dimensionale Array wird durch ein geschachteltes Array realisiert. Ein technisches Detail ist noch zu beachten: Arrays werden mit ganzen Zahlen indiziert; wir wollen das *next* Array aber mit einem Element des Alphabets indizieren. Hier kommt die Funktion *ord-Alphabet* ins Spiel, die jedem Zeichen eine ganze Zahl zuordnet. Für unser laufendes Beispiel erhalten wir die folgenden Daten — die Funktionen aus Abbildung 6.5 haben wir fortlaufend von oben nach unten beginnend mit 0 durchnummeriert.

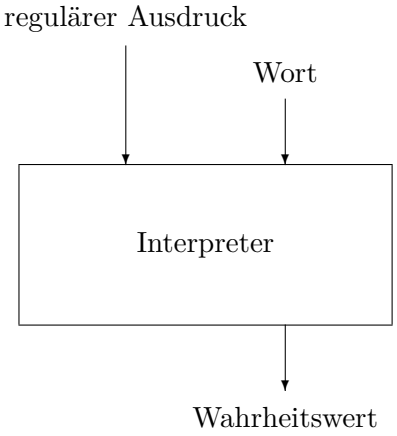
```
let control-r00 = { nullable = [| false; true; false; true |];
                    next     = [| [| 0; 0; 1; 1 |];       (* A *)
                                  [| 2; 2; 3; 3 |] |] |}   (* B *)
```

Das vollständige Programm ist in Abbildung 6.6 dargestellt. Der erste Teil ist unabhängig von einem konkreten regulären Ausdruck; der zweite Teil instantiiert das allgemeine Programm mit dem für unser laufendes Beispiel spezifischen Daten.

Verschaffen wir uns noch einmal einen Überblick. Der Übersetzer für reguläre Ausdrücke — den wir nicht angegeben haben — würde in diesem Szenario kein Mini–F# Programm erzeugen, sondern ein Element des Typs *Control*, in dem die Kontrollinformation des Akzeptors codiert ist. Der allgemeine Akzeptor *generic-accept* interpretiert dann diese Kontrollinformation, um zu einem gegebenen Wort zu entscheiden, ob es in der Sprache enthalten ist oder nicht. Bei dem allgemeinen Akzeptor handelt es sich somit um einen Interpreter, genauer um einen *Byte-code Interpreter*. Byte-code deswegen, weil die Rechtsfaktoren durch kleine Zahlen[6] repräsentiert werden und die Funktionen *nullable* und $r \;/\; x$ mit Hilfe von *nullable* und *next* codiert werden. Da die beiden Programme, der Übersetzer und der Byte-code Interpreter, über eine Datenstruktur miteinander kommunizieren, können sie in einem Programm zusammengefasst werden, siehe obige Abbildung.



Der Übersetzer generiert Byte-code, den der Byte-code Interpreter abarbeitet. Der Übersetzer hat sozusagen die Aufgabe reguläre Ausdrücke vorzuverdauen. Alternativ können wir einen Akzeptor schreiben, der direkt auf den regulären Ausdrücken arbeitet, sozusagen auf den unverdauten Eingaben. Dann haben wir keinen Byte-code Interpreter mehr vor uns, sondern einen „echten“ Interpreter, siehe Abbildung rechts.



Zu diesem Zweck müssen wir reguläre Ausdrücke durch einen Datentyp modellieren und die Funktionen *nullable* und $r \;/\; x$ in Mini–F# Programme überführen. Für die erste Aufgabe ziehen wir die abstrakte Syntax regulärer Ausdrücke heran und transliterieren diese in einen rekursiven Variantentyp.

module Grammar.Regular

```
type Reg =
    | Eps                  // das leere Wort
    | Sym of Alphabet      // einzelnes Zeichen\Terminalsymbol
    | Cat  of Reg * Reg    // Konkatenation\Sequenz
    | Empty                // die leere Sprache
    | Alt  of Reg * Reg    // Alternative
    | Rep  of Reg          // Wiederholung
```

Der reguläre Ausdruck r_{00} kann jetzt unmittelbar mittels einer Wertebindung definiert

[6] Der Begriff Byte, der für eine Folge von 8 Bits steht, ist hier nicht allzu wörtlich zu nehmen.

werden.

```
let any = Alt (Sym A, Sym B)
let r00 = Cat (Rep any, Cat (Sym B, any))
```

module Grammar. Interpreter

Die Funktion *nullable* lässt sich ebenfalls sehr direkt übertragen: Die sechs Gleichungen werden zu den sechs Zweigen einer Fallunterscheidung.

```
let rec nullable (reg : Reg) : Bool =
  match reg with
  | Eps          → true
  | Sym _        → false
  | Cat (r1, r2) → nullable r1 && nullable r2
  | Empty        → false
  | Alt (r1, r2) → nullable r1 || nullable r2
  | Rep r        → true
```

Ähnlich direkt ist die Umsetzung der mathematischen Funktion $r \,/\, x$.

```
let rec divide (reg : Reg, x : Alphabet) : Reg =
  match reg with
  | Eps          → Empty
  | Sym a        → if x = a then Eps else Empty
  | Cat (r1, r2) → if nullable r1
                   then alt (cat (divide (r1, x), r2), divide (r2, x))
                   else      cat (divide (r1, x), r2)
  | Empty        → Empty
  | Alt (r1, r2) → alt (divide (r1, x), divide (r2, x))
  | Rep r        → cat (divide (r, x), Rep r)
```

Jetzt können wir uns vom Mini–F# Interpreter ausrechnen lassen, ob r_{00} nullable ist und wie die Rechtsfaktoren von r_{00} aussehen.

```
Mini⟩ nullable r00
false
Mini⟩ divide (r00, A)
Alt (Cat (Cat (Eps, Rep (Alt (Sym A, Sym B))),
          Cat (Sym B, Alt (Sym A, Sym B))),
     Cat (Empty, Alt (Sym A, Sym B)))
Mini⟩ divide (r00, B)
Alt (Cat (Cat (Eps, Rep (Alt (Sym A, Sym B))),
          Cat (Sym B, Alt (Sym A, Sym B))),
     Cat (Eps, Alt (Sym A, Sym B)))
```

Es ist bemerkenswert, dass *divide* (r_{00}, A) einen regulären Ausdruck zurückgibt, der zwar semantisch äquivalent zu r_{00} ist, aber nicht syntaktisch gleich. Das liegt daran,

dass bei der Konstruktion des Rechtsfaktors keine algebraischen Vereinfachungen wie etwa $\epsilon\ r = r$ vorgenommen werden. (Bei manuellen Umformungen nehmen wir solche Vereinfachungen fast automatisch vor.) Für die Definition des Akzeptors spielt das aber keine Rolle (warum?).

```
let rec generic-accept (reg : Reg, input : List ⟨Alphabet⟩) : Bool =
  match input with
  | []        → nullable reg
  | a :: rest → generic-accept (divide (reg, a), rest)
```

Hier einige Beispielaufrufe:

```
Mini⟩ accept (r00, [A; A; A])
false
Mini⟩ accept (r00, [A; B; A])
true
Mini⟩ let even-no-of-as = Rep (Alt (Cat (Sym A, Cat (Rep (Sym B), Sym A)), Sym B))
even-no-of-as : Reg
Mini⟩ accept (even-no-of-as, [A; B; A])
true
Mini⟩ accept (even-no-of-as, [A; B; B])
false
```

In Abbildung 6.7 ist noch einmal der gesamte Mini–F# Code zusammengefasst. Die Funktionen sind etwas allgemeiner als die im Text beschriebenen, da von einem konkreten Alphabet abstrahiert wird. Zudem werden bei der Konstruktion der regulären Ausdrücke einfache algebraische Identitäten ausgenutzt, um die Ausdrücke zu verkleinern. Diese Aufgabe übernehmen die sogenannten *cleveren Konstruktoren cat* und *alt*.

Fassen wir zusammen: Aus einem regulären Ausdruck lässt sich automatisch ein passender Akzeptor konstruieren. Je nach Verzahnungsgrad kann man mindestens drei Ansätze unterscheiden:

1. reiner Übersetzer,
2. Mischform aus Übersetzer und Interpreter,
3. reiner Interpreter.

Übersetzer und Interpreter sind keineswegs spezifisch für reguläre Ausdrücke, sondern stellen allgemeine Konzepte dar. Im Allgemeinen übersetzt ein Übersetzer ein Wort einer Sprache, der Quellsprache, in ein Wort einer anderen Sprache, der Zielsprache. Ein Interpreter interpretiert ein Wort direkt. In diesem Abschnitt ist die Quellsprache die Sprache der regulären Ausdrücke und die Zielsprache Mini–F# bzw. *Control*. Natürlich kann es sich auch bei der Quellsprache um eine Programmiersprache handeln. Mini–F# zum Beispiel wird von einem Übersetzer in eine Zwischensprache, die wesentlich einfacher als Mini–F# aufgebaut ist, übersetzt. Die Zwischensprache wird dann von einem Interpreter abgearbeitet. Mehr zum Themenkreis Übersetzer und Interpreter erfahren Sie in der Vorlesung Übersetzerbau.

```
type Reg ⟨'a⟩ = | Eps
                | Sym of 'a
                | Cat  of Reg ⟨'a⟩ * Reg ⟨'a⟩
                | Empty
                | Alt  of Reg ⟨'a⟩ * Reg ⟨'a⟩
                | Rep  of Reg ⟨'a⟩
let cat (r1 : Reg ⟨'a⟩, r2 : Reg ⟨'a⟩) : Reg ⟨'a⟩ =
  match (r1, r2) with
  | (Eps, r)     | (r, Eps)     → r
  | (Empty, r) | (r, Empty) → Empty
  | _                       → Cat (r1, r2)
let alt (r1 : Reg ⟨'a⟩, r2 : Reg ⟨'a⟩) : Reg ⟨'a⟩ =
  match (r1, r2) with
  | (Empty, r) | (r, Empty) → r
  | _                       → Alt (r1, r2)
let rec nullable = function
  | Eps          → true
  | Sym _        → false
  | Cat (r1, r2) → nullable r1 && nullable r2
  | Empty        → false
  | Alt (r1, r2) → nullable r1 || nullable r2
  | Rep r        → true
let rec divide (reg : Reg ⟨'a⟩, x : 'a) : Reg ⟨'a⟩ =
  match reg with
  | Eps          → Empty
  | Sym a        → if x = a then Eps else Empty
  | Cat (r1, r2) → if nullable r1 then alt (cat (divide (r1, x), r2), divide (r2, x))
                   else  cat (divide (r1, x), r2)
  | Empty        → Empty
  | Alt (r1, r2) → alt (divide (r1, x), divide (r2, x))
  | Rep r        → cat (divide (r, x), Rep r)
let rec generic-accept (reg : Reg ⟨'a⟩, input : List ⟨'a⟩) : Bool =
  match input with
  | []        → nullable reg
  | a :: rest → generic-accept (divide (reg, a), rest)
(* Anwendung *)
let any = Alt (Sym A, Sym B)
let r00  = Cat (Rep any, Cat (Sym B, any))
let accept-r00 = fun input → generic-accept (r00, input)
```

Abbildung 6.7.: Generischer Akzeptor (Interpreter).

6.2.2. Scanner

module Grammar. Token

Die Akzeptoren aus dem letzten Abschnitt beantworten die Frage „Ist ein gegebenes Wort in der von einem regulären Ausdruck bezeichneten Sprache enthalten?“. Ein Scanner hingegen unterteilt ein Wort in einzelne Teilwörter, die sogenannten *Lexeme*. Ausgangspunkt für die Unterteilung ist nicht ein einzelner regulärer Ausdruck, sondern eine Menge regulärer Ausdrücke: $\{r_1, \ldots, r_n\}$. Jeder Ausdruck beschreibt den Aufbau einer Sorte von Lexemen, siehe zum Beispiel Abbildung 6.3. Ein Scanner beantwortet also in konstruktiver Weise die Frage „Ist ein gegebenes Wort in der von $(r_1 \mid \cdots \mid r_n)^*$ bezeichneten Sprache enthalten?“. Die Antwort ist konstruktiv, da eine Aufteilung der Eingabe gleich mitgeliefert wird. Die Aufteilung ist allerdings in der Regel nicht eindeutig; ist zum Beispiel $r_1 = \mathit{digit}^+$ und $r_2 = \mathit{letter}\ (\mathit{letter} \mid \mathit{digit})^*$, dann gibt es für `a11` vier mögliche Aufteilungen: `a`|`1`|`1`, `a1`|`1`, `a`|`11` und `a11`. Aus diesem Grund vereinbart man zusätzlich, dass jeweils das *längste passende Wort* abgetrennt wird (engl. longest match). In unserem Beispiel wird damit `a11` in genau ein Lexem „unterteilt“. Diese Zusatzvereinbarung macht es etwas mühsam, die Akzeptoren aus dem letzten Abschnitt ohne große Änderungen zu Scannern umzurüsten. Stattdessen schauen wir uns an, wie man für einfach gestrickte lexikalische Syntaxen einen Scanner von Hand programmiert.

Wir nehmen an, dass die Eingabe als Liste von Eingabezeichen gegeben ist. Die Eingabe `(4+7)*11` zum Beispiel wird durch die Liste

```
Mini⟩ explode "(4+7)*11"
['('; '4'; '+'; '7'; ')'; '*'; '1'; '1']
```

repräsentiert. (Die Funktion *explode* überführt eine Zeichenkette in eine Liste von Zeichen; umkehrt wird eine solche Liste von *implode* in eine Zeichenkette überführt.) Wir stellen uns die Aufgabe, einen Scanner für die lexikalische Syntax einfacher arithmetischer Ausdrücke zu schreiben: $\{\mathit{digit}^+, \texttt{(}, \texttt{)}, \texttt{*}, \texttt{+}, \mathit{space}\}$ ist die zugrundeliegende Menge regulärer Ausdrücke. Dabei steht *space* für einen beliebigen Zwischenraum (engl. white space), etwa ein Leerzeichen oder einen Zeilenvorschub. Als Ausgabe soll eine Liste von Lexemen erzeugt werden. Lexeme werden mit Hilfe des Variantentyps

```
type Token = | Num of Nat | LParen | RParen | Asterisk | Plus
```

repräsentiert. Die offene Klammer `(` wird zum Beispiel durch den Konstruktor *LParen* dargestellt; das Numeral `4711` durch den Wert *Num* 4711. Das Lexem (engl. token) für Numerale führt sozusagen seinen semantischen Wert mit. Zwischenräume werden in der Ausgabe nicht aufgeführt. Die Eingabe `(4+7)*11` sollte vom Scanner in die Liste

[*LParen*; *Num* 4; *Plus*; *Num* 7; *RParen*; *Asterisk*; *Num* 11]

überführt werden.

Die lexikalische Syntax ist besonders einfach, da das erste Zeichen der Eingabe das Lexem eindeutig bestimmt — im Gegensatz zu Mini–F#: `l` kann das erste Zeichen des Schlüsselworts `let` sein oder das erste Zeichen eines Bezeichners. Aus diesem Grund führt das Struktur Entwurfsmuster für Listen fast direkt zum Ziel, siehe Abbildung 6.8.

```
type Token = | Num of Nat | LParen | RParen | Asterisk | Plus
let rec lex (list : List ⟨Char⟩) : List ⟨Token⟩ =
  match list with
  | []        → []
  | c :: rest → if    c = '(' then LParen  :: lex rest
                elif  c = ')' then RParen  :: lex rest
                elif  c = '*' then Asterisk :: lex rest
                elif  c = '+' then Plus      :: lex rest
                elif  Char.IsDigit c then
                  let (xs1, xs2) = split-while Char.IsDigit list
                  in  Num (Nat.Parse (implode xs1)) :: lex xs2
                elif  Char.IsWhiteSpace c then lex rest
                else panic "illegal character"
```

Abbildung 6.8.: Handgeschriebener Scanner für einfache arithmetische Ausdrücke.

Die einzige „Schwierigkeit“ bereiten Numerale. Wir müssen die Liste weiterverfolgen, solange wir Ziffern sehen. Diese Aufgabe übernimmt eine nützliche Hilfsfunktion namens *split-while*: Der Aufruf *split-while pred list* teilt *list* in zwei Listen $list_1$ und $list_2$ auf; für alle Elemente x in $list_1$ gilt, dass *pred* x zu *true* auswertet; weiterhin ist $list_1$ das längste Anfangsstück mit dieser Eigenschaft.

```
Mini⟩ split-while (fun n → n % 2 = 0) (0 :: [2 .. 5])
([0; 2], [3; 4; 5])
Mini⟩ split-while Char.IsDigit (explode "24me")
(['2'; '4'], ['m'; 'e'])
Mini⟩ split-while Char.IsWhiteSpace (explode "24me")
([], ['2'; '4'; 'm'; 'e'])
```

Die Bibliotheksfunktion *Char.IsDigit* testet, ob das Argument eine Ziffer ist; entsprechend überprüft *Char.IsWhiteSpace*, ob das Argument ein Zwischenraum ist. Die Definition von *split-while* folgt exakt dem Struktur Entwurfsmuster.

```
let rec split-while (pred : 'a → Bool) = function
  | []      → ([], [])
  | x :: xs → if pred x then let (xs1, xs2) = split-while pred xs in (x :: xs1, xs2)
              else  ([], x :: xs)
```

Bleibt noch zu klären, was *Nat.Parse* und *panic* bedeuten: Die Funktion *Nat.Parse* wird benutzt, um einen Ziffernfolge in eine natürliche Zahl umzuwandeln: Der Aufruf

Nat.Parse `"4711"` zum Beispiel ergibt 4711. Die Funktion *panic* schließlich bricht eine Rechnung mit einer Fehlermeldung ab (siehe Abschnitt 7.4).

6.3. Kontextfreie Grammatiken⟍kontextfreie Syntax

Wir haben im letzten Abschnitt gesehen, dass reguläre Ausdrücke nicht besonders ausdrucksstark sind. Syntaktische Hygienevorschriften wie „zu jeder offenen Klammer muss es eine korrespondierende schließende Klammer geben“ lassen sich nicht formulieren.

$$\{\texttt{a}^n\ \texttt{b}^n \mid n \in \mathbb{N}\}$$

Die Sprache der wohlgeformten Klammerausdrücke lässt sich aber schrittweise (im Fachjargon: induktiv) definieren: ϵ ist wohlgeformt; wenn w wohlgeformt ist, dann auch `a` w `b`. Bei der Definition der Klammerausdrücke wird auf die definierte Sprache selbst zurückgegriffen. Mit einem Wort, wir benötigen *Rekursion*. Zur Erinnerung: Mini–F# erlaubt rekursiv definierte Funktionen und rekursiv definierte Variantentypen. Hier erweitern wir reguläre Ausdrücke um rekursiv definierte Sprachen. Die Sprache der wohlgeformten Klammerausdrücke zum Beispiel wird durch den Ausdruck

$$\textit{rec}\ x \to \epsilon \mid \texttt{a}\ x\ \texttt{b}$$

beschrieben, der ziemlich direkt die obige Bildungsvorschrift einfängt. Wenn man möchte, kann man den Pfeil „→“ als Implikation („wenn …, dann …“) lesen: Wenn x ein Element der Sprache ist, dann auch ϵ und `a` x `b`.

Die Erweiterung von regulären Ausdrücken um Rekursion bekommt einen neuen Namen: Wir sprechen von *kontextfreien Ausdrücken* oder *kontextfreien Grammatiken*.[7] Kontextfrei, weil der Bezeichner x in *rec* $x \to e$ ohne Einschränkungen, insbesondere ohne Berücksichtigung des jeweiligen Kontextes, verwendet werden kann. Neben regulären und kontextfreien Sprachen — das sind Sprachen, die durch reguläre bzw. kontextfreie Ausdrücke bezeichnet werden — gibt es auch *kontextsensitive Sprachen*. Die Sprachfamilien formen eine strikte Hierarchie: Jede reguläre Sprache ist auch kontextfrei, jede kontextfreie Sprache auch kontextsensitiv, aber nicht umgekehrt. Mehr zu diesem Thema später aus der Abteilung der Theoretischen Informatik.

Wir werden sehen, dass wir mit kontextfreien Ausdrücken die Syntax von Mini–F# hinreichend gut beschreiben können. Aber auch dieser Formalismus hat seine Grenzen: Kontextfreie Ausdrücke können eben keine kontextsensitiven Einschränkungen ausdrücken. Schauen wir uns ein kleines Beispiel an. Der folgende kontextfreie Ausdruck modelliert eine winzige Teilmenge von Mini–F#.

$$\textit{rec}\ \textit{expr} \to \texttt{0} \mid \texttt{false} \mid \textit{expr}\ \texttt{+}\ \textit{expr}$$

[7] Der Terminus „Grammatik“ wird eigentlich für einen *anderen*, aber äquivalenten Formalismus verwendet. Wir missbrauchen den Fachbegriff im Folgenden, um nicht immer von einem „kontextfreien Ausdruck“ sprechen zu müssen.

Der Bezeichner *expr*, der stellvertretend für einen beliebigen Mini–F# Ausdruck steht, kann ohne Einschränkungen rechts verwendet werden. Wir erinnern uns: Mini–F# Ausdrücke sind beliebig kombinierbar. Insbesondere erlaubt die Grammatik einen Booleschen Ausdruck, etwa `false`, in einem Kontext zu verwenden, in dem ein arithmetischer Ausdruck erwartet wird: `false + 0`. Diese Einschränkungen können wir nicht mit der kontextfreien Syntax ausdrücken. Müssen wir aber auch nicht; um diese Dinge kümmert sich ja gerade die statische Semantik. Durch die klare Trennung der Zuständigkeiten wird die Sprachdefinition von Mini–F# ungemein erleichtert.

6.3.1. Abstrakte Syntax

Kontextfreie Ausdrücke umfassen neben den Konstrukten regulärer Ausdrücke zusätzlich Bezeichner (auch *Nichtterminalsymbole* genannt) und Rekursion.

$x \in \mathsf{Id}$

$c \in \mathsf{CF} ::=$		*kontextfreie Ausdrücke:*
	a	einzelnes Zeichen↘Terminalsymbol
|	x	Bezeichner↘Nichtterminalsymbol
|	ϵ	das leere Wort
|	$c_1\ c_2$	Konkatenation↘Sequenz
|	$\emptyset$	die leere Sprache
|	$c_1 \mid c_2$	Alternative
|	$\mathit{rec}\ x \to c$	Rekursion

Die Wiederholung ist kein primitives Konzept mehr; sie kann mit Hilfe der Rekursion ausgedrückt werden: c^* durch $\mathit{rec}\ x \to \epsilon \mid c\ x$ oder $\mathit{rec}\ x \to \epsilon \mid x\ c$. (Streng genommen ist auch $\emptyset$, das Symbol für die leere Sprache, redundant. Es kann — wie wir in Kürze sehen werden — mit $\mathit{rec}\ x \to x$ ausgedrückt werden.)

6.3.2. Reduktionssemantik

Kommen wir zur Semantik kontextfreier Ausdrücke. Wir haben motiviert, dass der Ausdruck $\mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ die Sprache $\{\mathtt{a}^n\ \mathtt{b}^n \mid n \in \mathbb{N}\}$ bezeichnet. Das dem so ist, leuchtet vielleicht unmittelbar ein. Aber ist auch die Bedeutung von $\mathit{rec}\ x \to x$, $\mathit{rec}\ x \to \mathtt{a}\ x\ \mathtt{b}$ oder $\mathit{rec}\ x \to x^*$ unmittelbar klar? Vielleicht, vielleicht aber auch nicht. Legen wir also die Semantik präzise fest, um die Bedeutung dieser „extremen“ Beispiele ermitteln zu können. Merke: Semantik ist immer auch dazu da, Randfälle oder Extremfälle zu klären.

Diesmal fangen wir mit der *Reduktionssemantik* an, die — wir erinnern uns — eine lokale Sicht der Dinge vermittelt. Die Reduktionsregel für $\mathit{rec}\ x \to c$ formalisiert die Intuition. Ein rekursiver Ausdruck wird einmal „aufgefaltet“: Die Vorkommen von x in c werden durch den Ausdruck $\mathit{rec}\ x \to c$ selbst ersetzt, als Formel $c\{x \mapsto (\mathit{rec}\ x \to c)\}$. (Warum benötigen wir keine Regel für Bezeichner?)

$$\frac{}{(\mathit{rec}\ x \to c) \longrightarrow c\{x \mapsto (\mathit{rec}\ x \to c)\}}$$

Mit Hilfe dieser Regel können wir zum Beispiel `aabb` aus dem Ausdruck $\mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ ableiten.

$$
\begin{aligned}
& \mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b} \\
\longrightarrow\ & \epsilon \mid \mathtt{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\epsilon \mid \mathtt{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\mathtt{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\mathtt{a}\ (\epsilon \mid \mathtt{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b})\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\mathtt{a}\ \epsilon\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ \mathtt{a}\ \mathtt{b}\ \mathtt{b}
\end{aligned}
$$

Wie sieht es mit dem Ausdruck $\mathit{rec}\ x \to x$ aus? Nun, wir erhalten:

$$
\begin{aligned}
& \mathit{rec}\ x \to x \\
\longrightarrow\ & \mathit{rec}\ x \to x \\
& \vdots
\end{aligned}
$$

Man kann nicht behaupten, dass wir große Fortschritte erzielt haben: Der Rumpf x wird durch $\mathit{rec}\ x \to x$ ersetzt und wir erhalten wieder $\mathit{rec}\ x \to x$. Mit anderen Worten, wir können kein Wort ableiten; $\mathit{rec}\ x \to x$ bezeichnet demnach die leere Sprache. Das Gleiche gilt für den kontextfreien Ausdruck $\mathit{rec}\ x \to \mathtt{a}\ x\ \mathtt{b}$. Im Fall von $\mathit{rec}\ x \to x^*$ sieht es ein kleines bisschen „besser“ aus.

$$
\begin{aligned}
& \mathit{rec}\ x \to x^* \\
\longrightarrow\ & (\mathit{rec}\ x \to x^*)^* \\
\longrightarrow\ & \epsilon
\end{aligned}
$$

Ein anderes Wort lässt sich nicht ableiten. Also steht $\mathit{rec}\ x \to x^*$ für die Sprache $\{\epsilon\}$.

Die Ersetzung von Bezeichnern durch kontextfreie Ausdrücke nennt man übrigens *Substitution*: Ist c ein kontextfreier Ausdruck und $\sigma \in \mathsf{Id} \to_{\mathrm{fin}} \mathsf{CF}$ eine endliche Abbildung von Bezeichnern auf Ausdrücke, dann bezeichnet $c\sigma$ den Ausdruck, in dem die *frei* auftretenden Bezeichner aus $\mathit{dom}\ \sigma$ durch die zugeordneten Ausdrücke ersetzt werden.

$$
\begin{aligned}
a\sigma &= a \\
x\sigma &= \begin{cases} \sigma(x) & \text{falls } x \in \mathit{dom}\ (\sigma) \\ x & \text{sonst} \end{cases} \\
\epsilon\sigma &= \epsilon \\
(c_1\ c_2)\sigma &= (c_1\sigma)\ (c_2\sigma) \\
\emptyset\sigma &= \emptyset \\
(c_1 \mid c_2)\sigma &= (c_1\sigma) \mid (c_2\sigma) \\
(\mathit{rec}\ x \to c)\sigma &= \mathit{rec}\ x \to c(\sigma \setminus \{x\})
\end{aligned}
$$

Ersetzen wir Bezeichner in dem Ausdruck ***rec*** $x \to c$, müssen wir sicherstellen, dass der *gebundene* Bezeichner x *nicht* ersetzt wird. Zum Beispiel ist $(\textbf{\textit{rec}}\ x \to x)\{x \mapsto \epsilon\} = \textbf{\textit{rec}}\ x \to x$ und nicht etwa $\textbf{\textit{rec}}\ x \to \epsilon$. Zur Einnerung: $\sigma \setminus \{x\}$ bezeichnet die *Einschränkung* von σ auf $\mathit{dom}\ \sigma \setminus \{x\}$.

6.3.3. Denotationelle Semantik

Kommen wir zur denotationellen Semantik, der Semantik, die eine globale Sicht vermittelt. Bei der Motivation von ***rec*** $x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ haben wir überlegt, wie sich zum Beispiel $\mathtt{a}^n\ \mathtt{b}^n$ schrittweise bilden lässt: aus ϵ wird $\mathtt{ab}$, daraus wird $\mathtt{aabb}$ usw. Dieser Bildungsprozess lässt sich auch auf die *Sprache als Ganzes* übertragen. Aus dem Ausdruck ***rec*** $x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ leiten wir zunächst eine Funktion auf Sprachen ab.

$$F(L) = \{\epsilon\} \cup \{\mathtt{a}\} \cdot L \cdot \{\mathtt{b}\}$$

Jetzt können wir ausgehend von der leeren Menge durch wiederholte Anwendung von F uns der Semantik von ***rec*** $x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ nähern.

$$\begin{array}{lll}
\emptyset & & \\
F(\emptyset) & = \{\epsilon\} \cup \{\mathtt{a}\} \cdot \emptyset \cdot \{\mathtt{b}\} & = \{\epsilon\} \\
F(F(\emptyset)) & = \{\epsilon\} \cup \{\mathtt{a}\} \cdot \{\epsilon\} \cdot \{\mathtt{b}\} & = \{\epsilon, \mathtt{ab}\} \\
F(F(F(\emptyset))) & = \{\epsilon\} \cup \{\mathtt{a}\} \cdot \{\epsilon, \mathtt{ab}\} \cdot \{\mathtt{b}\} & = \{\epsilon, \mathtt{ab}, \mathtt{aabb}\} \\
F(F(F(F(\emptyset)))) & = \{\epsilon\} \cup \{\mathtt{a}\} \cdot \{\epsilon, \mathtt{ab}, \mathtt{aabb}\} \cdot \{\mathtt{b}\} & = \{\epsilon, \mathtt{ab}, \mathtt{aabb}, \mathtt{aaabbb}\}
\end{array}$$

Wir haben bewusst davon gesprochen, dass wir uns der Semantik *nähern*, da wir in endlich vielen Schritten niemals die unendliche Menge $\{\mathtt{a}^n\ \mathtt{b}^n \mid n \in \mathbb{N}\}$ konstruieren können. Mathematisch lässt sich die Menge aber sehr wohl definieren, sie ergibt sich als Vereinigung aller endlichen Approximationen.

$$\bigcup\{F^n(\emptyset) \mid n \in \mathbb{N}\}$$

Hier bezeichnet $F^n(\emptyset)$ die n-fache Anwendung der Funktion F auf $\emptyset$: $F^0(X) = X$ und $F^{n+1}(X) = F(F^n(X))$. Die obige Sprache, nennen wir sie S, hat die Eigenschaft, dass F sie unverändert lässt: $F(S) = S$. Man sagt auch, S ist ein *Fixpunkt* von F. Darüber hinaus ist S die *kleinste* Menge mit dieser Eigenschaft: S ist der *kleinste Fixpunkt*. Kurz: Alles Notwendige ist drin, mehr aber nicht. Für den mathematischen Nachweis dieser Tatsache müssten wir etwas ausholen, was wir an dieser Stelle nicht tun wollen, siehe aber Aufgabe 6.11.

Jetzt haben wir fast alles zusammen, um die semantischen Gleichungen der denotationellen Semantik aufzustellen. Wir müssen uns nur noch um die Bezeichner kümmern. Deren Bedeutung halten wir in einer Umgebung fest, die Bezeichner auf Sprachen abbildet.

$$\varrho \in \mathsf{Id} \to_{\text{fin}} \mathbb{P}(A^*)$$

Die Semantikfunktion bildet einen kontextfreien Ausdruck, der innerhalb der Oxfordklammern steht, und eine Umgebung auf eine Sprache ab.

$$
\begin{array}{ll}
[\![a]\!]\varrho & = \{a\} \\
[\![x]\!]\varrho & = \varrho(x) \\
[\![\epsilon]\!]\varrho & = \{\epsilon\} \\
[\![c_1\ c_2]\!]\varrho & = [\![c_1]\!]\varrho \cdot [\![c_2]\!]\varrho \\
[\![\emptyset]\!]\varrho & = \emptyset \\
[\![c_1 \mid c_2]\!]\varrho & = [\![c_1]\!]\varrho \cup [\![c_2]\!]\varrho \\
[\![\textbf{\textit{rec}}\ x \to c]\!]\varrho & = \bigcup\{F^n(\emptyset) \mid n \in \mathbb{N}\} \ \text{ mit } \ F(X) = [\![c]\!](\varrho, \{x \mapsto X\})
\end{array}
$$

Die letzte Gleichung formalisiert die oben beschriebene Fixpunktsemantik: Aus dem Ausdruck ***rec*** $x \to c$ wird eine Funktion F auf Sprachen abgeleitet (‘,’ ist der Kommaoperator); deren kleinster Fixpunkt ist die Bedeutung des Ausdrucks. Ist die Umgebung leer, $\varrho = \varnothing$, schreiben wir $[\![c]\!]\varrho$ kurz als $[\![c]\!]$.

Gehen wir die „extremen“ Beispiele noch einmal mit der denotationellen Semantik durch. Die dem Ausdruck ***rec*** $x \to x$ zugeordnete Funktion ist $F(L) = L$. Die Funktion hat unendlich viele Fixpunkte — jede Sprache ist Fixpunkt dieser Funktion. Der kleinste Fixpunkt ist die leere Sprache, somit ist $[\![\textbf{\textit{rec}}\ x \to x]\!] = \emptyset$. Ähnlich verhält es sich mit dem Ausdruck ***rec*** $x \to \texttt{a}\ x\ \texttt{b}$. Die zugehörige Funktion ist $G(L) = \{\texttt{a}\} \cdot L \cdot \{\texttt{b}\}$. Da $G(\emptyset) = \emptyset$ gilt, ist die Bedeutung von ***rec*** $x \to \texttt{a}\ x\ \texttt{b}$ ebenfalls die leere Sprache. Für das dritte Beispiel, ***rec*** $x \to x^*$, erhalten wir $H(L) = L^*$ mit

$$
\begin{array}{l}
\emptyset \\
H(\emptyset) \quad = \emptyset^* \quad = \{\epsilon\} \\
H(H(\emptyset)) = \{\epsilon\}^* = \{\epsilon\}
\end{array}
$$

Damit ist $[\![\textbf{\textit{rec}}\ x \to x^*]\!] = \{\epsilon\}$. Auch für kontextfreie Ausdrücke stimmen Reduktionssemantik und denotationelle Semantik überein.

6.3.4. Vertiefung

Im Folgenden spezifizieren wir schrittweise die kontextfreie Syntax von Mini–F#, eingeschränkt auf die in Kapitel 3 eingeführten Konstrukte. Dabei werden wir einige Schwierigkeiten zu bewältigen haben. Um diesen zu begegnen, wenden wir einige Tricks an und gehen verschiedene Kompromisse ein. Beides erinnert noch einmal daran, dass die konkrete Syntax erstens technischen Einschränkungen unterworfen ist und zweitens den Geschmack der Sprachdesigner widerspiegelt.

Das zugrundeliegende Alphabet ist die Menge aller Mini–F# Lexeme. Wir verwenden *num* als Bezeichner für die Sprache aller Numerale, *string* steht entsprechend für die Sprache aller Stringliterale und *id* bzw. *Id* für kleine bzw. große Bezeichner. Alle anderen Lexeme notieren wir, wie in Abbildung 6.3 aufgeführt.

Versuchen wir uns an der kontextfreien Syntax einfacher arithmetischer Ausdrücke.

rec *expr* → *num* | *expr* **+** *expr* | *expr* ***** *expr*

Der kontextfreie Ausdruck ist an die Baumsprache für Ausdrücke angelehnt: Ein arithmetischer Ausdruck ist entweder ein Numeral, oder ein Ausdruck gefolgt von dem Symbol **+** gefolgt von einem weiteren Ausdruck, oder ein Ausdruck gefolgt von dem Symbol

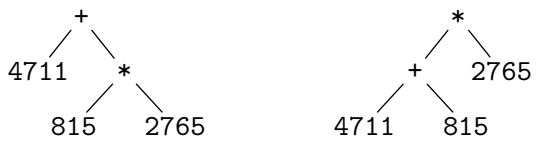
`*` gefolgt von einem weiteren Ausdruck. Aus dem kontextfreien Ausdruck lässt sich zum Beispiel das Wort `4711 + 815 * 2765` ableiten — wir kürzen den kontextfreien Ausdruck mit E ab und führen der Übersichtlichkeit halber nicht alle Reduktionsschritte auf:

$$
\begin{aligned}
E &\longrightarrow E \texttt{ + } E \\
&\longrightarrow \mathit{num} \texttt{ + } E \\
&\longrightarrow \texttt{4711 + } E \\
&\longrightarrow \texttt{4711 + } E \texttt{ * } E \\
&\longrightarrow \texttt{4711 + } \mathit{num} \texttt{ * } E \\
&\longrightarrow \texttt{4711 + 815 * } E \\
&\longrightarrow \texttt{4711 + 815 * } \mathit{num} \\
&\longrightarrow \texttt{4711 + 815 * 2765}
\end{aligned}
$$

Es gibt aber noch eine zweite mögliche Reduktionsfolge:

$$
\begin{aligned}
E &\longrightarrow E \texttt{ * } E \\
&\longrightarrow E \texttt{ + } E \texttt{ * } E \\
&\longrightarrow \mathit{num} \texttt{ + } E \texttt{ * } E \\
&\longrightarrow \texttt{4711 + } E \texttt{ * } E \\
&\longrightarrow \texttt{4711 + } \mathit{num} \texttt{ * } E \\
&\longrightarrow \texttt{4711 + 815 * } E \\
&\longrightarrow \texttt{4711 + 815 * } \mathit{num} \\
&\longrightarrow \texttt{4711 + 815 * 2765}
\end{aligned}
$$

Damit stehen wir dem ersten Problem gegenüber: Der obige kontextfreie Ausdruck ist *mehrdeutig*; ein Wort kann auf verschiedene Weisen abgeleitet werden.[8] Warum ist das ein Problem? Nun, die kontextfreie Syntax dient einzig und allein dem Zweck, aus der linearen Folge von Lexemen die hierarchische Struktur eines Programms zu rekonstruieren. (Im nächsten Abschnitt sehen wir uns an, wie man *automatisch* die konkrete Syntax in die abstrakte Syntax überführt.) Die beiden unterschiedlichen Reduktionsfolgen legen aber einen unterschiedlichen hierarchischen Aufbau nahe: Die erste Reduktionsfolge sieht den Operator `+` oben, die zweite den Operator `*`. Entsprechend haben wir zwei abstrakte Syntaxbäume zur Auswahl.



Die Semantik ordnet den beiden Syntaxbäumen eine unterschiedliche Bedeutung zu. Kurzum: Der obige kontextfreie Ausdruck ist ungeeignet zur Beschreibung arithmetischer Ausdrücke, da er nicht jedem Ausdruck einen eindeutigen abstrakten Syntaxbaum zuordnet.

[8] Diese Tatsache ist übrigens mit der denotationellen Semantik nicht fassbar, da jedem kontextfreien Ausdruck eine *Menge* und keine *Multimenge* zugeordnet wird.

Was ist zu tun? Nun, wir können die Syntax von Mini–F# Ausdrücken überdenken und zum Beispiel arithmetische Operatoren nicht zwischen die Operatoren schreiben, sondern davor oder dahinter. Entsprechend der Position spricht man von *Präfix-*, *Infix-* oder *Postfixnotation*. Präfix- und Postfixnotation lassen sich wie folgt einfangen:

rec *expr* → *num* | `+` *expr expr* | `*` *expr expr*

rec *expr* → *num* | *expr expr* `+` | *expr expr* `*`

Beide Syntaxen sind *eindeutig*. Wir haben in Abschnitt 2.2 erwähnt, dass die Programmiersprache *Scheme* Präfixnotation und die Programmiersprache *PostScript* Postfixnotation verwendet. Die allermeisten Sprachen notieren aber — wie auch Mini–F# — Operatoren infix. Warum? Wahrscheinlich, weil sie der mathematischen Tradition folgen. Das ist natürlich nur eine halbwegs befriedigende Antwort, klärt sie doch nicht, warum die Notation tatsächlich sinnvoll ist. Um den Gründen auf die Spur zu kommen, betrachten wir eine Summe aus drei Zahlen:

```
4711 + 815 + 2765
```

Notieren wir die Operatoren präfix, dann gibt es zwei alternative Ausdrücke, nämlich `+ 4711 + 815 2765` und `+ + 4711 815 2765`. Für die Postfixnotation gilt das gleiche. Semantisch sind beide Ausdrücke aber gleich, da die Addition *assoziativ* ist. Präfix- und Postfixnotation machen also einen Unterschied, wo es keinen gibt! Allein die Infixnotation stellt uns nicht vor die Wahl.

Assoziative Operatoren und Funktionen sind zahlreich: Disjunktion (`||`), Konjunktion (`&&`), Addition (+), Multiplikation (∗), Konkatenation von Strings (`^`), Minimum (*min*), Maximum (*max*), Konkatenation von Listen (@), der Kommaoperator (‘,’) sind Vertreter dieser Gattung. Wir sehen: Nicht alle assoziativen Funktionen notieren wir tatsächlich infix, bei allen würde es sich aber anbieten.

Wo Licht ist, ist auch Schatten. Es hat sich eingebürgert, auch nicht-assoziative Funktionen, wie zum Beispiel die Subtraktion infix zu notieren: `- 4711 - 815 2765` und `- - 4711 815 2765` sind sehr wohl unterschiedlich und die Infixschreibweise

```
4711 - 815 - 2765                                    (6.2)
```

klärt nicht, welche Variante gemeint ist. Ähnliches gilt für die Potenzfunktion (wenn wir uns entschließen, *power* (x, n) infix zu notieren, etwa als x `**` n). An dieser Stelle bedarf es einer Festlegung: Wir müssen vereinbaren, was mit (6.2) gemeint ist. Die Subtraktion wird allgemein als *linksassoziierend*[9] festgelegt (a `-` b `-` c steht für `- -` a b c), die Potenzfunktion als *rechtsassoziierend* (a `**` b `**` c steht für `**` a `**` b c).[10] Was aber machen wir, wenn wir die andere Variante benötigen, die wir ja infix nicht ausdrücken

[9] Diese syntaktische Konvention wird manchmal auch als „linksassoziativ“ bezeichnet; wir vermeiden diesen Begriff, da er zu sehr nach der semantischen Eigenschaft „assoziativ“ klingt.

[10] Eine solche Festlegung kann übrigens auch im Fall assoziativer Funktionen sinnvoll sein. Nämlich dann, wenn eine Variante effizienter ist als die andere. Ein Beispiel hierfür liefert die Konkatenation von Listen: xs_1 @ (xs_2 @ xs_3) und (xs_1 @ xs_2) @ xs_3 sind zwar gleichwertig, aber der erste Ausdruck ist schneller ausgerechnet.

können? In diesem Fall behilft man sich mit *Klammern*, die die Gruppierung explizit machen:

```
4711 - (815 - 2765)
```

Wir erinnern uns: Klammern sind ein Hilfsmittel der konkreten Syntax. Mit ihrer Hilfe lassen sich die Begriffe links- und rechtsassoziierend noch einmal verdeutlichen: Ist ein Operator '$\oplus$' linksassoziierend, dann meint $a \oplus b \oplus c$ den Ausdruck $(a \oplus b) \oplus c$ (Klammern links), ist der Operator rechtsassoziierend, dann ist der Ausdruck $a \oplus (b \oplus c)$ gemeint (Klammern rechts).

Bislang haben wir nur einen Operator für sich betrachtet. Bietet eine Sprache mehrere Operatoren an, dann muss man weiterhin klären, was passiert, wenn zwei Operatoren aufeinandertreffen. Damit kommen wir zu unserem Ausgangsbeispiel zurück.

```
4711 + 815 * 2765
```

Ist damit `(4711 + 815) * 2765` oder `4711 + (815 * 2765)` gemeint? Gängige Konvention — Punkt- vor Strichrechnung — gibt der zweiten Alternative den Vorzug. Hat man zwei beliebige Operatoren vor sich, zum Beispiel $\oplus$ und $\otimes$, so lässt sich der Konflikt mit Hilfe der sogenannten *Bindungsstärke* lösen.

$$a \oplus b \otimes c$$

Man stellt sich vor, dass $\oplus$ und $\otimes$ um den Operanden b streiten; der Operator mit der höheren Bindungsstärke zieht ihn an sich. Also, hat $\oplus$ die höhere Bindungsstärke, dann ist der Ausdruck $(a \oplus b) \otimes c$ gemeint, hat $\otimes$ die höhere Bindungsstärke, dann entsprechend der Ausdruck $a \oplus (b \otimes c)$.

Zurück zu unserer Aufgabe, der Aufstellung einer Syntax für arithmetische Ausdrücke. Wir können den kontextfreien Ausdruck E eindeutig machen, indem wir unser neu erworbenes Wissen über Operatoren in die Beschreibung „hineinprogrammieren“. Die grundlegende Idee ist, eine Hierarchie E_i von Sprachen zu definieren, so dass E_i nur Ausdrücke umfasst, deren oberster Operator eine Bindungsstärke von i oder mehr hat. Konkret: E_0 umfasst alle Ausdrücke, E_1 nur Produkte und E_2 nur atomare oder geklammerte Ausdrücke.

rec $expr_0 \rightarrow expr_1 \mid expr_1$ `+` $expr_0$
and $expr_1 \rightarrow expr_2 \mid expr_2$ `*` $expr_1$
and $expr_2 \rightarrow num \mid$ `(` $expr_0$ `)`

Die Grammatik ist *verschränkt rekursiv*: Alle Bezeichner sind in allen rechten Seiten sichtbar. Der kontextfreie Ausdruck ist eindeutig: Jeder arithmetische Ausdruck lässt sich auf genau eine Art und Weise ableiten. Zum Beispiel (E_i kürzt $expr_i$ ab):

$$
\begin{aligned}
E_0 &\longrightarrow E_1 \texttt{ + } E_0 \\
&\longrightarrow E_2 \texttt{ + } E_0 \\
&\longrightarrow \mathit{num} \texttt{ + } E_0 \\
&\longrightarrow \texttt{4711 + } E_0 \\
&\longrightarrow \texttt{4711 + } E_1 \\
&\longrightarrow \texttt{4711 + } E_2 \texttt{ * } E_1 \\
&\longrightarrow \texttt{4711 + } \mathit{num} \texttt{ * } E_1 \\
&\longrightarrow \texttt{4711 + 815 * } E_1 \\
&\longrightarrow \texttt{4711 + 815 * } E_2 \\
&\longrightarrow \texttt{4711 + 815 * } \mathit{num} \\
&\longrightarrow \texttt{4711 + 815 * 2765}
\end{aligned}
$$

Vorläufiges Fazit: Die naheliegende Syntax für arithmetische Ausdrücke ist mehrdeutig. Um die Syntax eindeutig zu machen, muss man Vereinbarungen über die Assoziierung (links- oder rechtsassoziierend) und die Bindungsstärke treffen. Lässt man die Vereinbarungen in die Sprachbeschreibung einfließen, nimmt diese an Umfang zu und an Leserlichkeit ab. In der Praxis belässt man es oft bei der mehrdeutigen Syntax und führt die zusätzlichen Vereinbarungen getrennt davon auf. So werden wir es auch halten.

Die Erweiterung von kontextfreien Ausdrücken um verschränkte Rekursion ist auch notwendig für die Beschreibung von ***in***-Ausdrücken. Diese involvieren eine zweite syntaktische Kategorie: Deklarationen.

$$
\begin{aligned}
&\textbf{\textit{rec}}\ \ \mathit{expr} \rightarrow \mathit{id} \mid \mathit{num} \mid \mathit{expr} \texttt{ + } \mathit{expr} \mid \mathit{expr} \texttt{ * } \mathit{expr} \mid \mathit{decl}^* \texttt{ in } \mathit{expr} \\
&\textbf{\textit{and}}\ \mathit{decl} \rightarrow \texttt{let } \mathit{id} \texttt{ = } \mathit{expr}
\end{aligned}
$$

Ausdrücke und Deklarationen sind verschränkt rekursiv: Ausdrücke beinhalten Deklarationen und umgekehrt.

Das Problem der Mehrdeutigkeiten ist leider nicht auf Infix-Operatoren beschränkt. Der Ausdruck ***let*** $n = 4711$ ***in*** $n + n$ ist zum Beispiel mehrdeutig: Ist damit (***let*** $n = 4711$ ***in*** n) $+ n$ oder ***let*** $n = 4711$ ***in*** $(n + n)$ gemeint? Der Unterschied ist groß, meint doch das zweite Vorkommen von n in beiden Ausdrücken etwas anderes! Wir treffen die Vereinbarung, dass sich ein ***in***-Ausdruck so weit nach rechts wie möglich erstreckt. Damit wird ***let*** $n = 4711$ ***in*** $(n + n)$ als Bedeutung des obigen Ausdrucks festgelegt. Genau wie die Vereinbarungen über Assoziierung und Bindungsstärke kann man auch diese Metaregel in die Syntax hineinprogrammieren oder als separate Bemerkung zur Sprachbeschreibung hinzufügen.

Alternativ könnte man das Ende von ***in***-Ausdrücken explizit markieren, zum Beispiel mit dem Schlüsselwort ***end***. Weder ***let*** $n = 4711$ ***in*** n ***end*** $+ n$ noch ***let*** $n = 4711$ ***in*** $n +$ n ***end*** ließen einen Interpretationsspielraum zu.

Die Vereinbarung „so weit nach rechts wie möglich“ kommt in Mini–F# auch bei Alternativen (***if*** e_1 ***then*** e_2 ***else*** e_3) und Funktionsausdrücken bzw. anonymen Funktionen (***fun*** $x \rightarrow e$) zum Zuge. Der Ausdruck ***fun*** $x \rightarrow x + x$ ist mehrdeutig: (***fun*** $x \rightarrow x$)$+x$ und ***fun*** $x \rightarrow (x + x)$ stehen als Interpretationen zur Wahl. Der erste Ausdruck ist nicht typkorrekt. Um Typkorrektheit kümmert sich zwar die kontextfreie Syntax nicht, trotzdem

ist das ein guter Grund der zweiten Interpretation den Vorzug zu geben. Da die Metaregel „so weit nach rechts wie möglich“ in der Regel die sinnvolle Variante auswählt, wird auch das Ende von Alternativen und Funktionsausdrücken nicht explizit markiert. Wir sagen bewusst „in der Regel“, denn auch die erste Interpretation kann durchaus Sinn ergeben.[11]

Verschiedene Programmiersprachen gehen mit dem Problem der Mehrdeutigkeit verschieden um. Einige setzen auf explizite „Terminierungssymbole“ — mittels gespiegelter Schlüsselwörter, wie in ***if*** e_1 ***then*** e_2 ***else*** e_3 ***fi*** und ***while*** e_1 ***do*** e_2 ***od*** oder mit Hilfe von Klammern, wie in ***if*** $(e_1)\,\{e_2\}$ ***else*** $\{e_3\}$. Andere Programmiersprachen setzen auf die Metaregel „so weit nach rechts wie möglich“. Zunehmend wird auch die Formatierung des Programmtextes, das *Layout*, benutzt, um Mehrdeutigkeiten aufzulösen. Nicht selten werden auch mehrere Ansätze gleichzeitig verfolgt: In F# kann man zum Beispiel zwischen Layout-sensitiver Syntax und expliziten Terminierungssymbolen wählen.

Abbildung 6.9 fasst die syntaktischen Konstrukte von Mini–F#, eingeschränkt auf die in Kapitel 3 eingeführten Konstrukte, zusammen. Die Beschreibung verwendet mehrere abkürzende Schreibweisen: c^+ steht für eine mindestens einmalige Wiederholung von c, c_s^+ (bzw. c_s^*) für eine mindestens einmalige (bzw. beliebige) Wiederholung, bei der die c Elemente durch s Elemente getrennt werden. In Formeln: c^+ kürzt ***rec*** $x \to c \mid c\ x$, c_s^+ kürzt ***rec*** $x \to c \mid c\ s\ x$ und c_s^* kürzt $\epsilon \mid c_s^+$ ab.

Neben den syntaktischen Kategorien für Ausdrücke und Deklarationen gibt es noch weitere für Muster und Typausdrücke. Ausdrücke teilen sich noch einmal auf in atomare Ausdrücke, *aexpr*, und Ausdrücke, *expr*. Atomare Ausdrücke können als aktuelle Parameter verwendet werden, *ohne* in Klammern gesetzt werden zu müssen. Ein atomarer Ausdruck ist entweder tatsächlich unteilbar (zum Beispiel ein Numeral) oder ein Ausdruck, der bereits geklammert ist. Die Aufteilung in atomare Konstrukte und nichtatomare Konstrukte findet sich auch bei Mustern und Typausdrücken wieder.

Die Syntax ist — wie wir bereits besprochen haben — hochgradig mehrdeutig. Die Mehrdeutigkeiten werden zum Ersten durch die Vereinbarung aufgelöst, dass sich eine lokale Deklaration, die Alternative und die Funktionsabstraktion so weit nach rechts wie möglich erstreckt. Zum Zweiten wird die Bindungsstärke und Assoziierung von Operatoren wie in Abbildung 6.10 angegeben festgelegt (die Tabelle umfasst fast alle Operatoren, nicht nur die von Mini-Mini–F#). Ist ein Operator linksassoziierend, dann muss der linke Operand die gleiche oder eine höhere Bindungsstärke besitzen und der rechte eine echt höhere Bindungsstärke. Für rechtsassoziierende Operatoren gilt Entsprechendes. Ein nicht assoziativer Operator darf nur Operanden mit höherer Bindungsstärke besitzen. Atomare Ausdrücke haben die höchste Bindungsstärke.

[11] In Abschnitt 7.2 führen wir einen Operator ein, mit dem zwei effektvolle Ausdrücke verknüpft werden können: $e_1; e_2$. Im Fall dieses Operators versagt die Metaregel: Beide Interpretationen des Ausdrucks ***if*** e_1 ***then*** e_2 ***else*** $e_3; e_4$ sind sinnvoll, so dass empfohlen wird, stets explizit zu klammern: ***if*** e_1 ***then*** e_2 ***else*** $(e_3; e_4)$ oder $(\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3); e_4$.

// Ausdrücke			
rec *expr*	→	*aexpr*	atomarer Ausdruck
	|	`if` *expr* `then` *expr* `else` *expr*	Alternative
	|	`let` $decl^*$ `in` *expr*	lokale Deklaration
	|	*expr* `||` *expr*	Disjunktion
	|	*expr* `&&` *expr*	Konjunktion
	|	*expr* `+` *expr*	Addition
	|	*expr* `-` *expr*	Subtraktion
	|	*expr* `*` *expr*	Multiplikation
	|	*expr* `/` *expr*	Division
	|	*expr* `%` *expr*	Divisionsrest
	|	*expr* `<` *expr*	kleiner
	|	*expr* `<=` *expr*	kleiner gleich
	|	*expr* `=` *expr*	gleich
	|	*expr* `<>` *expr*	ungleich
	|	*expr* `>=` *expr*	größer gleich
	|	*expr* `>` *expr*	größer
	|	*expr* `^` *expr*	Konkatenation von Strings
	|	`fun` $apat^+$ `->` *expr*	Funktionsabstraktion
	|	*aexpr* $aexpr^+$	Funktionsapplikation
	|	*expr* `:` *type*	Typangabe
and *aexpr*	→	*num*	Numeral
	|	*string*	Stringliteral
	|	*id*	Bezeichner
	|	`(` *expr* `)`	Gruppierung
// Deklarationen			
and *decl*	→	*pat* `=` *expr*	Wertedefinition
	|	*funcdecl*	Funktionsdefinition
	|	`rec` $fundecl^+_{\texttt{and}}$	rekursive Funktionsdefinitionen
and *fundecl*	→	*id* $apat^+$ `:` *type* `=` *expr*	Funktionsdefinition
// Muster			
and *pat*	→	*apat*	atomares Muster
	|	*pat* `:` *type*	Typangabe
and *apat*	→	*id*	Bezeichner
	|	`(` $pat^*_{,}$ `)`	Tupelmuster oder Gruppierung
// Typausdrücke			
and *type*	→	*atype*	atomarer Typ
	|	*type* `*` *type*	Tupeltyp
	|	*type* `->` *type*	Funktionstyp
and *atype*	→	*Id*	Typbezeichner
	|	`(` *type* `)`	Gruppierung

Abbildung 6.9.: Kontextfreie Syntax von Mini-Mini–F#.

Operator	Assoziierung
Ausdrücke	
`;`	rechtsassoziierend
`:=`	rechtsassoziierend
`,`	nicht assoziierend
`||`	linksassoziierend
`&&`	linksassoziierend
`< <= = <> >= >`	linksassoziierend
`^`	rechtsassoziierend
`:: @`	rechtsassoziierend
`+ -`	linksassoziierend (infix)
`* / %`	linksassoziierend
`**`	rechtsassoziierend
Funktionsapplikation	linksassoziierend
`+ - !`	linksassoziierend (prefix)
`.`	linksassoziierend
Muster	
`|`	rechtsassoziierend
`&`	rechtsassoziierend
Typen	
`->`	rechtsassoziierend
`*`	nicht assoziierend

Abbildung 6.10.: Assoziierung der Mini–F# Operatoren (die Operatoren sind aufsteigend nach ihrer Bindungsstärke angeordnet).

6.4. Parser⧹Akzeptoren⋆

Wie für reguläre Ausdrücke kann man auch für einen kontextfreien Ausdruck einen Akzeptor programmieren, der überprüft, ob ein gegebenes Wort in der Sprache enthalten ist. Konstruiert das Programm aus der Folge von Lexemen im Erfolgsfall zusätzlich einen abstrakten Syntaxbaum, so spricht man von einem *Parser*. Ein Parser ist für kontextfreie Ausdrücke das, was ein Scanner für reguläre Ausdrücke ist. Wenden wir uns zunächst der Programmierung von Akzeptoren zu.

6.4.1. Akzeptoren⋆

module
Grammar.
AcceptorDyck

Im Fall regulärer Ausdrücke haben wir einen Akzeptor generiert, indem wir systematisch alle Rechtsfaktoren konstruiert haben. Das Verfahren lässt sich leider nicht ohne weiteres auf kontextfreie Ausdrücke übertragen, da diese im Allgemeinen unendlich viele Rechtsfaktoren besitzen. Erinnern wir uns an die „Klammersprache“, die durch den kontextfreien Ausdruck $E = \mathit{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ gegeben ist. Wir erhalten als Rechtsfaktoren (der rekursive Ausdruck wird einfach aufgefaltet):

$$E \,/\, \mathtt{a} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b}) \,/\, \mathtt{a} = E\ \mathtt{b}$$
$$E \,/\, \mathtt{b} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b}) \,/\, \mathtt{b} = \emptyset$$

Wenn wir bereits ein a gesehen haben, erwarten wir als Rest einen korrekten Klammerausdruck gefolgt von einem b.

$$E\ \mathtt{b} \,/\, \mathtt{a} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b})\ \mathtt{b} \,/\, \mathtt{a} = E\ \mathtt{b}^2$$
$$E\ \mathtt{b} \,/\, \mathtt{b} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b})\ \mathtt{b} \,/\, \mathtt{b} = \epsilon$$

Sehen wir ein weiteres a, dann müssen nach dem Klammerausdruck zwei bs kommen. Allgemein gilt:

$$\begin{array}{lcll}
\mathit{nullable}(E\ \mathtt{b}^k) & = & k = 0 & \\
E\ \mathtt{b}^k \,/\, \mathtt{a} & = & E\ \mathtt{b}^{k+1} & \\
E\ \mathtt{b}^k \,/\, \mathtt{b} & = & \begin{cases} \emptyset & \text{falls } k = 0 \\ \mathtt{b}^{k-1} & \text{sonst} \end{cases} & \\
\mathit{nullable}(\mathtt{b}^k) & = & k = 0 & \\
\mathtt{b}^k \,/\, \mathtt{a} & = & \emptyset & \\
\mathtt{b}^k \,/\, \mathtt{b} & = & \begin{cases} \emptyset & \text{falls } k = 0 \\ \mathtt{b}^{k-1} & \text{sonst} \end{cases} &
\end{array}$$

Im Prinzip müssen wir die as zählen und überprüfen, ob genauso viele bs folgen. Damit können wir einen Akzeptor von Hand stricken.

```
type Alphabet = | A | B
type Acceptor = List ⟨Alphabet⟩ → Bool
let accept-expr : Acceptor =
  let rec accept-bs (k : Nat) : Acceptor = function
    | []          → k = 0
    | A :: rest → false
    | B :: rest → k > 0 && accept-bs (k − 1) rest
  let rec accept-expr-bs (k : Nat) : Acceptor = function
    | []          → k = 0
    | A :: rest → accept-expr-bs (k + 1) rest
    | B :: rest → k > 0 && accept-bs (k − 1) rest
  in accept-expr-bs 0
```

Die lokale Funktion *accept-bs* k implementiert einen Akzeptor für $\mathtt{b}^k$, die Funktion *accept-expr-bs* k entsprechend einen Akzeptor für $E\ \mathtt{b}^k$. Beide Definitionen basieren auf der (unbewiesenen) Tatsache, dass sich jeder kontextfreie Ausdruck c über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$ umschreiben lässt zu $\Delta(c) \mid \mathtt{a}\ (c\ /\ \mathtt{a}) \mid \mathtt{b}\ (c\ /\ \mathtt{b})$. Ein kontextfreier Ausdruck der Form $a_1\ c_1 \mid \cdots \mid a_n\ c_n$ bzw. $\epsilon \mid a_1\ c_1 \mid \cdots \mid a_n\ c_n$ lässt sich unmittelbar in den Mini–F#-Ausdruck

```
fun input → match input with
  | []          → false  bzw. true
  | A_1 :: rest → accept_1 rest
  | ...
  | A_n :: rest → accept_n rest
```

überführen, wobei A_i der zu a_i korrespondierende Konstruktor des Variantentyps *Alphabet* ist und $accept_i$ der Akzeptor für c_i. Diesen Sachverhalt haben wir auch schon bei der Generierung von Scannern in Abschnitt 6.2 ausgenutzt. Dort wie hier werden rekursiv definierte Sprachen — Wiederholung ist eine sehr spezielle Form der Rekursion — auf rekursiv definierte Akzeptoren abgebildet.

Die Herleitung des obigen Akzeptors ist stark auf die Klammersprache zugeschnitten, so dass sich die Frage stellt, ob wir die Entwicklung auch systematischer betreiben können. Betrachten wir noch einmal die Rechtsfaktoren von $E = \mathbf{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$.

$$E\ /\ \mathtt{a} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b})\ /\ \mathtt{a} = E\ \mathtt{b}$$
$$E\ /\ \mathtt{b} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b})\ /\ \mathtt{b} = \emptyset$$

Der Ausdruck E tritt im Rechtsfaktor wieder auf, allerdings gefolgt von dem Symbol $\mathtt{b}$. Wir können E selbst durch eine rekursiv definierte Funktion implementieren, wenn wir die Funktion mit dem Akzeptor für den Folgeausdruck parametrisieren. Mit anderen Worten, ein kontextfreier Ausdruck c wird durch eine Funktion implementiert, die einen Akzeptor für c' auf einen Akzeptor für die Sequenz $c\ c'$ abbildet, für beliebige kontextfreie Ausdrücke c'. Für unser Beispiel ergibt sich:

```
type Follow = List ⟨Alphabet⟩ → Bool
let accept-b (follow : Follow) : Follow = function
    | B :: rest → follow rest
    | _         → false
let rec accept-expr (follow : Follow) : Follow = function
    | A :: rest → accept-expr (accept-b follow) rest
    | _         → follow input
```

Der formale Parameter *follow* ist jeweils der Folgeakzeptor, der die verbleibende Eingabe verarbeitet. Für jedes gelesene `a` wird der Folgeakzeptor von *accept-expr* um *accept-b* erweitert; nach zwei `a`s zum Beispiel ist der Folgeakzeptor gleich *accept-b* (*accept-b follow*). Nach dem ersten `b` wird der akkumulierte Zopf von *accept-b*s abgearbeitet. Man sieht, wir können zählen, ohne die natürlichen Zahlen bemühen zu müssen. Den gewünschten Akzeptor für E erhalten wir schließlich, indem wir *accept-expr* mit dem Akzeptor für ϵ aufrufen: *accept-expr end-of-input* wobei *end-of-input* wie folgt definiert ist.

```
let end-of-input = function
    | []     → true
    | _ :: _ → false
```

Der Schritt von Funktionen des Typs *Acceptor* zu Funktionen des Typs *Follow* → *Follow* ist ein weiteres Beispiel für die Programmiertechnik der *Verallgemeinerung*: Wir verallgemeinern Akzeptoren für „isolierte“ kontextfreie Ausdrücke zu „Akzeptoren“ für in einen Kontext eingebettete kontextfreie Ausdrücke[12]. Mit Hilfe dieses Ansatzes können wir jetzt systematisch zu jedem kontextfreien Ausdruck einen korrespondierenden Mini–F# Ausdruck konstruieren. Gehen wir die Konstrukte der Reihe nach durch:

Das Terminalsymbol a wird durch den Mini–F# Ausdruck

```
fun follow → function
    | A :: rest → follow rest
    | _         → false
```

implementiert, wobei A das zu a korrespondierende Element des Alphabets ist.

Der Bezeichner x wird auf den Mini–F# Bezeichner *accept-x* abgebildet.

Das leere Wort ϵ wird durch *fun follow* → *fun input* → *follow input* bzw. kürzer durch

```
fun follow → follow
```

implementiert, also durch die *Identitätsfunktion*: Die Arbeit wird unmittelbar an den Folgeakzeptor delegiert.

Ist $accept_i$ die Implementierung von c_i, dann wird die Sequenz $c_1\ c_2$ durch

```
fun follow → accept1 (accept2 follow)
```

[12] Das hört sich merkwürdig an, löst sich aber wie folgt auf: „Kontextfreier Ausdruck“ ist ein feststehender Fachbegriff; Kontext meint das Umfeld, in den der Ausdruck eingebettet ist.

implementiert, also durch die *Komposition von Funktionen*. Beachte, dass $accept_i$ ein *Ausdruck* ist, nicht notwendigerweise ein Bezeichner.

Die leere Sprache $\emptyset$ entspricht dem Mini–F# Ausdruck

```
fun follow → fun input → false
```

Kommen wir zur Alternative $c_1 \mid c_2$; diese lässt sich wie folgt implementieren.

```
fun follow →
  fun input → accept1 follow input || accept2 follow input
```

Hier nutzen wir aus, dass die Sequenz über die Alternative distributiert: $(c_1 \mid c_2)\ c_3 = c_1\ c_3 \mid c_2\ c_3$. Die Alternative wird dann durch die logische Disjunktion realisiert.

Und schließlich die Rekursion: Ist *accept* die Implementierung von c, dann wird der rekursive Ausdruck $\mathbf{rec}\ x \to c$ durch die rekursive Mini–F# Definition

```
let rec accept-x (follow) =
  accept follow
in accept-x
```

implementiert. (Tritt x in c auf, so kommt entsprechend *accept-x* in *accept* vor.) Verschränkte Rekursion $\mathbf{rec}\ x_1 \to c_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n \to c_n$ wird entsprechend auf verschränkte Rekursion abgebildet.

```
let rec  accept-x1 (follow) → accept1 follow
    and ...
    and accept-xn (follow) → acceptn follow
in  accept-x1
```

Voilà. Setzen wir die Bausteine entsprechend zusammen, ergibt sich zum Beispiel für die Klammersprache $\mathbf{rec}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ die folgende kompakte Mini–F# Definition.

```
let rec accept-x (follow : Follow) : Follow =
  fun input →
    follow input || accept-a (accept-x (accept-b follow)) input
in accept-x
```

module Grammar. GenericAcceptor

Zu jedem kontextfreien Ausdruck korrespondiert ein Mini–F# Ausdruck. Wir können diese Korrespondenz auch explizit machen, indem wir den einzelnen Bausteinen einen Namen geben und in einer Bibliothek zusammenfassen, siehe Abbildung 6.11. (Die Bibliothek ist übrigens etwas allgemeiner als im Text beschrieben, da von einem konkreten Alphabet abstrahiert wird.) Elementare Bausteine haben den Typ *Acceptor* (wir redefinieren den Typ *Acceptor* an dieser Stelle):

```
type Acceptor = Follow → Follow
```

Dieser Typ ist sozusagen der Implementierungstyp für kontextfreie Ausdrücke. Die Bausteine für die Sequenz und die Alternative, *seq* und *alt*, besitzen entsprechend den Typ

```
type Follow ⟨'a⟩ = List ⟨'a⟩ → Bool
type Acceptor ⟨'a⟩ = Follow ⟨'a⟩ → Follow ⟨'a⟩
let eps : Acceptor ⟨'a⟩ =
  fun follow → follow
let symbol (a : 'a) : Acceptor ⟨'a⟩ =
  fun follow → function
    | []        → false
    | b :: rest → a = b && follow rest
let seq (accept1 : Acceptor ⟨'a⟩, accept2 : Acceptor ⟨'a⟩) : Acceptor ⟨'a⟩ =
  fun follow → accept1 (accept2 follow)
let seq3 (accept1 : Acceptor ⟨'a⟩, accept2 : Acceptor ⟨'a⟩, accept3 : Acceptor ⟨'a⟩) =
  fun follow → accept1 (accept2 (accept3 follow))
let empty : Acceptor ⟨'a⟩ =
  fun follow → fun input →
    false
let alt (accept1 : Acceptor ⟨'a⟩, accept2 : Acceptor ⟨'a⟩) : Acceptor ⟨'a⟩ =
  fun follow → fun input →
    accept1 follow input || accept2 follow input
let end-of-input : Follow ⟨'a⟩ = function
    | []    → true
    | _ :: _ → false
```

Abbildung 6.11.: Akzeptor-Kombinatoren für kontextfreie Ausdrücke.

$Acceptor * Acceptor \to Acceptor$. Das einzige Konstrukt, das wir nicht wiederfinden, ist *rec* $x \to c$; rekursive kontextfreie Ausdrücke müssen wir weiterhin von Hand umsetzen.[13] Mit Hilfe dieser Bibliothek können wir den Akzeptor für die Klammersprache etwas kompakter definieren.

```
let rec accept-expr : Acceptor = fun follow →
  alt (eps, seq (seq (symbol A, accept-expr), symbol B)) follow
```

Die Struktur der Mini–F# Definition spiegelt exakt die Struktur des kontextfreien Ausdrucks wider.

module Grammar.AcceptorExpr

Probieren wir die Technik an einem weiteren Beispiel aus, den einfachen arithmetischen Ausdrücken aus Abschnitt 6.3.

[13] Es ist zwar prinzipiell möglich, für *rec* $x \to c$ einen Baustein anzugeben; dieser taugt aber nicht für verschränkt rekursive Ausdrücke.

$$
\begin{aligned}
E_0 = \ & \textbf{\textit{rec}}\ \ expr_0 \to expr_1 \mid expr_1\ \texttt{+}\ expr_0 \\
& \textbf{\textit{and}}\ expr_1 \to expr_2 \mid expr_2\ \texttt{*}\ expr_1 \\
& \textbf{\textit{and}}\ expr_2 \to num \mid \texttt{(}\ expr_0\ \texttt{)}
\end{aligned}
$$

Das diesem Ausdruck zugrundeliegende Alphabet ist *Token*, siehe Abschnitt 6.2.2.

Die Umsetzung des kontextfreien Ausdrucks E_0 geht mechanisch vonstatten; die verschränkte Rekursion wird auf eine verschränkt rekursive Funktionsdefinition abgebildet.

```
let rec accept-expr0 : Acceptor ⟨Token⟩ = fun follow →
  alt (accept-expr1, seq3 (accept-expr1, symbol Plus, accept-expr0))      follow
and    accept-expr1 : Acceptor ⟨Token⟩ = fun follow →
  alt (accept-expr2, seq3 (accept-expr2, symbol Asterisk, accept-expr1))  follow
and    accept-expr2 : Acceptor ⟨Token⟩ = fun follow →
  alt (accept_num, seq3 (symbol LParen, accept-expr0, symbol RParen)) follow
```

Es bleibt noch, den Parser für Numerale, *accept-num*, nachzureichen.

```
let accept-num : Acceptor ⟨Token⟩ = fun follow → function
  | Num _ :: rest → follow rest
  | _             → false
```

Da der Konstruktor *Num* ein Argument besitzt, können wir *accept-num* nicht mit Hilfe von *symbol* definieren.

Vertiefung Genau wie andere Programme lassen sich auch Parser optimieren. Die Implementierung der Alternative ist nicht sehr zielgerichtet: *alt* $(accept_1, accept_2)$ probiert $accept_1$ und $accept_2$ nacheinander aus. Im Fall von E_0 wird dadurch Arbeit dupliziert: *alt* $(accept\text{-}expr_1, seq3\ (accept\text{-}expr_1, symbol\ Plus, accept\text{-}expr_0)$ probiert zunächst $accept\text{-}expr_1$ aus; ist die Eingabe eine Summe, scheitert dieser Aufruf; dann wird $accept\text{-}expr_1$ erneut ausprobiert, jetzt gefolgt von *symbol Plus* und $accept\text{-}expr_0$. Für jeden Teilausdruck werden also zwei Anläufe unternommen. Diese Verdopplung können wir vermeiden, indem wir den kontextfreien Ausdruck umschreiben: Mit Hilfe des Distributivgesetzes $(c_1\ c_2 \mid c_1\ c_3) = c_1\ (c_2 \mid c_3)$ erhalten wir

$$
\begin{aligned}
& \textbf{\textit{rec}}\ \ expr_0 \to expr_1\ (\epsilon \mid \texttt{+}\ expr_0) \\
& \textbf{\textit{and}}\ expr_1 \to expr_2\ (\epsilon \mid \texttt{*}\ expr_1) \\
& \textbf{\textit{and}}\ expr_2 \to num \mid \texttt{(}\ expr_0\ \texttt{)}
\end{aligned}
$$

Die gemeinsamen Faktoren werden jeweils nach links herausgezogen — man spricht daher auch von *Linksfaktorisierung*. Die Mini–F# Definition wird entsprechend umgeformt.

```
let rec accept-expr0 : Acceptor ⟨Token⟩ = fun follow →
  seq (accept-expr1, alt (eps, seq (symbol Plus, accept-expr0)))          follow
and    accept-expr1 : Acceptor ⟨Token⟩ = fun follow →
  seq (accept-expr2, alt (eps, seq (symbol Asterisk, accept-expr1)))      follow
and    accept-expr2 : Acceptor ⟨Token⟩ = fun follow →
  alt (accept_num, seq3 (symbol LParen, accept-expr0, symbol RParen)) follow
```

Zusammenfassung Fassen wir zusammen: Jeder kontextfreie Ausdruck lässt sich systematisch in ein Mini–F# Programm überführen. Rekursion wird auf Rekursion abgebildet, die anderen Konstrukte auf entsprechende Mini–F# Funktionen.

Ist damit der Fall abgeschlossen? Leider nein, die Umsetzung rekursiver Sprachen ist nicht perfekt: Der kontextfreie Ausdruck ***rec*** $x \to x$ wird auf den Mini–F# Ausdruck

let rec *accept-x* (*follow*) $\to$ *accept-x follow* ***in*** *accept-x*

abgebildet, das einfachste *nichtterminierende* Programm. Die Bedeutung von ***rec*** $x \to x$ ist aber die leere Sprache. Deren korrekte Implementierung lautet:

fun *follow* $\to$ ***fun*** *input* $\to$ *false*

ein einfaches, stets *terminierendes* Programm. Das Problem ist schnell ausgemacht: Im ersten Programm erfolgt der rekursive Aufruf, ohne dass das unsichtbare Argument, die Liste von Tokens, verkleinert wurde. Das Problem der Nichtterminierung tritt immer dann auf, wenn der kontextfreie Ausdruck *linksrekursiv* ist, wie zum Beispiel bei der folgenden Erweiterung der Klammersprache.

rec $x \to \epsilon \mid x$ `a` x `b`

Der zugehörige Akzeptor terminiert nicht, sobald die Eingabe mehr als ein Klammergebirge umfasst, zum Beispiel `abab`. Die äquivalente, rechtsrekursive Formulierung bereitet hingegen keine Probleme.

rec $x \to \epsilon \mid$ `a` x `b` x

Beim rekursiven Aufruf des Akzeptors ist sichergestellt, dass die Eingabe verkleinert wurde: Der zu `a` korrespondierende Akzeptor hat vorher einen Buchstaben konsumiert. Linksrekursion kann auch versteckt auftreten, etwa wenn der Ausdruck links vom Bezeichner „nullable“ ist.

rec $x \to$ `a` $\mid y\ x$ `b`
and $y \to \epsilon \mid$ `a`

In diesem Beispiel ist y nullable.

Fazit: Bei handgeschriebenen Akzeptoren muss man Sorge tragen, dass die rekursiven Aufrufe auf kleineren Eingaben arbeiten. *Linksrekursion ist um jeden Preis zu vermeiden.* Im Gegensatz dazu stellt bei Scannern das Konstruktionsverfahren sicher, dass die Eingabe stets kleiner wird.

6.4.2. Semantik, da capo⋆⋆

Waiting is a very funny activity:
you can't wait twice as fast.
— Edsger W. Dijkstra (1930–2002), *February 28, 1984*

Es lohnt sich die Diskrepanz zwischen rekursiv definierten Sprachen und rekursiv definierten Akzeptoren etwas genauer unter die Lupe zu nehmen. Wenn wir die Auswertungsregel für ***let rec fun*** $f\ (x) \to e$ ***in*** f mit der Reduktionsregel für ***rec*** $x \to x$ vergleichen, stellen wir fest, dass die Semantik der Rekursion sehr ähnlich ist: In beiden Fällen wird das gesamte „Objekt“ für den jeweiligen Bezeichner „eingesetzt“ — der Ausdruck wird einmal „aufgeklappt“. Trotzdem besteht offenbar ein Unterschied. Der Unterschied ist zunächst einmal ein formaler:

Die Bedeutung des Mini–F# Programms e ist ν, wenn $\varnothing \vdash e \Downarrow \nu$ mit Hilfe der Auswertungsregeln ableitbar ist. Wenn wir zurückblicken und die Auswertungsregeln Revue passieren lassen, stellen wir fest, dass auf jede „Situation“ genau eine Auswertungsregel passt. Für die Alternative zum Beispiel gibt es zwar zwei Regeln, aber beide Regeln verlangen, dass die Bedingung ausgewertet wird. Das Ergebnis dieser Auswertung bestimmt dann, welche Regel tatsächlich zur Anwendung kommt. Ähnliches gilt für die Disjunktion, die wir als Abkürzung für ***if*** e_1 ***then*** *true* ***else*** e_2 eingeführt haben. Aus den Regeln für die Alternative können wir die folgenden Regeln für die Disjunktion ableiten.

$$\frac{\delta \vdash e_1 \Downarrow \mathit{true}}{\delta \vdash (e_1 \mid\mid e_2) \Downarrow \mathit{true}} \qquad \frac{\delta \vdash e_1 \Downarrow \mathit{false} \qquad \delta \vdash e_2 \Downarrow \nu}{\delta \vdash (e_1 \mid\mid e_2) \Downarrow \nu}$$

Man sieht, e_1 wird auf jeden Fall ausgewertet, danach ist klar, welche Regel zum Zug kommt. Im Fachjargon sagt man, das Beweissystem ist *deterministisch*, in jedem Schritt gibt es nur eine anwendbare Regel. Der Mangel an Wahlmöglichkeiten ist erwünscht, da wir ja den Rechner rechnen lassen wollen. So ist dem Rechenknecht in jedem Schritt klar, was zu tun ist. Die Regeln sind sehr sorgfältig in Hinblick auf diese Eigenschaft gewählt. Nur zum Vergleich: Anstelle des asymmetrischen Regelpaars für die Disjunktion (erst e_1, dann e_2) hätten wir auch ein symmetrisches Regeltrio formulieren können:

$$\frac{\delta \vdash e_1 \Downarrow \mathit{true}}{\delta \vdash (e_1 \mid\mid e_2) \Downarrow \mathit{true}} \qquad \frac{\delta \vdash e_1 \Downarrow \mathit{false} \qquad \delta \vdash e_2 \Downarrow \mathit{false}}{\delta \vdash (e_1 \mid\mid e_2) \Downarrow \mathit{false}} \qquad \frac{\delta \vdash e_2 \Downarrow \mathit{true}}{\delta \vdash (e_1 \mid\mid e_2) \Downarrow \mathit{true}}$$

Jetzt ist die Auswertung nicht mehr festgelegt: Wir können e_1 oder e_2 auswerten; ist ein Ergebnis *true*, dann wird das Ergebnis des anderen Ausdrucks nicht mehr benötigt. Mit den neuen Regeln kann die Abarbeitung der Disjunktion nicht länger *sequentiell* erfolgen, sondern muss *parallel* durchgeführt werden.

Kommen wir zur Semantik von kontextfreien Ausdrücken. Die Bedeutung von c ist *die Menge aller Wörter* w, so dass $c \longrightarrow \cdots \longrightarrow w$ ableitbar ist. Das Beweissystem der Reduktionssemantik ist *nichtdeterministisch*. Im Fall der Alternative haben wir die freie Wahl.

$$\frac{}{c_1 \mid c_2 \longrightarrow c_1} \qquad \frac{}{c_1 \mid c_2 \longrightarrow c_2}$$

Wäre die Bedeutung eines kontextfreien Ausdrucks ein *einzelnes Wort*, würde sich an dieser Stelle ein mulmiges Gefühl einstellen: Die Bedeutung wäre nicht eindeutig oder im Fachjargon *nicht determiniert*. Determiniertheit ist etwas anderes als Determinismus. Das obige Regeltrio für die Disjunktion ist nichtdeterministisch; das Ergebnis der Auswertung ist aber determiniert: Unterschiedliche Rechenwege führen stets zum gleichen Ergebnis.

Vorläufiges Fazit: Die Semantik rekursiver Mini–F# Ausdrücke und die rekursiver kontextfreier Ausdrücke ist ähnlich aber nicht deckungsgleich. Der Unterschied tritt noch deutlicher zutage, wenn wir statt der „lokalen Semantik“ die „globale Semantik“ betrachten.

Wir erinnern uns: Die Semantik von ***rec*** $x \to c$ ist der kleinste Fixpunkt der Funktion $F(X) = [\![c]\!](\varrho, \{x \mapsto X\})$ Dieser Fixpunkt kann beliebig approximiert werden, indem man F wiederholt auf die leere Sprache anwendet.

$\emptyset$
$F(\emptyset)$
$F(F(\emptyset)$
...

Mit jedem Schritt wird das Wissen über die bezeichnete Sprache größer: $F^2\ (\emptyset)$ ist informativer als $F^1\ (\emptyset)$ und $F^1\ (\emptyset)$ ist informativer als $F^0\ (\emptyset)$.

Eine entsprechende denotationelle Semantik können wir auch für Mini–F# aufstellen. Wir deuten sie an dieser Stelle nur an; eine vollständige Ausführung würde den Rahmen der Vorlesung sprengen.

Die uninformativste Sprache ist die leere Sprache. Wenn wir die Idee der Approximation auf Programme übertragen wollen, müssen wir die Frage „Was ist das uninformativste Programm?“ beantworten. Ein Ratespiel hilft uns dabei, die Frage zu durchdringen: Harry Hacker hat eine Funktion *unknown* : *Bool* → *Bool* programmiert; wir können die Definition nicht einsehen, dürfen sie aber im Mini–F# Interpreter mit verschiedenen Werten aufrufen. Zum Beispiel:

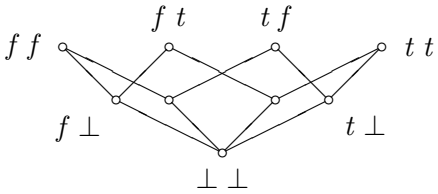
Mini⟩ *unknown false*
true
Mini⟩ *unknown true*
false

Das war sehr informativ, offenbar hat Harry die Negation programmiert. Zweite Runde mit einer anderen Definition von *unknown*:

Mini⟩ *unknown false*
true
Mini⟩ *unknown true*
...

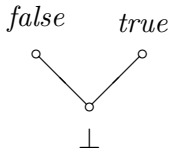
Der zweite Aufruf liefert kein Ergebnis, jedenfalls nicht in der Zeitspanne, die wir bereit waren zu warten. Zwei Möglichkeiten sind denkbar: Der Aufruf terminiert nicht oder der Aufruf terminiert, aber erst nach einer Weile. Wir sehen: Nichtterminierende Programme sind nicht sehr informativ, insbesondere, da wir uns der Nichtterminierung nicht sicher sein können; vielleicht ist das Programm ja nur extrem langsam. Der Informationsgehalt nimmt zu, je häufiger eine Funktion terminiert; die Funktion, die nie terminiert, ist das gesuchte Gegenstück zur leeren Sprache. Die Funktionen des Typs *Bool* → *Bool* lassen sich entsprechend ihres Informationsgehaltes in einer Hierarchie anordnen. Dazu ist es

nützlich der Nichtterminierung einen Namen zu geben (ähnlich dem Symbol 0 für die Zahl Null oder dem Symbol $\emptyset$ für die leere Menge). Es hat sich eingebürgert, die Nichtterminierung mit dem Symbol $\bot$ zu bezeichnen (engl. bottom). Mit $\bot$ im Gepäck lassen sich neun verschiedenen Funktionen unterscheiden (wir führen lediglich die Funktionswerte von *false* und *true* auf).



In der obersten Reihe sind die informativsten Funktionen aufgeführt: Die konstante Funktion ***fun*** $b \to$ *false*, die Identität, die Negation und die konstante Funktion ***fun*** $b \to$ *true*. Darunter finden sich die Funktionen wieder, die für einen Wahrheitswert terminieren und für den anderen nicht. Das kleinste Element ist schließlich die Funktion, die nie terminiert.

Die Ordnung lässt sich systematisch konstruieren. Funktionen sind zunächst einmal *punktweise* geordnet: $f \preccurlyeq g$ gdw. $f(x) \preccurlyeq g(x)$ für alle x. Mini–F# Funktionen des Typs $Bool \to Bool$ entsprechen mathematischen Funktionen des Typs $\mathbb{B} \to \mathbb{B}_\bot$, wobei $\mathbb{B} = \{false, true\}$ der Bereich der Booleschen Werte ist — $\mathbb{B}$ verhält sich zu $Bool$ wie $\mathbb{N}$ zu Nat — und $\mathbb{B}_\bot$ der um das Element $\bot$ erweiterte Bereich. Der Bereich $\mathbb{B}_\bot$ selbst ist wie folgt geordnet.



Der Bereich der Funktionen $\mathbb{B} \to \mathbb{B}_\bot$ enthält insgesamt $3^2 = 9$ Elemente, die punktweise wie oben angeordnet sind.

Mit diesem Hintergrundwissen können wir die Idee der schrittweisen Approximation auf rekursiv definierte Funktionen übertragen (wir missbrauchen im Folgenden Mini–F# Syntax, um mathematische Funktionen zu notieren). Hier ist Harry Hackers zweites Programm.

```
let rec unknown (b : Bool) : Bool =
  if b then unknown (unknown true) else true
```

Zunächst ordnen wir dem Programm eine *nicht-rekursive* Funktion zu, die mit *unknown* parametrisiert ist. (Vergleiche mit der denotationellen Semantik von Sprachen.)

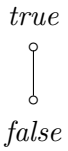
```
let F(unknown : Bool → Bool) : Bool → Bool =
  fun (b : Bool) → if b then unknown (unknown true) else true
```

Der Fixpunkt von F ist die gesuchte Bedeutung von *unknown*; der Fixpunkt kann ausgehend von der nie terminierenden Funktion $\perp\!\!\!\perp$ = ***fun*** $b \to \bot$ beliebig approximiert werden:

$$
\begin{array}{ll}
& \bot\!\!\bot \\
F(\bot\!\!\bot) & = \textbf{\textit{fun}}\ b \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ \bot\!\!\bot\ (\bot\!\!\bot\ \mathit{true})\ \textbf{\textit{else}}\ \mathit{true} \\
& = \textbf{\textit{fun}}\ b \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ \bot\ \textbf{\textit{else}}\ \mathit{true} \\
F(F\bot\!\!\bot) & = \textbf{\textit{fun}}\ b \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ F(\bot\!\!\bot)\ (F(\bot\!\!\bot)\ \mathit{true})\ \textbf{\textit{else}}\ \mathit{true} \\
& = \textbf{\textit{fun}}\ b \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ F(\bot\!\!\bot)\ (\bot)\ \textbf{\textit{else}}\ \mathit{true} \\
& = \textbf{\textit{fun}}\ b \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ \bot\ \textbf{\textit{else}}\ \mathit{true}
\end{array}
$$

Nach der zweiten Iteration haben wir einen Fixpunkt gefunden: Die Bedeutung der Funktion *unknown* ist $\textbf{\textit{fun}}\ b \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ \bot\ \textbf{\textit{else}}\ \mathit{true}$. Bei der Vereinfachung haben wir übrigens ausgenutzt, dass $f(\bot) = \bot$ gilt: Wenn die Auswertung des Funktionsarguments nicht terminiert, dann terminiert auch der Funktionsaufruf nicht. Dies gilt für alle Programmiersprachen, die Parameter „call by value“ übergeben.

Jetzt sind wir der Diskrepanz zwischen der Semantik von Sprachen und der Semantik von Mini–F# auf den Fersen. Die denotationelle Semantik ordnet Sprachen gemäß der Mengeninklusion, die Menge aller Sprachen ist der Potenzmengenverband $\mathbb{P}(A^*)$. Die einfachsten Akzeptoren, die wir geschrieben haben, besitzen den Typ $\mathit{List}\ \langle \mathit{Alphabet} \rangle \to \mathit{Bool}$ und können als *charakteristische Funktion* $A^* \to \mathbb{B}$ aufgefasst werden. Wenn wir die Inklusionsordnung auf charakteristische Funktionen übertragen, erhalten wir: $f \preccurlyeq g$ gdw. $f(x) \Longrightarrow g(x)$ für alle $x \in \mathbb{B}$. Die Implikation '$\Longrightarrow$' ist also die zugrundeliegende Ordnung auf $\mathbb{B}$: *false* ist kleiner als *true*. Mit anderen Worten, eine getreue Umsetzung der Semantik kontextfreier Ausdrücke verlangt, dass $\mathbb{B}$ wie folgt geordnet ist.



Wir haben aber gesehen, dass der Wertebereich von Akzeptoren in Wirklichkeit $\mathbb{B}_\bot$ ist und diesem Bereich eine andere Ordnung zugrundeliegt. Somit hat die Sprache $\textbf{\textit{rec}}\ x \to x$ die Bedeutung $\emptyset$, der Mini–F# Akzeptor

$$\textbf{\textit{let rec fun}}\ \mathit{accept\text{-}x}\ (\mathit{follow}) \to \mathit{accept\text{-}x}\ \mathit{follow}\ \textbf{\textit{in}}\ \mathit{accept\text{-}x}$$

aber die Bedeutung $\bot\!\!\bot$; eine bedauerliche, aber leider unvermeidbare Diskrepanz.

6.4.3. Parser⋆

Die Akzeptoren aus Abschnitt 6.4.1 beantworten die Frage „Ist das gegebene Wort in der von dem kontextfreien Ausdruck bezeichneten Sprache enthalten?“. Ein Parser beantwortet die gleiche Frage, gibt aber im positiven Fall zusätzlich einen *semantischen Wert* zurück. Im Fall arithmetischer Ausdrücke kann der semantische Wert tatsächlich der Wert des Ausdrucks sein, es kann aber auch der abstrakte Syntaxbaum des Ausdrucks sein. Der letztere Ansatz ist allgemeiner — aus dem abstrakten Syntaxbaum können wir den Wert berechnen, aber nicht umgekehrt.

module Grammar. Expr

Die abstrakte Syntax arithmetischer Ausdrücke fangen wir mit einer rekursiven Variantentypdefinition ein.

```
type Expr =
  | Const of Nat
  | Add   of Expr * Expr
  | Mul   of Expr * Expr
```

Wir erinnern uns: Variantentypen sind das Mini–F# Pendant zu Baumsprachen, so dass wir die Baumsprache für arithmetische Ausdrücke im Wesentlichen übernehmen können (hier aus Platzgründen nur in Teilen). Ein Auswerter für arithmetische Ausdrücke ist fix geschrieben. Das Struktur Entwurfsmuster für *Expr* bringt uns schnell und sicher ans Ziel.

```
let rec evaluate (expr : Expr) : Nat =
  match expr with
  | Const nat           → nat
  | Add (expr1, expr2) → evaluate expr1 + evaluate expr2
  | Mul (expr1, expr2) → evaluate expr1 * evaluate expr2
```

Der Auswerter, ein echter Interpreter (!), implementiert die Semantik aus Abschnitt 3.2.

Kommen wir zur Implementierung von Parsern. Ein Akzeptor liefert einen Booleschen Wert als Ergebnis; jetzt müssen wir im Erfolgsfall einen Wert zurückgeben (im Fall eines Misserfolgs wäre eine aussagekräftige Fehlermeldung schön — diesen Aspekt ignorieren wir hier geflissentlich). Aus dem Typ *Bool* wird somit der Typ *Option* ⟨*'value*⟩.

```
type Follow ⟨'value⟩ = List ⟨Token⟩ → Option ⟨'value⟩
```

Der Typparameter spezifiziert den Typ des resultierendes Wertes, für unser laufendes Beispiel ist der Typ zum Beispiel *Nat* oder *Expr*. Welche Änderungen ergeben sich für die Programme? Modifizieren wir zunächst den „handgeschriebenen“ Akzeptor aus Abbildung 6.12. Auf den ersten Blick sind es nicht allzu viele Änderungen: Nur *accept-RParen* und *end-of-input* enthalten konkrete Boolesche Werte. Die Änderung von *Bool* nach *Option* ⟨*'value*⟩ gibt vor, dass *false* zu *None* wird und *true* zu *Some expr*, wobei *expr* der konstruierte abstrakte Syntaxbaum ist (siehe auch Abschnitt 5.3.3). Damit erhalten wir für *end-of-input*:

```
let end-of-input : Follow ⟨Expr⟩ = function
  | []     → Some ??
  | _ :: _ → None
```

Aber woher nehmen wir das Argument für *Some*? Der Akzeptor *end-of-input* ist das letzte Glied in der Kette von Akzeptoren, er hat im Wesentlichen die Aufgabe abzunicken oder den Kopf zu schütteln. Es bleibt uns nichts anderes übrig, als *end-of-input* den semantischen Wert mit auf den Weg zu geben.

```
let end-of-input (e : Expr) : Follow ⟨Expr⟩ = function
  | []     → Some e
  | _ :: _ → None
```

Aus einer Funktion des Typs *Follow* ist eine Funktion des Typs $Expr \to Follow\,\langle Expr\rangle$ geworden. Die Funktion *end-of-input* ist ein Folgeakzeptor (der initiale), damit drängt sich die folgende Idee auf: *Jeder Parser übergibt seinen semantischen Wert an den Folgeakzeptor.* Aus der Funktion $accept\text{-}expr_0 : Follow \to Follow$ wird mit dieser Idee die Funktion $parse\text{-}expr_0 : (Expr \to Follow\,\langle Expr\rangle) \to Follow\,\langle Expr\rangle$. In der Definition von $parse\text{-}expr_0$ wird der abstrakte Syntaxbaum für die Summe konstruiert und an den Folgeakzeptor weitergegeben: $follow\ (Add\ (expr_1, expr_2))$. Ein Aufruf von $parse\text{-}expr_0$ hat typischerweise die Form $parse\text{-}expr_0\ (\textbf{\textit{fun}}\ expr \to \ldots)$; der semantische Wert von $parse\text{-}expr_0$ wird dann im Rumpf der Funktionsabstraktion weiterverarbeitet. Der vollständige Programmcode ist in Abbildung 6.12 aufgeführt.

Es lohnt sich, ein konkretes Beispiel durchzurechnen: die Analyse von `4711+815*2765` bzw. als Liste von Tokens:

$$[Num\ 4711; Plus; Num\ 815; Asterisk; Num\ 2765]$$

Da die Mini–F# Ausdrücke recht groß sind, führen wir nur die (rekursiven) Aufrufe auf und kürzen die Folgeakzeptoren jeweils ab (die Folgeakzeptoren können aus dem Programmtext, siehe Abbildung 6.12, rekonstruiert werden).

$$\begin{array}{lll}
& parse\text{-}expr_0\ \ end\text{-}of\text{-}input & [Num\ 4711, \ldots] \\
= & parse\text{-}expr_1\ \ follow_0 & [Num\ 4711, \ldots] \\
= & parse\text{-}expr_2\ \ follow_1 & [Num\ 4711, \ldots] \\
= & follow_1\ \ (Const\ 4711) & [Plus, \ldots] \\
= & follow_0\ \ (Const\ 4711) & [Plus, \ldots] \\
= & parse\text{-}expr_0\ \ follow_2 & [Num\ 815, \ldots] \\
= & parse\text{-}expr_1\ \ follow_3 & [Num\ 815, \ldots] \\
= & parse\text{-}expr_2\ \ follow_4 & [Num\ 815, \ldots] \\
= & follow_4\ \ (Const\ 815) & [Asterisk, \ldots] \\
= & parse\text{-}expr_1\ \ follow_5 & [Num\ 2765] \\
= & parse\text{-}expr_2\ \ follow_6 & [Num\ 2765] \\
= & follow_6\ \ (Const\ 2765) & [\,] \\
= & follow_5\ \ (Const\ 2765) & [\,] \\
= & follow_3\ \ (Mul\ (Num\ 815, Num\ 2765)) & [\,] \\
= & follow_2\ \ (Add\ (Num\ 4711, Mul\ (Num\ 815, Num\ 2765))) & [\,] \\
= & end\text{-}of\text{-}input\ (Add\ (Num\ 4711, Mul\ (Num\ 815, Num\ 2765))) & [\,] \\
= & Some\ (Add\ (Num\ 4711, Mul\ (Num\ 815, Num\ 2765))) &
\end{array}$$

Die Abfolge der Funktionsaufrufe entspricht im Wesentlichen der in Abschnitt 6.3 aufgeführten Reduktionsfolge von `4711+815*2765`. Der semantische Werte des bereits verarbeiteten Teils der Eingabe (in der Reduktionsfolge entspricht das dem Anfangsstück aus Terminalsymbolen) wird jeweils an den Folgeakzeptor weitergereicht; dieser ergänzt den semantischen Wert und reicht ihn an seinen Folgeakzeptor weiter. Insgesamt ergibt sich das Bild einer Kollekte: Um Geld für besondere Zwecke zu sammeln, wird ein Korb herumgereicht; jeder fügt ein paar Münzen hinzu und reicht den Korb an den Nachbarn (den Folgeakzeptor) weiter. Am Ende der Kette wird der Korb geleert (*end-of-input*).

```
type Token = | Num of Nat | LParen | RParen | Asterisk | Plus
type Follow 'v = List Token -> Option 'v
type Expr =
  | Const of Nat
  | Add   of Expr * Expr
  | Mul   of Expr * Expr
let parse-RParen (follow : Follow Expr) : Follow Expr = function
  | RParen :: rest -> follow rest
  | _              -> None
let rec parse-expr0 (follow : Expr -> Follow Expr) : Follow Expr =
    parse-expr1 (fun e1 -> function
      | Plus :: rest  -> parse-expr0 (fun e2 -> follow (Add (e1, e2))) rest
      | _             -> follow e1 input)
  and parse-expr1 (follow : Expr -> Follow Expr) : Follow Expr =
    parse-expr2 (fun e1 -> function
      | Asterisk :: rest -> parse-expr1 (fun e2 -> follow (Mul (e1, e2))) rest
      | _                -> follow e1 input)
  and parse-expr2 (follow : Expr -> Follow Expr) : Follow Expr = function
    | Num n :: rest  -> follow (Const n) rest
    | LParen :: rest -> parse-expr0 (fun e -> parse-RParen (follow e)) rest
    | _              -> None
let end-of-input (e : Expr) : Follow Expr = function
  | []     -> Some e
  | _ :: _ -> None
let abstract-syntax-tree : String -> Option Expr =
  explode >> lex >> parse-expr0 end-of-input
```

Abbildung 6.12.: Parser für einfache arithmetische Ausdrücke.

Beim Parsen, wie bei der Kollekte kann es passieren, dass der Korb unverändert weitergereicht wird oder dass sogar etwas entnommen wird. (Letzteres ist im Fall einer Kollekte unerwünscht.) Der Parser *parse-expr*$_0$ zum Beispiel reicht das Ergebnis von *parse-expr*$_1$ unverändert an *follow* weiter, wenn er kein *Plus* Symbol sieht.

Übungen.

1. Welche Sprachen bezeichnen die folgenden regulären Ausdrücke?

1. `0 (0 | 1)* 0`,
2. $((\epsilon \mid \texttt{0})\ \texttt{1}^*)^*$,
3. `(0 | 1)* 0 (0 | 1) (0 | 1)`,
4. `0* 1 0* 1 0* 1 0*`.

Geben Sie die Sprachen sowohl umgangssprachlich als auch formal an.

2. Geben Sie für die folgenden Sprachen reguläre Ausdrücke an.

1. Alle Buchstabenfolgen, die die fünf Vokale jeweils genau einmal in der Reihenfolge des Alphabets beinhalten.
2. Alle Buchstabenfolgen, bei denen die Buchstaben lexikographisch aufsteigend geordnet sind.
3. Kommentare, die aus Zeichenketten über dem Alphabet $A = \{\texttt{a}, \texttt{b}, \ldots, \texttt{y}, \texttt{z}, \texttt{(}, \texttt{)}, \texttt{*}, \texttt{"}\}$ bestehen, die in `(*` und `*)` eingeschlossen sind und die weder `(*` noch `*)` enthalten, es sei denn innerhalb von Anführungszeichen (`"` ... `"`).
4. Alle Folgen von `0` und `1`, bei denen nicht mehrere `1`en direkt nebeneinander stehen.
5. Alle nicht-leeren Folgen von `0` und `1`, bei denen das erste und das letzte Zeichen identisch sind.
6. Alle Ziffernfolgen, die keine Ziffer mehrfach enthalten.
7. Alle Ziffernfolgen, die höchstens eine Ziffer mehrfach enthalten.
8. Alle Folgen aus `0` und `1`, in denen `001` nicht enthalten ist.

3. Um größere Zahlen besser lesen zu können, setzt man oft einen Punkt zwischen Gruppen aus drei Ziffern: zum Beispiel 4.711 oder 1.234.567. Geben Sie einen regulären Ausdruck für derartige Numerale an.

4. Welche Sprache bezeichnet der folgende reguläre Ausdruck?

$$\mathit{riddle} = (\texttt{a a} \mid \texttt{b b})^*\ ((\texttt{a b} \mid \texttt{b a})\ (\texttt{a a} \mid \texttt{b b})^*\ (\texttt{a b} \mid \texttt{b a})\ (\texttt{a a} \mid \texttt{b b})^*)^*$$

Welche Sprachen bezeichnen *riddle* / `a` und *riddle* / `b`?

5. Wenn wir Durchschnitt und Komplement zu den regulären Ausdrücken hinzunehmen, lassen sich viele Sprache einfacher formulieren.

$r ::= \cdots$	*reguläre Ausdrücke:*
$\mid\ r_1\ \&\ r_2$	Durchschnitt
$\mid\ \widehat{\ }r$	Komplement

1. Legen Sie die denotationelle Semantik dieser Erweiterung fest.
2. Stellen Sie die Reduktionssemantik für diese Erweiterung auf (schwierig).

6. Formulieren Sie eine Auswertungsemantik für reguläre Ausdrücke, indem Sie Beweisregeln für die Auswertungsrelation $r \Downarrow w$ aufstellen.

7. Schreiben Sie Akzeptoren für die folgenden regulären Ausdrücke:

1. $(\mathtt{a}^* \mid \mathtt{b}^*)^*$,
2. $\mathtt{a}^*\ (\epsilon \mid \mathtt{b})\ (\mathtt{a}\ \mathtt{a}^*\ \mathtt{b})^*\ \mathtt{a}^*$,
3. $(\mathtt{b} \mid \mathtt{ab})^*\ \mathtt{a}^*$.

8. Geben Sie für jede der folgenden Sprachen einen kontextfreien Ausdruck an, der die jeweilige Sprache beschreibt. Das zugrundeliegende Alphabet ist $\{\mathtt{a}, \mathtt{b}\}$.

1. Die Sprache aller Wörter, bei denen auf jedes `a` direkt ein `b` folgt.
2. Die Sprache aller Wörter, die eine gleiche Anzahl von `a`'s und `b`'s enthalten.
3. Die Sprache $\{\mathtt{a}^{n+3}\ \mathtt{b}^{n+2} \mid n \in \mathbb{N}\}$.
4. Die Sprache aller Wörter, die nicht das Teilwort `abb` enthalten.

9. Leiten Sie die Worte `aaaba` und `baaba` aus dem folgenden kontextfreien Ausdruck ab:

rec $x \to \mathtt{a} \mid x\ x \mid y\ x$
and $y \to \mathtt{b} \mid x\ y$

10. Schreiben Sie einen Akzeptor für die Sprache der arithmetischen Ausdrücke in Präfixnotation.

rec *expr* $\to$ *num* | `+` *expr expr* | `*` *expr expr*

11. Ziel dieser Aufgabe ist es, zu zeigen, dass die dem kontextfreien Ausdruck ***rec*** $x \to c$ zugeordnete Funktion stets einen kleinsten Fixpunkt besitzt. Dazu benötigen wir etwas mathematisches Rüstzeug. Die Vereinigung $\bigcup \mathcal{X}$ einer Menge von Mengen ist durch die folgende Eigenschaft charakterisiert:

$$\bigcup \mathcal{X} \subseteq A \quad \Longleftrightarrow \quad \forall\, X \in \mathcal{X} \,.\, X \subseteq A$$

Eine Funktion $F : \mathbb{P}(A^*) \to \mathbb{P}(A^*)$ heißt *monoton* genau dann, wenn

$$X \subseteq Y \quad \Longrightarrow \quad F(X) \subseteq F(Y)$$

Eine Funktion $F : \mathbb{P}(A^*) \to \mathbb{P}(A^*)$ heißt *stetig* genau dann, wenn

$$F(\bigcup \mathcal{X}) = \bigcup \{ F(X) \mid X \in \mathcal{X} \}$$

1. Sei F stetig. Zeigen Sie, dass $\bigcup \{ F^n(\emptyset) \mid n \in \mathbb{N} \}$ *ein* Fixpunkt von F ist.
2. Zeigen Sie, dass eine stetige Funktion stets monoton ist.
3. Sei F stetig. Zeigen Sie, dass $\bigcup \{ F^n(\emptyset) \mid n \in \mathbb{N} \}$ der *kleinste* Fixpunkt von F ist.
4. Zeigen Sie, dass die dem kontextfreien Ausdruck ***rec*** $x \to c$ zugeordnete Funktion stetig ist (schwierig).

7. Effekte⧹Effektvolles Rechnen

I amar prestar aen.
Die Welt ist im Wandel.

— John R. R. Tolkien (1892–1973), *Herr der Ringe*

Nach dem längeren Ausflug in die Welt der formalen Sprachen wenden wir uns wieder unserer Programmiersprache zu. Frischen wir unser Gedächtnis auf. In Kapitel 3 haben wir uns mit den grundlegenden Bestandteilen von Mini–F# vertraut gemacht: mit der Verarbeitung von Wahrheitswerten und natürlichen Zahlen, mit der Möglichkeit, Ausdrücken einen Namen zu geben und sich später auf die Werte dieser Ausdrücke zu beziehen und schließlich mit der Formulierung von Rechenregeln in Form von Funktionen und rekursiven Funktionen. Am Ende des Kapitels steht eine Sprache, mit der man die prinzipiellen Möglichkeiten eines Rechners ausschöpfen kann. In Kapitel 4 haben wir das bis dato bescheidene Repertoire an Typen erweitert: Wir haben gesehen, wie man mehrere Daten zu einer Einheit zusammenfasst und alternative Angaben als Einheit behandelt. Die Möglichkeiten zur Darstellung und Verwaltung von Daten sind vielfältig: Tupel, Records, Varianten, Listen, Bäume, Arrays usw. stehen zur Auswahl. Die verschiedenen Typen unterscheiden sich hinsichtlich Flexibilität, Bequemlichkeit und Schnelligkeit der sie manipulierenden Operationen.

Wir haben viele Funktionen kennengelernt und selbst programmiert. So unterschiedlich die Programme auch sein mögen, sie eint ein charakteristisches Merkmal: Funktionen verdienen ihren Namen! Das Ergebnis einer Funktion — die Ausgabe, wenn man so will — wird allein durch den aktuellen Parameter — die Eingabe, wenn man möchte — bestimmt. Das ist sowohl ein Segen als auch ein Fluch.

Es ist ein Segen, weil man Funktionen „lokal“ lesen und verstehen kann. Um zu begreifen, was eine Funktion berechnet, muss ich nur ihre Definition studieren (und die Definitionen, von denen sie statisch abhängt). Rufe ich eine Funktion mit dem gleichen Argument auf, erhalte ich das gleiche Ergebnis.

Es ist ein Fluch, weil Funktionen in ihren Interaktionsmöglichkeiten stark eingeschränkt sind. Eine Funktion verhält sich in gewisser Weise autistisch: Ich stelle eine Frage, indem ich die Funktion aufrufe, nach einer Periode der Stille und des Wartens erhalte ich die Antwort in Form des Funktionsergebnisses. Weiteren Einfluss auf den Verlauf der Rechnung habe ich nicht; weiteres Feedback erhalte ich nicht. Das gilt im übrigen für alle Ausdrücke, nicht nur für Funktionsaufrufe: Ein Ausdruck wird zu einem Wert ausgerechnet; die Rechnung selbst kann nicht beeinflusst oder beobachtet werden.

Halten wir fest: Die Kommunikation mit einem Programm ist stark eingeschränkt und passt sicherlich nicht in das Bild, das die meisten von einem Rechner und von dem Umgang mit diesem Medium haben. Persönliche Rechner interagieren mit der Benutzerin

oder dem Benutzer über Bildschirm, Tastatur und Maus; Steuerungsrechner interagieren mit der Umwelt über Sensoren und Aktoren. Um auch diesen Einsatzgebieten gerecht zu werden, erweitern wir in diesem Kapitel die Idee des Rechnens: Ein Ausdruck kann neben dem Wert zusätzlich einen *Effekt* haben. Ein Effekt ist zum Beispiel eine Ausgabe auf dem Bildschirm, die Anforderung einer Eingabe, das Einlesen von Sensordaten oder die Steuerung eines Motors. Mit diesen *externen Effekten* beschäftigt sich Abschnitt 7.1. Wir beschränken uns dabei auf die Ein- und Ausgabe von Strings.

Neben externen gibt es auch *interne Effekte*: Eine Rechnung kann von einem Gedächtnis abhängen oder das Gedächtnis verändern; eine Rechnung kann ergebnislos abgebrochen werden und an anderer Stelle wiederaufgenommen werden. Das (Kurzzeit-) Gedächtnis eines Rechners ist der sogenannte (Haupt-) Speicher; Abschnitt 7.2 macht uns mit dem Konzept des Speichers vertraut.

Abschnitt 7.3 führt Sprachkonstrukte ein, die es auf einfache Art und Weise ermöglichen, effektvolle Ausdrücke miteinander zu kombinieren: Effekte aneinanderzureihen (Sequenz), Effekte von Bedingungen abhängig zu machen (Alternative), Effekte zu wiederholen (Iteration).

Eine Rechnung führt nicht immer zu einem Ergebnis: Treffen wir im Laufe einer Rechnung zum Beispiel auf den Teilausdruck $1 \div 0$, müssen wir die Segel streichen. Abschnitt 7.4 zeigt, wie wir mit diesen und ähnlichen Ausnahmesituationen umgehen können, wie wir Rechnungen abbrechen und an anderer Stelle wiederaufnehmen.

Mit dem Einzug von Effekten verändert sich die Natur des Rechnens. Die Reihenfolge, in der Teilrechnungen bearbeitet werden, spielt plötzlich eine zentrale Rolle: Wenn e_1 und e_2 Effekte haben, ist es nicht mehr egal, wie zum Beispiel der Ausdruck $e_1 + e_2$ ausgerechnet wird. Ist eine Funktion effektvoll, weil sie Ein- oder Ausgaben tätigt oder weil sie vom Gedächtnis abhängt oder das Gedächtnis verändert, dann handelt es sich nicht mehr um eine Funktion im mathematischen Sinne. Eine effektvolle Funktion kann bei gleichen Argumenten unterschiedliche Resultate liefern! Vielleicht ahnt man die Gefahr. Setzt man die neuen Sprachkonstrukte nicht mit Bedacht ein, so wird aus dem Segen schnell ein Fluch: Klarheit und Lesbarkeit von Programmen leiden. Wenn eine Funktion auf vielfältigen Wegen mit ihrer Umwelt interagiert, dann kann die Funktion nicht mehr isoliert verstanden werden, sondern die vielfältigen Verflechtungen müssen zusätzlich berücksichtigt werden. Diese warnenden Worte sollte man beim Studium der folgenden Seiten stets im Gedächtnis behalten.

7.1. Ein- und Ausgabe

In Abschnitt 3.6 haben wir ein 2-Personenspiel programmiert, bei dem Spielerin B eine von Spieler A ausgedachte Zahl raten musste. Beide Parteien wurden bisher vom Rechner gestellt. Das wollen wir jetzt ändern: Die Benutzerin bzw. der Benutzer soll die Rolle von Spieler A übernehmen. Natürlich wird dieser weiterhin durch eine Mini–F# Funktion realisiert, etwa

let human-player (*guess* : *Nat*) : *Bool*

aber es soll eine Funktion sein, die über Ein- und Ausgaben mit der Benutzer*in interagiert und deren Antworten weiterleitet. Dazu benötigen wir grundlegende Funktionen zur Ein- und Ausgabe.

Ausgaben auf dem Bildschirm werden mit Hilfe der Funktion *putstring* getätigt. Der Aufruf

$$\mathit{putstring}\ \texttt{"Hello, world!"}$$

wertet zu dem leeren Tupel '()' aus und hat zusätzlich den Effekt, dass der String `"Hello, world!"` ausgegeben wird. Die Funktion *putstring* ist das erste Beispiel für eine nicht-mathematische Funktion. Der Funktionswert steht schon vor dem Aufruf fest; ignorieren wir die Effekte, dann entspricht *putstring* der Funktion $\mathbf{\mathit{fun}}\ (s : \mathit{String}) \to ()$. Die Funktion *putstring* wird allein wegen ihres Effektes aufgerufen.

Man gewöhnt sich schnell an die Tatsache, dass ein Ausdruck neben einem Wert zusätzlich einen Effekt haben kann. Genauso schnell sollte man sich klarmachen, dass sich das Rechnen mit dem Einzug von Effekten verändert: Die Reihenfolge und die Multiplizität von Rechnungen spielen plötzlich eine Rolle. Bisher galt, dass zum Beispiel die Komponenten des Paarausdrucks

$$(\mathit{factorial}\ 9, \mathit{factorial}\ 10)$$

in beliebiger Reihenfolge ausgerechnet werden konnten. Die Auswertungsregel für Paarausdrücke

$$\frac{\delta \vdash e_1 \Downarrow \nu_1 \qquad \delta \vdash e_2 \Downarrow \nu_2}{\delta \vdash (e_1, e_2) \Downarrow (\nu_1, \nu_2)}$$

mag eine Auswertung von links nach rechts suggerieren, aber das ist eine Illusion. Auswertungsregeln wie die obige werden verwendet, um die Relation $\delta \vdash e \Downarrow \nu$ festzulegen. An dieser Relation ändert sich nichts, wenn wir die Voraussetzungen der Beweisregel umordnen.

$$\frac{\delta \vdash e_2 \Downarrow \nu_2 \qquad \delta \vdash e_1 \Downarrow \nu_1}{\delta \vdash (e_1, e_2) \Downarrow (\nu_1, \nu_2)}$$

Zurück zu unserem Beispiel: Wenn *factorial* zusätzlich einen Effekt hat, dann spielt die Reihenfolge der Teilrechnungen sehr wohl eine Rolle. Ebenso ist relevant, wie oft eine Rechnung durchgeführt wird. Der Ausdruck

$$\mathbf{\mathit{let}}\ f = \mathit{factorial}\ 9\ \mathbf{\mathit{in}}\ (f, f * 10)$$

hat zwar den gleichen Wert, aber wahrscheinlich einen anderen Effekt. Hier ist eine effektvolle Version von *factorial*, bei der der Unterschied zwischen den obigen Ausdrücken zu Tage tritt.

```
let rec factorial (n : Nat) : Nat =
  let () = putstring (show n ^ "\n")
  in if n = 0 then 1 else factorial (n − 1) * n
```

Die Ausgabeanweisung *putstring* (*show* n `^ "\n"`) ist ein Beispiel für einen Ausdruck, der nur seines Effektes willen, nicht aber seines Wertes wegen ausgerechnet wird. Ein wiederkehrendes Idiom in diesem Zusammenhang ist ***let*** $() = e_1$ ***in*** e_2. Die lokale Bindung dient dazu, die Auswertung von zwei Ausdrücken und damit das Auftreten von Effekten zu sequentialisieren. Zunächst wird e_1 ausgerechnet, das Ergebnis wird mit dem Muster '()' abgeglichen, anschließend wird e_2 ausgerechnet. Der Wert von e_1 spielt für die zweite Teilrechnung keine Rolle. Der Wert des gesamten Ausdrucks ist der Wert von e_2. Wir kürzen dieses Idiom mit $e_1; e_2$ ab (lies: e_1 gefolgt von e_2). Tatsächlich können wir das Semikolon auch weglassen, wenn wir e_1 und e_2 untereinander schreiben und durch das „Layout" unsere Intention ausdrücken. Weiterhin erlauben wir, die Deklaration ***let*** $() = e$ mit ***do*** e abzukürzen (diese Abkürzung werden wir in Kapitel 8 häufiger verwenden). Mit Hilfe dieses „syntaktischen Zuckers" lässt sich die effektvolle Version von *factorial* etwas kompakter aufschreiben.

```
let rec factorial (n : Nat) : Nat =
  putstring (show n ^ "\n")
  if n = 0 then 1 else factorial (n − 1) ∗ n
```

Der aktuelle Parameter n wird nach jedem Aufruf auf dem Bildschirm ausgegeben. Die Auswertung von (*factorial* 9, *factorial* 10) resultiert somit in 21 Ausgaben, wohingegen ***let*** $f =$ *factorial* 9 ***in*** $(f, f * 10)$ nur 10 Ausgaben erzeugt.

Eingaben von der Tastatur können mit Hilfe der Funktion *getline* eingefangen werden. Der Aufruf

```
getline ()
```

liest eine einzelne Zeile ein, eine Folge von Zeichen, die von einem Zeilenvorschub abgeschlossen wird.[1] Der String *ohne* den Zeilenvorschub wird als Ergebnis zurückgegeben. Auch *getline* ist keine mathematische Funktion. Wäre sie eine, dann müsste sie stets den gleichen String zurückgeben. Wie auch bei Ausgaben spielt bei Eingaben die Reihenfolge und die Multiplizität von Rechnungen eine Rolle. Der Ausdruck

```
(getline (), getline ())
```

fordert zwei Eingaben an; der Aufruf

```
let s = getline () in (s, s)
```

module Effects. Input

hingegen nur eine.

Wir können *putstring* und *getline* kombinieren, um eine Funktion zu programmieren, die die Benutzer*in zu einer Eingabe auffordert.

```
let query (prompt : String) : String =
  putstring prompt; getline ()
```

[1] Die Benutzerin bzw. der Benutzer schließt die Eingabe einer Zeile mit der Eingabetaste (auch: Return- oder Enter-Taste) ab.

Mit Hilfe von *query* können wir *human-player* kurz und knapp definieren.

```
let human-player (guess : Nat) : Bool =
  query ("Ist die Zahl gleich oder kleiner als " ^ show guess ^ "? ") = "ja"
```

Der Ratekandidat *guess* wird ausgegeben; die Eingabe der Benutzer*in wird in einen Booleschen Wert verwandelt. Hier sehen wir die Funktion in Aktion:

```
Mini⟩ player-B (human-player, 0, 99)
Ist die Zahl gleich oder kleiner als 49 ? ja
Ist die Zahl gleich oder kleiner als 24 ? nein
Ist die Zahl gleich oder kleiner als 37 ? nein
Ist die Zahl gleich oder kleiner als 43 ? nein
Ist die Zahl gleich oder kleiner als 46 ? nein
Ist die Zahl gleich oder kleiner als 48 ? ja
Ist die Zahl gleich oder kleiner als 47 ? ja
47
```

Nach sieben Runden hat der Rechner die Zahl ermittelt.

Mit Hilfe des Tests *query* ... = `"ja"` überprüfen wir, ob die Benutzer*in `ja` eingegeben hat oder etwas anderes. Um verschiedenen Eingaben den Wahrheitswert *true* zuzuordnen, ist die Funktion *contains* : *String* → *List* ⟨*String*⟩ → *Bool* nützlich. Zur Erinnerung: Die Funktion überprüft, ob das erste Argument in der angegebenen Liste enthalten ist. Der Aufruf *contains* (*query* ...) $[s_1, \ldots, s_n]$ zum Beispiel testet, ob die Benutzer*in einen der aufgeführten Strings eingegeben hat. Das nachfolgende Programm, das einen Wert vom Typ *Person* einliest, illustriert die Verwendung dieses Idioms.

```
let input-person () : Person =
  if contains (query "gender: ") ["f";"female"] then
    Female {name = query "name:    "}
  else
    Male   {name = query "name:    ";
            bald = contains (query "bald?:   ") ["y";"yes"]}
```

Zunächst wird das Geschlecht abgefragt; in Abhängigkeit von der Antwort werden ein oder zwei weitere Eingaben getätigt. Die folgende Interaktion illustriert die Verwendung von *input-person*.

```
Mini⟩ input-person ()
gender : male
name   : Ralf
bald?  : yes
Male {name = "Ralf"; bald = true}
```

Es ist bemerkenswert, dass *input-person* eine Funktion des Typs *Unit* → *Person* ist und nicht ein Wert des Typs *Person*. Wäre *input-person* durch eine Wertebindung der

Form ***let*** *input-person* = ... gegeben, dann würde *ein* Datensatz unmittelbar bei Abarbeitung der Deklaration eingelesen. Der Bezeichner *input-person* stände nachfolgend für den eingegebenen Datensatz. Die obige Funktions*definition* ist hingegen effektfrei, da eine Funktion unmittelbar zu sich selbst auswertet! Die Eingaben werden erst angefordert, wenn die Funktion aufgerufen wird — mit '()' als Dummyargument. Und: Jeder weitere Aufruf führt zu einer erneuten Eingabe. Es besteht also ein großer Unterschied zwischen einem Ausdruck vom Typ *Person* und einem Ausdruck vom Typ $Unit \to Person$. Funktionen des Typs $Unit \to t$ bzw. $t \to Unit$ werden uns in diesem Kapitel häufig begegnen. Der Dummytyp '*Unit*' ist in der Regel ein Indiz dafür, dass eine Funktion effektvoll ist.

7.1.1. Abstrakte Syntax

Die Funktionen *putstring* und *getline* lassen sich auf Funktionen zurückführen, die ein einzelnes Zeichen ausgeben bzw. einlesen.

$e ::= \cdots$	*Ein- und Ausgabeoperationen:*
$\mid$ *getchar* e	Einlesen eines Zeichens
$\mid$ *putchar* e	Ausgabe eines Zeichens
$\mid$ *readFromFile* e	Einlesen von einer Datei
$\mid$ *writeToFile* e	Ausgabe in eine Datei

Zusätzlich gibt es Funktionen, die eine Textdatei einlesen bzw. einen String in eine Textdatei ausgeben. Eine Textdatei ist eine Folge von Zeichen, die auf einem Speichermedium abgelegt ist. Eine *Datei* kann die Laufzeit eines Programms überdauern und wird für die Verwaltung *persistenter* Daten verwendet (lat. anhaltend, beharrlich). In vielen Anwendungen sind Daten tatsächlich wichtiger als Programme. Mehr zum diesem Themenkreis erfahren Sie im weiteren Studium in der Vorlesung Informationssysteme.

7.1.2. Statische Semantik

Die Ein- und Ausgabeoperationen verarbeiten einzelne Zeichen.

$$\frac{\Sigma \vdash e : Unit}{\Sigma \vdash getchar\ e : Char} \qquad \frac{\Sigma \vdash e : Char}{\Sigma \vdash putchar\ e : Unit}$$

$$\frac{\Sigma \vdash e : String}{\Sigma \vdash readFromFile\ e : String} \qquad \frac{\Sigma \vdash e : String * String}{\Sigma \vdash writeToFile\ e : Unit}$$

Die Funktion *readFromFile* bzw. *writeToFile* erwartet als Argument bzw. als erstes Argument den Namen einer Datei.

7.1.3. Dynamische Semantik

Wir haben schon mehrfach angesprochen, dass sich die Auswertung mit dem Einzug von Effekten verändert. Diese Veränderung findet ihren Niederschlag bei der Aufstellung der dynamischen Semantik. Zunächst einmal ist nicht unmittelbar klar, wie wir die

Beweisregeln modifizieren müssen, damit wir Interaktionen mit der Umwelt modellieren können. Schließlich sind Beweisregeln genauso wenig interaktiv wie mathematische Funktionen. Überlegen wir: Ein Ausdruck hat neben einem Wert zusätzlich einen Effekt. Die dreistellige Relation $\delta \vdash e \Downarrow \nu$ taugt nicht mehr für dieses Szenario. Es liegt nahe, die Auswertungsrelation zu einer vierstelligen Relation zu erweitern, die eine Umgebung mit einem Ausdruck, einem Effekt und einem Wert in Beziehung setzt.

$$\delta \vdash e \Downarrow_t \nu$$

Den Effekt t notieren wir als Index an den Pfeil. Bleibt zu klären, was ein Effekt ist. Wir modellieren einen Effekt als Sequenz von externen Ereignissen, wobei ein einzelnes Ereignis zum Beispiel die Ein- oder Ausgabe eines einzelnen Unicode-Zeichens[2] ist.

$c \in \mathsf{Unicode}$

$t \in \mathsf{Event}^*$	Sequenz von Ereignissen
$\mathsf{Event} ::=$	Ereignis
$\mid \; in(c)$	Eingabe von c
$\mid \; out(c)$	Ausgabe von c

Wir beschränken uns auf die Definition von *getchar* und *putchar*; die dateiverarbeitenden Funktionen *readFromFile* und *writeToFile* werden analog behandelt.

$$\frac{\delta \vdash e \Downarrow_t ()}{\delta \vdash \mathit{getchar}\ e \Downarrow_{t \cdot in(c)} c} \qquad \frac{\delta \vdash e \Downarrow_t c}{\delta \vdash \mathit{putchar}\ e \Downarrow_{t \cdot out(c)} ()}$$

Wie üblich wird zunächst das Funktionsargument ausgerechnet. Dabei kann eine Folge von Ereignissen auftreten; *getchar* erweitert diese Folge um ein Eingabeereignis, *putchar* entsprechend um ein Ausgabeereignis. Schauen wir uns ein einfaches Beispiel an.

$$\frac{\dfrac{\varnothing \vdash () \Downarrow_\epsilon ()}{\varnothing \vdash \mathit{getchar}\ () \Downarrow_{in(\mathtt{h})} \mathtt{'h'}}}{\varnothing \vdash \mathit{putchar}\ (\mathit{getchar}\ ()) \Downarrow_{in(\mathtt{h}) \cdot out(\mathtt{h})} ()}$$

Das leere Tupel '()' wertet zu sich selbst aus; die Auswertung hat keinen Effekt. Weiterhin sehen wir, dass *putchar* (*getchar* ()) insgesamt zu '()' auswertet und dabei die Ereignisfolge $in(\mathtt{h}) \cdot out(\mathtt{h})$ auftritt. Das ist nicht die einzige mögliche Ereignisfolge: Auch $in(\mathtt{a}) \cdot out(\mathtt{a})$ oder $in(\mathtt{l}) \cdot out(\mathtt{l})$ usw. sind denkbar, nicht aber $in(\mathtt{h}) \cdot out(\mathtt{a})$ oder $in(\mathtt{a}) \cdot out(\mathtt{h})$. Die Auswertungsrelation ist eine „echte“ Relation — eine Umgebung, ein Ausdruck, eine Folge von Ereignissen und ein Wert werden zueinander in Beziehung gesetzt — und trägt damit der Tatsache Rechnung, dass viele unterschiedliche Interaktionen mit der Benutzer*in möglich sind.

[2] Unicode ist ein internationaler Standard, der die Kodierung und Darstellung von Schriftzeichen und Textelementen regelt. Insbesondere definiert Unicode verschiedene Zeichenkodierungen, Zuordnungen von Zeichen zu Zahlen. Zu den gängisten Kodierungen zählen UTF-8, UTF-16 und UTF-32. In der aktuellen Version, Unicode 11.0, werden ingesamt 137.439 verschiedene Zeichen definiert, eine beeindruckende Zahl, die die immense Vielfalt von Sprachen und Schriften in Kultur und Wissenschaft widerspiegelt.

Jetzt, da wir die Auswertungsrelation um ein Argument erweitert haben, müssen wir *alle* bisher aufgeführten Auswertungsregeln anpassen! Schließlich kann jeder Teilausdruck einen Effekt haben, auch zum Beispiel die Summanden einer Addition:

$$(\mathit{putstring}\ \texttt{"Hello, "}; 4700) + (\mathit{putstring}\ \texttt{"world!"}; 11)$$

Nicht, dass dieser Programmierstil empfehlenswert ist — im Gegenteil —, aber möglich sind solche Ausdrücke und somit müssen wir uns um deren Semantik kümmern. Glücklicherweise ist die Anpassung der Regeln recht einfach. Die Paarregel wird zum Beispiel wie folgt abgeändert.

$$\frac{\delta \vdash e_1 \Downarrow_{t_1} \nu_1 \qquad \delta \vdash e_2 \Downarrow_{t_2} \nu_2}{\delta \vdash (e_1, e_2) \Downarrow_{t_1 \cdot t_2} (\nu_1, \nu_2)}$$

Beide Teilausdrücke haben einen Effekt, t_1 bzw. t_2; der kumulierte Effekt des Paarausdrucks ist $t_1 \cdot t_2$. Somit treten die Effekte der ersten Komponente vor den Effekten der zweiten Komponente auf. Allgemein werden Ausdrücke von links nach rechts abgearbeitet und Effekte werden in dieser Reihenfolge sichtbar. Eine Auswertungsregel der Form

$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \qquad \cdots \qquad \delta_n \vdash e_n \Downarrow \nu_n}{\delta \vdash e \Downarrow \nu}$$

muss somit wie folgt erweitert werden:

$$\frac{\delta_1 \vdash e_1 \Downarrow_{t_1} \nu_1 \qquad \cdots \qquad \delta_n \vdash e_n \Downarrow_{t_n} \nu_n}{\delta \vdash e \Downarrow_{t_1 \cdot \ldots \cdot t_n} \nu}$$

Die Reihenfolge der Effekte wird durch die Konkatenation der Ereignissequenzen festgelegt.

Die Einführung von effektvollen Ausdrücken hat einen dramatischen Effekt auf die Semantik unserer Programmiersprache. Diese legt nunmehr pedantisch fest, in welcher Reihenfolge ein Programm abgearbeitet wird. Das ist in gewisser Weise ein Rückschritt. Bis dato konnten zum Beispiel die Teilausdrücke in $e_1 + e_2$ gleichzeitig oder im Fachjargon *parallel* ausgerechnet werden. Jetzt ist die sequentielle Auswertung die Norm. Wenn wir eine parallele Auswertung wegen des möglichen Geschwindigkeitsvorteils bevorzugen, dann müssen wir sicherstellen, dass e_1 keine Effekte hat, dass also stets $e_1 \Downarrow_\epsilon \nu_1$ gilt. Diese Eigenschaft ist wie viele andere nicht formal entscheidbar, so dass man sich mit Näherungen an das tatsächliche Verhalten zufriedengeben muss. Tatsächlich ist die Parallelisierung von Programmen eine der großen Herausforderungen der Informatik. Trotz existierender Hardwareunterstützung (Stichwort: Multi-Core Prozessor), werden viele Programme weiterhin streng sequentiell ausgewertet.

7.1.4. Vertiefung

Eingabe mit Validierung Wir haben schon angesprochen, dass interaktive Programme wegen ihrer Interaktionen schwieriger zu lesen und zu verstehen sind als effektfreie

Programme. Aus diesem Grund sollte man versuchen, Effekte auf einige wenige Funktionen zu beschränken und so viel wie möglich effektfrei zu rechnen. Wie dies auch für hochgradig interaktive Programme gelingen kann, wollen wir uns im Folgenden ansehen.

Die Programme aus der Einleitung zeigen, wie man systematisch Daten einliest; sie prüfen die Eingaben aber nicht auf Plausibilität.

```
Mini⟩ query "age: "
age : Hello, world !
"Hello, world!"
```

Offenbar soll das Alter abgefragt werden, als Eingabe wird aber ein beliebiger String akzeptiert. Es wäre wünschenswert, wenn fehlerhafte Eingaben erkannt werden und die Eingabe entsprechend wiederholt wird. Was aber ist eine fehlerhafte Eingabe? Das hängt vom jeweiligen Anwendungsfall ab: Eine Altersangabe ist eine höchstens dreistellige natürliche Zahl; ein Vorname besteht aus Buchstaben und vielleicht einigen Sonderzeichen. Mit anderen Worten, die interaktive Funktion *query* kann nicht beurteilen, ob eine Eingabe gültig ist. Diese Erkenntnis legt nahe, *query* mit einem *Validator* zu parametrisieren. Ein Validator bildet einen String auf ein Element des folgenden Datentyps ab.

```
type Result ⟨'value⟩ =
  | Okay of 'value
  | Error of String
```

Der Variantentyp ist eine Variante `;-)` von *Option*: *Okay* entspricht *Some*, *Error* korrespondiert zu *None*. Der Konstruktor *Error* bietet im Unterschied zu *None* zusätzlich die Möglichkeit, anzugeben, warum eine Validierung gescheitert ist. Also: Ist der String zulässig, wird *Okay value* zurückgegeben, wobei *value* der semantische Wert des Strings ist, zum Beispiel eine natürliche Zahl; schlägt die Validierung fehl, wird *Error msg* zurückgegeben, wobei *msg* eine aussagekräftige Fehlermeldung ist. Nach diesen Vorarbeiten können wir eine validierende Version von *query* definieren:

```
let rec checked-query (prompt : String,
                       check : String → Result ⟨'value⟩) : 'value =
  match check (query (prompt ^ ": ")) with
  | Okay v      → v
  | Error msg → putline ("*** " ^ msg);
                checked-query (prompt, check)
```

Es werden solange Eingaben angefordert, bis die Eingabe von *check* abgesegnet wird. Im Fehlerfall wird die Benutzer*in auf den Fehler hingewiesen.

Validatoren sind einfache, effektfreie Funktionen: *is-nat* zum Beispiel überprüft, ob die Eingabe eine nicht-leere Folge von Ziffern ist.

```
let is-nat (s : String) : Result ⟨Nat⟩ =
  if s <> "" && String.forall Char.IsDigit s then
    Okay (Nat.Parse s)
  else
    Error "natural number expected"
```

Bei der Definition greifen wir auf die Funktion

forall : (*Char* → *Bool*) → *String* → *Bool*

zurück, die überprüft, ob alle Zeichen eines Strings das angegebene Prädikat erfüllen. Die Funktion *is-nat* ist im Prinzip ein einfacher Parser (!), der einen String in einen semantischen Wert überführt oder fehlschlägt. Sind die Eingabedaten komplizierter, so bietet es sich in der Tat an, eine Grammatik für die Menge aller zulässigen Eingaben aufzustellen und mit den Methoden aus Abschnitt 6.4.3 einen Parser zu konstruieren. Wenn der Parser die Eingabe nicht akzeptiert, muss man sich zusätzlich darum kümmern, eine aussagekräftige Fehlermeldung zu generieren.

Die folgende Interaktion zeigt *checked-query* in Aktion.

```
Mini⟩ checked-query ("age", is-nat)
age : Hello, world !
*** natural number expected
age : 4711
4711
```

Als Ergebnis des Aufrufs wird tatsächlich eine Zahl zurückgegeben, kein String. Natürlich ist 4711 ein extrem hohes Alter. Wir sollten zusätzlich verlangen, dass die Angabe kleiner als 123 ist.[3]

```
Mini⟩ checked-query ("age", both (is-nat, is-less 123))
age : Ralf
*** natural number expected
age : 4711
*** number must be less than 123
age : 41
41
```

Die Funktion *both* kombiniert zwei Validatoren: *both* (*is-nat*, *is-less* 123) fordert, dass die Eingabe eine Folge von Ziffern ist *und* dass die korrespondierende Zahl kleiner als 123 ist. Um den String nicht wiederholt in eine Zahl umwandeln zu müssen, wird der semantische Wert des ersten Validators an den zweiten Validator weitergereicht: *is-nat* vom Typ *String* → *Result* ⟨*Nat*⟩ wird so mit *is-less* 123 vom Typ *Nat* → *Result* ⟨*Nat*⟩ kombiniert.

```
let both (first : 'a → Result ⟨'b⟩, second : 'b → Result ⟨'c⟩) : 'a → Result ⟨'c⟩ =
  fun x → match first x with
    | Okay y     → second y
    | Error msg → Error msg
```

Die Funktion ist „sehr polymorph“: *first* und *second* dürfen fast beliebige Argument- und Ergebnistypen haben; lediglich der Ergebnistyp des ersten Validators muss zum

[3] Warum 123? Als bis dato ältester Mensch der Welt gilt Jeanne Calment (1875–1997) mit 122 Jahren und 164 Tagen.

Argumenttyp des zweiten passen. Die Funktion *is-less* kleidet im Wesentlichen die Vergleichsoperation $<$ in *Okay* bzw. *Error* ein.

```
let is-less (n : Nat) : Nat → Result ⟨Nat⟩ = fun m →
  if m < n then Okay m
           else Error ("number must be less than " ^ show n)
```

Mit Hilfe von *checked-query* können wir auch Eingaben auf eine vorgegebene Auswahl von Strings beschränken.

```
let choice (prompt : String, choices : List ⟨String⟩) : String =
  checked-query (prompt,
    fun s → if contains s choices
            then Okay s
            else Error ("choices: " ^ concat ", " choices))
```

Die Funktion

```
concat : String → List ⟨String⟩ → String
```

konkateniert eine Liste von Strings und fügt zwischen je zwei Elemente den angegebenen Separator, erstes Argument, ein. Die folgende Interaktion illustriert die Verwendung von *choice*.

```
Mini⟩ choice ("gender", ["f"; "m"; "female"; "male"])
gender : sehr maskulin
*** choices : f, m, female, male
gender : m
"m"
```

Mit diesen Zutaten können wir die Funktion *input-person* neu definieren, diesmal inklusive Validierung der getätigten Eingaben.

```
let input-person () : Person =
  if contains (choice ("gender", ["f"; "m"; "female"; "male"])) ["f"; "female"]
  then
    Female {name = checked-query ("name   ", is-name)}
  else
    Male {name = checked-query ("name   ", is-name);
          bald = contains (choice ("bald? ", ["y"; "n"; "yes"; "no"])) ["y"; "yes"]}
```

Die Funktion *is-name* überprüft, ob die Eingabe ein gültiger Name ist. Die Definition ist der Phantasie der Leserin oder des Lesers überlassen.

module Effects.Trace

Tracing Wechseln wir das Thema: Ausgaben können auch beim Testen von Programmen nützliche Dienste leisten. Etwa, wenn wir wissen wollen, mit welchen aktuellen Parametern eine Funktion aufgerufen wird oder welche Ergebnisse sie zurückliefert.

```
let rec factorial (n : Nat) : Nat =
  if n = 0 then
    Return 1
  else
    Return (Call factorial (n − 1) ∗ n)
```

Mit Hilfe von *Return* wird der (Rückgabe-) Wert eines Ausdrucks protokolliert; mit Hilfe von *Call* der aktuelle Parameter einer Funktion. Wir stellen die Definition von *Return* und *Call* zurück und schauen uns zunächst einige Anwendungen an.

```
Mini⟩ Call factorial 10
call   10
call   9
call   8
call   7
call   6
call   5
call   4
call   3
call   2
call   1
call   0
return 1
return 1
return 2
return 6
return 24
return 120
return 720
return 5040
return 40320
return 362880
return 3628800
3628800
```

Man sieht sehr schön, wie der aktuelle Parameter auf dem Hinweg zur Rekursionsbasis schrittweise verkleinert wird und wie auf dem Rückweg der Funktionswert anschwillt.

Die Rekursion muss nicht immer linear sein. Die sogenannte *Fibonacci-Funktion* demonstriert ein lebhafteres Auf und Ab.

```
let rec fibonacci (n : Nat) : Nat =
  Return (if n ⩽ 1 then n
                   else Call fibonacci (n − 1) + Call fibonacci (n − 2))
```

Die Funktion ist nach dem italienischen Mathematiker Leonardo da Pisa, genannt Fibonacci, benannt, der mit dieser Funktion das Wachstum einer Kaninchenpopulation

modellierte. Die Funktion beantwortet die Frage „Wie viele Kaninchenpaare entstehen nach n Monaten aus einem einzigen Paar, wenn jedes Paar ab dem zweiten Lebensmonat ein weiteres Paar auf die Welt bringt?“. Nach dem ersten Monat gibt es ein Paar; danach erhöht sich die Zahl der Paare, *fibonacci* $(n-1)$, um die Zahl der geschlechtsreifen Paare, *fibonacci* $(n-2)$. Abbildung 7.1 zeigt das Auf und Ab der rekursiven Aufrufe von *fibonacci* 6 (und lässt reichlich Platz für Notizen). Man sieht sehr schön, dass wiederholt die gleichen Aufrufe getätigt werden (*call* 2 tritt fünfmal auf) und immer wieder neu berechnet werden. In Abschnitt 7.2 werden wir uns damit beschäftigen, wie man diese wiederholten Berechnungen vermeiden kann.

Es bleibt, die Definitionen von *Return* und *Call* nachzureichen.

```
let Return (x : 'a) : 'a =
  putline ("return " ^ show x); x
```

Wertmäßig ist *Return* die Identität: Das Argument wird als Ergebnis zurückgegeben. Damit *Return* für beliebige Werte verwendet werden kann, abstrahieren wir vom Typ des Arguments und verwenden *show*, um den beliebigen Wert in einen String zu überführen.

```
let Call (f : 'a → 'b) : 'a → 'b =
  fun x → putline ("call   " ^ show x); f x
```

Wertmäßig ist *Call* die Identität von Funktionen. Deswegen abstrahieren wir über zwei Typen: den Argument- und den Ergebnistyp der Funktion.

7.2. Zustand

> *Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics.*
>
> — C.A.R. Hoare (1934–), *CACM 12(10), 1969*

Wir haben im letzten Abschnitt gesehen, wie wir die Abarbeitung rekursiver Funktionen am Bildschirm protokollieren können. Versieht man verschiedene Funktionen mit Kontrollausgaben, so werden die Protokolle schnell unübersichtlich, so dass der Wunsch aufkommt, die Ausgaben gezielt an- und ausschalten zu können. Eine solche *Außensteuerung* ist auch nützlich, wenn man die Protokollierung insgesamt ausschalten, die Protokollanweisungen *Return* und *Call* aber im Programm belassen möchte.

Funktionen können bis dato nur über das Funktionsargument von außen gesteuert werden. Diese Art der Steuerung ist für unser Szenario ungeeignet: Um das Protokoll an- oder auszuschalten, müssten wir das Programm an vielen verschiedenen Stellen ändern — überall dort, wo Protokollanweisungen stehen. Wir wünschen uns einen globalen Schalter, nicht viele lokale. Mit den bisherigen Mitteln lässt sich ein globaler Schalter nicht bewerkstelligen; wir benötigen ein neues Sprachkonstrukt. Die grundlegende Idee ist, ein Gedächtnis bzw. einen Speicher einzuführen, der abgefragt und manipuliert werden kann. Ein Schalter kann dann durch einen Speicher bzw. eine *Speicherzelle* realisiert werden, die einen Booleschen Wert enthält.

```
Mini⟩ Call fibonacci 6
call   6
call   5
call   4
call   3
call   2
call   1
return 1
call   0
return 1
return 2
call   1
return 1
return 3
call   2
call   1
return 1
call   0
return 1
return 2
return 5
call   3
call   2
call   1
return 1
call   0
return 1
return 2
call   1
return 1
return 3
return 8
call   4
call   3
call   2
call   1
return 1
call   0
return 1
return 2
call   1
return 1
return 3
call   2
call   1
return 1
call   0
return 1
return 2
return 5
return 13
```

Abbildung 7.1.: Trace von *fibonacci* 6.

let *trace* = ***ref*** *false*

Der Ausdruck ***ref*** *false* legt eine neue Speicherzelle an. Im Fachjargon sagt man, die Speicherzelle wird *allokiert*. Eine Speicherzelle ist wie ein Behälter. Der initiale *Inhalt* der allokierten Speicherzelle ist *false*. Als Ergebnis des Aufrufs gibt ***ref*** die *Adresse* der allokierten Speicherzelle zurück. Diese Adresse benötigen wir, um den Inhalt der Speicherzelle später abzufragen oder zu verändern. Aus diesem Grund binden wir die Adresse an den Bezeichner *trace*. Die Adresse hat den Typ *Ref*⟨*Bool*⟩. Der Bezeichner *trace* ist also kein Boolescher Wert, sondern er verweist auf einen Booleschen Wert (***ref*** bzw. *Ref* kürzt *reference* ab, engl. für *Verweis* oder Referenz).

Bezeichnet e die Adresse einer Speicherzelle, so ermittelt $!e$ den Inhalt der Speicherzelle (lies: „bang e“, engl. für Knall nicht deutsch für ängstlich); e muss ein Ausdruck vom Typ *Ref*⟨t⟩ sein, $!e$ ist dann entsprechend ein Ausdruck vom Typ t. Mit $!$*trace* lässt sich somit der Zustand des Schalters abfragen. Damit können wir eine Version von *Return* definieren, die sich von außen über den Schalter steuern lässt (*Call* wird analog behandelt).

```
let Return (x :'value) :'value =
  (if !trace then putline ("return " ^ show x) else ()); x
```

Nur wenn der Inhalt der Speicherzelle *true* ist, erfolgt die Kontrollausgabe. Die Alternative ***if*** e_1 ***then*** e_2 ***else*** () lässt sich übrigens zu ***if*** e_1 ***then*** e_2 abkürzen. (Die einarmige Alternative ergibt aber nur Sinn, wenn der ***then***-Zweig den Typ *Unit* hat.)

Mit Hilfe der *Zuweisung* $e_1 := e_2$ können wir schließlich den Inhalt der von e_1 referenzierten Speicherzelle durch den Wert von e_2 ersetzen (lies: e_1 wird zu e_2); e_1 muss den Typ *Ref*⟨t⟩ besitzen und e_2 den Typ t. Der Wert der Zuweisung ist ‘()’, unser liebgewonnener Dummywert. Der Ausdruck *trace* := *true* schaltet somit die Protokollierung an und *trace* := *false* entsprechend aus.

```
Mini⟩  factorial 1
1
Mini⟩  trace := true
()
Mini⟩  !trace
true
Mini⟩  factorial 1
call    0
return 1
return 1
1
Mini⟩  trace := false
()
Mini⟩  !trace
false
Mini⟩  factorial 1
1
```

module
Effects.
Bank

Die Interaktion zeigt, dass auch '!' keine Funktion im mathematischen Sinne ist: Der gleiche Aufruf, !*trace*, führt zu zwei unterschiedlichen Ergebnissen.

Speicherzellen können beliebige Werte enthalten: Boolesche Werte, natürliche Zahlen, Funktionen, Adressen anderer Speicherzellen usw. Eine Speicherzelle vom Typ $Ref\langle Nat\rangle$ kann zum Beispiel verwendet werden, um ein Bankkonto zu modellieren.

```
module Account =
  let private funds = ref 0
  let deposit (amount : Nat) =
    funds := !funds + amount
  let withdraw (amount : Nat) =
    let old = !funds
    funds := !funds − amount
    old − !funds
  let balance () = !funds
```

Mit *ref* 0 wird eine mit 0 initialisierte Speicherzelle allokiert, die Repräsentation des Kontostands. Die Funktion *deposit* modelliert eine Einzahlung, *withdraw* entsprechend eine Auszahlung. Der Ausdruck *funds* := !*funds* + *amount* erhöht den Inhalt der Speicherzelle um den Betrag *amount* und ist ein typisches Idiom für Speicherzellen vom Typ $Ref\langle Nat\rangle$. Die Funktion *withdraw* gibt als Ergebnis den tatsächlich ausgezahlten Betrag zurück; (fast) wie im richtigen Leben kann der Kontostand nicht negativ werden. Mit *balance* kann der Kontostand abgefragt werden.

Die Deklaration der Speicherzelle ist lokal zu den Deklarationen der drei Funktionen. Der Zusatz *private* stellt sicher, dass der Bezeichner nur innerhalb des Moduls, nicht aber außerhalb sichtbar ist. Warum schränken wir die Sichtbarkeit ein? Nun, wir wollen sicherstellen, dass der Kontostand nur indirekt mit Hilfe von *deposit* und *withdraw* manipuliert wird und nicht etwa direkt über eine Zuweisung. Heimliche Kontomanipulationen sind ein in der Bankenwelt ungern gesehener Vorgang. Das Bankkonto existiert zwar — Speicherzellen leben ewig —, aber es kann außerhalb des Moduls nicht direkt angesprochen werden. (Das ist ähnlich wie bei der Post: Um jemandem einen Brief zu schicken, benötigt man ihre oder seine Adresse; hat man die Adresse nicht, so folgt daraus nicht, das die- oder derjenige nicht existiert.) Die folgende Interaktion demonstriert Ein- und Auszahlungen.

```
Mini⟩ Account.deposit 4711
()
Mini⟩ Account.withdraw 815
815
Mini⟩ Account.withdraw 2765
2765
Mini⟩ Account.withdraw 2765
1131
Mini⟩ Account.withdraw 2765
0
```

Auch hier wird deutlich, dass *withdraw* keine mathematische Funktion ist: drei Aufrufe der Form *withdraw* 2765, drei unterschiedliche Funktionsergebnisse. Diese Eigenschaft — nicht-mathematisch — ähnelt einer ansteckenden Krankheit: *withdraw* greift auf '!' zurück und steckt sich an.

7.2.1. Abstrakte Syntax

Wir erweitern Mini–F# um Speichermanipulationen.

$e ::= \cdots$	*Speichermanipulation:*
$\mid$ ***ref*** e	Allokation
$\mid\ !e$	Dereferenzierung
$\mid\ e_1 := e_2$	Zuweisung

Der Ausdruck ***ref*** e allokiert eine Speicherzelle und gibt die Adresse der bzw. eine Referenz auf die Speicherzelle zurück. Aus diesem Grund heißt der Zugriff $!e$ auch Dereferenzierung. Die Zuweisung ist charakteristisches Merkmal der meisten Programmiersprachen (siehe auch das Zitat am Anfang des Abschnitts).

7.2.2. Statische Semantik

Adressen erhalten einen Referenztyp; dieser ist mit dem Typ des Inhaltes parametrisiert.

$t ::= \cdots$	*Typen:*
$\mid Ref\langle t\rangle$	Referenztyp

Bevor wir uns die Typregeln anschauen, lohnt es sich, noch einmal eine der Grundprinzipien von Mini–F# ins Gedächtnis zu rufen: Ausdrücke können beliebig miteinander kombiniert werden. Die Zuweisung $e_1 := e_2$ ist ein Ausdruck und kann überall dort verwendet werden, wo ein Ausdruck verlangt wird. Die linke Seite der Zuweisung ist ebenfalls ein Ausdruck. Als einzige Einschränkung verlangen wir, dass e_1 zu einer Adresse auswertet und dass die rechte Seite zur linken passt. Diese Einschränkungen formulieren wie immer die Typregeln.

$$\frac{\Sigma \vdash e : t}{\Sigma \vdash \textbf{\textit{ref}}\ e : Ref\langle t\rangle} \qquad \frac{\Sigma \vdash e : Ref\langle t\rangle}{\Sigma \vdash\ !e : t} \qquad \frac{\Sigma \vdash e_1 : Ref\langle t\rangle \qquad \Sigma \vdash e_2 : t}{\Sigma \vdash e_1 := e_2 : Unit}$$

Die Deklaration ***let*** $p = (\textbf{\textit{ref}}\ false, \textbf{\textit{ref}}\ 0)$ zum Beispiel bindet p an einen Wert vom Typ $Ref\langle Bool\rangle * Ref\langle Nat\rangle$. Bezüglich dieser Deklaration sind sowohl $!(fst\ p)$ als auch $snd\ p := 4711$ zulässige Ausdrücke. Die linke Seite der Zuweisung hat den Typ $Ref\langle Nat\rangle$, ist also eine Adresse, und die rechte Seite vom Typ Nat passt dazu.

7.2.3. Dynamische Semantik

Die drei neuen Konstrukte manipulieren einen sogenannten (Haupt-) Speicher. Der Zusatz 'Haupt' grenzt den Speicher vom Hintergrundspeicher ab, auf dem Daten persistent

gespeichert werden. Daten im Hauptspeicher sind flüchtig: Sie überdauern eine Programmausführung bzw. eine interaktive Sitzung nicht.

Ein Speicher ist eine endliche Abbildung von Adressen auf Werte.

$a \in \mathsf{Addr}$	*Adressen*
$\sigma \in \mathsf{Addr} \to_{\mathrm{fin}} \mathsf{Val}$	*Speicher*

Den Bereich der Adressen lassen wir abstrakt; wir fordern nur, dass es unendlich viele Adressen gibt.

Die speichermanipulierenden Konstrukte haben wie die Ein- und Ausgabefunktionen neben einem Wert zusätzlich einen Effekt. Der Effekt besteht hier in der Veränderung des Speichers. Wie auch im letzten Abschnitt müssen wir die Auswertungsrelation erweitern, um den Effekt beschreiben zu können. Aus der dreistelligen Relation $\delta \vdash e \Downarrow \nu$ wird eine fünfstellige Relation

$$\delta \vdash \sigma \parallel e \Downarrow \nu \parallel \sigma'$$

Der Ausdruck e wertet zu ν aus und bewirkt zusätzlich eine Zustandsänderung: σ ist der Speicher vor der Auswertung von e und σ' ist der Speicher nach der Auswertung. Eigentlich müssten wir die vierstellige Relation $\delta \vdash e \Downarrow_t \nu$ des letzten Abschnitts zu einer sechsstelligen Relation

$$\delta \vdash \sigma \parallel e \Downarrow_t \nu \parallel \sigma'$$

erweitern, aber das wollen wir aus Gründen der Übersichtlichkeit nicht tun. In der Tat verändern die Ein- und Ausgabefunktionen nicht den Zustand und umgekehrt berühren die speichermanipulierenden Konstrukte nicht die Ereignisfolge. Ebenfalls im Interesse der Leserlichkeit lassen wir in diesem Abschnitt die Umgebung δ unter den Tisch fallen — keine der folgenden Regeln benötigt oder manipuliert die Umgebung.

Da der Ausdruck ***ref*** e eine Adresse zurückgibt, müssen wir noch den Bereich der Werte um Adressen erweitern.

$\nu ::= \cdots$	*Werte:*
$\mid\ a$	Adresse

Nach diesen Vorarbeiten können wir die dynamische Semantik der Konstrukte festlegen.

$$\frac{\sigma \parallel e \Downarrow \nu \parallel \sigma'}{\sigma \parallel \textbf{\textit{ref}}\ e \Downarrow a \parallel \sigma', \{a \mapsto \nu\}} \qquad a \notin \mathit{dom}\ \sigma'$$

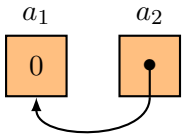
$$\frac{\sigma \parallel e \Downarrow a \parallel \sigma'}{\sigma \parallel\ !e \Downarrow \sigma'(a) \parallel \sigma'}$$

$$\frac{\sigma \parallel e_1 \Downarrow a \parallel \sigma_1 \qquad \sigma_1 \parallel e_2 \Downarrow \nu \parallel \sigma_2}{\sigma \parallel e_1 := e_2 \Downarrow () \parallel \sigma_2, \{a \mapsto \nu\}}$$

Generell gilt, dass zunächst die Teilausdrücke ausgerechnet werden. Im Fall von ***ref*** e wird zunächst e ausgewertet; diese Auswertung kann den Speicher verändern: σ wird

zu σ'. Dann wird eine beliebige Adresse bestimmt, die nicht im Definitionsbereich von σ' enthalten ist: a ist diese „frische“ Adresse. Schließlich wird σ' um eine neue Speicherzelle, $\{a \mapsto \nu\}$, erweitert. Schauen wir uns die Auswertung von ***ref*** (***ref*** 0) an.

$$\dfrac{\dfrac{\emptyset \parallel 0 \Downarrow 0 \parallel \emptyset}{\emptyset \parallel \textbf{\textit{ref}}\ 0 \Downarrow a_1 \parallel \{a_1 \mapsto 0\}}}{\emptyset \parallel \textbf{\textit{ref}}\ (\textbf{\textit{ref}}\ 0) \Downarrow a_2 \parallel \{a_1 \mapsto 0, a_2 \mapsto a_1\}}$$



Zunächst wird die Auswertung von ***ref*** 0 und dann die Auswertung von 0 angestoßen; 0 wertet zu 0 aus, ohne den (leeren) Speicher zu verändern; dann wird eine Speicherzelle angelegt $\{a_1 \mapsto 0\}$ und schließlich eine zweite $\{a_2 \mapsto a_1\}$.

Die Dereferenzierung ist ein *lesender Speicherzugriff*. Die Auswertung von e kann den Speicher verändern; der eigentliche Zugriff ist aber effektfrei: Der modifizierte Speicher σ' wird nicht mehr verändert.

Im Gegensatz dazu ist $e_1 := e_2$ ein *schreibender Speicherzugriff*. Die Auswertung von e_1 kann den Speicher verändern: σ wird zu σ_1. Die Auswertung von e_2 „sieht“ den modifizierten Speicher und ändert ihn zu σ_2. Die Zuweisung selbst schließlich modifiziert σ_2 zu $\sigma_2, \{a \mapsto \nu\}$: der Inhalt der durch a adressierten Speicherzelle wird mit ν überschrieben.

Ähnlich wie im letzten Abschnitt müssen wir *alle* bisher aufgeführten Auswertungsregeln anpassen! Wie illustrieren die Änderungen zunächst an der Paarregel.

$$\dfrac{\sigma_0 \parallel e_1 \Downarrow \nu_1 \parallel \sigma_1 \qquad \sigma_1 \parallel e_2 \Downarrow \nu_2 \parallel \sigma_2}{\sigma_0 \parallel (e_1, e_2) \Downarrow (\nu_1, \nu_2) \parallel \sigma_2}$$

Beide Teilausdrücke verändern potentiell den Speicher: e_1 ändert σ_0 zu σ_1, wie bei der Zuweisung „sieht“ e_2 den modifizierten Speicher und ändert ihn zu σ_2. Die kumulierte Änderung von σ_0 zu σ_2 ist der Effekt des Paarausdrucks. Allgemein gilt, wie auch im letzten Abschnitt, dass Ausdrücke von links nach rechts abgearbeitet werden und Effekte auch in dieser Reihenfolge sichtbar werden. Eine Auswertungsregel der Form

$$\dfrac{e_1 \Downarrow \nu_1 \qquad e_2 \Downarrow \nu_2 \qquad \cdots \qquad e_n \Downarrow \nu_n}{e \Downarrow \nu}$$

wird wie folgt angepasst:

$$\dfrac{\sigma_0 \parallel e_1 \Downarrow \nu_1 \parallel \sigma_1 \qquad \sigma_1 \parallel e_2 \Downarrow \nu_2 \parallel \sigma_2 \qquad \cdots \qquad \sigma_{n-1} \parallel e_n \Downarrow \nu_n \parallel \sigma_n}{\sigma_0 \parallel e \Downarrow \nu \parallel \sigma_n}$$

Der Zustand wird jeweils von Teilausdruck zu Teilausdruck weitergereicht. Man sagt auch, der Zustand wird durchgefädelt (engl. threading).

7.2.4. Vertiefung

Tracing Kommen wir noch einmal auf die Protokollierung von Funktionsargumenten und Funktionswerten zurück. Wir haben in der Einleitung eine Speicherzelle vom Typ *Ref*⟨*Bool*⟩ eingeführt, um die Protokollierung von außen an- und auszuschalten. Diese

Speicherzelle war in allen nachfolgenden Definitionen und Ausdrücken sichtbar. Man sollte sich frühzeitig angewöhnen, die Sichtbarkeit von Speicherzellen mit lokalen ***let***-Bindungen oder mit Hilfe von Modulen und ***private***-Annotationen zu begrenzen, so wie wir das im Fall des Bankkontos gemacht haben. Eine entsprechend modifizierte Version könnte zum Beispiel wie folgt aussehen:

```
module Trace
  let private trace = ref false
  let trace-on () =
    trace := true
  let trace-off () =
    trace := false
  let traceline (s : String) =
    if !trace then putline s
```

Die Sichtbarkeit von *trace* wird auf drei Funktionen eingeschränkt: *traceline* ist eine „*trace*-sensitive“ Version von *putline*, mit Hilfe von *trace-on* bzw. *trace-off* wird die Protokollierung an- bzw. abgeschaltet. Die Funktionen *Return* und *Call* können wir damit wie folgt definieren (der Aufruf von *putline* in der ursprünglichen Fassung ist einfach durch *traceline* ersetzt worden).

```
let Return (x : 'a) : 'a =
  Trace.traceline ("return " ^ show x); x
let Call (f : 'a -> 'b) : 'a -> 'b =
  fun x → Trace.traceline ("call   " ^ show x); f x
```

Der Zustand des Schalters kann nur mit Hilfe von *trace-on* und *trace-off* manipuliert werden. Man spricht in diesem Zusammenhang auch von *Kapselung*. Die Tatsache, dass die Funktionen eine Speicherzelle vom Typ $Ref\langle Bool\rangle$ verwenden, ist nach außen nicht sichtbar. Was ist der Vorteil der gekapselten Version? Auf den ersten Blick macht es keinen großen Unterschied, ob die Protokollierung mit *trace* := *true* oder mit *trace-on* () eingeschaltet wird. Der Vorteil der gekapselten Version ist, dass sich die Implementierung leicht ändern und erweitern lässt. Zum Beispiel, wenn wir das An- und Ausschalten selbst protokollieren wollen. Bei der gekapselten Version ist die Änderung lokal: Wir müssen nur die Definition von *trace-on* und *trace-off* modifizieren.

```
let trace-on () =
  trace := true; putline "TRACE ON"
let trace-off () =
  trace := false; putline "TRACE OFF"
```

Bei der ursprünglichen Variante ist die Änderung global: Jedes Vorkommen von *trace* := *true* muss um *putline* `"TRACE ON"` erweitert werden.

Memoisierung Wenden wir uns einem anderen Thema zu: Wenn wir ein Programm optimieren wollen, dann ist es nützlich zu wissen, wie oft eine bestimmte Funktion aufgerufen wird. (Wir erinnern uns: Man sollte nicht blindlings optimieren; es gilt, den oder die Laufzeitverbrecher zu finden. Dazu muss man, wie in jedem guten Krimi, zunächst Daten sammeln.) Zu diesem Zweck können wir einen Zähler, eine Speicherzelle vom Typ $Ref\langle Nat\rangle$, verwenden.

```
let counter = ref 0
let rec fibonacci (n : Nat) : Nat =
  counter := !counter + 1
  if n ⩽ 1 then
    n
  else
    fibonacci (n − 1) + fibonacci (n − 2)
```

Die Berechnung von *fibonacci* n ist aufwändig, wie die folgende Interaktion zeigt.

```
Mini⟩ counter := 0; let f = fibonacci 10 in (f, !counter)
(55, 177)
Mini⟩ counter := 0; let f = fibonacci 20 in (f, !counter)
(6765, 21891)
```

Die Anzahl der Aufrufe übersteigt den Wert der Fibonaccifunktion in beiden Fällen. Natürlich können wir auch *effektfrei* zählen, indem wir neben dem eigentlichen Wert zusätzlich die Anzahl der Aufrufe zurückgeben.

```
let rec counting-fibonacci (n : Nat) : Nat * Nat =
  if n ⩽ 1 then
    (n, 1)
  else
    let (f1, c1) = counting-fibonacci (n − 1)
    let (f2, c2) = counting-fibonacci (n − 2)
    (f1 + f2, c1 + c2 + 1)
```

Im Basisfall zählen wir nur den Aufruf selbst; im Rekursionsschritt kommt zu diesem Aufruf noch die Summe aus den beiden rekursiven Aufrufen hinzu. Man sieht, wir müssen uns bei der effektfreien Version mehr abstrampeln, aber der Einsatz wird auch belohnt: Aus der resultierenden Definition lässt sich ablesen, dass die Zahl c der Aufrufe für die Berechnung von *fibonacci* n fast doppelt so groß ist wie der Funktionswert von *fibonacci* $(n+1)$, es gilt: $c = 2 \cdot fibonacci\ (n+1) - 1$.

module Effects. Memoization

Das Problem bei der Berechnung von *fibonacci* ist, dass die gleichen Funktionswerte wiederholt berechnet werden. So wird beispielsweise *fibonacci* 8 bei der Berechnung von *fibonacci* 20 insgesamt 233 mal (= *fibonacci* 13) neu ausgerechnet. Da *fibonacci* eine mathematische Funktion ist, sind die wiederholten Berechnungen redundant, der Wert ist stets der gleiche. Eine naheliegende Idee ist, sich die Werte vorheriger Aufrufe zu merken

und dann auf die memorierten Werte zurückzugreifen. Mit Hilfe von Arrays lässt sich eine Funktion leicht tabellieren: [| **for** *i* **in** 0 .. 99 → *fibonacci i* |]. Abstrahieren wir von 99 und von *fibonacci*, erhalten wir

```
let memo (dom : Nat, func : Nat → 'value) : Nat → 'value =
  let
    memo-table = [| for i in 0 .. dom − 1 → func i |]
  in
    fun (n : Nat) → memo-table.[n]
let memo-fibonacci = memo (100, fibonacci)
```

Unglücklicherweise dauert die Auswertung von *memo-fibonacci* extrem (!) lange, da zunächst die Tabelle vollständig gefüllt wird. Die Werte von *fibonacci* 0 bis *fibonacci* 99 werden berechnet, ohne dass klar ist, ob diese später überhaupt benötigt werden. Eigentlich schwebt uns eine *bedarfsgetriebene* Füllung der Tabelle vor: Erst wenn ein Funktionswert angefordert wird, berechnet *memo* den entsprechenden Tabelleneintrag. Wenn wir Tabelleneinträge ändern wollen, müssen wir statt einem Array von Zahlen ein Array von Speicherzellen verwenden. Jede Speicherzelle nimmt dabei einen von zwei möglichen Zuständen an: „der Funktionswert wurde noch nicht berechnet“ oder „der Wert wurde berechnet und er lautet …“. Die beiden Zustände können wir mit Elementen des Datentyps *Option* modellieren: *None* bzw. *Some value*, wobei *value* der berechnete Wert ist. Die Memotabelle hat somit insgesamt den furchteinflößenden Typ *Array* ⟨*Ref* ⟨*Option* ⟨'*a*⟩⟩⟩ statt *Array* ⟨'*a*⟩ wie bisher.

```
let memo (dom : Nat, func : Nat → 'value) : Nat → 'value =
  let
    memo-table = [| for i in 0 .. dom − 1 → ref None |]
  in
    fun (n : Nat) → match !memo-table.[n] with
                    | None   → let v = func n
                               memo-table.[n] := Some v
                               v
                    | Some v → v
let memo-fibonacci = memo (100, fibonacci)
```

Die Auswertung der beiden Deklarationen dauert nunmehr keinen Fingerschnips. Jetzt tritt eine lange (!) Stille ein, wenn wir zum Beispiel *memo-fibonacci* 99 aufrufen. Ist der Wert nach Hunderten von Jahren bestimmt, dann wird jeder weitere Aufruf sofort bedient. Aber, warum das lange Warten? Nun, die Memoisierung greift nicht beim Füllen der Tabelle: Die *rekursiven* Aufrufe der Fibonacci Funktion werden nicht memoisiert. Der Programmcode macht das deutlich: Die Definition von *memo-fibonacci* ist nicht rekursiv — *memo-fibonacci* ist durch eine Wertebindung gegeben —, der zweite Parameter von *memo* ist die rekursive Funktion. Die rekursiven Aufrufe namens *fibonacci* beziehen sich auf diese Funktion, nicht auf die memoisierte Version namens *memo-fibonacci*.

Was ist zu tun? Wertedefinition dürfen nicht rekursiv sein; die Wertedefinition in eine Funktionsdefinition zu überführen, ist verführerisch, aber nicht sinnvoll. Dann wird für jeden rekursiven Aufruf eine *neue* Memotabelle angelegt. (Nachdenken!) Wir können die rekursiven Aufrufe memoisieren, indem wir *memo* nicht eine Funktion des Typs $Nat \to 'a$ übergeben, sondern eine Funktion des Typs $(Nat \to 'a) \to (Nat \to 'a)$, die über die rekursiven Aufrufe abstrahiert.

```
let rec-memo (dom : Nat,
              functional : (Nat → 'value) → (Nat → 'value)) : Nat → 'value =
  let memo-table = [| for i in 0 .. dom − 1 → ref None |]
  let rec memo-f (n : Nat) : 'value =
    match !memo-table.[n] with
    | None   → let v = functional memo-f n
               memo-table.[n] := Some v
               v
    | Some v → v
  in memo-f
let memo-fibonacci =
  rec-memo (100, fun fib → fun n →
                   if n ⩽ 1 then n
                            else fib (n − 1) + fib (n − 2))
```

Aus der rekursiven Funktion, zweites Argument von *memo*, ist eine nicht-rekursive Funktion geworden, zweites Argument von *rec-memo*, die die rekursiven Aufrufe zum Parameter macht. Dieser Parameter wird in der Definition von *rec-memo* mit der memoisierten Funktion belegt: *functional memo-f*. (Betrachten wir die Definition durch die semantische Brille, so bestimmt *rec-memo* den Fixpunkt des zweiten Arguments bei gleichzeitiger Memoisierung, siehe auch Abschnitt 6.4.2.) Der Lohn der Anstrengungen: Sowohl die Deklaration von *memo-fibonacci* als auch alle Aufrufe von *memo-fibonacci* sind in Windeseile ausgerechnet; auch die rekursiven Aufrufe füllen die Tabelle.

7.2.5. Über den Tellerrand

L- und R-Werte Zum Abschluss wollen wir noch einen Blick über den Tellerrand und auf andere Programmiersprachen werfen. In den historisch ersten Programmiersprachen und auch in den meisten aktuellen Sprachen spielen Speicherzellen eine weitaus größere Rolle als in Mini–F#. Es wird gerechnet, indem die Inhalte von Speicherzellen schrittweise verändert werden. Schauen wir uns die Unterschiede anhand zweier konkreter Beispiele an: der Programmiersprache *Pascal* und der Programmiersprache *C*. Ein Bezeichner in diesen Sprachen oder in gängiger Terminologie eine *Variable* bezeichnet stets eine Speicherzelle. In Pascal führt die Deklaration `var i:integer` eine Speicherzelle ein, die eine ganze Zahl enthält; die C Deklaration `int i` unterscheidet sich syntaktisch etwas, bedeutet aber das Gleiche. Die korrespondierende Mini–F# Deklaration lautet *let* i = *ref* 0. Dereferenzierung ist in Pascal und C implizit: Statt wie in Mini–F# $i := !i + 4711$ programmiert man in Pascal `i := i + 4711` und in C `i = i + 4711`

module
Effects.
CallByRef

oder etwas kürzer `i += 4711`. Auf der rechten Seite der Zuweisung werden Speicherzellen automatisch dereferenziert — machmal spricht man auch vom *L-Wert* und *R-Wert* einer Variablen; links ist die Adresse gemeint und rechts der Inhalt der Speicherzelle.

Automatische Dereferenzierung erscheint auf den ersten Blick bequem, hat aber wie alle Automatismen seine Nachteile: Was passiert, wenn auf der rechten Seite tatsächlich die Adresse gemeint ist und nicht der Inhalt? Schauen wir uns ein Beispiel an. Die folgende Version der Fakultät kommuniziert das Funktionsergebnis nicht über den Rückgabewert, sondern über das Argument! Das hört sich paradox an; werfen wir also einen Blick auf die Definition (in Mini–F#), um das Rätsel zu lösen.

```
let rec factorial (n : Nat, result : Ref⟨Nat⟩) =
  if n = 0 then
    result := 1
  else
    factorial (n − 1, result)
    result := !result ∗ n
```

Das zweite Argument ist vom Typ $Ref\langle Nat\rangle$. Der Funktion wird die Adresse übergeben, unter der sie das Ergebnis ablegen soll — ähnlich einem Postfach. Diese Technik (engl. destination passing style) wird in Pascal und C oft verwendet, um Funktionen mit mehreren Rückgabewerten zu simulieren — Funktionen dürfen in diesen Sprachen keine aggregierten Daten wie etwa ein Paar zurückgeben. Um diese Version der Fakultät zu verwenden, muss man ein Postfach anlegen und später dort nachschauen:

```
Mini⟩ let post-office-box = ref 10
val post-office-box : Ref⟨Nat⟩
Mini⟩ factorial ( !post-office-box, post-office-box)
()
Mini⟩ !post-office-box
3628800
```

Der zweite Ausdruck verwendet sowohl den L-Wert (*post-office-box*) als auch den R-Wert (!*post-office-box*) einer Variablen. Sprachen, die automatisch dereferenzieren, sehen sich an dieser Stelle mit einem Problem konfrontiert. In Pascal wird das Problem gelöst, indem bei der Definition der Funktion der zweite Parameter besonders gekennzeichnet wird: `procedure factorial (n : integer, var result : integer)`. Man spricht von einem *Variablen-* oder *Referenzparameter* (engl. *call by reference*). Die Deklaration bewirkt, dass der korrespondierende Aufruf in Pascal, `factorial (pob, pob)`, korrekt behandelt wird: Das erste Vorkommen von `pob` (kurz für post office box) wird dereferenziert, das zweite nicht.

In C wird ein Operator bereitgestellt, der sogenannte Adressoperator `&`, um die automatische Dereferenzierung zurückzunehmen: `factorial (pob, &pob)`. In der Deklaration der Funktion muss zusätzlich vermerkt werden, dass der zweite Parameter eine Adresse ist: `int factorial (int n, int* result)`.

Fassen wir zusammen: Etwas vereinfachend kann man sagen, dass in Pascal L- und R-Werte syntaktisch durch die Grammatik unterschieden werden, in C hingegen ähnlich

wie in Mini–F# anhand des Typs. Eine syntaktische Unterscheidung ist notgedrungen restriktiver als eine typmäßige Unterscheidung — Typregeln sind ausdrucksstärker als Grammatiken. Der konservative Ansatz in Pascal hat aber seinen Grund, der mit der *Lebensdauer* (engl. extent) von Variablen zusammenhängt. Speicherzellen in Mini–F# leben prinzipiell ewig (der Speicher σ wird stets größer, nie kleiner); in Pascal und C endet die Lebensdauer in der Regel mit dem Ende der Sichtbarkeit: Ist eine Variable nicht mehr sichtbar, wird sie deallokiert. Dahinter steht die Vorstellung oder vielmehr die Hoffnung, dass eine Speicherzelle nicht mehr benötigt wird, wenn ihre Adresse nicht mehr sichtbar ist. In Mini–F# gilt dies sicherlich nicht: In der Implementierung des Bankkontos ist die Speicherzelle *funds* nach Abarbeitung der Moduldefinition nicht mehr sichtbar, benötigt wird sie aber sehr wohl.

Die Adresse überlebt das Ende der Sichtbarkeit durch ihre Verwendung in den Funktionen *deposit*, *withdraw* und *balance*. Es lohnt sich, diesen Punkt mit Hilfe eines vereinfachten Beispiels noch einmal genauer zu beleuchten.

```
let inc =
  let counter = ref 0
  fun () : Nat →
    counter := !counter + 1
    !counter
```

Die lokale Definition wertet zu der Umgebung $\{\, counter \mapsto a \,\}$ aus, wobei a die Adresse der angelegten Speicherzelle ist. Bezüglich dieser Umgebung wird der Funktionsausdruck ausgewertet; das Ergebnis ist ein Funktionsabschluss. Die globale Definition wertet somit zu der Umgebung $\{\, inc \mapsto \langle \{\, counter \mapsto a \,\}, (), counter := !counter + 1;\ !counter \rangle \,\}$ aus. So entweicht die Adresse a aus ihrem Sichtbarkeitsbereich: a ist nicht mehr direkt via *counter* verfügbar, aber indirekt via *inc*.

Die Einschränkungen von Pascal garantieren, dass eine solche Situation nicht auftreten kann — das geht natürlich mit einem Verlust an Ausdruckskraft einher. In C ist die Situation verzwickter: Das obige Beispiel lässt sich wegen ähnlicher Einschränkungen nicht nachprogrammieren; eine Adresse kann also nicht auf diesem Wege entweichen. Aber C hat den Adressoperator im Repertoire: Dieser erlaubt es, die Adresse einer Variablen zu bestimmen. Merkt man sich diese Adresse, so kann es passieren, dass man später auf eine Speicherzelle zugreift, die längst nicht mehr existiert — mit unabsehbaren Folgen. (Die ungültige Adresse nennt man auch *dangling pointer*.)

Die obige Diskussion wirft mindestens zwei weitere Fragen auf: Warum werden in Pascal und C Variablen am Ende ihrer Sichtbarkeit deallokiert. Wann und wie werden in Mini–F# Speicherzellen deallokiert? Die zweite Frage ist schnell beantwortet: In Mini–F# werden Speicherzellen deallokiert, wenn sie tatsächlich nicht mehr benötigt werden und wenn der Speicherplatz knapp wird. Diese Aufgabe übernimmt eine sogenannter *Garbage collector* (engl. für Müllmann), ein wichtiger Bestandteil des Mini–F# Interpreters. Ob ein Speicherplatz nicht mehr benötigt wird, ist eine globale Eigenschaft. Aus diesem Grund sollte sich jemand um die Müllentsorgung kümmern, der den Überblick hat — und das ist nicht notwendigerweise die Programmiererin oder der Programmierer.

Pascal und C verfügen über keinen Garbage collector. Sie verwenden daher eine einfachere, aber durchaus bewährte Form der Speicherorganisation: Variablen am Anfang der Sichtbarkeit allokieren, am Ende deallokieren (Stichwort: stackartige Speicherorganisation).

module
Effects.
ImperativeList

Listen, da capo Da in Pascal und C das Konzept des Speichers dominierend ist, gehen diese Sprachen auch die Implementierung von Datenstrukturen wie Listen oder Bäumen anders an. Um die Unterschiede zu diskutieren, programmieren wir eine typische Listenimplementierung in Mini–F# nach. Der grundlegende Unterschied zu der Definition von Listen aus Abschnitt 4.2.2 ist, dass eine Liste selbst, sowie sämtliche Restlisten Speicherzellen sind.

```
type List ⟨'elem⟩  = Ref ⟨Item ⟨'elem⟩⟩
and  Item ⟨'elem⟩ =
        | Nil
        | Cons of 'elem * List ⟨'elem⟩
```

Die Typen *List* und *Item* sind verschränkt rekursiv definiert: *List* verwendet *Item* und *Item* verwendet *List*. Analog zu verschränkt rekursive Funktionen müssen auch verschränkt rekursive Typen mit ***and*** verknüpft werden. Entfernen wir den Typkonstruktor *Ref*, so erhalten wir unsere ursprüngliche Definition von Listen. Ein „Item“ ist entweder leer oder besteht aus einem Listenelement und der Adresse des nächsten Items.

Wenn wir „smarte“ Konstruktoren definieren,

```
let nil () = ref Nil
let cons (x, xs) = ref (Cons (x, xs))
```

dann ändert sich die Konstruktion von Listen im Vergleich zur Listendefinition aus Abschnitt 4.2.2 nur marginal. (Abbildung 7.2 erklärt, warum *nil* eine Funktion sein muss.)

```
Mini⟩  let primes = cons (2, cons (3, cons (5, nil ())))
val primes : Ref ⟨Item ⟨Nat⟩⟩
Mini⟩  primes
{ contents = Cons (2, { contents = Cons (3, { contents = Cons (5,
    { contents = Nil }) }) }) }
```

Lediglich die Ausgabe zeigt an, dass Speicherzellen involviert sind.[4]

Auch die Definition von „beobachtenden“ Funktionen wie der Listenlänge ändert sich nur unwesentlich. Vor jeder Fallunterscheidung muss zunächst die Adresse dereferenziert werden:

[4] Die Ausgabe verrät, wie der Mini–F# Interpreter Speicherzellen implementiert: als Records mit genau einer Komponente, dem Inhalt (engl. contents) der Speicherzelle, siehe Seite 337.

Speicherzellen und Polymorphie stehen in einem gewissen Spannungsverhältnis. So ist es *nicht* erlaubt, eine polymorphe Speicherzelle zu definieren.

```
let nil = ref Nil          // nicht zulässig (value restriction), nil : List ⟨'elem⟩
```

Wäre die Definition zulässig, hätte *nil* den Typ $List\,\langle'elem\rangle$ und wir könnten mit ihrer Hilfe Listen verschiedensten Typs konstruieren.

```
let bools = ref (Cons (false, ref (Cons (true, nil))))
let nats  = ref (Cons (4711, ref (Cons (815, nil))))
```

Vielleicht sieht man schon das Unheil nahen. Da *nil* eine veränderbare Speicherzelle ist, können wir ihren Inhalt modifizieren. Da *nil* einen polymorphen Typ vorgibt, können wir mit dem Inhalt auch den Typ abändern.

```
nil := Cons ("oh je", ref Nil)
```

Aus der leeren Liste ist eine einelementige Liste geworden, als Nebeneffekt sind sowohl *bools* als auch *nats* um ebendieses Element verlängert worden. Die Listen sind nicht länger homogen! Aus diesem Grund ist die Definition polymorpher Werte verboten; versucht man es trotzdem, meldet die Typprüfung eine Verletzung der *Wertebeschränkung* (engl. *value restriction*). Keine Regel ohne Ausnahme: polymorphe Funktionen und polymorphe Datenkonstruktoren sind erlaubt.

```
let nil () = ref Nil       // zulässig, nil : Unit → Ref ⟨Item ⟨'elem⟩⟩
let nil = Nil              // zulässig, nil : Item ⟨'elem⟩
```

Abbildung 7.2.: Wertebeschränkung (engl. value restriction).

```
let length (list : List ⟨'elem⟩) =
  let rec worker n p =
    match !p with
      | Nil          → n
      | Cons (x, xs) → worker (n + 1) xs
  in worker 0 list
```

(Aus Gründen der Effizienz verwendet *length* eine „endrekursive“ Arbeiterfunktion mit einem akkumulierenden Parameter; die gleiche Optimierung ist auch auf unsere Listen aus Abschnitt 4.2.2 anwendbar und tatsächlich sinnvoll. Wir wenden uns diesem Thema ausführlich in Abschnitt 7.3.3 zu.)

Anders verhält es sich mit „transformierenden“ Funktionen: Listen werden manipuliert, indem die Inhalte der adressierten Speicherzellen verändert werden. Die Definition der Listenkonkatenation illustriert die Vorgehensweise.

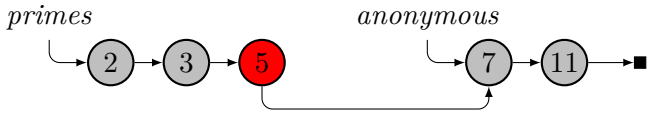
```
let rec append (list₁ : List ⟨'elem⟩, list₂ : List ⟨'elem⟩) : Unit =
  match !list₁ with
    | Nil          → list₁ := !list₂
    | Cons (x, xs) → append (xs, list₂)
```

Ist die erste Liste leer, so wird ihr der Inhalt der zweiten Liste zugewiesen. Anderenfalls wird die zweite Liste an die Restliste angehängt. Vergleichen wir diese Definition mit der Definition aus Abschnitt 4.2.2, so ist auffällig, dass *append* keine neue Liste konstruiert; *append* arbeitet „in situ“ (lat. am Platz) mit den bestehenden Strukturen.

Probieren wir *append* aus:

```
Mini⟩ append (primes, cons (7, cons (11, nil ())))
()
Mini⟩ primes
{contents = Cons (2, {contents = Cons (3, {contents = Cons (5,
    {contents = Cons (7, {contents = Cons (11, {contents = Nil})})})})})}
```

Wir sehen: *append* hängt das zweite Argument an das erste an und modifiziert dieses dabei. Die ursprüngliche Liste hat sich nach diesem Aufruf verflüchtigt. Aus diesem Grund spricht man auch von einer *ephemeren Datenstruktur* (griech. nur einen Tag dauernd, vorübergehend). Die folgende Grafik visualiert das resultierende Speichergeflecht (der rote Knoten ist überschrieben worden).



Vergleichen wir das Verhalten mit der Listenimplementierung aus Abschnitt 4.2.2:

```
Mini⟩  let primes = Cons (2, Cons (3, Cons (5, Nil)))
val primes : List ⟨Nat⟩
Mini⟩  primes
Cons (2, Cons (3, Cons (5, Nil)))
Mini⟩  append (primes, Cons (7, Cons (11, Nil)))
Cons (2, Cons (3, Cons (5, Cons (7, Cons (11, Nil)))))
Mini⟩  primes
Cons (2, Cons (3, Cons (5, Nil)))
Mini⟩  append (primes, primes)
Cons (2, Cons (3, Cons (5, Cons (2, Cons (3, Cons (5, Nil))))))
```

Diese Version ist nicht destruktiv: *append* $(list_1, list_2)$ konkateniert die Listen $list_1$ und $list_2$; die Argumentlisten sind nach dem Aufruf unverändert, sie überdauern den Aufruf (klar, das Konzept des Speichers haben wir ja erst in diesem Abschnitt eingeführt). Aus diesem Grund spricht man auch von einer *persistenten Datenstruktur* (lat. anhaltend, beharrlich).[5] Persistente Datenstrukturen sind in der Regel ephemeren Datenstrukturen vorzuziehen. Da sie, zumindest nach außen, effektfrei sind, sind sie leichter zu verstehen, zu verwenden und darüber hinaus auch flexibler.

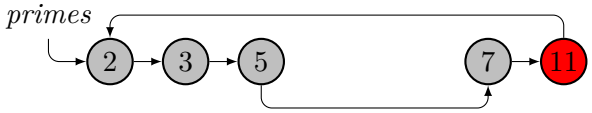
Die Gefahr von Effekten illustriert die folgende interaktive Sitzung.

```
Mini⟩  append (primes, primes)
()
Mini⟩  primes
{ contents = Cons (2, { contents = Cons (3, { contents = Cons (5, { contents =
    Cons (7, { contents = Cons (11, { contents = ...})})})})})}
```

Wir hängen die Liste der ersten fünf Primzahlen an die Liste der ersten fünf Primzahlen und erhalten — ja, was eigentlich? Um die Ausgabe überschaubar zu halten, gibt der Interpreter nur die ersten Elemente aus. Dass die Liste tatsächlich länger ist, wird durch die Ellipse '...' angedeutet. Aber wie lang?

```
Mini⟩  length primes
...
```

Dem Aufruf von *length primes* folgt Stille, die Auswertung terminiert nicht! Der Aufruf *append* $(primes, primes)$ hat nicht etwa eine 10-elementige Liste konstruiert, sondern eine *zyklische Liste*: Das ursprüngliche Ende der Liste verweist nunmehr auf den Anfang der Liste selbst.



[5] Diese Verwendung des Begriffes Persistenz sollte nicht mit der für Daten auf einem Hintergrundspeicher verwechselt werden, siehe vorheriger Abschnitt. Hier ist gemeint, dass die für die Datenstruktur bereitgestellten Funktionen ihre Argumente nicht verändern.

Das Beispiel zeigt, dass nicht alle Effekte erwartet oder gar erwünscht sind. Und noch etwas wird deutlich: Das Struktur Entwurfsmuster garantiert nicht länger, dass die nach dem Muster gestrickten Funktionen auch terminieren! Wir können — gewollt oder ungewollt — zyklische Speichergeflechte konstruieren, so dass die listenverarbeitenden Funktionen keinem Basisfall mehr zustreben.

```
Mini⟩  let nihil : List ⟨Nat⟩ = nil ()
val nihil : Ref⟨Item ⟨Nat⟩⟩
Mini⟩  let cycle = cons (0, nihil)
val cycle : Ref⟨Item ⟨Nat⟩⟩
Mini⟩  nihil := !cycle
()
Mini⟩  append (cycle, cons (1, nil ()))
...
```

Natürlich kann man solche Speichergeflechte in einem Programm auch bewusst einsetzen. In diesem Fall muss man die Terminierung mit anderen Mitteln gewährleisten, etwa indem man sich merkt, welche Speicherzellen schon verarbeitet wurden. Auf zyklische Geflechte kommen wir am Ende des Abschnitts noch einmal zu sprechen.

Speicherzellen verändern also die Natur des Rechnens — es gibt nicht länger eine Terminierungsgarantie für strukturell rekursive Funktionen. Aber es kommt noch schlimmer: Selbst die Terminierung von nicht-rekursiven Funktionen ist nicht mehr gewährleistet! Die folgenden beiden Wertebindungen implementieren die Fakultät (ersetzen wir die gesamte Alternative durch den ***else***-Zweig, so erhalten wir eine nicht-terminierende Funktion).

```
let fac = ref (fun (n : Nat) → 0)
let factorial =
  fac := (fun (n : Nat) → if n = 0 then 1
                          else  n ∗ ( !fac) (n − 1))
  !fac
```

Die erste Wertebindung allokiert eine Speicherzelle, die eine Funktion enthält: *fac* hat den Typ $Ref\langle Nat \to Nat\rangle$. Die zweite Wertebindung weist dieser Speicherzelle eine Funktion zu, die die in dieser Speicherzelle enthaltene Funktion aufruft, also dank der Zuweisung sich selbst: ein zyklisches Speichergeflecht, das Rekursion simuliert. Damit sollen nicht etwa „funktionale“ Speicherzellen diffamiert werden — ähnlich wie Funktionen höherer Ordnung sind auch Speicherzellen, die Funktionen enthalten, nützlich (Stichwort: hooks).

Variablen⟍Veränderliche Neben Speicherzellen bietet F# noch ein weiteres Konstrukt an, um Programme mit einem Gedächtnis anzustatten: *Veränderliche* bzw. *Variablen*.

```
let mutable funds = 0
```

Mit ***mutable*** wird ein Bezeichner als veränderlich gekennzeichnet: Aus einem Namen für einen Wert wird eine Speicherzelle. Zuweisungen an ***mutable***s werden wie folgt notiert.

$$funds \leftarrow funds + amount$$

Lies: funds wird zu *funds*+*amount* (engl. *funds* becomes *funds*+*amount*). Wie in C oder Pascal wird ein veränderlicher Bezeichner automatisch dereferenziert. Der Bezeichner *funds* hat tatsächlich den Typ *Nat* und kann wie gewohnt in jedem Kontext verwendet werden, in dem eine natürliche Zahl erwartet wird. Zusätzlich darf *funds* auf der linken Seite einer Zuweisung der Form $e_1 \leftarrow e_2$ stehen. Die Vor- und Nachteile der automatischen Dereferenzierung haben wir bereits am Anfang des Abschnitts diskutiert; sie gelten in gleicher Weise für ***mutable***s.

Was wir bisher verschwiegen haben: Auch Arrays sind veränderlich; die Elemente eines Arrays können mittels einer Zuweisung der Form $a.[e_1] \leftarrow e_2$ verändert werden. (Ein Array ist sozusagen eine große Speicherzelle mit nummerierten Fächern.) Mit diesem Wissen lässt sich die Implementierung der Memotabellen verbessern: Der furchteinflößende Typ $Array\,\langle Ref\,\langle Option\,\langle'a\rangle\rangle\rangle$ kann zu $Array\,\langle Option\,\langle'a\rangle\rangle$ vereinfacht werden.

```
let memo (dom : Nat, func : Nat → 'value) : Nat → 'value =
  let
    memo-table = [| for i in 0 .. dom − 1 → None |]
  in
    fun (n : Nat) → match memo-table.[n] with
                    | None   → let v = func n
                               memo-table.[n] ← Some v
                               v
                    | Some v → v
```

Der Code ist fast identisch: Die Allokation von Speicherzellen mit ***ref*** entfällt, ebenso wie die Dereferenzierung mit '!'; die Zuweisung $e_1 := e_2$ wird in $e_1 \leftarrow e_2$ abgeändert.

module Effects.Cell

Nicht nur Bezeichner können als veränderlich gekennzeichnet werden, auch Komponenten von Records. Wie bereits angedeutet, sind Speicherzellen tatsächlich Records mit einer veränderlichen Komponente, dem Inhalt (engl. contents) der Speicherzelle. Der Typ $Ref\,\langle'value\rangle$ und die Operationen auf Speicherzellen sind wie folgt definiert.

```
type Ref⟨'value⟩ =
  {mutable contents : 'value}
let ref value =
  {contents = value}
let (!) cell =
  cell.contents
let (:=) cell value =
  cell.contents ← value
```

Um symbolische Bezeichner wie '!' oder ':=' zu definieren, müssen diese in Klammern eingeschlossen werden. Der Unterschied zwischen einer Speicherzelle, sprich einer Adresse, und ihrem Inhalt wird hier noch einmal sehr deutlich: *cell* ist jeweils die Speicherzelle und *cell.contents* ihr Inhalt. (Harry Hacker würde übrigens die Zuweisung so modifizieren, dass der R-Wert zusätzlich als Ergebnis zurückgegeben wird.

```
let (:=) cell value =
  cell.contents ← value
  value
```

Damit lassen sich Zuweisungen aneinanderreihen, $x := y := 4711$, oder mit anderen Operationen kombinieren $(x := !x + 1) \leqslant 99$. Wenn Sie bereits das Vergnügen hatten in C zu programmieren, wird Ihnen dieses Idiom bekannt vorkommen.)

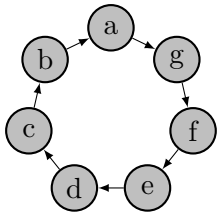
module Effects. Necklace

Perlenketten ⟍ Zyklische Listen

> *The following anecdote is told of William James. […] After a lecture on cosmology and the structure of the solar system, James was accosted by a little old lady.*
>
> *"Your theory that the sun is the centre of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.*
>
> *„And what is that, madam?" inquired James politely.*
>
> *"That we live on a crust of earth which is on the back of a giant turtle."*
>
> *Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.*
>
> *„If your theory is correct, madam,"he asked, "what does this turtle stand on?"*
>
> *"You're a very clever man, Mr. James, and that's a very good question," replied the little old lady, „but I have an answer to it. And it's this: The first turtle stands on the back of a second, far larger, turtle, who stands directly under him."*
>
> *„But what does this second turtle stand on?" persisted James patiently.*
>
> *To this, the little old lady crowed triumphantly,*
>
> *„It's no use, Mr. James — it's turtles all the way down."*
>
> — J. R. Ross, *Constraints on Variables in Syntax*

Besteht ein Record aus mehreren Komponenten, kann man selektiv für jede Komponente entscheiden, ob sie veränderlich sein soll oder eben nicht. Zum Beispiel:

```
type Necklace ⟨'elem⟩ =
  {bead            : 'elem
   mutable next : Necklace ⟨'elem⟩}
```

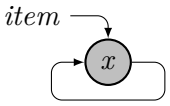


Elemente dieses Typs sind *Perlenketten* (engl. necklaces) oder, etwas prosaischer, nicht-leere zyklisch verkettete Listen, siehe Abbildung zur Rechten. Die nicht veränderliche Komponente enthält eine Perle (engl. bead), ein Listenelement; die veränderliche Komponente verweist auf das nächste Kettenglied (engl. next). Der Typ ist bemerkenswert, da er zum Ersten rekursiv definiert ist, aber zum Zweiten keinen Basisfall anbietet. Eine Kette besteht aus einer Perle und einer Kette, die wiederum aus einer Perle besteht und einer Kette, die … — it's turtles all the way

down. Der einzige Ausweg aus der Zwickmühle: Eine Kette muss notwendigerweise ein zyklisches, wenn auch nicht unbedingt ein zirkuläres Geflecht sein. (Über diese Geflechte sind wir schon gestolpert, als wir uns mit ephemeralen Listen beschäftigt haben. Hier setzen wir sie bewusst ein — a bug turned into a feature.)

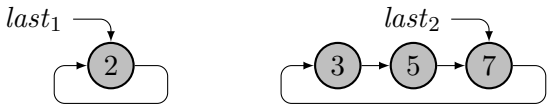
Wie konstruieren wir ein zyklisches Geflecht? Die Antwort ist so zwingend, wie vielleicht überraschend: durch eine rekursive *Werte*definition.

```
let single x =
  let rec item =
    {bead = x
     next = item}
  in item
```



Bis dato haben wir nur rekursive *Funktions*definitionen kennengelernt (es sei denn, Sie haben Abschnitt 4.5.5 gelesen). Aus gutem Grund. Rekursive Wertedefinition machen im Allgemeinen keinen Sinn: *let rec* $n = n+1$ ist zum Beispiel nicht zulässig, da der Bezeichner n als Teil seiner eigenen Definition ausgewertet würde. (Auch als mathematische Gleichung hat $n = n + 1$ keine Lösung.) Funktionsdefinitionen sind hingegen unproblematisch, da sie unmittelbar zu Funktionsabschlüssen auswerten, im Fall einer rekursiven Definition zu einem zyklischen Geflecht, siehe Abschnitt 3.6. Die obige rekursive Wertedefinition ist ebenfalls unproblematisch, wenn auch ihre Auswertung etwas trickreich vonstatten geht. Hinter den Kulissen wird item in zwei Schritten konstruiert: Zunächst wird *let* $item = \{bead = x; next = \bot\}$ angelegt, wobei $\bot$ ein beliebiger Dummywert ist; dann wird mittels $item.next \leftarrow item$ der Dummywert durch den zyklischen Verweis ersetzt. Wiederum wird aus einer rekursiven Definition ein zyklisches Geflecht.

Wir können Perlenketten verwenden, um nicht-leere, *endliche* Folgen von Elementen darzustellen — wir erhalten eine weitere Variation von ephemeralen Listen. Die grundlegende Idee ist, eine Folge von Elementen durch eine Kette mit entsprechenden Perlen zu repräsentieren und dabei auf das *letzte* Element der zu repräsentierenden Folge zu verweisen. Die einelementige Folge 2 und die dreielementige Folge 3 5 7 werden zum Beispiel durch



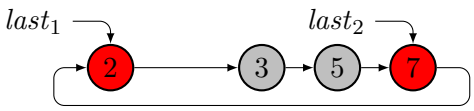
repräsentiert. (Wenn man möchte, kann man $last_1$ bzw. $last_2$ als Verschluss der jeweiligen Perlenkette ansehen.) Die Darstellung macht es *sehr* einfach, Folgen zu rotieren: $last_2.next$ repräsentiert 5 7 3, $last_2.next.next$ repräsentiert 7 3 5 und $last_2.next.next.next$ ergibt die ursprüngliche Folge — der Ausdruck ist identisch zu $last_2$.

Das Kopfelement der Folge *last* ist $last.next.bead$; die Restfolge erhalten wird, indem wir das erste Element überspringen: $last.next \leftarrow last.next.next$.

```
let head (last : Necklace ⟨'elem⟩) : 'elem =
  last.next.bead
let tail (last : Necklace ⟨'elem⟩) : Necklace ⟨'elem⟩ =
  last.next ← last.next.next
  last
```

Perlenketten sind ephemeral: Die Funktion *tail* verändert ihr Argument, das lediglich aus Gründen der Bequemlichkeit zusätzlich als Funktionsergebnis zurückgegeben wird (so dass wir zum Beispiel *tail* (*tail xs*) schreiben können).

Wir haben noch nicht verraten, warum wir ausgerechnet auf das letzte und nicht etwa auf das erste Element verweisen. Diese Darstellung erlaubt es, zwei Folgen in *konstanter* Zeit zu konkatenieren. Wir müssen lediglich $last_1.next$ und $last_2.next$ vertauschen (die roten Knoten sind überschrieben worden).



Die Funktion *append* implementiert die Vertauschung; das Ergebnis der Konkatenation ist die zweite Folge. (Nachdenken!)

```
let append (last1 : Necklace ⟨'elem⟩, last2 : Necklace ⟨'elem⟩) : Necklace ⟨'elem⟩ =
  // swap last1.next and last2.next
  let tmp = last1.next
  last1.next ← last2.next
  last2.next ← tmp
  last2
let cons (x, xs) = append (single x, xs)
```

Auch *append* verändert ihre Argumente: repräsentiert $last_i$ die Folge xs_i, dann repräsentiert $last_2$ *nach* dem Aufruf von *append* die Folge $xs_1 \mathbin{@} xs_2$, während $last_1$ die Folge $xs_2 \mathbin{@} xs_1$ darstellt — eine perfekte Symmetrie. Tatsächlich implementiert *append* so etwas wie die „symmetrische Konkatenation“ zweier Folgen. Hier sind die verschiedenen Operationen in Aktion:

```
Mini⟩  let mutable primes = single 2
Mini⟩  primes
{ bead = 2; next = ...}
Mini⟩  primes ← append (primes, cons (3, cons (5, single 7)))
Mini⟩  primes
{ bead = 7; next = { bead = 2; next = { bead = 3; next = { bead = 5; next = ...}}}}
Mini⟩  tail primes
{ bead = 7; next = { bead = 3; next = { bead = 5; next = ...}}}
Mini⟩  tail primes
{ bead = 7; next = { bead = 5; next = ...}}
Mini⟩  tail primes
{ bead = 7; next = ...}}
Mini⟩  tail primes
{ bead = 7; next = ...}}
```

Man sieht jeweils sehr schön, dass auf das letzte Element der Folge verwiesen wird. Die letzten Aufrufe zeigen, dass sich eine einelementige Folge nicht weiter verkleinern lässt: *tail* (*single* x) ist und bleibt *single* x.

7.3. Kontrollstrukturen

Mit der Einführung von Operationen zur Ein- und Ausgabe und zur Speichermanipulation wird es notwendig, Effekte präzise zu kontrollieren: Welche Effekte treten auf, in welcher Reihenfolge treten sie auf und wie oft werden sie gegebenenfalls wiederholt. Mit dem Einzug von Effekten verändert sich nicht nur die Natur des Rechnens, auch der Programmierstil wird potentiell ein anderer. Der *problemnahe*, *deskriptive* Charakter — Was soll berechnet werden? — weicht einem *maschinennahen*, *präskriptiven* Stil — Wie genau wird die Rechnung durchgeführt? Das „Wie“, der Kontrollfluss, kann mit speziellen Sprachmitteln, den Kontrollstrukturen, präzisiert werden und ist wesentlicher Inhalt dieses Abschnitts. Im Einzelnen haben wir Folgendes vor:

In Abschnitt 7.3.1 führen wir Sprachmittel ein, um auf einfache Art und Weise Listen und Arrays zu bilden. Ironischerweise sind die Konstrukte deskripter Natur — sie *beschreiben* Listen und Arrays, anstatt *vorzuschreiben*, wie diese im Detail zu konstruieren sind. Abschnitt 7.3.1 bereitet den Boden für Abschnitt 7.3.2, in dem Kontrollstrukturen, inbesondere Konstrukte zur *Iteration*, diskutiert werden: ***for***- und ***while***-Schleifen. Abschnitt 7.3.3 beleuchtet das Verhältnis zwischen Rekursion und Iteration. Daran anknüpfend zeigt Abschnitt 7.3.4, wie der Kontrollfluss mit den bisherigen Bordmitteln gesteuert werden kann.

7.3.1. Listen- und Arraybeschreibungen

In Abschnitt 4.4 haben wir zwei Möglichkeiten kennengelernt, Arrays zu konstruieren: explizit durch Aufzählung der Elemente bzw. implizit durch Angabe einer Bildungsvorschrift. So konstruiert $[\,|\,2;3;5;7;11\,|\,]$ das Array der ersten fünf Primzahlen und

[|*for* i *in* $0\,..\,99 \to i * i$|]] das Array der ersten hundert Quadratzahlen. Beide linguistischen Konstrukte sind auch für Listen verfügbar: $[2;3;5;7;11]$ ist entsprechend die Liste der ersten fünf Primzahlen und [*for* i *in* $0\,..\,99 \to i * i$] die Liste der ersten hundert Quadratzahlen. Die Konstruktion mittels Angabe einer Bildungsvorschrift ist tatsächlich sehr viel allgemeiner als bisher vorgestellt: Generatoren dürfen aneinandergereiht und geschachtelt werden; zusätzlich können Filter oder Alternativen eingebaut werden. Die folgenden Beispiele illustrieren einige der Möglichkeiten:

```
Mini⟩ [yield 4711; for i in 0 .. 5 do yield i * i]
[4711; 0; 1; 4; 9; 16; 25]
Mini⟩ [for i in 0 .. 9 do if i % 2 = 1 then yield i * i]
[1; 9; 25; 49; 81]
```

Zunächst einmal ist [*for* i *in* $e_1 \to e_2$] eine Abkürzung für [*for* i *in* e_1 *do yield* e_2]. Der Ausdruck *yield* e erzeugt bzw. generiert ein einzelnes Listenelement. Die Listenbeschreibung [*yield* 4711; *for* i *in* $0\,..\,5$ *do yield* $i * i$] generiert somit zunächst 4711 und dann für alle i im Intervall $(0, 5)$ den Wert $i * i$. Mit einer einarmigen Alternative können Elemente herausgefiltert werden: *if* $i \,\%\, 2 = 1$ lässt nur ungerade Werte für i zu.

Wenn die Beschreibungen komplizierter werden, deutet man die hierarchische Struktur am besten durch Einrückung an. Das folgende Beispiel generiert die Positionen aller schwarzen Felder auf einem 8×8-Schachbrett.

```
Mini⟩ [for i in 1 .. 8 do
         for j in 1 .. 8 do
           if (i + j) % 2 = 0 then
             yield (i, j)]
[(1, 1); (1, 3); (1, 5); (1, 7); (2, 2); (2, 4); (2, 6); (2, 8); (3, 1); (3, 3); (3, 5);
 (3, 7); (4, 2); (4, 4); (4, 6); (4, 8); (5, 1); (5, 3); (5, 5); (5, 7); (6, 2); (6, 4);
 (6, 6); (6, 8); (7, 1); (7, 3); (7, 5); (7, 7); (8, 2); (8, 4); (8, 6); (8, 8)]
```

(Die Verwendung von Layout ist auch hilfreich, um Mehrdeutigkeiten zu vermeiden, die bei einer linearen, einzeiligen Notierung zuweilen auftreten.

```
[yield 815                          [yield 815
 for i in 0 .. 9 do                  for i in 0 .. 9 do
   yield i                             yield i
 yield 4711]                         yield 4711]
```

Links wird das Element 4711 genau einmal generiert; rechts 10-mal. Die lineare Variante [*yield* 815; *for* i *in* $0\,..\,9$ *do yield* i; *yield* 4711] ist hingegen mehrdeutig.)

Mit Hilfe von *for* können wir auch über die Elemente einer Liste bzw. eines Arrays iterieren.

```
Mini⟩ let js = [yield 4711; for i in 0 .. 5 do yield i * i]
Mini⟩ [for j in js do yield 2 * j + 1]
[9423; 1; 3; 9; 19; 33; 51]
```

Der Bezeichner j wird nacheinander an die Elemente der Liste js gebunden. Für jede Bindung wird mit ***yield*** $2{*}j{+}1$ ein Element der resultierenden Liste generiert. Der Generator ***for*** i ***in*** $l\,..\,u$ ist übrigens eine Abkürzung für ***for*** i ***in*** $[l\,..\,u]$. Listenbeschreibungen eignen sich insbesondere, um aus bestehenden Listen neue Listen zu generieren. „Aus Alt mach Neu."

Abstrakte Syntax Wir erweitern Ausdrücke um Listen- bzw. Arraybeschreibungen.

$e \in \mathsf{Expr} ::=$	*Listen- und Arraybeschreibungen:*
$\mid\ [\ se\ ]$	Listenbeschreibung
$\mid\ [\mid se \mid]$	Arraybeschreibung

Innerhalb der Klammern steht ein sogenannter *Sequenzausdruck*, eine neue syntaktische Kategorie, deren Elemente Sequenzen von Werten bezeichnen. Wir verwenden hier Sequenz als semantischen Oberbegriff für Liste und Array.

$se \in \mathsf{SExpr} ::=$	*Sequenzausdrücke:*
$\mid$ ***yield*** e	Element⟍einelementige Sequenz
$\mid$ ***yield!*** e	mehrere Elemente⟍Sequenz
$\mid\ d$ ***in*** se	lokale Definition
$\mid\ se_1; se_2$	Konkatenation
$\mid$ ***if*** e ***then*** se	Filter
$\mid$ ***if*** e ***then*** se_1 ***else*** se_2	Alternative
$\mid$ ***for*** x ***in*** e ***do*** se	Generator⟍beschränkte Iteration

Der Sequenzausdruck ***yield*** e steht für eine einelementige Sequenz, dessen einziges Element e ist; ***yield!*** e ist eine Abkürzung für ***for*** x ***in*** e ***do yield*** x und überführt die Liste e in eine Sequenz. Diese beiden Sequenzausdrücke sind übrigens *keine* Ausdrücke; sie haben keinerlei Bedeutung außerhalb der $[\ldots]$ bzw. $[\![...]\!]$ Klammern. Lokale Definitionen, Konkatenation, Filter und Alternativen sind hingegen Zwitterwesen: Sie dienen sowohl als Ausdrücke als auch als Sequenzausdrücke mit (fast) identischer Bedeutung.

Der *Generator* heißt auch ***for****-Schleife*, der Teilausdruck se *Rumpf* der Schleife. Schleife, da die Auswertung des Rumpfs se für alle Elemente von e wiederholt wird. Der Bezeichner x, die sogenannte *Schleifenvariable*, wird dabei neu eingeführt; seine bzw. ihre Sichtbarkeit erstreckt sich auf den Rumpf der Schleife. (Im nächsten Abschnitt werden wir sehen, dass eine ***for***-Schleife ebenfalls als Ausdruck verwendet werden kann.)

Listen- und Arraybeschreibungen unterscheiden sich nur im Hinblick auf den resultierenden Typ — einmal interpretieren wir eine Sequenz als Liste, einmal als Array. Aus diesem Grund beschränken wir uns im Folgenden auf die präzise Definition von Listenbeschreibungen.

Statische Semantik Die eckigen Klammern überführen eine Sequenz in eine Liste.

$$\frac{\Sigma \vdash^* se : t}{\Sigma \vdash [\ se\ ] : \mathit{List}\ \langle t \rangle}$$

In der Voraussetzung der Typregel wird von einer neuen Relation Gebrauch gemacht: $\Sigma \vdash^* se{:}t$. *Lies:* „bezüglich der Signatur Σ haben die *Elemente* der Sequenz se den Typ t“.

$$\frac{\Sigma \vdash e : t}{\Sigma \vdash^* \textbf{\textit{yield}}\ e : t} \qquad \frac{\Sigma \vdash e : \mathit{List}\,\langle t\rangle}{\Sigma \vdash^* \textbf{\textit{yield!}}\ e : t}$$

$$\frac{\Sigma \vdash d : \Sigma' \qquad \Sigma, \Sigma' \vdash^* se : t}{\Sigma \vdash^* (d\ \textbf{\textit{in}}\ se) : t} \qquad \frac{\Sigma \vdash^* se_1 : t \qquad \Sigma \vdash^* se_2 : t}{\Sigma \vdash^* (se_1; se_2) : t}$$

$$\frac{\Sigma \vdash e : \mathit{Bool} \qquad \Sigma \vdash^* se : t}{\Sigma \vdash^* \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ se : t}$$

$$\frac{\Sigma \vdash e : \mathit{Bool} \qquad \Sigma \vdash^* se_1 : t \qquad \Sigma \vdash^* se_2 : t}{\Sigma \vdash^* \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ se_1\ \textbf{\textit{else}}\ se_2 : t}$$

$$\frac{\Sigma \vdash e_1 : \mathit{List}\,\langle t_1\rangle \qquad \Sigma, \{x_1 \mapsto t_1\} \vdash^* se : t}{\Sigma \vdash^* \textbf{\textit{for}}\ x_1\ \textbf{\textit{in}}\ e_1\ \textbf{\textit{do}}\ se : t}$$

Die Typregel für die ***for***-Schleife macht noch einmal deutlich, dass die Schleifenvariable neu eingeführt wird und sich ihre Sichtbarkeit auf den Rumpf der Schleife erstreckt.

Dynamische Semantik Listenbeschreibungen sind „syntaktischer Zucker“ im besten Sinne des Wortes: Sie versüßen das Leben beim Programmieren, sind aber nicht lebensnotwendig. Alle Konstrukte lassen sich mit den bisherigen Bordmitteln formulieren: Zum Beispiel ist $[\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,9\ \textbf{\textit{do}}\ \textbf{\textit{yield}}\ i{*}i]$ gleichwertig zu $\mathit{map}\ (\textbf{\textit{fun}}\ i \to i{*}i)\ [0\,..\,9]$; $[\textbf{\textit{if}}\ i\,\%\,2 = 1\ \textbf{\textit{then}}\ \textbf{\textit{yield}}\ i*i]$ kürzt $\textbf{\textit{if}}\ i\,\%\,2 = 1\ \textbf{\textit{then}}\ [i*i]\ \textbf{\textit{else}}\ [\,]$ ab. Die Bedeutung der Konstrukte können wir erklären, indem wir sie entzuckern, in uns bekannte Ausdrücke *übersetzen*: Die Listenbeschreibung $[se]$ wird in den listenwertigen Ausdruck $[\![se]\!]$ übersetzt, wobei $[\![se]\!]$ wie folgt definiert ist.

$$\begin{aligned}
[\![\textbf{\textit{yield}}\ e]\!] &= [e] \\
[\![\textbf{\textit{yield!}}\ e]\!] &= e \\
[\![d\ \textbf{\textit{in}}\ se]\!] &= d\ \textbf{\textit{in}}\ [\![se]\!] \\
[\![se_1; se_2]\!] &= [\![se_1]\!] \mathbin{@} [\![se_2]\!] \\
[\![\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ se]\!] &= \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ [\![se]\!]\ \textbf{\textit{else}}\ [\,] \\
[\![\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ se_1\ \textbf{\textit{else}}\ se_2]\!] &= \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ [\![se_1]\!]\ \textbf{\textit{else}}\ [\![se_2]\!] \\
[\![\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e\ \textbf{\textit{do}}\ se]\!] &= \mathit{collect}\ (\textbf{\textit{fun}}\ x \to [\![se]\!])\ e
\end{aligned}$$

Die einarmige Alternative, ein Filter, ist eine Abkürzung für eine zweiarmige Alternative, deren ***else***-Zweig die leere Liste $[\,]$ zurückgibt. Ein Generator wird mit Hilfe der vordefinierten Funktion $\mathit{collect} : (\mathit{'a} \to \mathit{'b\ list}) \to (\mathit{'a\ list} \to \mathit{'b\ list})$ implementiert: $\mathit{collect}\ f\ \mathit{list}$ wendet die listenwertige Funktion f auf jedes Element von list an und konkateniert die resultierenden Listen (das Rekursionsmuster kennen wir von *sum-by*).

```
let rec collect f = function
    | []      → []
    | x :: xs → f x @ collect f xs
```

Das folgende Beispiel illustriert die Übersetzung und anschließende Auswertung einer Listenbeschreibung.

$$
\begin{aligned}
& [\textit{for}\ i\ \textit{in}\ [0\,..\,9]\ \textit{do if}\ i\,\%\,2 = 1\ \textit{then yield}\ i * i] \\
= \quad & \{\ \text{Übersetzung Listenbeschreibung}\ \} \\
& [\![\textit{for}\ i\ \textit{in}\ [0\,..\,9]\ \textit{do if}\ i\,\%\,2 = 1\ \textit{then yield}\ i * i]\!] \\
= \quad & \{\ \text{Übersetzung Generator}\ \} \\
& \mathit{collect}\ (\textit{fun}\ i \to [\![\ \textit{if}\ i\,\%\,2 = 1\ \textit{then yield}\ i * i]\!])\ [0\,..\,9] \\
= \quad & \{\ \text{Übersetzung Filter}\ \} \\
& \mathit{collect}\ (\textit{fun}\ i \to \textit{if}\ i\,\%\,2 = 1\ \textit{then}\ [\![\textit{yield}\ i * i]\!]\ \textit{else}\ [])\ [0\,..\,9] \\
= \quad & \{\ \text{Übersetzung}\ \textit{yield}\ \} \\
& \mathit{collect}\ (\textit{fun}\ i \to \textit{if}\ i\,\%\,2 = 1\ \textit{then}\ [i * i]\ \textit{else}\ [])\ [0\,..\,9] \\
= \quad & \{\ \text{Definition von}\ [l\,..\,u]\ \} \\
& \mathit{collect}\ (\textit{fun}\ i \to \textit{if}\ i\,\%\,2 = 1\ \textit{then}\ [i * i]\ \textit{else}\ [])\ [0;1;2;3;4;5;6;7;8;9] \\
= \quad & \{\ \text{Definition von}\ \mathit{collect}\ \} \\
& [\,] \mathbin{@} [1] \mathbin{@} [\,] \mathbin{@} [9] \mathbin{@} [\,] \mathbin{@} [25] \mathbin{@} [\,] \mathbin{@} [47] \mathbin{@} [\,] \mathbin{@} [81] \\
= \quad & \{\ \text{Definition von `@'}\ \} \\
& [1;9;25;47;81]
\end{aligned}
$$

Der Funktionsausdruck $\textit{fun}\ i \to \ldots$ überführt eine ungerade Zahl in eine einelementige Liste und eine gerade Zahl in die leere Liste. Die Konkatenation der resultierenden Listen ist das Ergebnis der Listenbeschreibung.

module Effects. Database

Vertiefung: Hauptspeicherdatenbank Das Motto von Listenbeschreibungen „Aus Alt mach Neu“ lässt sich prima mit Datenbanktabellen und Datenbankanfragen illustrieren. Die folgenden selbsterklärenden (?) Typdefinitionen modellieren Studierende, Module und Prüfungsergebnisse.

```
type Name    = String // student name
type GUID    = Int    // student id (Globally Unique Identifier)
type Course  = | ALG | FPR | IPR | OOP
type Mark    = | Insufficient | Sufficient | Good | VeryGood | Excellent
type Student = {name : Name; guid : GUID}
type Result  = {course : Course; guid : GUID; mark : Mark}
```

Eine *Datenbanktabelle* ist eine Liste von Records; eine *Datenbankanfrage* selektiert oder extrahiert Informationen aus einer Tabelle oder kombiniert Daten aus mehreren Tabellen, siehe Abbildung 7.3. Die Abarbeitung der Anfragen folgt dem oben beschriebenen Übersetzungsschema und taugt durchaus für kleine Tabellen mit mehreren hundert Einträgen. Für die Verarbeitung größerer Datenmengen müssen Tabellen geschickter dargestellt und Anfragen entsprechend optimiert werden. Mehr zum diesem Thema erfahren Sie im weiteren Studium in der Vorlesung Informationssysteme.

```
let students : Student list =
  [{ name = "Ralf"; guid = 4711};
   { name = "Lisa"; guid = 815  };...]
let results : Result list =
  [{ course = ALG; guid = 4711; mark = Good      };
   { course = FPR; guid = 815;  mark = Excellent };
   { course = FPR; guid = 4711; mark = Sufficient};...]
```

Wie heißt der Student mit der Matrikelnummer 4711?

```
let query1 : Name =
  List.head [for student in students do
               if student.guid = 4711 then
                 yield student.name]
```

Liste die Ergebnisse der Vorlesung *ALG*.

```
let query2 : Mark list =
  [for result in results do
     if result.course = ALG then
       yield result.mark]
```

Wer hat das beste Ergebnis in der Vorlesung *FPR* erzielt?

```
let query3 : GUID =
  snd (List.max [for result in results do
                   if result.course = FPR then
                     yield (result.mark, result.guid)])
```

Welche Ergebnisse hat der Student namens `"Ralf"` erzielt?

```
let query4 : Mark list =
  [for student in students do
     if student.name = "Ralf" then
       for result in results do
         if student.guid = result.guid then
           yield result.mark]
```

Welche Student*in hat mindestens ein exzellentes Ergebnis in einem ihrer Kurse?

```
let query5 : Name list =
  [for result in results do
     if result.mark = Excellent then
       for student in students do
         if result.guid = student.guid then
           yield student.name]
```

Abbildung 7.3.: Eine Prüfungsdatenbank und verschiedene Datenbankanfragen.

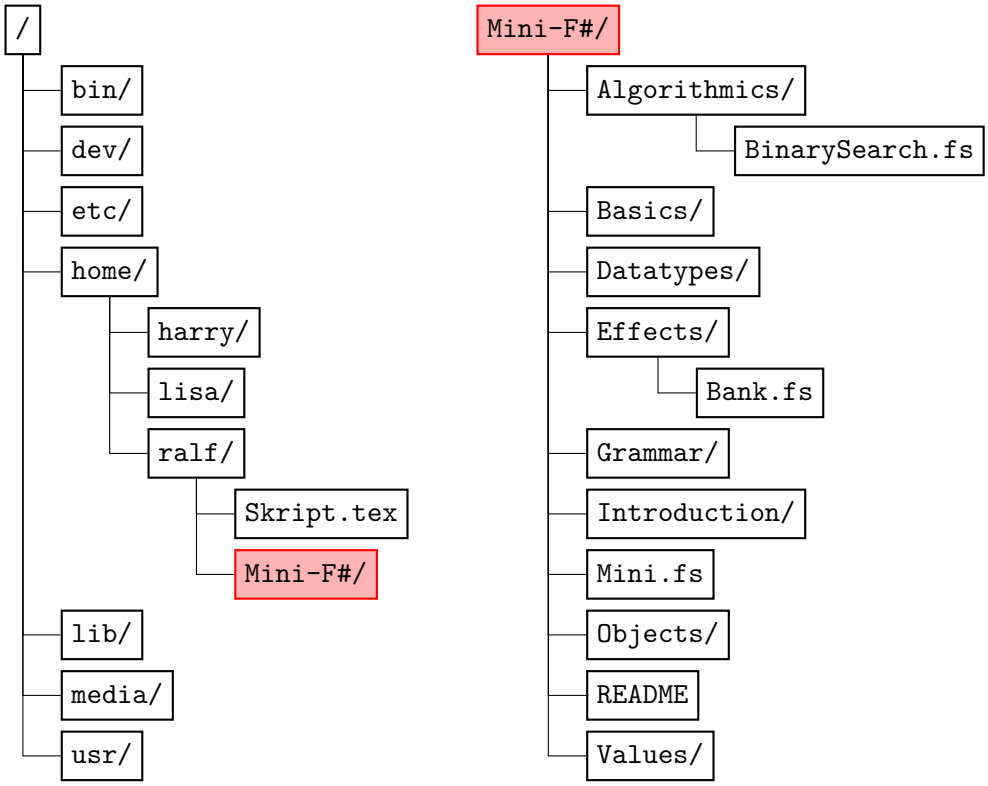


Abbildung 7.4.: Verzeichnisstruktur eines Linux-Dateisystems.

module Effects.Files

Vertiefung: „Dateisystem-Crawler“ Die in Abbildung 7.3 aufgeführten Listenbeschreibungen sind frei von Effekten — passen also thematisch eher ins Kapitel 4. Listenbeschreibungen können aber auch gewinnbringend mit effektvollen Ausdrücken kombiniert werden. Die Standardbibliothek von F# bietet verschiedene Funktionen an, um auf das Dateisystem zuzugreifen. Kurz zum Hintergrund: Das *Dateisystem* (engl. file system) ist Bestandteil des Betriebssystems und organisiert die persistente Speicherung von Daten in sogenannten Dateien (engl. files). Moderne Dateisysteme sind hierarchisch organisiert: Dateien werden in Verzeichnissen (engl. folder) abgelegt; Verzeichnisse dürfen neben Dateien Unterverzeichnisse enthalten, die auf die gleiche Art und Weise organisiert sind. Die Grafik in Abbildung 7.4 zeigt einen *Auszug* aus der Verzeichnisstruktur eines Linux-Dateisystems.

Die F# Bibliothek *System.IO* enthält unter anderem die Funktionen

$$\begin{aligned} &\mathit{Directory.GetFiles} &&: \mathit{String} \to \mathit{Array}\,\langle \mathit{String} \rangle \\ &\mathit{Directory.GetDirectories} &&: \mathit{String} \to \mathit{Array}\,\langle \mathit{String} \rangle \end{aligned}$$

die die in einem Verzeichnis abgelegten Dateien bzw. Unterverzeichnisse ermitteln.

```
Mini⟩ open System.IO
Mini⟩ Directory.GetFiles "."
[|"./Mini.fs";"./README"|]
Mini⟩ Directory.GetDirectories "."
[|"./Algorithmics";"./Basics";"./Datatypes";"./Effects";"./Grammar";
  "./Introduction";"./Objects";"./Values"|]
```

Das Verzeichnis „`.`“ ist eine Abkürzung für das aktuelle Verzeichnis, im obigen Beispiel `/home/ralf/Mini-F#/`. Die folgende Funktion bestimmt alle Dateien, die in *und* unterhalb eines Verzeichnisses abgelegt sind.

```
open System.IO
let rec list-files dir =
  [| yield! Directory.GetFiles dir
     for d in Directory.GetDirectories dir do
       yield! list-files d |]
```

Zunächst werden mit ***yield!*** alle Dateien im aktuellen Verzeichnis aufgezählt; für jedes Unterverzeichnis erfolgt ein rekursiver Aufruf, der entsprechend abgearbeitet wird. Die Rekursion terminiert, wenn ein Verzeichnis keine weiteren Unterverzeichnisse enthält, der Aufruf *Directory.GetDirectories dir* das leere Array zurückgibt. Die Funktion *list-files* in Aktion:

```
Mini⟩ list-files "."
[|"./Mini.fs";"./README";"./Algorithmics/BinarySearch.fs";
  "./Algorithmics/Neighbours.fs";...;"./Datatypes/Person.fs";...;
  "./Effects/Bank.fs";"./Effects/Files.fs";... |]
```

7.3.2. Schleifen

Eine ***for***-Schleife kann auch verwendet werden, um effektvolle Ausdrücke aneinanderzureihen. Die folgenden Beispiele illustrieren einige der Möglichkeiten:

```
Mini⟩ print 4711; for i in 0 .. 5 do print (i * i)
4711
0
1
4
9
16
25
Mini⟩ for i in 0 .. 9 do if i % 2 = 1 then print (i * i)
1
9
25
49
81
Mini⟩ for i in 1 .. 8 do
          for j in 1 .. 8 do
            if (i + j) % 2 = 0 then
              print (i, j)
(1, 1)
(1, 3)
...
(8, 6)
(8, 8)
```

Die Ausdrücke sind fast identisch zu denen aus Abschnitt 7.3.1. Wir haben lediglich ***yield*** durch *print* ersetzt und die Listenklammern entfernt. Anstatt Elemente mit ***yield*** in einer Liste zu sammeln, wird jetzt jedes Element mit *print* ausgegeben. Anstatt Elemente aneinanderzureihen, werden Effekte aneinandergereiht. Die Funktion *print* gibt übrigens einen beliebigen Wert aus: ***let*** *print a* = *putline* (*show a*).

Es ist natürlich nicht zwingend, die Elemente auszugeben. Das folgende Beispiel illustriert einen anderen Effekt.

```
let sum (xs : List ⟨Int⟩) : Int =
  let mutable acc = 0
  for x in xs do
    acc ← acc + x
  acc
```

Die Funktion *sum* addiert die Elemente der angegebenen Liste auf. Wir verwenden eine Veränderliche namens *acc* (kurz für accumulator, engl. für Akkumulator), um uns die jeweilige Zwischensumme zu merken. Für jedes Listenelement wird die Zwischensumme um ebendieses Element erhöht — diese Zuweisung ist der effektvolle Ausdruck, der für jedes Listenelement ausgeführt wird. Ist das Listenende erreicht, wird die Zwischensumme zur Gesamtsumme.

Ganz entsprechend lassen sich Arrays aufaddieren.

```
let sum (a : Array ⟨Int⟩) : Int =
  let mutable acc = 0
  for x in a do
    acc ← acc + x
  acc
```

```
let sum (a : Array ⟨Int⟩) : Int =
  let mutable acc = 0
  for i in 0 .. a.Length − 1 do
    acc ← acc + a.[i]
  acc
```

Wir können entweder direkt über die Arrayelemente selbst iterieren oder über ihre Hausnummern. Bezüglich der Laufzeit ergeben sich keine Unterschiede, da die Subskription, der Zugriff über die Hausnummer, in konstanter Zeit erfolgt.

```
module Effects.Sort
```

Eine ***for***-Schleife wiederholt einen Effekt für *alle* Elemente einer Liste oder eines Arrays. Diese Kontrollstruktur ist nicht für alle Aufgabenstellungen adäquat. Erinnern wir uns an die lineare Suche: Ist ein Element mit der gewünschten Eigenschaft gefunden, wird die Suche unmittelbar beendet. In Abschnitt 3.6 haben wir die lineare Suche *rekursiv* formuliert. Das folgende Programm zeigt, wie man die lineare Suche *iterativ* mit Hilfe einer ***while***-Schleife implementiert.

```
let linear-search (key : 'elem) (a : Array ⟨'elem⟩) : Option ⟨Int⟩ =
  let mutable i = 0
  while i < a.Length && a.[i] < key do
    i ← i + 1
  if i < a.Length && a.[i] = key then Some i
                                 else None
```

Die Funktion verlangt ein sortiertes Feld und bestimmt die kleinste Hausnummer i, so dass $a.[i] = key$. Der Rumpf der ***while***-Schleife wird wiederholt, solange die Schleifenbedingung wahr ist. Also, solange das Ende des Arrays noch nicht erreicht ist, $i < a.Length$, und das aktuelle Arrayelement kleiner ist als der Suchschlüssel, $a.[i] < key$, wird der Index erhöht, $i \leftarrow i+1$. Eine ***while***-Schleife ist *nur* im Zusammenspiel mit Effekten sinnvoll. Die Schleifenbedingung wird wiederholt ausgerechnet; damit sich der Wahrheitswert ändert, muss der Ausdruck von Benutzereingaben oder vom Speicher abhängen. Das ist hier der Fall: Beide Bestandteile der Konjunktion enthalten die Veränderliche i, deren Wert im Rumpf auch tatsächlich verändert wird.

Auch die binäre Suche lässt sich iterativ formulieren — die Funktion *binary-search* aus Abschnitt 3.6 ist ebenso wie die lineare Suche *endrekursiv*; endrekursive Funktionen können relativ einfach in iterative Programme übersetzt werden, siehe Abschnitt 7.3.3. Das folgende Programm löst das gleiche Problem wie *linear-search*, basiert aber auf der Idee der binären Suche, der Intervallschachtelung. (Da die Funktion einen 3-Wege Vergleich verwendet, haben wir sie auf dem Namen *ternary-search* getauft.)

```
let ternary-search (key : 'elem) (a : Array ⟨'elem⟩) : Option ⟨Int⟩ =
  let mutable l = 0
  let mutable u = a.Length − 1
  let mutable found = None
  while l ⩽ u && found = None do
    let m = (l + u) ÷ 2
    if   key < a.[m]   then u ← m − 1
    elif key = a.[m]   then found ← Some m
    (* key > a.[m] *) else  l ← m + 1
  found
```

Wir definieren drei Veränderliche: die beiden Intervallgrenzen und den Suchstatus. Die Schleifenbedingung involviert alle drei Veränderliche; im Schleifenrumpf wird genau eine von ihnen modifiziert. Nach endlich vielen Schritten ist entweder das Suchintervall $l \,..\, u$ leer oder der Suchschlüssel *key* gefunden.

Abstrakte Syntax Ausdrücke vom Typ *Unit*, die *nur* ihres Effektes wegen ausgerechnet werden, heißen auch *Anweisungen*. Eine Funktion des Typs $t \to \mathit{Unit}$, die *nur* ihres Effektes wegen aufgerufen wird, wird auch *Prozedur* genannt.

Die folgende Tabelle fasst die wichtigsten Kontrollstrukturen zusammen, mit deren Hilfe elementare Anweisungen wie zum Beispiel Ausgabeoperationen oder Zuweisungen zu komplexen Anweisungen zusammengebaut werden.

$e ::= \cdots$	*Kontrollstrukturen:*
$\mid\ e_1; e_2$	Sequenz
$\mid$ **if** e_1 **then** e_2	einarmige Alternative
$\mid$ **if** e_1 **then** e_2 **else** e_3	zweiarmige Alternative
$\mid$ **for** x **in** e_1 **do** e_2	beschränkte Wiederholung⁄Iteration
$\mid$ **while** e_1 **do** e_2	unbeschränkte Wiederholung⁄Iteration

Zur Erinnerung: Die Sequenz $e_1; e_2$ entspricht dem **in**-Ausdruck **let** _ = e_1 **in** e_2, der die Auswertung von e_1 und e_2 sequentialisiert. Die einarmige Alternative **if** e_1 **then** e_2 ist eine Abkürzung für **if** e_1 **then** e_2 **else** (). Der Dummywert '()' zeigt an, dass der **else**-Zweig keinen Effekt hat. Der Teilausdruck e_1 in **while** e_1 **do** e_2 heißt *Schleifenbedingung* oder *Laufbedingung*; e_2 heißt *Schleifenrumpf*. (Laufbedingung, weil die Schleife „läuft", solange die Bedingung wahr ist; es gibt in anderen Programmiersprachen auch Schleifenkonstrukte mit inverser Logik, die eine *Abbruchbedingung* verwenden.)

Statische Semantik Wir beschränken uns auf die Formalisierung von **for**-Schleifen, die über Listen iterieren. Analoge Überlegungen gelten für Iterationen über Arrays. Die **for**-Schleife ist ein „Binder": Die Schleifenvariable ist im Rumpf der Schleife sichtbar.

$$\frac{\Sigma \vdash e_1 : \mathit{List}\,\langle t_1 \rangle \qquad \Sigma, \{\, x_1 \mapsto t_1 \,\} \vdash e : \mathit{Unit}}{\Sigma \vdash \textbf{for}\ x_1\ \textbf{in}\ e_1\ \textbf{do}\ e : \mathit{Unit}}$$

$$\frac{\Sigma \vdash e_1 : \mathit{Bool} \qquad \Sigma \vdash e_2 : \mathit{Unit}}{\Sigma \vdash \textbf{\textit{while}}\ e_1\ \textbf{\textit{do}}\ e_2 : \mathit{Unit}}$$

Der Schleifenrumpf ist jeweils eine Anweisung, ein Ausdruck vom Typ *Unit*. Beide Schleifen werden nur des Effektes willen ausgeführt.

module
Effects.
Control

Dynamische Semantik Aus Gründen des Lesbarkeit beschränken wir uns wie in Abschnitt 7.2 auf die Modellierung von Speichereffekten und ignorieren externe Effekte.

$$\frac{\sigma \parallel \delta \vdash e_1 \Downarrow [\nu_0; \ldots; \nu_{n-1}] \parallel \sigma_0 \qquad \sigma_i \parallel \delta, \{x_1 \mapsto \nu_i\} \vdash e \Downarrow () \parallel \sigma_{i+1} \quad | \quad i \in \mathbb{N}_n}{\sigma \parallel \delta \vdash \textbf{\textit{for}}\ x_1\ \textbf{\textit{in}}\ e_1\ \textbf{\textit{do}}\ e \Downarrow () \parallel \sigma_n}$$

Zunächst wird die Liste e_1 ausgewertet; die Schleifenvariable x_1 wird nacheinander an die Listenelemente gebunden; bezüglich jeder Bindung wird der Schleifenrumpf e ausgerechnet. Zur Erinnerung: $\phi_i \mid i \in \mathbb{N}_n$ ist eine kompakte Schreibweise für die n Voraussetzungen ϕ_0, …, ϕ_{n-1}. Jede Auswertung verändert potentiell den Speicher, der von Auswertung zu Auswertung weitergereicht wird. Die ***for***-Schleife „kommuniziert“ mit ihrer Umwelt mittels des Speichers — ihr Wert, das einzige Element vom Typ *Unit*, steht ja bereits vor der Auswertung fest.

Für die ***while***-Schleife gelten ähnliche Überlegungen.

$$\frac{\sigma \parallel \delta \vdash e_1 \Downarrow \mathit{false} \parallel \sigma_1}{\sigma \parallel \delta \vdash \textbf{\textit{while}}\ e_1\ \textbf{\textit{do}}\ e_2 \Downarrow () \parallel \sigma_1}$$

$$\frac{\sigma \parallel \delta \vdash e_1 \Downarrow \mathit{true} \parallel \sigma_1 \qquad \sigma_1 \parallel \delta \vdash e_2 \Downarrow \nu \parallel \sigma_2 \qquad \sigma_2 \parallel \delta \vdash \textbf{\textit{while}}\ e_1\ \textbf{\textit{do}}\ e_2 \Downarrow () \parallel \sigma_3}{\sigma \parallel \delta \vdash \textbf{\textit{while}}\ e_1\ \textbf{\textit{do}}\ e_2 \Downarrow () \parallel \sigma_3}$$

Zunächst wird die Schleifenbedingung e_1 ausgewertet. Ist das Ergebnis *false*, ist die Auswertung damit abgeschlossen; anderenfalls wird der Schleifenrumpf e_2 ausgewertet und *die Schleife wird erneut ausgewertet*. Ein wichtiges Detail: Die wiederholte Auswertung „sieht“ den modifizierten Speicher σ_2. Die letzte Auswertungsregel zählt zu den ungewöhnlichsten, die uns bisher begegnet sind. Normalerweise wird die Bedeutung eines Ausdrucks auf die Bedeutung seiner *Teil*ausdrücke zurückgeführt. Die ***while***-Regel folgt nicht dieser kompositionalen Herangehensweise: Eine der Voraussetzung enthält den zu definierenden Ausdruck selbst.

Der Dämon der Nichtterminierung lugt um die Ecke. Verändern weder e_1 noch e_2 den Speicher, ***while*** *true* ***do*** $i \leftarrow i + 1$, dann tritt die Schlussfolgerung unverändert als Voraussetzung auf, ein unendlicher Regress. Mit anderen Worten: Es lässt sich kein Beweisbaum für die Auswertung konstruieren. Es ist zwingend notwendig, dass der Speicher modifiziert wird. Notwendig, aber nicht hinreichend: $i \leftarrow 1$; ***while*** $i > 0$ ***do*** $i \leftarrow i + 1$ terminiert ebenfalls nicht. Wie die ungebändigte Rekursion — Rekursion, die nicht den Vorgaben eines Entwurfsmusters folgt — birgt die ***while***-Schleife die Gefahr der Nichtterminierung und sollte daher mit Bedacht verwendet werden.

Schleifen sind wie Listen- und Arraybeschreibungen syntaktischer Zucker. Wir können ihre Bedeutung auch klären, indem wir sie in uns bereits bekannte Konstrukte übersetzen, sprich, indem wir die Schleifen programmieren. Die Schleife ***for*** x ***in*** $l \,..\, u$ ***do*** e lässt sich zum Beispiel in den Aufruf *foreach* (l, u) (***fun*** $x \rightarrow e$) übersetzen, wobei *foreach* wie

folgt definiert ist. (Umgekehrt kann *foreach* (l, u) *body* durch ***for*** x ***in*** $l \,..\, u$ ***do*** *body* x implementiert werden.)

```
let rec foreach (lower : Nat, upper : Nat) (body : Nat → Unit) : Unit =
  if lower ⩽ upper then
    body lower
    foreach (lower + 1, upper) body
```

Die ***for***-Schleife ist ein Binder; in der Übersetzung wird die Bindung auf einen Funktionsausdruck abgebildet, der Mutter aller variablenbindenden Konstrukte. Die Definition von *foreach* folgt dem Entwurfsmuster für Intervalle: Die Terminierung ist garantiert, sofern der Rumpf stets terminiert. Zur Übung: Programmieren sie entsprechende Funktionen, um über alle Elemente einer Liste bzw. eines Arrays zu iterieren, siehe Aufgabe 7.7.

In die Übersetzung der ***while***-Schleife müssen wir mehr Grips stecken. Die Typregel suggeriert, eine Funktion der Form ***let rec*** *whiledo* (*test* : *Bool*) (*body* : *Unit*) : *Unit* zu definieren. Das ist allerdings zu kurz gedacht. Da Parameter in Mini–F# vor dem Aufruf ausgewertet werden (Stichwort: call by value), würde der Test und der Rumpf genau einmal beim Aufruf von *whiledo* ausgewertet. (Mit den entsprechenden Konsequenzen: Die Schleife wird entweder gar nicht ausgeführt oder endlos.) Die Schleifenbedingung ist kein Boolescher Wert, sondern eine *Rechnung*, die einen Wahrheitswert zum Ergebnis hat. Wir müssen den Test durch eine *Funktion* des Typs *Unit* → *Bool* modellieren und den Rumpf entsprechend durch eine Funktion des Typs *Unit* → *Unit*. Die Schleife ***while*** e_1 ***do*** e_2 wird dann in *whiledo* (***fun*** () → e_1) (***fun*** () → e_2) übersetzt. Wir erinnern uns: „Funktionsausdrücke werten zu sich selbst aus“; ihre Auswertung wird eingefroren. Beim Aufruf von *whiledo* wird auf diese Weise weder e_1 noch e_2 ausgewertet. (Umgekehrt kann *whiledo test body* durch ***while*** *test* () ***do*** *body* () implementiert werden.)

```
let rec whiledo (test : Unit → Bool) (body : Unit → Unit) : Unit =
  if test () then
    body ()
    whiledo test body
```

Die Funktionsabschlüsse werden zum Leben erweckt, sie werden aufgetaut, indem sie auf das Dummyargument ‘()’ angewendet werden. Die Definition von *whiledo* führt uns die Gefahr der Nichtterminierung noch einmal plastisch vor Augen: Der rekursive Aufruf ist identisch zu dem ursprünglichen Aufruf, kein Parameter wird verändert.

Die Definitionen von *foreach* und *whiledo* sind übrigens *endrekursiv*: Der rekursive Aufruf ist die letzte Aktion im Funktionsrumpf. Endrekursive Programme lassen sich in iterative Programme übersetzen, siehe Abschnitt 7.3.3 — der Kreis schließt sich.

module Effects. Sort

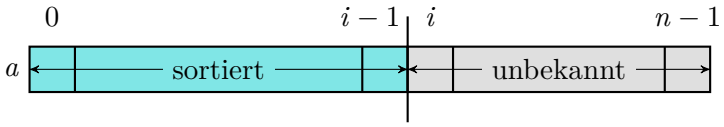
Vertiefung In Abschnitt 5.1.1 haben wir zwei einfache Algorithmen kennengelernt, um Folgen von Elementen zu sortieren: Sortieren durch Einfügen und Sortieren durch Auswählen. Die Implementierungen der Sortierverfahren im besagten Abschnitt arbeiten auf

Listen: Sie nehmen eine unsortierte Liste als Eingabe und geben eine sortierte Permutation als Ausgabe zurück. Im folgenden schauen wir uns an, wie man Arrays sortieren kann. Genauer: Wir überlegen, wie man die Elemente an „Ort und Stelle“ (engl. in place, lat. in situ) sortieren kann, *ohne* zusätzlichen Speicher zu allokieren. Die folgende Interaktion zeigt die Vorgehensweise.

```
Mini⟩ let revenues = [|815; 4; 7; 1; 1|]
Mini⟩ selection-sort revenues
()
Mini⟩ revenues
[|1; 1; 4; 7; 815|]
```

Die Sortierfunktion hat kein interessantes Ergebnis — sie hat den Ergebnistyp *Unit* — aber einen bemerkenswerten Effekt: Nach dem Aufruf ist das Eingabearray sortiert. Wir werden sehen, dass die Reimplementierung der Sortierverfahren sich stark von den Programmen aus Abschnitt 5.1.1 unterscheidet — nichtdestotrotz setzen sie jeweils die gleiche algorithmische Idee um.

Wenden wir uns zunächst dem Sortieren durch Auswählen zu. Da wir mit dem zur Verfügung stehenden Platz in Form des Eingabearrays auskommen müssen, organisieren wir die Sortierung durch wiederholtes Vertauschen von Arrayelementen. Dazu teilen wir das gegebene Array a gedanklich in zwei Zonen auf:



wobei $n = a.Length$. Die linke, grüne Zone ist sortiert; über die rechte, graue Zone wissen wir nichts. Die grüne Zone wird schrittweise nach rechts ausgedehnt, bis sie sich über das gesamte Areal erstreckt. Ein Schritt besteht darin, das Minimum der grauen Zone zu bestimmen und mit dem ersten Element der grauen Zone auszutauschen.

```
let selection-sort (a : Array ⟨'elem⟩) =
  let n = a.Length
  for i in 0 .. n − 2 do
    // determine minimum of a.[i], . . . , a.[n − 1]
    let mutable m = i
    for j in i + 1 .. n − 1 do
      if a.[j] < a.[m] then
        m ← j;
    // swap a.[i] and a.[m]
    let tmp = a.[i]
    a.[i]  ← a.[m]
    a.[m] ← tmp
```

Um die Vertauschung einfach durchführen zu können, wird tatsächlich die *Position* des Minimums bestimmt: m ist die Position und $a.[m]$ das eigentliche Minimum.

Ist das Programm korrekt? Ähnlich wie wir die Korrektheit eines rekursiven Programms mit Hilfe einer Rekursionsinvariante nachweisen können (siehe Abschnitt 5.2.4), können wir die Korrektheit eines iterativen Programms mit Hilfe einer *Schleifeninvariante* zeigen. Die Invariante fängt typischerweise die algorithmische Idee ein, in unserem Beispiel die Unterteilung des Arrays in zwei Zonen.

Invariante (i): $\quad a.[0] \leqslant a.[1] \leqslant \cdots \leqslant a.[i-1]$
$\forall\, j,k \;.\; 0 \leqslant j < i \;\wedge\; i \leqslant k < n \;\Rightarrow\; a.[j] \leqslant a.[k]$

Die Invariante ist mit i, der Zonengrenze, parametrisiert. Die Invariante fordert, dass die grüne Zone sortiert ist und dass die Elemente der grünen Zone höchstens so groß sind, wie die Elemente der grauen Zone.

Auch Schleifeninvarianten durchlaufen in ihrem Leben drei Phasen:

1. die Invariante wird etabliert (vor der Schleife);
2. die Invariante wird erhalten (bei einem Schleifendurchlauf);
3. aus der Invariante folgt das gewünschte Ergebnis (nach der Schleife).

Für unser Beispiel ergeben sich die folgenden Überlegungen:

Schritt 1: Wenn $i = 0$ gilt, dann ist die grüne Zone leer und die Anforderungen gelten trivialerweise.

Schritt 2: Wir müssen zeigen, dass nach einem Schleifendurchlauf die Invariante für $i+1$ gilt, unter der Annahme, dass sie vor dem Durchlauf für i gilt. Da der grüne Bereich um das Element $a.[i]$ erweitert wird, reicht es zu zeigen, dass dieses Element größer $(\geqslant)$ ist als alle Elemente zur Linken und kleiner $(\leqslant)$ als alle Elemente zur Rechten. Ersteres gilt auf Grund der Invariante für i und der Tatsache, dass $a.[i]$ aus der grauen Zone stammt. Letzteres gilt, da $a.[i]$ das Minimum der grauen Zone ist.

Schritt 3: Wenn $i = n-1$ ist, dann besteht die graue Zone aus genau einem Element, das zudem größer ist als alle Elemente der grünen Zone. Somit ist das gesamte Array sortiert. Was zu beweisen war.

Terminiert das Programm? Klare Antwort: Ja! Eine ***for***-Schleife gibt ein Terminierungsversprechen. Da nur über endliche viele Elemente eines Intervalls, einer Liste oder eines Arrays iteriert wird, ist die Terminierung sichergestellt, sofern der Schleifenrumpf stets terminiert. Das obige Programm besteht aus zwei ineinandergeschachtelten ***for***-Schleifen und einigen elementaren Ausdrücken, so dass nichts anbrennen kann.

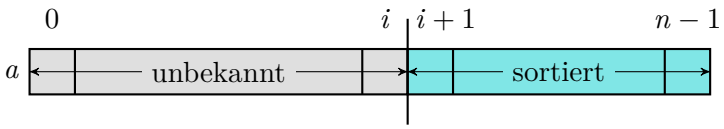
Welche Laufzeit hat das Programm? Laufzeit ist die kleine Schwester der Terminierung. Eine ***for***-Schleife gibt nicht nur ein Terminierungsversprechen, sondern sie erlaubt auch klare Aussagen über die Laufzeit. Ein Schleife der Form ***for*** x ***in*** $l\,..\,u-1$ ***do*** e wiederholt den Schleifenrumpf e insgesamt $u-l$ mal; die Kosten für die Abarbeitung des Rumpfes müssen entsprechend aufsummiert werden. Hängt die Laufzeit nicht von der Schleifenvariablen x ab, dann ergibt sich die Laufzeit als Produkt von $u-l$ und der Anzahl der benötigten Schritte für e. Besteht eine Abhängigkeit, müssen wir mittels einer Summenformel die Kosten aufaddieren. Unser Beispiel illustriert beide Fälle.

(Wir zählen der Einfachheit halber nur die Anzahl der Vergleiche $a.[j] < a.[m]$). Die innere Schleife wiederholt die einarmige Alternative $n-(i+1)$ mal; die äußere Schleife wiederholt die innere Schleife $n-1$ mal, allerdings mit unterschiedlichen Werten für die Schleifenvariable i. Mit einer Summenformel ausgedrückt:

$$\sum_{i=0}^{n-2} n-(i+1) = \sum_{i=1}^{n-1} i = (n-1)\cdot n/2$$

Das Ergebnis stimmt exakt mit der Laufzeit der listenbasierten Implementierung aus Abschnitt 5.1.1 überein. Sortieren durch Auswählen ist und bleibt ein quadratisches Sortierverfahren.

Kommen wir zum Sortieren durch Einfügen. Wir teilen das zu sortierende Array wiederum gedanklich in zwei Zonen auf.



Dual zum Sortieren durch Einfügen befindet sich die sortierte, grüne Zone jetzt rechts. Wir lassen sie schrittweise wachsen, indem wir jeweils ein graues Element in die grüne Zone einfügen. (Aufgabe 7.5 klärt, warum wir von rechts nach links vorgehen.)

```
let insertion-sort (a : Array ⟨'elem⟩) =
  let n = a.Length
  for i in n − 2 .. −1 .. 0 do
    let key = a.[i]
    // insert key into the sorted sub-array a.[i + 1], .., a.[n − 1]
    let mutable j = i + 1
    while j < n && key > a.[j] do
      a.[j − 1] ← a.[j]
      j ← j + 1
    a.[j − 1] ← key
```

Die ***for***-Schleife verwendet ein Konstrukt, das uns bisher noch nicht untergekommen ist, ein *Intervall mit Schrittweite*: $l\,..\,d\,..\,r$ generiert die Elemente l, $l+d$, $l+2\cdot d$, ..., r. Mit $0\,..\,1\,..\,n-1$ oder kurz $0\,..\,n-1$ werden die Hausnummern des Arrays a aufsteigend, von links nach rechts aufgezählt; mit $n-1\,..\,-1\,..\,0$ absteigend, von rechts nach links; kurz: spiegelverkehrt.

Um das Element $a.[i]$ in die grüne Zone einzufügen, verwenden wir eine ***while***-Schleife: Solange wir noch nicht den rechten Rand der grünen Zone erreicht haben und die richtige Position nicht ermittelt ist, gehen wir nach rechts. „Einfügen“ hört sich etwas einfacher an, als es tatsächlich ist. Um ein Element in ein Array einzufügen, müssen wir Platz schaffen. Auf dem Weg durch die grüne Zone werden die Arrayelemente um eine Position nach links verschoben. Das erste Arrayelement, das *überschrieben* wird, ist das einzufügende Element selbst. Aus diesem Grund merken wir uns das Element, indem wir $a.[i]$

an den Bezeichner *key* binden. Summa summarum werden die Arrayelemente $a.[i]$, …, $a.[j-1]$ zyklisch nach links vertauscht, wobei $a.[j-1]$ das größte Element der grünen Zone ist, das echt kleiner als $a.[i]$ ist.

Terminiert das Programm? Vorsicht ist geboten: Im Unterschied zu einer ***for***-Schleife gibt eine ***while***-Schleife kein Terminierungsversprechen: Zum Beispiel terminiert der Ausdruck ***while*** *true* ***do*** *print* `"busy"` nicht. Im Fall von *insertion-sort* können wir allerdings Entwarnung geben: Das erste Glied der Konjunktion, $j < n$, im Zusammenspiel mit der letzten Zuweisung im Schleifenrumpf, $j \leftarrow j + 1$, stellt die Terminierung sicher.

Welche Laufzeit hat das Programm? Die ***while***-Schleife wird im schlechtesten Fall $n - (i + 1)$ mal durchlaufen. Die ***for***-Schleife wiederholt die ***while***-Schleife $n - 1$ mal, allerdings mit unterschiedlichen Werten für die Schleifenvariable i. Insgesamt werden $(n-1) \cdot n/2$ Schritte benötigt.

Die ***while***-Schleife implementiert eine lineare Suche. Können wir die Suche beschleunigen, indem wir binär suchen? Ja, aber das zahlt sich leider nicht aus, da für das einzufügende Element Platz geschaffen werden muss. Der konstante Zugriff auf Arrayelemente lässt sich in diesem Fall nicht zu unserem Vorteil nutzen.

Auch Sortieren durch Einfügen ist und bleibt ein quadratisches Sortierverfahren. Unterschiede ergeben sich allerdings, wie bereits besprochen, im besten Fall: Ist die Eingabe bereits sortiert, benötigt Sortieren durch Einfügen nur $n - 1$ Vergleiche, während Sortieren durch Auswählen weiterhin eine quadratische Laufzeit hat. Diese Unterschiede lassen sich an Äußerlichkeiten festmachen. Sortieren durch Auswählen verwendet zwei ineinandergeschachtelte ***for***-Schleifen, die stets über die gleichen Intervalle iterieren; die Anzahl der Schritte ist somit unabhängig von den Eingabeelementen. Sortieren durch Auswählen ist „vergesslich“ (engl. oblivious). Sortieren durch Einfügen ersetzt die innere Schleife durch eine ***while***-Schleife, die *maximal* $n - (i + 1)$ mal durchlaufen wird, aber unter Umständen auch gar nicht. Sortieren durch Einfügen zeigt ein *adaptives* Verhalten.

Wir haben in Abschnitt 5.1.3 besprochen, dass jedes Sortierverfahren, das auf dem Vergleichen von Elementen basiert, im schlechtesten Fall mindestens $n \lg n$ Vergleiche benötigt. Nun ist es nicht immer zwingend zu vergleichen; um zum Beispiel ein Array zu sortieren, das nur Zahlen aus einem gegebenen Intervall, $0\,..\,m$, enthält, bietet es sich an, zu zählen, wie häufig jedes Element aus dem Intervall vorkommt.

```
let counting-sort (m : Int, array : Array ⟨Int⟩) : Array ⟨Int⟩ =
  let counts = [| for i in 0 .. m → 0 |]
  for j in array do
    counts.[j] ← counts.[j] + 1
  [| for i in 0 .. m do
       for c in 1 .. counts.[i] do
         yield i |]
```

In dem Array *counts* halten wir die Häufigkeit der einzelnen Elemente fest. Im Ausgabearray wird jedes Element entsprechend seiner Häufigkeit im Eingabearray wiederholt. Die Laufzeit der Verfahrens, *Sortieren durch Zählen*, ist proportional zu $\max\{m, n\}$, wobei n die Größe des Eingabearrays ist. Ein Laufzeitgewinn — es sei denn, m ist sehr viel größer als n.

Die Funktion *counting-sort* kombiniert geschickt ***for***-Schleifen mit Arraybeschreibungen, um ein sortiertes Ausgabearray zu erstellen. *Zur Übung:* wie muss das Programm modifiziert werden, um das Eingabearray in situ zu sortieren?

7.3.3. Endrekursion

To iterate is human;
to recurse, divine.

— L. Peter Deutsch

Rekursion ist wie ein Einkaufsbummel von der Wohnung in die Fußgängerzone: Sowohl auf dem Hin- als auch auf dem Rückweg gibt es interessante Aktivitäten. Auf dem Hinweg werden Probleme in immer kleinere Teilprobleme zerteilt; auf dem Rückweg werden Teillösungen zu immer größeren Lösungen kombiniert, siehe Abbildung 5.1. *Iteration* ist in gewissem Sinne Einwegrekursion: Aktivitäten finden nur auf dem Hinweg statt; ist man am Ziel angelangt, wird man zurück ins Heim gebeamt. (Oder umgekehrt: der Hinweg führt ereignislos ans Ziel; nur auf dem Rückweg werden die Besorgungen erledigt.) Die Idee der "Einwegrekursion" wollen wir im Folgenden genauer unter die Lupe nehmen.

In Abschnitt 4.5 haben wir uns mit ähnlichen Themen beschäftigt (Stichworte: Endrekursion, Fortsetzungen, Stack und Heap), dort aus Implementierungssicht, auf der Ebene einer abstrakten Maschine. In diesem und im nächsten Abschnitt machen wir gewissermaßen einen zweiten Durchlauf durch den Themenkomplex, diesmal aus Programmiersicht, auf der Ebene der Programmiersprache. Trotz einer gewissen, durchaus gewollten Redundanz ergänzen sich die Abschnitte: Hier liegt der Fokus auf Programmiertechniken, dort auf Implementierungsdetails, die diese Techniken unterstützen.

Tail Recursion Elimination Betrachten wir noch einmal die Funktion *product*, die die Elemente einer Liste miteinander multipliziert.

```
let rec product = function
  | []      -> 1
  | x :: xs -> x * product xs
```

Die Definition ist exakt nach dem Struktur Entwurfsmuster für Listen gestrickt: Der Basisfall wird ad hoc gelöst; im Rekursionsschritt wird *product* auf die Restliste angewendet; die erhaltene Teillösung, das Produkt der Restliste, wird durch Multiplikation mit dem Kopfelement zur Gesamtlösung erweitert. Das Rekursionsmuster wird greifbar,

wenn man sich eine konkrete Auswertung anschaut:

$$
\begin{aligned}
& \mathit{product}\ (4 :: 7 :: 11 :: [\,]) \\
= \quad & \{\ \text{Definition von } \mathit{product}\ \} \\
& 4 * \mathit{product}\ (7 :: 11 :: [\,]) \\
= \quad & \{\ \text{Definition von } \mathit{product}\ \} \\
& 4 * (7 * \mathit{product}\ (11 : [\,])) \\
= \quad & \{\ \text{Definition von } \mathit{product}\ \} \\
& 4 * (7 * (11 * \mathit{product}\ [\,])) \\
= \quad & \{\ \text{Definition von } \mathit{product}\ \} \\
& 4 * (7 * (11 * 1)) \\
= \quad & \{\ \text{Definition von } `*`\ \} \\
& 4 * (7 * 11) \\
= \quad & \{\ \text{Definition von } `*`\ \} \\
& 4 * 77 \\
= \quad & \{\ \text{Definition von } `*`\ \} \\
& 308
\end{aligned}
$$

Beim rekursiven Abstieg wird ein Turm von Multiplikationen aufgebaut; beim rekursiven Aufstieg wird dieser Turm Schritt für Schritt abgebaut. Die Laufzeit, die „Höhe“ der Rechnung, ist linear zur Länge der Liste — was zu erwarten ist. Allerdings ist auch der Speicherbedarf, die „Breite“ der Rechnung, linear zur Listenlänge. Sind wir im Basisfall angelangt, hat sich für eine n-elementige Liste ein Turm von n Multiplikationen aufgebaut. Um sich diesen Ausdruck zu merken, wird Speicherplatz benötigt, *zusätzlich* zu dem Platz, den die Liste selbst verschlingt. Das ist ein Problem, wenn der Speicherplatz knapp oder die Liste *sehr* lang ist. Das Ergebnis ist in beiden Fällen das Gleiche: Die Rechnung wird mit einem „Stack Overflow“ abgebrochen. (Auf dem Rechner des Dozenten führt zum Beispiel der Aufruf *product* $[0\,..\,999999]$ zum besagten Fehlerabbruch.) Kurz zum Hintergrund: Die Buchhaltung von Funktionsaufrufen wird in der Regel stackartig organisiert. Beim Funktionsaufruf werden die Parameter auf dem *Rekursionsstack* abgelegt und nach Abarbeitung des Funktionsrumpfes wieder entfernt. Leider kann der Rekursionsstack nicht in den Himmel wachsen; seine Größe ist beschränkt. Ist die maximale Größe erreicht, wird die Rechnung unvermittelt mit der obigen Fehlermeldung abgebrochen.

Der iterativen Variante von *product* ist dieses Problem nicht zu eigen.

```
let product list =
  let mutable acc = 1
  for x in list do
    acc ← acc * x
  acc
```

Der zusätzlich benötigte Speicherplatz ist konstant: Wir verwenden eine Speicherzelle, *acc*, um uns jeweils das aktuelle Zwischenergebnis zu merken. Die gesamte Arbeit wird sozusagen auf dem Hinweg durch die Liste erledigt; ist das Listenende erreicht, wird das Zwischenergebnis unmittelbar zum Endergebnis.

Um das Verhältnis von Rekursion zu Iteration genauer zu beleuchten, lassen Sie uns die rekursive Definition von *product* so umschreiben, dass ebenfalls die gesamte Arbeit auf dem Hinweg erledigt wird. Wir spezifizieren:

$$worker\ acc\ list = acc * product\ list \tag{7.1}$$

Die Funktion *worker* erledigt zwei Aufgaben auf einmal: Sie bildet das Produkt einer Liste und multipliziert zusätzlich das Ergebnis mit ihrem ersten Argument. Die rechte Seite der Spezifikation ähnelt stark dem Rekursionsschritt von *product* — das ist kein Zufall; die zusätzliche Multiplikation ist ja gerade der Speicherplatzverbrecher, den wir eliminieren wollen. Die Spezifikation ist übrigens ein Beispiel für die Programmiertechnik der Verallgemeinerung, des Rekursionsparadoxons. Die noch zu definierende Funktion *worker* verallgemeinert *product*; haben wir eine effiziente Implementierung von *worker* gefunden, programmieren wir:

$$\textbf{\textit{let}}\ product\ list = worker\ 1\ list$$

Um sicherzustellen, dass wir keine Programmierfehler machen, *leiten* wir die Definition von *worker* aus der Spezifikation *her*.

Fall $list = [\,]$**:**

$$\begin{aligned}
& worker\ a\ [\,] \\
= \quad & \{\ \text{Spezifikation von } worker\ (7.1)\ \} \\
& a * product\ [\,] \\
= \quad & \{\ \text{Definition von } product\ \} \\
& a * 1 \\
= \quad & \{\ 1 \text{ ist das neutrale Element von } `*'\ \} \\
& a
\end{aligned}$$

Fall $list = x :: xs$**:**

$$\begin{aligned}
& worker\ a\ (x :: xs) \\
= \quad & \{\ \text{Spezifikation von } worker\ (7.1)\ \} \\
& a * product\ (x :: xs) \\
= \quad & \{\ \text{Definition von } product\ \} \\
& a * (x * product\ xs) \\
= \quad & \{\ `*' \text{ ist assoziativ}\ \} \\
& (a * x) * product\ xs \\
= \quad & \{\ \text{Spezifikation von } worker\ (7.1)\ \} \\
& worker\ (a * x)\ xs
\end{aligned}$$

Fassen wir den ersten und den letzten Ausdruck jeweils zu einer Rechenregel zusammen, erhalten wir das folgende Programm (a haben wir zu acc umbenannt).

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ worker\ acc = \textbf{\textit{function}} \\
\quad \mid [\,] \quad\ \ \rightarrow acc \\
\quad \mid x :: xs \rightarrow worker\ (acc * x)\ xs
\end{array}
$$

Es ist wichtig festzuhalten, dass die Herleitung algebraische Eigenschaften der Multiplikation verwendet: '$*$' ist assoziativ und 1 ist das neutrale Element von '$*$'. Die gleiche Herleitung funktioniert somit auch für die Addition, '$+$' und 0; *nicht* aber für die Subtraktion, '$-$' und 0. Insbesondere wird deutlich, dass *product* und *worker* nicht die identischen Operationen ausführen:

$$
\begin{array}{rcl}
product\ [a; b; c; d] & = & a * (b * (c * (d * 1))) \\
worker\ 1\ [a; b; c; d] & = & ((((1 * a) * b) * c) * d)
\end{array}
$$

Die Funktion *product* klammert rechtsassoziierend; *worker* 1 hingegen linksassoziierend. Da die Multiplikation assoziativ mit 1 als neutralem Element ist, unterscheiden sich die jeweiligen Ergebnisse trotz der unterschiedlichen Klammerung nicht.

Zurück zur Definition von *worker*: Der Parameter *acc* ist ein sogenannter *akkumulierender Parameter* oder kurz *Akkumulator*; die Funktion *worker* selbst ist *endrekursiv*. Nach dem rekursiven Aufruf, auf dem Rückweg, ist nichts mehr zu tun, wie die folgende Rechnung zeigt.

$$
\begin{array}{ll}
 & worker\ 1\ (4 :: 7 :: 11 :: [\,]) \\
= & \quad \{\ \text{Definition von } worker\ \} \\
 & worker\ 4\ (7 :: 11 :: [\,]) \\
= & \quad \{\ \text{Definition von } worker\ \} \\
 & worker\ 28\ (11 :: [\,]) \\
= & \quad \{\ \text{Definition von } worker\ \} \\
 & worker\ 308\ [\,] \\
= & \quad \{\ \text{Definition von } worker\ \} \\
 & 308
\end{array}
$$

Die Auswertung illustriert, warum *acc* Akkumulator heißt. Im Gegensatz zum Listenargument wird der erste Parameter nicht kleiner, sondern schwillt mit jedem rekursiven Aufruf an. Das Beispiel macht zudem die Vorteile der endrekursiven Variante deutlich. Ist das Listenende erreicht, wird der Akkumulator unmittelbar als Ergebnis zurückgegeben. Wie bei der iterativen Variante ist der zusätzliche Speicherbedarf konstant; die Rechnung geht nicht in die Breite. Da der rekursive Aufruf die letzte „Aktion" im Funktionsrumpf von *worker* ist, wird der Parameter *acc* nach dem rekursiven Aufruf nicht mehr benötigt; der Speicherplatz kann entsprechend wiederverwendet werden. Ein guter Übersetzer führt die sogenannte „*Tail Recursion Elimination*"[6] (TRE) durch: Aus

[6] Leider unterstützen nicht alle Programmiersprachen bzw. deren Implementierungen diese wichtige Optimierung.

der Rekursion wird eine Schleife; aus dem Parameter wird eine Veränderliche; aus der Änderung des Parameter wird die Zuweisung $acc \leftarrow acc * x$. Kurzum: Das endrekursive Programm wird in das (maschinennähere) iterative Programm umgeschrieben.

```
let product list =
  let rec worker acc = function
    | []      → acc
    | x :: xs → worker (acc * x) xs
  worker 1 list
```

```
let product list =
  let mutable acc = 1
  for x in list do
    acc ← acc * x
  acc
```

Der Effekt ist nachprüfbar: *product* $[0\,..\,999999]$ wertet jeweils problemlos zu 0 aus.

Tail Call Optimization Allgemein lassen sich Funktionsaufrufe in *Endpositionen*, sogenannte „*tail calls*“, bezüglich des Speicherverbrauchs optimieren. Dabei wird der Speicherplatz, der von den Funktionsparametern der aufrufenden Funktion belegt wird, bereits *vor* dem Aufruf freigegeben bzw. für den Aufruf wiederverwendet. Der Aufruf von g (bzw. g_1 und g_2) befindet sich in den folgenden Beispielen jeweils an einer Endposition; der Wert des Parameters x wird nach dem Aufruf von g nicht mehr benötigt und somit kann der von x belegte Speicherplatz für den „tail call“ wiederverwendet werden.

```
let f x = g e
let f x = if e then g1 e1 else g2 e2
let f x = let p1 = e1 in g e
```

Im Unterschied dazu befindet sich der Aufruf von g in `let f x = x * g e` *nicht* an einer Endposition — der Parameter x wird nach dem Aufruf von g für die Berechnung des Produkts benötigt.

Die Unterschiede lassen sich an den jeweiligen Auswertungsregeln festmachen.

$$\frac{\delta \vdash e_1 \Downarrow \mathit{true} \qquad \delta \vdash e_2 \Downarrow \nu}{\delta \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow \nu} \qquad \frac{\delta \vdash e_1 \Downarrow \mathit{false} \qquad \delta \vdash e_3 \Downarrow \nu}{\delta \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow \nu}$$

Man sieht, dass das Ergebnis des **then**- bzw. des **else**-Zweigs *unverändert* als Ergebnis der gesamten Alternative übernommen wird. Bei der Multiplikation ist das nicht der Fall:

$$\frac{\delta \vdash e_1 \Downarrow n_1 \qquad \delta \vdash e_2 \Downarrow n_2}{\delta \vdash e_1 * e_2 \Downarrow n_1 \cdot n_2}$$

Das Zwischenergebnis n_2 muss noch mit n_1 multipliziert werden (oder umgekehrt).

module Effects. Tree

Tail Calls und nicht-lineare Strukturen Die Umschreibung von *product* in eine endrekursive Form gelingt deshalb so problemlos, weil die zugrundeliegende Datenstruktur linear ist. Betrachten wir statt Listen zum Beispiel Binärbäume, sehen wir uns vor neue Herausforderungen gestellt.

```
let rec product tree =
  match tree with
  | Leaf          → 1
  | Node (l, x, r) → product l * x * product r
```

Die Definition ist exakt nach dem Struktur Entwurfsmuster für Binärbäume gestrickt: Der Basisfall wird ad hoc gelöst; Im Rekursionsschritt wird *product* auf den linken und auf den rechten Teilbaum angewendet; die erhaltenen Teillösungen werden durch Multiplikation mit dem Wurzelelement zur Gesamtlösung erweitert.

Um zu testen, wie robust *product* ist — wann tritt ein Stack Overflow auf? —, definieren wir zwei Baumgeneratoren, einen, der linksentartete, und einen, der rechtsentartete Binärbäume erzeugt.

```
let left-skewed size (x : 'elem) =
  let rec bottom-up n (tree : Tree ⟨'elem⟩) =
    if n = 0 then tree else bottom-up (n − 1) (Node (tree, x, Leaf))
  bottom-up size Leaf
let right-skewed size (x : 'elem) =
  let rec bottom-up n (tree : Tree ⟨'elem⟩) =
    if n = 0 then tree else bottom-up (n − 1) (Node (Leaf, x, tree))
  bottom-up size Leaf
```

Im Unterschied zu den Definitionen aus Abschnitt 5.3.4 sind die obigen Varianten *endrekursiv*. Die Bäume werden nicht von oben nach unten („top down“) konstruiert, sondern von unten nach oben („bottom up“). Warum? Nun, wir müssen vermeiden, dass bereits bei der Generierung eines Testbaums selbst ein Stack Overflow auftritt! Zur Übung: Formulieren Sie die Funktionen iterativ mit Hilfe einer ***for***- oder einer ***while***-Schleife.

Wie zu erwarten, verursacht sowohl der Aufruf *product* (*left-skewed* 1000000 1) als auch *product* (*right-skewed* 1000000 1) einen Stack Overflow. Versuchen wir also, die Funktion robuster zu machen. Die Idee des akkumulierenden Parameters führt zu der folgenden Definition.

```
let rec product-acc acc tree =
  match tree with
  | Leaf          → acc
  | Node (l, x, r) → product-acc (product-acc acc l ∗ x) r
```

Leider ist das nur die halbe Miete. Da ein Binärknoten zwei Teilbäume besitzt, benötigen wir zwei rekursive Aufrufe. Da diese irgendwie geschachtelt werden müssen, kann nur ein Aufruf in einer Endposition stehen. Da die Abarbeitung des linken Teilbaums, *product-acc acc l*, kein „tail call“ ist, verursacht *product-acc* 1 (*left-skewed* 1000000 1) weiterhin einen Stack Overflow. Immerhin wertet *product-acc* 1 (*right-skewed* 1000000 1) problemlos zu 1 aus.

Wollen wir sämtliche „non-tail calls“ eliminieren, müssen wir radikaler ans Werk gehen. Eine vielleicht revolutionäre Idee ist, den impliziten Rekursionsstack (die Ursache der Überläufe) durch einen expliziten Stack, der unter unserer Kontrolle steht, zu ersetzen. Wir repräsentieren den Stack durch eine Liste und spezifizieren:

```
product-acc-many acc trees = worker acc [for t in trees → product t]        (7.2)
```

Die Funktion *product-acc-many* bildet das Produkt einer Liste von Bäumen und multipliziert das Ergebnis zusätzlich mit dem Akkumulator. Aus dieser Spezifikation lässt sich die folgende Implementierung herleiten:

```
let rec product-acc-many acc = function
  | []                  → acc
  | Leaf :: ts          → product-acc-many acc ts
  | Node (l, x, r) :: ts → product-acc-many (acc * x) (l :: r :: ts)
let product tree = product-acc-many 1 [tree]
```

Wenn man möchte, kann man das zweite Argument als *Todo-Liste* lesen. Auf ihr merken wir uns, welche Teilbäume noch bearbeitet werden müssen. Ist die Todo-Liste leer, wird der Akkumulator zurückgegeben. Anderenfalls betrachten wir den ersten Baum. Im Falle eines Blatts arbeiten wir die Restliste ab; im Fall eines Knotens wird der Akkumulator mit dem Wurzelelement multipliziert und die beiden Teilbäume werden zur Todo-Liste hinzugefügt. Dabei wird allerdings die Reihenfolge der Faktoren vertauscht — die Programmtransformation basiert auf der Tatsache, dass die Multiplikation *kommutativ* ist, und ist somit für nicht-kommutative Operationen nicht anwendbar, siehe aber Aufgabe 7.6. Der Lohn der Mühen: Sowohl *product-acc-many* 1 [*left-skewed* 1000000 1] als auch *product-acc-many* 1 [*right-skewed* 1000000 1] werten klaglos zu 1 aus. Zur Übung: Überführen Sie *product-acc-many* in ein iteratives Programm, das heißt, führen Sie die „tail recursion optimization“ von Hand durch.

7.3.4. Fortsetzungen⋆

Mit Hilfe von akkumulierenden Parametern lassen sich Funktionen wie *length*, *sum*, *product* usw. in eine endrekursive Form bringen. Endrekursive Programme sind vorteilhaft, da sie mit dem Stackspeicher sparsam umgehen und so unerwünschte Stack Overflows verhindern. (Zur Klarstellung: Diese treten nur auf, wenn *sehr* große Datenmengen verarbeitet werden.) Wir haben im letzten Abschnitt gesehen, dass die Transformation in eine endrekursive Form von Eigenschaften der beteiligten Operationen abhängt (z.B. Assoziativität oder Kommutativität) und somit nicht immer angewendet werden kann. In diesem Abschnitt lernen wir eine allgemeine Technik kennen, mit der die Transformation immer gelingt, *ohne* zusätzliche Annahmen machen zu müssen.

Kommen wir noch einmal zu dem Ausgangspunkt unserer Überlegungen zurück.

```
let rec product = function
  | []      → 1
  | x :: xs → x * product xs
```

Das Ziel ist klar: Wir wollen die zusätzliche Arbeit nach dem rekursiven Aufruf vermeiden, die Multiplikation mit x. Dazu haben wir im letzten Abschnitt eine Funktion *worker* definiert, die x als zusätzlichen Parameter mit auf den Weg nimmt, siehe (7.1). Aber warum nur x? Die Restarbeit ist ja ‘$x *$ ’, eine *Multiplikation* mit x. Allgemeiner formuliert: *product* wird in einem bestimmten Kontext aufgerufen, *cont* (*product list*), der die

Berechnung *fortsetzt* und die Rückgabe von *product* verarbeitet. Die Idee ist nun, diese restliche Arbeit, die sogenannte *Fortsetzung* (engl. *continuation*), der Funktion *worker* mit auf den Weg geben. Wir spezifizieren (*cont* kürzt continuation ab):

$$worker\ list\ cont = cont\ (product\ list) \tag{7.3}$$

Im Normalfall wird *worker* sein Ergebnis *nicht* zurückgeben, sondern als letzte Aktion an die Fortsetzung *cont* weiterreichen. Die Spezifikation ist ein weiteres Beispiel für die Programmiertechnik der Verallgemeinerung, des Rekursionsparadoxons. Wie gewohnt verallgemeinert die noch zu definierende Funktion *worker* die ursprüngliche Funktion *product*; haben wir eine effiziente Implementierung von *worker* gefunden, definieren wir:

$$\textbf{\textit{let}}\ product\ list = worker\ list\ (\textbf{\textit{fun}}\ r \to r)$$

Als initiale Fortsetzung wählen wir die identische Abbildung $\textbf{\textit{fun}}\ r \to r$, auch bekannt unter dem Namen *id*.

Um Programmierfehler zu vermeiden, leiten wir die Definition von *worker* wiederum aus der Spezifikation her.

Fall $list = [\,]$**:**

$$\begin{aligned}
& worker\ [\,]\ c \\
= \quad & \{\ \text{Spezifikation von } worker\ (7.3)\ \} \\
& c\ (product\ [\,]) \\
= \quad & \{\ \text{Definition von } product\ \} \\
& c\ 1
\end{aligned}$$

Fall $list = x :: xs$**:**

$$\begin{aligned}
& worker\ (x :: xs)\ c \\
= \quad & \{\ \text{Spezifikation von } worker\ (7.3)\ \} \\
& c\ (product\ (x :: xs)) \\
= \quad & \{\ \text{Definition von } product\ \} \\
& c\ (x * product\ xs) \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& (\textbf{\textit{fun}}\ r \to c\ (x * r))\ (product\ xs) \\
= \quad & \{\ \text{Spezifikation von } worker\ (7.3)\ \} \\
& worker\ xs\ (\textbf{\textit{fun}}\ r \to c\ (x * r))
\end{aligned}$$

Im vorletzten Schritt wird der Kontext von *product xs* in eine Funktion überführt. Also, aus $c\ (x * product\ xs)$ wird $c\ (x * \bullet)$, ein Ausdruck mit einem Platzhalter; aus diesem wird die Funktion $\textbf{\textit{fun}}\ r \to c\ (x * r)$.

Fassen wir den ersten und den letzten Ausdruck jeweils zu einer Rechenregel zusammen, erhalten wir das folgende Programm (*c* haben wir zu *cont* umbenannt).

```
let rec worker xs cont =
  match xs with
  | []      → cont 1
  | x :: xs → worker xs (fun r → cont (x * r))
```

Die Funktion *worker* ist wie gewünscht endrekursiv. Bei der Herleitung haben wir keinerlei Annahmen gemacht, keine Eigenschaften der beteiligten Operationen ausgenutzt.

Schauen wir uns ein Beispiel für eine Auswertung an.

$$
\begin{aligned}
& worker\ (4 :: 7 :: 11 :: [])\ (\mathbf{fun}\ r_1 \to r_1) \\
= \quad & \{\ \text{Definition von } worker\ \} \\
& worker\ (7 :: 11 :: [])\ (\mathbf{fun}\ r_2 \to (\mathbf{fun}\ r_1 \to r_1)\ (4 * r_2)) \\
= \quad & \{\ \text{Definition von } worker\ \} \\
& worker\ (11 :: [])\ (\mathbf{fun}\ r_3 \to (\mathbf{fun}\ r_2 \to (\mathbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ (7 * r_3)) \\
= \quad & \{\ \text{Definition von } worker\ \} \\
& worker\ []\ (\mathbf{fun}\ r_4 \to (\mathbf{fun}\ r_3 \to (\mathbf{fun}\ r_2 \to (\mathbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ (7 * r_3))\ (11 * r_4)) \\
= \quad & \{\ \text{Definition von } worker\ \} \\
& (\mathbf{fun}\ r_4 \to (\mathbf{fun}\ r_3 \to (\mathbf{fun}\ r_2 \to (\mathbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ (7 * r_3))\ (11 * r_4))\ 1 \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& (\mathbf{fun}\ r_3 \to (\mathbf{fun}\ r_2 \to (\mathbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ (7 * r_3))\ 11 \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& (\mathbf{fun}\ r_2 \to (\mathbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ 77 \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& (\mathbf{fun}\ r_1 \to r_1)\ 308 \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& 308
\end{aligned}
$$

Man sieht sehr schön die endrekursive Natur von *worker*: Ein Aufruf wird durch den nächsten ersetzt. Ebenso deutlich wird, dass die Rechnung in die Breite geht: Die jeweiligen Fortsetzungen, Multiplikation mit 4, Multiplikation mit 7, Multiplikation mit 11, werden im zweiten Parameter protokolliert. Es entsteht ein Turm von Fortsetzungen, ineinandergeschachtelte Funktionsausdrücke. Nichtdestotrotz werden auch größere Listen klaglos ausmultipliziert: $worker\ [0\,..\,999999]\ (\mathbf{fun}\ r \to r)$ ergibt wie gewünscht 0.

Warum tritt kein Stack Overflow auf? Wie bereits angedeutet, wird eine stack-artige Organisation für die Verwaltung von Parametern bei (geschachtelten) Funktionsaufrufen eingesetzt. Die tatsächlichen Daten, auf die die Parameter verweisen, also natürliche Zahlen (!), Paare, Records, Listen, Bäume und eben auch Funktionsabschlüsse, müssen in der Regel Funktionsaufrufe überleben. Sie werden dauerhaft gespeichert, solange bis sie tatsächlich nicht mehr benötigt werden. Der zu diesem Zweck verwendete Speicherbereich heißt im Fachjargon „*Heap*“ (engl. für Haufen) und enthält recht wörtlich einen

ungeordneten Haufen von Daten. Die Größe des Heaps ist in der Regel nur durch die tatsächliche Speicherkapazität des Rechners begrenzt. Wird der Speicherplatz auf dem Heap knapp, identifiziert der *Garbage collector* nicht mehr benötigte Daten und „recycelt“ den belegten Speicherplatz.

Jetzt da wir zwei endrekursive Varianten von *product* implementiert haben, eine, die einen akkumulierenden Parameter verwendet, und eine, die auf Fortsetzungen basiert, stellt sich natürlich die Frage, wie sich die beiden Varianten im Vergleich schlagen? Das Anlegen von Funktionsabschlüssen ist mit Aufwand verbunden, Aufwand, den man spüren bzw. hören kann: Der Aufruf *worker* $[0\,..\,999999]$ (***fun*** $r \to r$) ist ungleich langsamer als der korrespondierende Aufruf *worker* 1 $[0\,..\,999999]$ aus Abschnitt 7.3.3. Die akkumulierende Variante verarbeitet jedes Listenelement in einem Zug: Der Akkumulator wird mit dem Element multipliziert. Die fortsetzungsbasierte Variante arbeitet zweischrittig: Eine Fortsetzung wird angelegt, die dann zu einem späteren Zeitpunkt abgearbeitet wird. Die Ausnutzung algebraischer Eigenschaften ist also segensreich.

module Effects. Tree

Die Stärke des fortsetzungsbasierten Ansatzes liegt in seiner Allgemeinheit: Mit dem gleichen Ansatz können wir auch einen Binärbaum endrekursiv ausmultiplizieren. Im Fall von Listen legen wir eine Fortsetzung an; im Fall von Binärbäumen haben wir entsprechend zwei Fortsetzungen.

```
let rec product-cont tree cont =
  match tree with
  | Leaf          → cont 1
  | Node (l, x, r) → product-cont l (fun pl →
                       product-cont r (fun pr →
                       cont (pl * x * pr)))
```

Der linke Teilbaum wird ausmultipliziert; das Ergebnis wird an die erste Fortsetzung weitergereicht (***fun*** $pl \to \ldots$); dann wird der rechte Teilbaum ausmultipliziert; das Ergebnis wird an die zweite Fortsetzung weitergereicht (***fun*** $pr \to \ldots$); die Ergebnisse werden multipliziert und an die ursprüngliche Fortsetzung (*cont*) übergeben. Insgesamt ergibt sich das Bild einer Kollekte: Um Geld zu sammeln, wird ein Korb herumgereicht; jeder fügt ein paar Münzen hinzu und reicht den Korb an den Nachbarn (die Fortsetzung) weiter. Der Lohn der Mühen: Beide Aufrufe, *product-cont* (*left-skewed* 1000000 1) (***fun*** $n \to n$) und *product-cont* (*right-skewed* 1000000 1) (***fun*** $n \to n$) werden fehlerlos abgearbeitet.

Beide Programmiertechniken, Akkumulatoren und Fortsetzungen, lassen sich auch mit Gewinn kombinieren:

```
let rec product-acc-cont acc tree cont =
  match tree with
  | Leaf          → cont acc
  | Node (l, x, r) → product-acc-cont acc l (fun pl →
                       product-acc-cont (pl * x) r cont)
```

Die Zahlen in den linken Teilbäumen werden akkumuliert; für die rechten Teilbäume werden Fortsetzungen angelegt. Zur Übung: Wenn Sie mögen, überführen Sie die endrekursive Funktion *product-acc-cont* in eine iterative Form.

Die Technik der Fortsetzungen (engl. *continuation passing style*, kurz CPS) haben wir übrigens bereits bei der Implementierung von Parsern in Abschnitt 6.4.3 eingesetzt: Der Folgeakzeptor entspricht gerade einer Fortsetzung, die die syntaktische Analyse fortsetzt.

7.4. Ausnahmen

Nicht jede Rechnung lässt sich einem erfolgreichen Ende zuführen: Treffen wir im Laufe einer Rechnung auf den Teilausdruck $2 \div 0$, dann bleibt uns nichts anderes übrig, als mit der Schulter zu zucken und aufzugeben. Die dynamische Semantik äußert sich bis dato nicht zu dieser Situation — wie wir in Abschnitt 3.2 angemerkt haben, ist die Auswertungsregel für '÷' schlicht und einfach nicht anwendbar. Es ist klar, dass wir die Teilrechnung an dieser Stelle nicht weiterführen können, daraus folgt aber nicht notwendigerweise, dass wir damit auch die Gesamtrechnung abbrechen müssen. Ganz im Gegenteil: Der Divisor könnte aus einer interaktiven Eingabe resultieren; das Programm sollte dann die Benutzer*in auf den Fehler hinweisen und sich nicht klang- und sanglos verabschieden. Nun ist Division durch Null nicht die einzige Ausnahmesituation: Eine Fallunterscheidung kann unvollständig sein, ein Arrayzugriff kann außerhalb der Arraygrenzen liegen usw. Da viele Dinge während einer Rechnung schieflaufen können, liegt es nahe, einen allgemeinen Mechanismus einzuführen, um Ausnahmesituationen zu signalisieren, Rechnungen abzubrechen und an anderer Stelle wieder aufzunehmen.

Eine Rechnung wird abgebrochen, indem eine sogenannte *Ausnahme* (engl. exception) ausgelöst wird. Die ausgelöste Ausnahme kann an einer anderen Stelle behandelt werden; an dieser anderen Stelle wird die Rechnung dann fortgesetzt. Etwas bildlicher spricht man auch davon, dass eine Ausnahme *geworfen* und *gefangen* wird. Die Divisionsoperation wirft zum Beispiel die Ausnahme *Div*, wenn der Divisor Null ist.

```
Mini⟩ 815 ÷ (4711 − 7 ∗ 673) + 1
uncaught exception : Div
```

Da die Ausnahme nicht gefangen wird, trifft der Wurf auf die Benutzungsoberfläche und führt zu einer Fehlermeldung. Eine Ausnahme kann auch explizit mit ***raise*** geworfen werden. Der nachfolgende Ausdruck hat somit den gleichen Effekt wie der obige.

```
Mini⟩ raise Div + 1
uncaught exception : Div
```

Da die Rechnung abgebrochen wird, besitzt der Ausdruck ***raise*** *Div* + 1 *keinen* Wert, sondern hat nur einen Effekt.

Eine geworfene Ausnahme kann mit dem ***try***-Konstrukt gefangen werden.

```
Mini⟩ try show (815 ÷ (4711 − 7 ∗ 673) + 1) with | Div → "oops"
"oops"
Mini⟩ try show (815 ÷ (4711 − 6 ∗ 773) + 1) with | Div → "oops"
"12"
```

Zwischen den Schlüsselwörtern ***try*** und ***with*** steht der auszurechnende Ausdruck. Gelingt dessen Auswertung, so ist der berechnete Wert auch der Wert des gesamten Ausdrucks. Wird hingegen während der Auswertung eine Ausnahme geworfen, dann kommt der Teil nach dem Schlüsselwort ***with*** zum Einsatz. Dieser entspricht dem Rumpf einer erweiterten Fallunterscheidung mit ***match***. Passt die Ausnahme auf die linke Seite einer Regel, so wird mit der Auswertung der rechten Seite fortgefahren. Passt kein Muster, so wird die Ausnahme weitergeworfen.

```
Mini⟩ try show (815 ÷ (4711 − 7 ∗ 673) + 1) with | Match → "oops"
uncaught exception : Div
```

Die Ausnahme *Div* passt nicht auf das Muster *Match*, so dass *Div* weitergeworfen wird. Ausnahmen sind normale Werte, Elemente des Typs *Exception* (Kurzform: *exn*), und können wie Elemente eines Variantentyps verwendet werden.

```
Mini⟩ let exns = (Div, Match)
val exns : Exception ∗ Exception
Mini⟩ match snd exns with | Div → "div" | Match → "match"
"match"
```

Zunächst wird ein Paar von Ausnahmen konstruiert; anschließend wird die zweite Komponente mit einer Fallunterscheidung analysiert.

Die Ausnahme *Match* signalisiert übrigens, dass eine Fallunterscheidung fehlgeschlagen ist, dass keines der angegebenen Muster auf den analysierten Wert gepasst hat. Unvollständige Fallunterscheidungen sollten zwar so weit wie möglich vermieden werden, aber manchmal lässt sich nicht jeder Fall behandeln — zumindest auf den ersten Blick.

```
let head (list : List ⟨'elem⟩) : 'elem =
  match list with
  | x :: _ → x
```

Die Funktion *head* bestimmt das erste Element einer Liste; *head* ist somit eine *partielle Funktion*: Der Aufruf *head e* wirft die Ausnahme *Match*, wenn *e* zur leeren Liste auswertet.

```
Mini⟩ head (head ([] :: [4711] :: []))
uncaught exception : Match
Mini⟩ try head (head ([] :: [4711] :: [])) with | Match → 0
0
```

Eine *Match* Ausnahme ist nicht sehr spezifisch; sie drückt aus, dass irgendwo im Programm eine Fallunterscheidung fehlgeschlagen ist. Nun kann ein Programm viele Fallunterscheidungen verwenden. Dass eine Fallunterscheidung nicht alle Fälle abdeckt, kann gewollt sein, kann aber auch auf einen *Programmierfehler* (engl. *bug*) hindeuten. Aus diesem Grund sollte man Fallunterscheidungen vervollständigen und eventuelle Fehlerfälle explizit machen, zum Beispiel, mit einer maßgeschneiderten Ausnahme.

```
exception Head
let head (list : List ⟨'elem⟩) : 'elem =
  match list with
  | []      → raise Head
  | x :: _ → x
```

Die Deklaration ***exception*** führt eine neue Ausnahme ein; der Datentyp *Exception* wird so um ein Element erweitert. Die neue Ausnahme wird geworfen, wenn *head* mit der leeren Liste aufgerufen wird; der Programmtext dokumentiert auf diese Weise, dass *head* diesen Fall nicht behandeln kann (oder will).

module Effects. Bank

Ausnahmen wie *Div* und *Head* können wie Konstruktoren verwendet werden. Wie Konstruktoren können Ausnahmen auch ein Argument haben, zum Beispiel um Informationen vom Ort des Abbruchs zum Ort der Wiederaufnahme zu transportieren. Die folgende Re-Implementierung des Bankkontos illustriert diese Möglichkeit.

```
exception Insufficient of Nat
module Junior-Account =
  let private min-funds = 10
  let private funds     = ref 0
  let deposit (amount : Nat) =
      funds := !funds + amount
  let withdraw (amount : Nat) =
    if !funds − amount ⩾ min-funds then
      funds := !funds − amount
    else
      raise (Insufficient ( !funds − min-funds))
  let balance () =
      !funds
```

Die Funktion *withdraw* ist abgeändert worden: Der gewünschte Betrag wird nur abgebucht, wenn das Konto gedeckt ist — ein Betrag von 10 € muss auf dem Konto verbleiben. Anderenfalls wird die Ausnahme *Insufficient n* geworfen, wobei *n* der Betrag ist, der höchstens abgehoben werden kann. Das ursprüngliche Verhalten von *withdraw* — soviel abheben wie möglich — lässt sich nachprogrammieren, indem man die geworfene Ausnahme fängt und dann das Konto leerräumt.

```
let maximal-withdraw (amount : Nat) : Nat =
  try
    Junior-Account.withdraw amount; amount
  with
  | Insufficient rest → Junior-Account.withdraw rest; rest
```

7.4.1. Abstrakte Syntax

Wir erweitern Deklarationen um Ausnahmedeklarationen und Ausdrücke um Konstrukte zum Werfen und Fangen von Ausnahmen.

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***exception*** C ***of*** t	Deklaration einer Ausnahme

Wird bei einer ***exception*** Deklaration kein Typargument angegeben, dann fassen wir das, wie bei „normalen“ Konstruktoren, als Abkürzung für C ***of*** *Unit* auf.

$e ::= \cdots$	*Ausnahmebehandlung:*
$\mid$ ***raise*** e	Werfen einer Ausnahme
$\mid$ ***try*** e ***with*** m	Fangen einer Ausnahme

Syntaktisch entspricht ***try*** e ***with*** m einer erweiterten Fallunterscheidung mit ***match***: e ist ein Ausdruck und m eine Folge von Regeln der Form $p \to m$, siehe Abschnitt 4.2.3.

7.4.2. Statische Semantik

Ausnahmen haben den Typ *Exception*.

$t ::= \cdots$	*Typen:*
$\mid$ *Exception*	Typ der Ausnahmen

Der Typ entspricht im Wesentlichen einem Variantentyp mit dem Unterschied, dass die Konstruktoren nicht sofort, sondern peu à peu mit Hilfe von ***exception*** Deklarationen eingeführt werden.

Die Ausnahmedeklaration ***exception*** C ***of*** t wird ähnlich wie eine Variantentypdeklaration gehandhabt: Der Ausnahmekonstruktor korrespondiert zu einer Funktion vom Typ $t \to$ *Exception*. Wir erlauben es *nicht*, Ausnahmekonstruktoren zu redefinieren oder lokal zu definieren — aus den gleichen Gründen, die die Redefinition von Record- und Variantentypen verbieten, siehe Abschnitt 4.1.

$$\frac{\Sigma \vdash e : \mathit{Exception}}{\Sigma \vdash \textbf{\textit{raise}}\ e : t}$$

Das Argument von ***raise*** muss ein Ausdruck sein, der zu einer Ausnahme auswertet; ***raise*** e selbst hat einen beliebigen Typ! Mit anderen Worten, ***raise*** e kann überall verwendet werden, als Boolescher Wert (***raise*** *Div* `&&` e), als natürliche Zahl (***raise*** *Div* $+$ 4711), als Funktion ((***raise*** *Div*) 4711), als Paar (*snd* (***raise*** *Div*)) usw. Warum? Nun, ***raise*** e bricht die aktuelle Rechnung ab, deswegen kann der Ausdruck in jedem beliebigen Kontext stehen; der Kontext bekommt den Wert von ***raise*** e ja niemals zu Gesicht.

$$\frac{\Sigma \vdash e : t \qquad \Sigma \vdash m\ (\mathit{Exception}) : t}{\Sigma \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m : t}$$

Der Rumpf des ***try***-Ausdrucks muss, angewendet auf eine Ausnahme, zu einem Element des Typs t auswerten, wobei t der Typ des Ausdrucks ist, der „ausprobiert“ wird.

7.4.3. Dynamische Semantik

Wie auch in den letzten beiden Abschnitten müssen wir die Auswertungsregeln etwas abändern. Dies führt uns zunächst zu der Frage, zu welchem Wert zum Beispiel der Ausdruck ***raise*** *Div* auswertet. Der Ausdruck hat sicherlich keinen „normalen“ Wert; er wirft ja die Ausnahme *Div*. Diese Ausnahme können wir als „außergewöhnlichen“ Wert des Ausdrucks ansehen. Mit der Einführung von Ausnahmen kann die Auswertung eines Ausdrucks zwei mögliche Resultate produzieren: normale Werte und Ausnahmen. Um im Bilde zu bleiben, nennt man die geworfenen Ausnahmen auch *Pakete*.

$r \in \mathsf{Result} ::=$		*Resultate*
	ν	Wert
$\mid$	$\boxed{\nu}$	Paket

Das Kästchen $\boxed{\nu}$ soll verdeutlichen, dass es sich um ein Paket handelt. Der Inhalt des Pakets, zum Beispiel *Div* oder *Insufficient* 4711, ist ein normaler Wert — statisch gesehen ein Element des Typs *Exception*. Die Auswertungsrelation $\delta \vdash e \Downarrow \nu$ wird nun abgeändert zu

$$\delta \vdash e \Downarrow r$$

um der Tatsache Rechnung zu tragen, dass eine Auswertung zwei mögliche Resultate haben kann. Das ist die dritte und letzte Änderung der Auswertungsrelation. Wenn wir es genau nehmen, müssten wir natürlich die geänderte Relation ändern: Aus $\delta \vdash e \Downarrow_t \nu$ wird

$$\delta \vdash e \Downarrow_t r$$

bzw. eigentlich müssten wir die geänderte geänderte Relation ändern: Aus der Formel $\delta \vdash \sigma \parallel e \Downarrow_t \nu \parallel \sigma'$ wird

$$\delta \vdash \sigma \parallel e \Downarrow_t r \parallel \sigma'$$

Glücklicherweise kommen sich die jeweiligen Erweiterungen, Ein- und Ausgabe, Zustand und Ausnahmen, nicht allzu sehr ins Gehege, so dass wir jede Erweiterung separat betrachten können und dies aus Gründen der Lesbarkeit auch tun.

Kommen wir zu den Auswertungsregeln: Für ***raise*** gibt es überraschenderweise zwei Regeln.

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash \textbf{\textit{raise}}\ e \Downarrow \boxed{\nu}} \qquad \frac{\delta \vdash e \Downarrow \boxed{\nu}}{\delta \vdash \textbf{\textit{raise}}\ e \Downarrow \boxed{\nu}}$$

Wie immer wird zunächst das Argument ausgewertet. Die Auswertung von e kann zwei mögliche Ergebnisse haben: einen Wert oder ein Paket. Wenn e zum dem Wert ν auswertet, schnürt ***raise*** daraus ein Paket $\boxed{\nu}$ und wirft es. Wenn bei der Auswertung von e aber

bereits ein Paket geworfen wird, so wirft ***raise*** dieses Paket weiter. (Werte werden niemals doppelt verpackt: Ein Ergebnis der Form $\boxed{\nu}$ gibt es *nicht*.) Zwei künstlich konstruierte Beispiele für den letzten Fall sind ***raise*** (***raise*** *Div*) oder ***raise*** (***raise*** *Div*; *Match*). Beide Ausdrücke sind äquivalent zu ***raise*** *Div*.

$$\dfrac{\dfrac{}{\varnothing \vdash \mathit{Div} \Downarrow \mathit{Div}}}{\dfrac{\varnothing \vdash \textbf{\textit{raise}}\ \mathit{Div} \Downarrow \boxed{\mathit{Div}}}{\varnothing \vdash \textbf{\textit{raise}}\ (\textbf{\textit{raise}}\ \mathit{Div}) \Downarrow \boxed{\mathit{Div}}}}$$

Sie sind äquivalent, weil beide bzw. alle drei Ausdrücke das gleiche Ergebnis liefern.

Kommen wir zum Fangen von Paketen.

$$\dfrac{\delta \vdash e \Downarrow \nu}{\delta \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m \Downarrow \nu}$$

$$\dfrac{\delta \vdash e \Downarrow \boxed{\nu} \qquad m\ (\nu) \Downarrow r}{\delta \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m \Downarrow r} \qquad \dfrac{\delta \vdash e \Downarrow \boxed{\nu} \qquad \delta \vdash m\ (\nu) \Downarrow \lightning}{\delta \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m \Downarrow \boxed{\nu}}$$

Zunächst wird e ausgewertet („die Auswertung wird versucht“); resultiert die Auswertung in einem Wert, so ist dieser auch der Wert des gesamten Ausdrucks („der Versuch ist erfolgreich“). Wird bei der Auswertung das Paket $\boxed{\nu}$ geworfen („die Auswertung im bisherigen Sinne misslingt“), so fängt ***try*** das Paket auf und packt es aus. Jetzt sind zwei bzw. eigentlich sogar drei Fälle denkbar: ν passt auf die linke Seite einer der in m aufgeführten Regeln, dann wird die entsprechende rechte Seite ausgerechnet. Das Ergebnis dieser Rechnung ist auch das Ergebnis des gesamten Ausdrucks — das schließt ausdrücklich Ausnahmen ein, die bei der Abarbeitung geworfen werden. Passt ν hingegen nicht auf eine der aufgeführten Regeln, $m\ (\nu) \Downarrow \lightning$, dann wird das Paket weitergeworfen. Es gibt also insgesamt vier unterschiedliche Konstellationen:

$$\begin{array}{llll}
\textbf{\textit{try}}\ 4711 & \textbf{\textit{with}} \mid \mathit{Div} & \to 815 & \Downarrow 4711 \\
\textbf{\textit{try}}\ 4711 \div 0 & \textbf{\textit{with}} \mid \mathit{Div} & \to 815 & \Downarrow 815 \\
\textbf{\textit{try}}\ 4711 \div 0 & \textbf{\textit{with}} \mid \mathit{Div} & \to \textbf{\textit{raise}}\ \mathit{Match} & \Downarrow \boxed{\mathit{Match}} \\
\textbf{\textit{try}}\ 4711 \div 0 & \textbf{\textit{with}} \mid \mathit{Match} & \to 815 & \Downarrow \boxed{\mathit{Div}}
\end{array}$$

Im ersten Beispiel wertet der von ***try*** und ***with*** eingeschlossene Ausdruck zu einem Wert aus; in den anderen drei Beispielen wird eine Ausnahme geworfen. Diese Ausnahme wird im zweiten und dritten Beispiel abgefangen, im vierten nicht. Die Behandlung der Ausnahme führt im zweiten Beispiel zu einem Wert, im dritten Beispiel wird eine Ausnahme geworfen.

Wie in den Abschnitten 7.1 und 7.2 müssen wir auch in diesem Abschnitt die bisher aufgestellten Auswertungsregeln anpassen. Aber nicht nur das, jetzt haben wir endlich die Mittel in der Hand, um die fehlenden Auswertungsregeln nachzutragen. Erinnern wir uns: Wir haben Ausnahmen insbesondere eingeführt, um die Semantik von Ausdrücken wie $1 \div 0$, ***match*** $[\,]$ ***with*** $x :: xs \to 4711$ oder $[\lvert 1; 2 \rvert].[4711]$ festlegen zu können. Fangen wir mit den arithmetischen Operationen an.

$$\dfrac{\delta \vdash e_2 \Downarrow 0}{\delta \vdash e_1 \div e_2 \Downarrow \boxed{\mathit{Div}}} \qquad \dfrac{\delta \vdash e_2 \Downarrow 0}{\delta \vdash e_1\ \%\ e_2 \Downarrow \boxed{\mathit{Mod}}}$$

Wertet der Divisor zu 0 aus, wird eine entsprechende Ausnahme geworfen.

$$\frac{\delta \vdash e \Downarrow \nu \qquad \delta \vdash m\,(\nu) \Downarrow \lightning}{\delta \vdash \textbf{\textit{match}}\ e\ \textbf{\textit{with}}\ m \Downarrow \boxed{\mathit{Match}}}$$

Passt keine der Regeln auf den Wert des Diskriminatorausdrucks, wird eine *Match* Ausnahme geworfen.

$$\frac{\delta \vdash e_1 \Downarrow s \qquad \delta \vdash e_2 \Downarrow i}{\delta \vdash e_1.[e_2] \Downarrow \boxed{\mathit{Subscript}}} \quad i \notin \mathit{dom}\ s$$

Ist der Index i nicht im Definitionsbereich des Arrays, dann wird eine *Subscript* Ausnahme geworfen. Die folgenden Ausnahmen sind vordefiniert:[7]

```
exception Div
exception Mod
exception Match
exception Subscript
```

Kommen wir auf die Änderungen an den bestehenden Regeln zu sprechen, zunächst exemplarisch an der Paarbildung. Die folgenden beiden Regeln kommen zu der ursprünglichen *hinzu*:

$$\frac{\delta \vdash e_1 \Downarrow \boxed{\nu}}{\delta \vdash (e_1, e_2) \Downarrow \boxed{\nu}} \qquad \frac{\delta \vdash e_1 \Downarrow \nu_1 \qquad \delta \vdash e_2 \Downarrow \boxed{\nu}}{\delta \vdash (e_1, e_2) \Downarrow \boxed{\nu}}$$

Die Rechnung wird abgebrochen, wenn eine der beiden Komponenten eine Ausnahme wirft. Diese Ausnahme ist auch das Ergebnis des Paarausdrucks. Im Allgemeinen müssen wir für eine Regel mit n Voraussetzungen n zusätzliche Regeln angeben. Die Regel

$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \qquad \delta_2 \vdash e_2 \Downarrow \nu_2 \qquad \cdots \qquad \delta_n \vdash e_n \Downarrow \nu_n}{\delta \vdash e \Downarrow \nu}$$

wird um die folgenden n Regeln *ergänzt*:

$$\frac{\delta_1 \vdash e_1 \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$

$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \qquad \delta_2 \vdash e_2 \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$

$$\vdots$$

[7] Dichtung und Wahrheit: In F# wird bei einem ungültigen Arrayzugriff tatsächlich die Ausnahme *IndexOutOfRangeException* geworfen. Diese Ausnahme ist ein sogenanntes Objekt, eine Instanz der gleichnamigen Klasse *IndexOutOfRangeException*, einem Untertyp von *Exception*, siehe Kapitel 8. Möchte man diese Ausnahme behandeln, schreibt man tatsächlich statt ... ***with*** | *Subscript* → *e* etwas länglicher ... ***with*** | :? *System.IndexOutOfRangeException* → *e*. Auch *Match* hat einen etwas längeren Namen, *Microsoft.FSharp.Core.MatchFailureException*, und erwartet in Wirklichkeit drei Argumente.

$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \qquad \delta_2 \vdash e_2 \Downarrow \nu_2 \qquad \cdots \qquad \delta_n \vdash e_n \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$

Jeder Teilausdruck kann eine Ausnahme werfen, die dann auch das Ergebnis der Rechnung darstellt. Auf diese Weise wird eine Ausnahme bis zum nächsten umgebenden ***try***-Ausdruck propagiert. Dabei können beliebig viele Teilrechnungen abgebrochen werden, etwa wenn der Wurf einer Ausnahme in einer rekursiven Funktion erfolgt.

7.4.4. Vertiefung

Panik Nicht immer lässt sich eine abgebrochene Rechnung sinnvoll weiterführen; manchmal ist sie schlicht und einfach Symptom eines Programmierfehlers. Unter anderem für die Dokumentation dieser „unmöglichen Fälle“ ist die in Mini–F# vordefinierte Ausnahme *Panic* gedacht.

```
exception Panic of String
let panic (message : String) : 'a =
  raise (Panic message)
```

Die Funktion *panic* hat den Ergebnistyp $'a$ und zeigt damit an, dass sie keinen Wert zurückgibt. (Ein Ausdruck vom Typ $'a$ kann in jedem Kontext verwendet werden, etwa als natürliche Zahl oder als Wahrheitswert.)

Wenn man möchte, kann man *Panic* Ausnahmen abfangen — am besten mit einem ***try***-Ausdruck um das Hauptprogramm — und der Benutzer*in mit der Bitte um einen Fehlerbericht präsentieren.

```
try
  ...
with
| Panic msg →
  putline ("panic! (the 'impossible' happened)\n" ^ msg ^
           "\nPlease report it as a bug to support@harry-hacker.org.")
```

module Effects. Product

Ausnahmen als Kontrollstruktur Der Begriff Ausnahme suggeriert, dass die Konstrukte ***raise*** und ***try*** tatsächlich nur in Ausnahmesituationen verwendet werden sollten. Das ist zwar prinzipiell nicht ganz falsch, schöpft aber das Potential der Konstrukte nicht aus. Wir können ***raise*** und ***try*** auch gezielt einsetzen, um die Auswertung zu steuern. Nehmen wir an, wir wollen das Produkt einer Liste von Zahlen bestimmen. Mit dem Struktur Entwurfsmuster erhalten wir die folgende, uns wohlbekannte Definition.

```
let rec product (list : List ⟨Nat⟩) : Nat =
  match list with
    | []      → 1
    | n :: ns → n * product ns
```

Hat die Eingabeliste die Länge n, so werden n Produkte berechnet, unabhängig vom Wert der Listenelemente. Enthält die Liste irgendwo eine Null, dann wird viel unnötige Arbeit verrichtet. Die folgende Implementierung vermeidet diese Arbeit: Die Funktion *product* terminiert unmittelbar mit dem Ergebnis 0, sobald sie auf eine 0 trifft.

```
exception Zero
let product (list : List ⟨Nat⟩) : Nat =
  try
    let rec worker = function
      | []      → 1
      | n :: ns → if n = 0 then raise Zero else n * worker ns
    in worker list
  with
  | Zero → 0
```

Wenn im Rumpf der Hilfsfunktion *worker* die Ausnahme *Zero* geworfen wird, werden sämtliche rekursiven Aufrufe abgebrochen und es wird mit der rechten Seite der Regel $Zero \to 0$ weitergerechnet. Wir springen sozusagen aus der Rekursion heraus.

module Effects.Input2

Eingabe mit Validierung, da capo Themenwechsel: Mit Hilfe von Ausnahmen können wir *einfache* Parser programmieren, hier vorgeführt an der Funktion *read-nat*, die einen String in eine natürliche Zahl überführt.

```
exception Read
let read-nat (s : String) : Nat =
  let rec nat = function
    | []      → 0
    | c :: cs → if Char.IsDigit c then
                  Nat.ord c − Nat.ord '0' + 10 * nat cs
                else
                  raise Read
  if s = "" then raise Read
            else  nat (List.rev (explode s))
```

Ist die Syntaxanalyse erfolgreich, wird der semantische Wert direkt zurückgegeben; im Fall eines Syntaxfehlers wird eine Ausnahme geworfen: *read-nat* beklagt sich, wenn der String leer ist oder wenn andere Zeichen als Ziffern vorkommen. (Mit Hilfe der Funktion *explode* wird eine Zeichenkette in eine Liste von Zeichen überführt; diese wird mit *rev* gespiegelt, so dass die niederwertigste Ziffer an den Anfang kommt; *Nat.ord* bildet ein Zeichen auf seinen Zeichencode ab.) Die Funktion *read-nat* können wir zum Beispiel verwenden, um den Validator *is-nat* aus Abschnitt 7.1 etwas kürzer zu definieren.

```
let is-nat (s : String) : Result ⟨Nat⟩ =
  try
    Okay (read-nat s)
  with
  | Read → Error "natural number expected"
```

Die eventuell von *read-nat* geworfene Ausnahme wird gefangen und in ein Element des Datentyps *Result* überführt.

Der Datentyp *Result* mit seinen Konstruktoren *Okay* und *Error* ist unserer Modellierung von Ausnahmen mit den möglichen Resultaten ν und $\boxed{\nu}$ sehr ähnlich: *Okay value* und ν stehen für Ergebnisse geglückter Rechnungen, *Error msg* und $\boxed{\nu}$ signalisieren Fehler. In der Tat sind ***try*** und ***raise*** in gewisser Hinsicht überflüssig: Alle Effekte lassen sich unter Verwendung des Datentyps *Result* nachprogrammieren! Umgekehrt lässt sich der Datentyp *Result* durch Ausnahmen ersetzen. Welchen Ansatz man wählt, ist zum Teil eine Frage des guten Geschmacks und zum Teil eine Frage der Bequemlichkeit. Machen wir die Unterschiede explizit, indem wir die Validatoren aus Abschnitt 7.1 noch einmal mit Ausnahmen nachprogrammieren. Erinnern wir uns: Ein Validator hat den Typ $t_1 \to \mathit{Result}\,\langle t_2\rangle$. Wenn wir statt *Result* Ausnahmen verwenden, können wir den Typ zu $t_1 \to t_2$ vereinfachen. Eine fehlerhafte Eingabe wird jetzt durch das Werfen einer Ausnahme signalisiert.[8] Wir missbrauchen die oben eingeführte *Panic* Ausnahme zu diesem Zweck. Damit sieht die Definition von *checked-query* wie folgt aus:

```
let rec checked-query (prompt : String, check : String → 'a) : 'a =
  try
    check (query (prompt ^ ": "))
  with
  | Panic msg → putline ("*** " ^ msg); checked-query (prompt, check)
```

Die Fallunterscheidung mit ***match*** ist einem ***try***-Ausdruck gewichen.

Die Validatoren haben, wie gesagt, einen einfacheren Typ: Sie geben entweder den semantischen Wert zurück oder werfen eine *Error* Ausnahme.

```
let is-nat (s : String) : Nat =
  try readnat s with
  | Read → panic "natural number expected"
let is-less (n : Nat) : Nat → Nat =
  fun m → if m < n
          then m
          else  panic ("number must be less than " ^ show n)
```

Die Funktion *both* ist jetzt schlicht und einfach die Vorwärtskomposition von Funktionen. (Die übliche Komposition $f \cdot g$ mit $(f \cdot g)(x) = f(g(x))$ komponiert Funktionen rückwärts: Erst wird g auf x angewendet, dann f auf das Ergebnis.)

```
let both (first : 'a → 'b, second : 'b → 'c) : 'a → 'c =
  fun x → second (first x)
```

[8] Interessanterweise spiegelt sich die Tatsache, dass ein Validator eine Ausnahme werfen kann, nicht im Typ wider. Man könnte sich durchaus vorstellen, diesen Effekt dort zu vermerken, zum Beispiel in der Form $t_1 \to t_2$ **throws** *Panic*. Einen solchen Ansatz verwendet die Sprache *Java* — das sogenannte „Catch or Specify Requirement“ verlangt, dass eine Ausnahme entweder behandelt wird oder dass im Typ vermerkt wird, dass sie *möglicherweise* geworfen wird.

(Zur Erinnerung: Die Vorwärtskomposition ist auch als Infixoperator vordefiniert: $f \gg g$. *Lies:* f dann g — der „Pfeil“ $\gg$ zeigt an, dass das Ergebnis von f in die Funktion g eingespeist wird.)

Für den Anwender von *checked-query* ändert sich nichts; die Aufrufe funktionieren unverändert:

```
Mini⟩ checked-query ("age", both (is-nat, is-less 123))
age : 41
41
```

Kommen wir zu den Vor- und Nachteilen. Verwendet man den Datentyp *Result*, dann macht der Typ von Ausdrücken und Funktionen explizit, wer „Ausnahmen wirft“ und wer nicht. Das ist zunächst ein Vorteil: Man kann nicht vergessen, Ausnahmen zu behandeln, und man kommt nicht in Versuchung, Ausnahmen zu fangen, die gar nicht geworfen werden. Die Tatsache, dass alles explizit gemacht wird, ist gleichzeitig aber auch ein Nachteil: Im Prinzip müssen die Auswertungsregeln für Ausnahmen nachprogrammiert werden. Um zum Beispiel ein Paar vom Typ $Result\,\langle t_1 * t_2\rangle$ aus zwei Ausdrücken e_i vom Typ $Result\,\langle \tau_i\rangle$ zu konstruieren — beide Ausdrücke können Ausnahmen werfen —, muss man die drei Paarregeln in Programmcode überführen.

```
match e1 with
| Okay v1 →
    match e2 with
    | Okay v2    → Okay (v1, v2)
    | Error msg → Error msg
| Error msg → Error msg
```

Jeder der drei Zweige korrespondiert zu einer Auswertungsregel. Zum Vergleich: Verwendet man Ausnahmen, programmiert man einfach (e_1, e_2).

Fassen wir zusammen: Variantentypen wie *Result* oder *Option* dienen dem gleichen Zweck wie Ausnahmen. Ausnahmen sind fest in Mini–F# verdrahtet, aber nicht unbedingt notwendig: Mit Fleiß und Ausdauer können wir Ausnahmen mit Hilfe von Variantentypen programmieren. Kurzum: Einmal formalisieren wir Ausnahmen in der Objektsprache (mit den Mitteln von Mini–F#), einmal in der Metasprache (in der dynamischen Semantik mit Hilfe von Auswertungsregeln).

module
Effects.
UPNCalculator

Mini-Projekt: interaktiver Taschenrechner Lösen wir zum Schluss des Kapitels noch eine letzte Aufgabe, die Programmierung eines interaktiven Taschenrechners. Wir stellen zwei verschiedene Varianten vor: einen UPN-Rechner und einen Mini^2-F# Rechner. Fangen wir mit dem altmodischen Gerät an: *UPN* steht für Umgekehrte Polnische Notation, ein Synonym für Postfixnotation, siehe Abschnitt 6.3. Schauen wir uns eine kurze Interaktion an:

```
UPN⟩ 4711
UPN⟩ 815
UPN⟩ 2765
UPN⟩ *
UPN⟩ +
UPN⟩ .
2258186
```

Der Postfix-Ausdruck wird Zeile für Zeile eingegeben: 4711 815 2765 ∗ + entspricht dem Infix-Ausdruck 4711+815∗2765. Ausdrücke in Postfixnotation lassen sich besonders leicht ausrechnen: Wir merken uns die eingegebenen Zahlen; jede Operation ersetzt die letzten beiden Zahlen durch das Ergebnis. Die obigen Eingaben lassen sich wie folgt abarbeiten:

4711	4711
815	4711 815
2765	4711 815 2765
∗	4711 2253475
+	2258186

Die Liste der Zahlen nennt man auch *Stack* oder *Stapel*: Optisch muss man die Listen um 90° nach links drehen. Damit wird jede eingegebene Zahl oben auf dem Stapel abgelegt und eine Operation nimmt die beiden obersten Elemente vom Stapel und legt das Ergebnis dort wieder ab.

Einen Stapel, das Gedächtnis des UPN-Rechners, können wir durch eine Speicherzelle repräsentieren, die eine Liste von Zahlen enthält.

```
exception Pop
exception Top
module Stack =
  let private stack = ref []
  let push (x : Nat) =
    stack := x :: !stack
  let pop () : Nat =
    match !stack with
    | []      → raise Pop
    | x :: xs → stack := xs; x
  let top () : Nat =
    match !stack with
    | []      → raise Top
    | x :: xs → x
```

Die Funktion *push* legt ein Element auf dem Stapel ab, *pop* entfernt ein Element und *top* inspiziert das oberste Element. Die letzten beiden Funktionen werfen gleichnamige Ausnahmen, wenn der Stapel leer ist. Die Speicherzelle, auf der die Funktionen arbeiten, ist wie immer gekapselt, also nur lokal sichtbar.

Der UPN-Taschenrechner implementiert ähnlich wie der Mini–F# Interpreter eine einfache Kommandozeile: Ein Prompt wird ausgegeben, ein Kommando eingelesen und anschließend ausgeführt.

```
let rec upn-calculator () =
  try
    match query "UPN > " with
    | "+" → Stack.push (Stack.pop () + Stack.pop ())
    | "*" → Stack.push (Stack.pop () * Stack.pop ())
    | "." → print (Stack.top ())
    | s   → Stack.push (read-nat s)
  with
  | Pop | Top → putline "stack is empty"
  | Read      → putline "enter a number or an operator"
  upn-calculator ()
```

Typisch für einen Kommandozeileninterpreter ist der *endrekursive* Aufruf am Ende: Der Interpreter terminiert nicht, zumindest nicht auf den ersten Blick. Das Verhalten einschließlich der Terminierung wird von der Benutzer*in gesteuert. Signalisiert sie das Ende der Eingabe[9], dann wirft *query* eine *EOF* Ausnahme. Diese Ausnahme wird im Gegensatz zu *Pop*, *Top* und *Read* nicht abgefangen und führt damit zum Programmabbruch. Die Erweiterung des Taschenrechners in Abbildung 7.5 fängt auch diese Ausnahme und verabschiedet sich von der Benutzer*in. Auch die Eingabe von `quit` oder Synonymen terminiert den Taschenrechner.

module Effects. Mini2

Kommen wir zum Mini^2-F# Rechner, einem Taschenrechner, der Infixnotation unterstützt. Wie man sieht, simuliert der Taschenrechner den Mini–F# Interpreter.

```
Mini-Mini⟩  4711 + 815 * 2765
2258186
Mini-Mini⟩  1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
55
Mini-Mini⟩  1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10
3628800
Mini-Mini⟩  Hello, world !
lexical error : illegal character
Mini-Mini⟩  4711 815 2765 * +
syntax error
```

Die Implementierung, siehe Abbildung 7.6, ist etwas kürzer als die des UPN-Taschenrechners, da wir auf die Vorarbeiten aus Kapitel 6 zurückgreifen können. So verwenden wir die Funktion *abstract-syntax-tree*, um den eingegebenen String in einen abstrakten Syntaxbaum zu überführen und *evaluate* um den Baum auszuwerten. Zwei Arten von Fehlern

[9] Unter Linux und UNIX wird das Ende der Eingabe vermittels der Tastenkombination Control-D, respektive Strg-D auf einer deutschen Tastatur, signalisiert; unter Microsofts DOS verwendet man Control-Z respektive Strg-Z.

```
let upn-calculator () =
  let rec read-eval-print-loop () =
    try
      match query "UPN > " with
       | ""  → ()
       | "+" → Stack.push (Stack.pop () + Stack.pop ())
       | "*" → Stack.push (Stack.pop () * Stack.pop ())
       | "." → print (Stack.top ())
       | "exit" | "halt" | "q" | "quit" | "stop"
             → raise EOF
       | s   → Stack.push (read-nat s)
    with
     | Pop | Top → putline "stack is empty"
     | Read      → putline "enter a number or an operator"
    read-eval-print-loop ()
  in
    try
      read-eval-print-loop ()
    with
     | EOF → putline "bye bye"
```

Abbildung 7.5.: Ein UPN-Taschenrechner.

```
let rec read-eval-print-loop () =
  try
    let s = query "Mini-Mini > "
    if s = "" then
      ()
    elif contains s ["exit"; "halt"; "q"; "quit"; "stop"] then
      raise EOF
    else
      match abstract-syntax-tree s with
      | None   → putline "syntax error"
      | Some e → print (evaluate e)
  with
  | Panic s  → putline ("lexical error: " ^ s)
  read-eval-print-loop ()
let mini2 () =
  try
    read-eval-print-loop ()
  with
  | EOF → putline "bye bye"
```

Abbildung 7.6.: Ein Taschenrechner (Mini^2-F# in Mini–F#).

werden erkannt: lexikalische Fehler und Syntaxfehler. Der Scanner aus Abschnitt 6.2.2 wirft eine *Panic* Ausnahme, wenn ein unzulässiges Zeichen auftritt. Dieser Fehler wird mit einem ***try***-Ausdruck abgefangen. Ein Syntaxfehler liegt vor, wenn der Parser den Wert *None* zurückgibt. Dieser Fehler wird mit einem ***match***-Ausdruck abgefangen. Der Mini^2-F# Interpreter realisiert eine echte „read-eval-print loop“: Ein Ausdruck wird eingelesen, der Ausdruck wird ausgewertet und das Ergebnis wird ausgegeben. Der Mini–F# Interpreter selbst folgt dem gleichen Schema, nur dass die einzelnen Phasen etwas aufwändiger sind, zum Beispiel wird der abstrakte Syntaxbaum erst typgeprüft, dann wird der Ausdruck in eine einfachere Sprache übersetzt und erst dann erfolgt die eigentliche Auswertung.

Übungen.

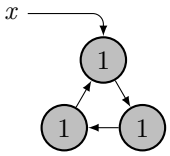
1. Schreiben Sie Funktionen, um eine Liste von natürlichen Zahlen auszugeben bzw. einzulesen. Wie lassen sich die Funktionen auf beliebige Listen verallgemeinern?

2. Ein Array lässt sich in linearer Zeit in eine Liste überführen. Die Konvertierung einer Liste in ein Array ist schwieriger: Mit den Mitteln von Kapitel 3 oder 4 lässt sich dies nur in quadratischer Zeit bewerkstelligen. Welche in diesem Kapitel eingeführten Konzepte benötigt man, um die Konvertierung in linearer Zeit zu schaffen? Geben Sie eine Implementierung an.

3. In Abschnitt 5.2.1 haben wir Funktionen für die Verwaltung von Personaldaten implementiert. Schreiben Sie ein interaktives Programm, um Personaldaten einzugeben und Personaldaten herauszusuchen. Verwenden Sie *readFromFile* und *writeToFile*, um die Stammdaten über eine interaktive Sitzung hinaus persistent zu speichern.

4. In Abschnitt 7.2.5 haben wir Sequenzen, endliche Folgen von Elementen, mit Hilfe von Perlenketten implementiert. Definieren Sie eine Funktion, die die Länge einer Perlenkette bestimmt. Die unten abgebildete Perlenkette x hat zum Beispiel die Länge 3. Für die Lösung dieser Aufgabe benötigen Sie noch zusätzliches Hintergrundwissen. Perlenketten werden tatsächlich durch die folgende Typdefinition eingeführt.

```
[< ReferenceEquality >]
type Necklace ⟨'elem⟩ =
  { bead          : 'elem
    mutable next  : Necklace ⟨'elem⟩ }
```



Das sogenannte *Attribut* (engl. attribute) [< *ReferenceEquality* >] instruiert den F# Übersetzer den Test auf Gleichheit nicht durch *Wertegleichheit* (engl. *value equality*), sondern durch *Verweisgleichheit* (engl. *reference equality*) zu implementieren. Die folgende Interaktion illustriert die Unterschiede.

```
Mini⟩ let x = single 4711
Mini⟩ let y = single 4711
Mini⟩ x = y
false
Mini⟩ x = x
true
```

Die Bezeichner x und y haben zwar den gleichen Wert, sie sind aber nicht identisch — im Hauptspeicher werden sie unter zwei unterschiedlichen Adressen geführt. In der obigen Grafik sind alle drei Perlen wertegleich, aber nicht verweisgleich, nicht identisch. Verweisgleichheit ist

die diskrimierenste Gleichheit: Sind zwei Ausdrücke verweisgleich, dann sind sie auch wertegleich; die Umkehrung gilt nicht, wie die Interaktion demonstriert. Um die Länge einer Perlenkette zu bestimmen, muss die Möglichkeit bestehen, Perlen auf Identität zu testen. (Nachdenken!) Entfernen wir übrigens das Attribut [< *ReferenceEquality* >], dann folgt dem Aufruf $x = y$ eine lange Stille, die Auswertung terminiert nicht. (Warum?)

5. Ein Sortierverfahren heißt *stabil*, wenn es die relative Reihenfolge von äquivalenten Elemente nicht verändert, siehe Aufgabe 5.2. Zeigen Sie, dass die imperativen Varianten von Sortieren durch Einfügen und Sortieren durch Auswählen aus Abschnitt 7.3.2 stabil sind.

6. Die Funktion *product* berechnet das „Produkt“ der Elemente eines Binärbaums.

```
let rec product = function
  | Leaf           → e
  | Node (l, x, r) → product l ⊗ x ⊗ product r
```

Von der Operation '⊗' wissen wir lediglich, dass sie assoziativ mit e als neutralem Element ist. (Insbesondere dürfen wir nicht annehmen, dass '⊗' kommutativ ist.) Programmieren Sie eine endrekursive Variante von *product*. *Hinweis:* Eine Idee ist, die in Abschnitt 7.3.3 eingeführte Metapher der Todo-Liste konsequent weiterzuverfolgen, indem man erlaubt, verschiedene „Items“ auf die Todo-Liste zu setzen.

```
type Item =
  | Mul of Nat
  | Sub of Tree ⟨Nat⟩
type Todo =
  Item list
```

Orientieren Sie sich bei der Lösung der Aufgabe an der Funktion *product-acc-many*.

7. Definieren Sie die Funktionen,

```
foreach-list  (list  : List ⟨'elem⟩  ) (body : 'elem → Unit) : Unit
foreach-array (array : Array ⟨'elem⟩) (body : 'elem → Unit) : Unit
```

die über eine Liste bzw. ein Array iterieren, mit den Mitteln aus Kapitel 3 und 4. Übersetzen Sie die Schleifen *for x in list do e* und *for x in array do e* in Aufrufe dieser Funktionen.

8. Erweitern Sie den UPN-Taschenrechner um weitere arithmetische Funktionen wie Subtraktion und Division. Fangen Sie die Division durch Null ab und machen Sie die Benutzer*in auf den Fehler aufmerksam.

8. Objekte\Rechnen im Großen

I made up the term 'object-oriented', and
I can tell you I didn't have C++ in mind.
— Alan Kay, *OOPSLA '97*

Object oriented programming is, in some sense, just a programming trick using indirection. It's a trick good programmers have been using for years.
— Bjarne Stroustrup

There are only two things wrong with C++.
The initial concept and the implementation.
— Bertrand Meyer

In den vorangegangenen Kapiteln haben wir uns mit der *Programmierung im Kleinen* beschäftigt, mit Algorithmen und Datenstrukturen. Wie werden Rechenregeln aufgestellt? Wie lassen sich Daten und Informationen repräsentieren? Wie kann ein Programm mit der Umwelt interagieren? Wie kann man Rechenregeln mit einem Gedächtnis ausstatten? Wie kann man mit Ausnahmesituationen umgehen?

In diesem, dem letzten und damit abschließenden Kapitel wenden wir uns der *Programmierung im Großen* zu und diskutieren, wie sich Rechenregeln zu größeren konzeptionellen Einheiten zusammenfassen lassen, wie diese Einheiten ihrerseits organisiert und komponiert werden können.

Es gibt im Wesentlichen zwei Mechanismen, um größere Programme oder Softwaresysteme zu strukturieren:

- *Modulsysteme* und
- *Objektsysteme*.

Grob gesprochen ist ein Objekt eine homogene Sammlung von Operationen, während ein Modul eine inhomogene Sammlung von Werten, Typen und anderen Modulen ist. Die beiden Mechanismen stehen in einer gewissen Konkurrenzsituation. Die meisten Programmiersprachen bieten entweder ein ausgefeiltes Modulsystem oder ein ausgefeiltes Objektsystem an. Mini–F# gehört zur letzten Kategorie: Ihr Modulsystem ist bewusst einfach gestrickt und dient im Wesentlichen der Verwaltung von Namensräumen (der Namenshygiene). Das zentrale Thema dieses Kapitels ist somit ihr Objektsystem.

Objektsysteme können auf Prototypen oder auf Klassen basieren. Sie werden schnell merken, dass es neben den eigentlichen Sprachkonzepten, einiges an technischem Jargon zu lernen gibt. Objekte werden gerne in eine biologische Metapher eingekleidet: „Ein Objekt sendet eine Nachricht an ein anderes Objekt“; „ein Objekt erbt Verhalten von

einem anderen Objekt.“ Prototypenbasierte Systeme sind im gewissen Sinne näher an der biologische Metapher: Ein neues Objekt wird erzeugt, indem ein bereits bestehendes Objekt geklont wird. In klassenbasierten Systemen werden neue Objekte erzeugt, indem Lücken von vorgefertigten Schablonen, den sogenannten Klassen, mit Leben erfüllt werden. Wir betrachten beide Varianten, den prototypenbasierten Ansatz vorwiegend in den Abschnitten 8.1 und 8.2 und den klassenbasierten Ansatz in den Abschnitten 8.3 und 8.5.

Die Zitate am Anfang dieses Kapitels deuten an, dass es durchaus sehr konträre Meinungen gibt, was eine objektorientierte Sprache ausmacht. Fragt man 10 Experten, so erhält man 20 Meinungen. Relativ unstrittig sind die folgenden Charakteristika:

- *Dynamische Bindung* (engl. dynamic dispatch): Wird eine Operation auf einem Objekt durchgeführt, bestimmt das Objekt selbst, welcher Code ausgeführt wird. Objekte mit derselben Schnittstelle können unterschiedlich implementiert sein. Die Implementierungen der Operationen heißen *Methoden*; der Anwendung einer Operation entsprechend *Methodenaufruf* (engl. method invocation). Man sagt auch, dem Objekt wird ein Nachricht geschickt.

- *Kapselung* (engl. encapsulation): Die interne Repräsentation eines Objekts ist außen nicht sichtbar und kann somit lokal geändert werden.

- *Untertypen* (engl. interface inheritance): Auf Schnittstellen lässt sich eine natürliche Untertypbeziehung definieren: Ein Objekt lässt sich in einem Kontext verwenden, in dem nur eine Teilmenge der Methoden benötigt wird. Auf diese Weise lassen sich polymorphe Funktionen definieren, die unterschiedliche Objekte uniform behandeln.

- *Vererbung* (engl. implementation inheritance): Vererbung erlaubt es, die Implementierung verschiedener Objekten zu faktorisieren, so dass gemeinsames Verhalten nur einmal implementiert werden muss. Prototypenbasierte Sprachen verwenden *Delegation*, klassenbasierte Sprachen bieten Klassen und *Unterklassen* an.

Mit all diesen Konzepten werden wir uns intensiv auseinandersetzen. Das folgende Konzept fällt dabei unter den Tisch, obwohl es von vielen Proponenten der Objektorientierung als ebenso zentral angesehen wird (siehe aber Übung 8.9).

- *Objektidentität* (engl. object identity): Jedes Objekt hat eine eigene Identität („all objects are born unequal“).

Im Einzelnen haben wir Folgendes vor.

Abschnitt 8.1 führt den Markenkern von Objektsystemen ein, Schnittstellen und Objekte, und zeigt, wie Objekte mittels Delegation komponiert werden können. Das Objektsystem von Mini–F# basiert streng genommen nicht auf Prototypen; die leichtgewichtigen Objekte aus Abschnitt 8.1 kommen der Idee aber sehr nahe. Wir kontrastieren Objekte mit uns bereits bekannten und vertrauten Konzepten: Modulen, Records, Varianten und abstrakten Datentypen.

Abschnitt 8.2 beschäftigt sich mit einer zweiten Spielart der Polymorphie, der Inklusionspolymorphie, die es erlaubt, überqualifizierte Objekte in Kontexten zu verwenden, die eine geringere Funktionalität erfordern.

Abschnitt 8.3 wendet sich dem klassenbasierten Ansatz zu und zeigt, dass Klassen im Prinzip Sprachkonzepte von Modulen mit denen von Objekten kombinieren.

Abschnitt 8.4 unternimmt einen kurzen Ausflug in die Welt der Modellierung und diskutiert den Entwurf von Schnittstellen am Beispiel von Sequenzen. Folgen von Elementen lassen sich nicht nur explizit durch Daten („Was sind die Elemente der Sequenz?“), sondern auch implizit durch Programme darstellen („Wie zähle ich die Elemente der Sequenz auf?“).

Abschnitt 8.5 zeigt schließlich, wie im klassenbasierten Ansatz, Objekte mittels Unterklassen kompositional definiert werden können. Das Konzept der Vererbung wird an Hand von zwei objektorientierten Entwurfsmustern, dem Beobachter-Entwurfsmuster und dem Schablonen-Entwurfsmuster, illustriert und dem Konzept der Delegation aus Abschnitt 8.1 gegenübergestellt.

8.1. Schnittstellen und Objekte

module
Objects.
Bank

Rufen wir uns noch einmal die Implementierung eines Bankkontos aus Kapitel 7 ins Gedächtnis (*Account* aus Abschnitt 7.2 bzw. *Junior-Account* aus Abschnitt 7.4).

```
exception Insufficient of Nat
let monus (n1 : Nat, n2 : Nat) : Nat =
  if n1 ⩾ n2 then n1 ∸ n2
             else raise (Insufficient n1)
module Account =
  let mutable private funds = 0
  let deposit (amount : Nat) =
    funds ← funds + amount
  let withdraw (amount : Nat) =
    funds ← monus (funds, amount)
  let balance () = funds
```

Wir haben mehrere kleinere Änderungen vorgenommen: An die Stelle einer Speicherzelle (*let funds = ref* 0) ist ein veränderlicher Bezeichner getreten (*let mutable funds* = 0); aus Gründen der Übersichtlichkeit haben wir die „Subtraktion mit Ausnahme“ aus dem Rumpf der Funktion *withdraw* herausgenommen und als getrennte Funktion implementiert; und schließlich verzichten wir darauf, auf dem Konto einen Minimalbetrag *min-funds* zu belassen.

Erinnern wir uns: Die Funktionen *deposit*, *withdraw* und *balance* verwenden die Veränderliche *funds*, die den jeweils aktuellen Kontostand repräsentiert. Nach außen ist dieses Implementierungsdetail nicht sichtbar: Der Zustand ist gekapselt. Mit Hilfe des

Modulsystems werden die drei Operationen weiterhin zu einer konzeptionellen Einheit zusammengefasst: Das lokale Modul *Account* modelliert ein Bankkonto.

Trotz dieser wünschenswerten Eigenschaften scheint die Verwendung des Modulsystems nicht adäquat: Auf diese Weise wird genau *ein* Bankkonto realisiert — es ist unklar, ob und wie man ein Bankinstitut mit einer Vielzahl von Konten modellieren kann. Um dieser Anforderung Rechnung zu tragen, führen wir *Schnittstellentypen* ein.

```
type IAccount =
  interface
    abstract member Deposit  : Nat → Unit
    abstract member Withdraw : Nat → Unit
    abstract member Balance  : Nat with get
  end
```

Die Schnittstelle (engl. interface) legt fest, was mit einem Konto, einem Element des Typs *IAccount*, gemacht werden kann:

- Mit Hilfe von *Deposit* kann ein Betrag in ein Konto eingezahlt werden.
- Die sogenannte Methode *Withdraw* erlaubt es, einen Betrag abzuheben.
- Der Kontostand kann mit Hilfe der Eigenschaft *Balance* eingesehen werden.

Die interne Repräsentation eines Kontos ist im Typ *nicht* festgelegt und kann, wie wir sehen werden, sehr unterschiedlich sein. Eine Schnittstelle basiert auf der Idee des *abstrakten Datentyps* — wie das Schlüsselwort ***abstract*** andeutet. Später mehr zu diesem Thema.

Etwas Fachjargon: Eine Schnittstelle heißt auch Objekttyp; Elemente eines Objekttyps heißen entsprechend *Objekte*. In der Schnittstelle aufgeführte Funktionen werden zu *Methoden* (engl. methods), andere Werte zu *Eigenschaften* (engl. properties). Im obigen Beispiel sind *Deposit* und *Withdraw* Methoden, während *Balance* eine Eigenschaft ist. Der Ergebnistyp der beiden Methoden, *Unit*, deutet an, dass diese um ihres Effektes und nicht um ihres Wertes willen verwendet werden. Die Eigenschaft *Balance* ist lesbar (*get* ist nach dem Schlüsselwort ***with*** aufgeführt), aber nicht schreibbar (*set* fehlt).

Ist der Schnittstellentyp *IAccount* gegeben, dann kann ein einzelnes Bankkonto mit einem sogenannten *Objektausdruck* der Form {***new*** *IAccount* ***with*** ...} angelegt werden. Nach dem Schlüsselwort ***with*** wird konkretisiert, wie die verschiedenen Mitglieder der Schnittstelle, Methoden und Eigenschaften, implementiert werden.

```
let lisas : IAccount =
  let mutable funds = 0
  {
    new IAccount with
      member self.Deposit (amount : Nat) =
        funds ← funds + amount
      member self.Withdraw (amount : Nat) =
        funds ← monus (funds, amount)
      member self.Balance
        with get () = funds
  }
```

Allgemein erzeugt {*new* T *with* ...} ein Element des Typs T, ein *Objekt*. Ein Objekt selbst ist ein Wert und kann somit mittels einer Wertedefinition an einen Bezeichner gebunden werden. Wie in der modulbasierten Variante hat das Objekt *lisas* einen internen Zustand, die Speicherzelle *funds*, um den aktuellen Kontostand zu repräsentieren. Durch die lokale Bindung mit *let* ist dieses Implementierungsdetail nicht nach außen sichtbar — wie vorher ist der Zustand gekapselt.

Auf die Mitglieder eines Objekts wird mit Hilfe der Punktnotation zugegriffen, die wir schon von Modulen und Records kennen.

```
Mini⟩ lisas.Deposit 4711
()
Mini⟩ lisas.Withdraw 815
()
Mini⟩ lisas.Withdraw 2765
()
Mini⟩ lisas.Balance
1131
```

Man sagt auch dem Objekt *lisas* wird die *Nachricht* *Deposit* geschickt. In der Definition von *lisas* steht der frei wählbare Bezeichner *self* jeweils für das Objekt selbst, dem Empfänger der Nachricht. (Methoden können verschränkt rekursiv definiert werden; mit Hilfe von *self* kann das Objekt im Rumpf einer Methode oder einer Eigenschaft Nachrichten an sich selbst schicken. Wenn man möchte, kann man ein Objekt in erster Näherung als ein rekursiv definiertes Record auffassen, siehe Abschnitt 8.1.4.)

Ein Objekt definiert sich einzig und allein durch sein Verhalten (engl. behaviour). Das gleiche Verhalten kann auf sehr unterschiedliche Art und Weise realisiert werden. Die folgende Implementierung eines Bankkontos merkt sich die letzten n Kontostände in einer Art „Ringpuffer“.

```
let ludwigs : IAccount =
  let size        = 10
  let history     = [| for k in 0 .. size − 1 → 0 |]
  let mutable i = 0
  {
    new IAccount with
      member self.Deposit (amount : Nat) =
        let i′ = (i + 1) % size
        history.[i′] ← history.[i] + amount
        i ← i′
      member self.Withdraw (amount : Nat) =
        let i′ = (i + 1) % size
        history.[i′] ← monus (history.[i], amount)
        i ← i′
      member self.Balance
        with get () = history.[i]
  }
```

Jetzt da wir zwei verschiedene Bankkonten zur Verfügung haben, können wir eine Überweisung tätigen.

```
Mini⟩ lisas.Balance
1131
Mini⟩ ludwigs.Deposit 4711
()
Mini⟩ let n = 1000 in lisas.Withdraw n; ludwigs.Deposit n
()
Mini⟩ lisas.Balance
131
Mini⟩ ludwigs.Balance
5711
```

Da die beiden Objekte die gleiche Schnittstelle implementieren, verstehen sie die gleichen Nachrichten. Die interne Umsetzung ist aber ganz unterschiedlich. Jedes Objekt entscheidet selbst, wie es auf eine Nachricht reagiert, sprich welcher Programmcode ausgeführt wird (engl. *dynamic dispatch*).

Objekte sind wie gesagt normale Werte; sie können insbesondere Argument oder Ergebnis von Funktionen sein. Die Funktion *transfer*, die eine Überweisung realisiert, illustriert die Verwendung von Objekten als Funktionsargumente.

```
let transfer (sender : IAccount, amount : Nat, receiver : IAccount) =
  sender.Withdraw amount
  receiver.Deposit  amount
```

Die Funktion *transfer* kann lediglich die in der Schnittstelle zur Verfügung gestellten Operationen nutzen. Wie potentielle Argumente gestrickt sind, ist für die ihre Funkti-

onsweise uninteressant und unerheblich. Tatsächlich können wir *transfer* programmieren, ohne dass Implementierungen der Schnittstelle existieren — nur testen könnten wir *transfer* in einem solchen Fall nicht.

```
Mini⟩ (lisas.Balance, ludwigs.Balance)
(131, 5711)
Mini⟩ transfer (ludwigs, 815, lisas)
()
Mini⟩ (lisas.Balance, ludwigs.Balance)
(946, 4896)
```

Kommen wir von einem Bankkonto zu vielen Bankkonten, sprich einer Bank. Ein einzelnes Bankinstitut kann durch eine Funktion implementiert werden, die ein Bankkonto als Ergebnis hat und auf diese Weise die Eröffnung eines Kontos modelliert.

```
let account (seed : Nat) : IAccount =
  let mutable funds = seed
  {
    new IAccount with
      member self.Deposit (amount : Nat) =
        funds ← funds + amount
      member self.Withdraw (amount : Nat) =
        funds ← monus (funds, amount)
      member self.Balance
        with get () = funds
  }
```

Die Funktion *account* illustriert die Verwendung von Objekten als Funktionsergebnis. Die Funktion heißt auch *Objektkonstruktor* oder kurz *Konstruktor*, da sie Objekte konstruiert.

module Objects. Person

„Getter“ und „Setter“ Hinter der öffentlichen Eigenschaft *Balance* steht die interne Veränderliche *funds* (im Englischen wird *funds* auch als „backing store“ von *Balance* bezeichnet). Ihr Wert wird von *Balance* unverändert bekannt gemacht. Dass muss nicht zwangsläufig so sein: Eine Eigenschaft kann sich auch aus einem Datum oder aus mehreren Daten ableiten; sie kann eine *Sicht* (engl. view) auf die Daten anbieten.

```
type Person =
  interface
    abstract member Name : String with get
    abstract member Age  : Nat    with get, set
  end
let doctor name alias age =
  let mutable age = age
  {
    new Person with
      member self.Name
        with get () = "Doctor " ^ name ^
                      if alias = "" then "" else " (aka " ^ alias ^ ")"
      member self.Age
        with get ()   = age
        and  set inc = age ← age + inc
  }
```

Die Eigenschaft *Name* definiert eine spezielle *Sicht* (engl. view) auf die zugrundeliegenden „Rohdaten“. Die Schnittstelle *Person* bietet neben dieser nur lesbaren Eigenschaft auch eine Eigenschaft an, die sowohl les- als auch schreibbar ist. Schreibbare Eigenschaften können auf der linken Seite einer Zuweisung der Form $e_1 \leftarrow e_2$ stehen.

```
Mini⟩ let dolittle = doctor "Dolittle" "King Jong Thinkalot" 42
Mini⟩ dolittle.Name
"Doctor Dolittle (aka King Jong Thinkalot)"
Mini⟩ dolittle.Age ← 2
()
Mini⟩ dolittle.Age
44
```

Es ist bemerkenswert, dass die Zuweisung nicht das Alter auf den angegebenen Wert setzt, sondern um diesen Wert erhöht. (Die Zuweisung *dolittle.Age* ← 2 entspricht somit der C Zuweisung `dolittle.Age += 2`.) Anders als im Fall von Veränderlichen können beim Lesen und beim Schreiben von Eigenschaften beliebige Berechnungen durchgeführt werden.

8.1.1. Abstrakte Syntax

Eine *Eigenschaft* ist tatsächlich syntaktischer Zucker für zwei Methoden, die sogenannten „Setter“ und „Getter“ der Eigenschaft. Mit anderen Worten, die Semantik von Eigenschaften können wir erklären, indem wir die Sprachkonstrukte übersetzen und so auf die Semantik von Methoden zurückführen. Die Deklaration der Eigenschaft

```
abstract member ℓ : t with get, set
```

ist eine Abkürzung für die Methodendeklarationen

abstract member *get-*ℓ : *Unit* $\to t$
abstract member *set-*ℓ : $t \to$ *Unit*

Fehlt bei der Deklaration der Eigenschaft der Zusatz *get* oder *set*, dann ist entsprechend nur eine der beiden Methoden verfügbar. Ist t ein nicht-funktionaler Typ, dann ist ***abstract member*** $\ell : t$ eine Abkürzung für ***abstract member*** $\ell : t$ ***with*** *get*.

Somit können wir uns auf die formale Definition von Methoden konzentrieren. Wie üblich beschränken wir uns auf den binären Fall und formalisieren nur Objekte mit exakt zwei Methoden. Alle Sprachkonstrukte verallgemeinern sich in naheliegender Weise auf n Methoden.

Ein Schnittstellentyp (engl. interface type) wird durch eine Definition eingeführt.

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***type*** $T =$	Schnittstellentypdefinition ($\ell_1 \neq \ell_2$)
interface	
abstract member $\ell_1 : t_1$	
abstract member $\ell_2 : t_2$	
end	

Der Bezeichner T wird durch die Definition neu eingeführt, ebenso die Bezeichner für die *Methoden*, ℓ_1 und ℓ_2, die verschieden sein müssen: $\ell_1 \neq \ell_2$. (Das Schlüsselwort ***member*** kann übrigens weggelassen werden, ebenso wie die ***interface*** ... ***end*** Klammer.)

Wir erweitern Ausdrücke um Sprachkonstrukte, die Objekte konstruieren und analysieren, um Objektausdrücke (engl. object expressions) und Methodenaufrufe (engl. method invocations).

$e ::= \cdots$	*Ausdrücke:*
$\mid$ { ***new*** T ***with***	Objektausdruck\anonymes Objekt ($\ell_1 \neq \ell_2$)
member $s_1.\ell_1 = e_1$	
member $s_2.\ell_2 = e_2$	
}	
$\mid$ $e.\ell$	Methodenaufruf

So wie ein Funktionsausdruck eine anonyme Funktion konstruiert, so erzeugt ein Objektausdruck ein anonymes Objekt. Nach dem Schlüsselwort ***member*** wird jeweils ein beliebiger Bezeichner aufgeführt, der den Empfänger der Nachricht repräsentiert, das heißt, der für das durch den Objektausdruck erzeugte Objekt selbst steht. Auf Grund dieser Selbstbezüglichkeit wird als Name oft *me*, *self* oder *this* gewählt (wir verwenden durchgehend *self*). Benötigt man keine Referenz auf das Objekt selbst, kann eine anonyme Variable angegeben werden. Ähnlich wie bei Funktionsdefinitionen wird nach dem Gleichheitszeichen der *Methodenrumpf* aufgeführt. Mit $e.\ell$ wird die Methode ℓ aufgerufen; man sagt auch, dem Objekt e wird die Nachricht ℓ geschickt.

In der konkreten Syntax wird zwischen Methoden und Eigenschaften unterschieden. Für Methoden *muss* die Syntax ***member*** $s.\ell\ x = e$ benutzt werden, das heißt nach dem Methodennamen wird der Parametername aufgeführt; für Eigenschaften wird die Syntax

$$\textbf{\textit{member}}\ s.\ell\ \textbf{\textit{with}}\ get\ () = e_1\ \textbf{\textit{and}}\ set\ v = e_2$$

verwendet. Die Definition ist syntaktischer Zucker für die entsprechenden „Setter“ und „Getter“ der Eigenschaft ℓ:

$$\begin{array}{l}\textbf{\textit{member}}\ s.get\text{-}\ell\ () = e_1\\ \textbf{\textit{member}}\ s.set\text{-}\ell\ v\ = e_2\end{array}$$

Wir fassen im Folgenden die konkrete Syntax

$$\textbf{\textit{member}}\ s.\ell\ x = e \text{ als Abkürzung für } \textbf{\textit{member}}\ s.\ell = \textbf{\textit{fun}}\ x \to e$$

auf. Auf diese Weise wird vermieden, dass wir die Typ- und Auswertungsregeln für Funktionen für Methoden duplizieren. In der konkreten Syntax ist die obige Abkürzung leider nicht zulässig, da sie für die Definition von Eigenschaften reserviert ist: $\textbf{\textit{member}}\ s.\ell = e$ kürzt $\textbf{\textit{member}}\ s.\ell\ \textbf{\textit{with}}\ get\ () = e$ ab.

Ist ℓ eine Eigenschaft, so wird mit $e.\ell$ die Eigenschaft des Objekts e abgefragt und mit $e.\ell \leftarrow e'$ gesetzt. Der Ausdruck $e.\ell$ ist syntaktischer Zucker für den Methodenaufruf $e.get\text{-}\ell\ ()$, die Zuweisung $e.\ell \leftarrow e'$ entsprechend für $e.set\text{-}\ell\ e'$.

8.1.2. Statische Semantik

Die folgenden Typregeln setzen voraus, dass die Typdefinition

$$\begin{array}{l}\textbf{\textit{type}}\ T =\\ \quad\textbf{\textit{interface}}\\ \quad\quad\textbf{\textit{abstract member}}\ \ell_1 : t_1\\ \quad\quad\textbf{\textit{abstract member}}\ \ell_2 : t_2\\ \quad\textbf{\textit{end}}\end{array}$$

bekannt ist. Die Typregeln sind denen für Records nicht unähnlich.

$$\frac{\Sigma, \{\, s_1 \mapsto T \,\} \vdash e_1 : t_1 \qquad \Sigma, \{\, s_2 \mapsto T \,\} \vdash e_2 : t_2}{\begin{array}{l}\Sigma \vdash \{\, \textbf{\textit{new}}\ T\ \textbf{\textit{with member}}\ s_1.\ell_1 = e_1\\ \qquad\qquad\qquad\qquad \textbf{\textit{member}}\ s_2.\ell_2 = e_2 \,\} : T\end{array}}$$

$$\frac{\Sigma, \{\, s_1 \mapsto T \,\} \vdash e_1 : t_1 \qquad \Sigma, \{\, s_2 \mapsto T \,\} \vdash e_2 : t_2}{\begin{array}{l}\Sigma \vdash \{\, \textbf{\textit{new}}\ T\ \textbf{\textit{with member}}\ s_2.\ell_2 = e_2\\ \qquad\qquad\qquad\qquad \textbf{\textit{member}}\ s_1.\ell_1 = e_1 \,\} : T\end{array}}$$

$$\frac{\Sigma \vdash e : T}{\Sigma \vdash e.\ell_i : t_i} \quad i \in \{1, 2\}$$

Bei der Konstruktion eines Objekts müssen stets alle in der Schnittstelle aufgeführten Mitglieder definiert werden; die Reihenfolge der Definitionen ist allerdings beliebig. Auf diese Weise wird sichergestellt, dass ein Objekt alle Nachrichten versteht. Der „*self*-Bezeichner“ s_i ist im Methodenrumpf e_i sichtbar und besitzt den gleichen Typ wie das Objekt selbst.

8.1.3. Dynamische Semantik

Wir erweitern Werte um Methodentabellen, die Objekte repräsentieren, und Methodenabschlüsse, zu denen Methoden auswerten.

$\mu \in \mathsf{Lab} \to_{\mathrm{fin}} \mathsf{Val}$	*Methodenumgebungen*$\diagdown$*Methodentabellen*
$\nu ::= \cdots$	*Werte:*
$\mid \mu$	Objekt
$\mid \langle\!\langle \delta,\ s,\ e \rangle\!\rangle$	Methodenabschluss

Die Methodentabelle (engl. method table or dispatch table) bildet Methodennamen auf Werte ab. Der Methodenabschluss $\langle\!\langle \delta,\ s,\ e \rangle\!\rangle$ korrespondiert zu dem *rekursiven* Funktionsabschluss $\langle \delta,\ s,\ x,\ e \rangle$ mit dem Unterschied, dass der formale Parameter x fehlt.

Ein Objekt wertet im Wesentlichen zu sich selbst aus.

$$\delta \vdash \{\, \textbf{\textit{new}}\ T\ \textbf{\textit{with member}}\ s_1.\ell_1 = e_1\ \textbf{\textit{member}}\ s_2.\ell_2 = e_2 \,\} \Downarrow \{\, \ell_1 \mapsto \langle\!\langle \delta,\ s_1,\ e_1 \rangle\!\rangle, \ell_2 \mapsto \langle\!\langle \delta,\ s_2,\ e_2 \rangle\!\rangle \,\}$$

$$\delta \vdash \{\, \textbf{\textit{new}}\ T\ \textbf{\textit{with member}}\ s_2.\ell_2 = e_2\ \textbf{\textit{member}}\ s_1.\ell_1 = e_1 \,\} \Downarrow \{\, \ell_1 \mapsto \langle\!\langle \delta,\ s_1,\ e_1 \rangle\!\rangle, \ell_2 \mapsto \langle\!\langle \delta,\ s_2,\ e_2 \rangle\!\rangle \,\}$$

Dem Objekt wird eine Methodentabelle zugeordnet, in der die Methodennamen an die jeweiligen Methodenanschlüsse gebunden sind. Ähnlich wie bei Funktionsausdrücken wird der Methodenrumpf e_i *nicht* ausgewertet.

Erst wenn dem Objekt eine Nachricht geschickt wird, erfolgt die Auswertung des Methodenrumpfs.

$$\frac{\delta \vdash e \Downarrow \mu \qquad \delta', \{\, s_i \mapsto \mu \,\} \vdash e_i \Downarrow \nu_i}{\delta \vdash e.\ell_i \Downarrow \nu_i} \qquad \text{mit } \mu(\ell_i) = \langle\!\langle \delta',\ s_i,\ e_i \rangle\!\rangle$$

Das Label ℓ_i wird zur Laufzeit in der zu dem Objekt e gehörigen Methodentabelle nachgeschlagen (engl. dynamic dispatch). Die statische Semantik stellt sicher, dass $\ell_i \in \mathit{dom}\ \mu$. Der „*self*-Bezeichner“ s_i wird an das Objekt selbst (engl. self) gebunden; in der erweiterten Umgebung wird der Rumpf der Methode ausgewertet. Das Ergebnis ist auch das Ergebnis des Methodenaufrufs.

Die Auswertungsregel für Methodenaufrufe ist der Regel für die Applikation rekursiver Funktionen nicht unähnlich, siehe Abschnitt 3.6 — in beiden Fällen wird aus einer rekursiven Definition ein zyklisches Geflecht.

8.1.4. Vertiefung

Ein Record aggregiert Werte, ein Modul aggregiert Definitionen und ein Objekt aggregiert Methoden bzw. etwas weniger prosaisch Verhalten. Auf die Komponenten der Aggregation wird jeweils mit der Punktnotation zugegriffen. Drei ähnliche Sprachkonzepte, deren Verhältnis wir im Folgenden näher beleuchten wollen.

module
Objects.
Clock

Objekte versus Records Ein Record aggregiert Werte; da Funktionen Werte sind, können insbesondere Funktionen aggregiert werden. Wird ein Record mit funktionalen Komponenten rekursiv definiert, erhalten wir faktisch ein Objekt. Schauen wir uns ein Beispiel an: In der folgenden Gegenüberstellung wird das Konzept einer Uhr modelliert, einmal mit Hilfe einer Schnittstelle und eines Objektkonstruktors und ein zweites Mal unter Verwendung eines Recordtyps und eines rekursiv definierten Records.

```
type Clock =
  interface
    abstract Tick : Nat → Nat
    abstract Time : Nat with get
  end
let clock () =
  let mutable hours = 0

  {new Clock with
      member self.Tick inc =
        hours ← hours + inc
        self.Time
      member self.Time
        with get () = hours % 24
  }
```

```
type Clock =
  {
    Tick : Nat → Nat
    Time : Unit → Nat
  }
let clock () =
  let mutable hours = 0
  let rec self =
    {
      Tick = fun inc →
               hours ← hours + inc
               self.Time ()
      Time = fun () →
               hours % 24
    }
  self
```

Die Eigenschaft *Time* gibt die „aktuelle“ Uhrzeit in Stunden an. Die Methode *Tick* stellt die Uhr um die angegebene Stundenzahl vor und gibt die Uhrzeit nach der Zeitumstellung zurück. Zu diesem Zweck ruft *Tick* die Methode *Time* auf; das Objekt *self* schickt sich selbst eine Nachricht. Diese Selbstbenachrichtigung wird in der record-basierten Variante durch eine rekursive Wertedefinition realisiert (siehe auch Abschnitt 7.2.5) — mit *let rec self* wird das Record namens *self* rekursiv definiert und anschließend als Ergebnis von *clock* () zurückgegeben. In der objekt-basierten Version rückt der „*self*“-Bezeichner zu jeder einzelnen Methode und steht dort für den Empfänger der Nachricht, also für das Objekt selbst. Die übrigen Unterschiede sind kosmetischer Natur: Eine Methodendefinition entspricht syntaktisch einer Funktionsdefinition; für Recordkomponenten ist diese Syntax nicht verfügbar, so dass wir die Funktionsdefinitionen in Funktionsausdrücke überführen müssen.

In der praktischen Handhabung unterscheiden sich die beiden Varianten nicht, wie die folgenden Interaktionen demonstrieren.

```
Mini⟩ let now = clock ()
Mini⟩ now.Tick 4711
7
Mini⟩ now.Time
7
```

```
Mini⟩ let now = clock ()
Mini⟩ now.Tick 4711
7
Mini⟩ now.Time ()
7
```

In der record-basierten Variante muss bei der Abfrage der Zeit explizit das Dummyargument '()' angegeben werden; in der objekt-basierten Version erledigt das die „Entzuckerung“: *now.Time* wird hinter den Kulissen in den Aufruf der „Getter“-Methode *now.get-Time ()* übersetzt. Auch auf die Gefahr hin etwas Offensichtliches zu betonen: Die Komponente *Time* muss eine Funktion sein; ersetzen wir $Time : Unit \to Nat$ durch $Time : Nat$ und entsprechend $Time = \textbf{\textit{fun}}\ () \to hours \,\%\, 24$ durch $Time = hours \,\%\, 24$, dann wird die Zeit ein unveränderlicher Wert, sie wird auf die Stundenzahl bei der Erzeugung des Records eingefroren, nämlich 0.

Vorläufiges Fazit: Bezüglich der bisher eingeführten Funktionalität sind Objekte und rekursive Records austauschbar. Das wird nicht so bleiben: Wir werden in Abschnitt 8.2 sehen, dass eine Schnittstelle um Funktionalität erweitert werden kann; dies ist bei einem Recordtyp nicht möglich. Aber wir greifen vor.

Objekte versus Module In Kapitel 7 haben wir ein Bankkonto durch ein Modul modelliert. An die Stelle eines einzelnen Moduls ist jetzt eine Sammlung von Objekten getreten. Mit der Einführung von Objekten verlieren Module aber nicht ihre Daseinsberechtigung. Es bietet sich zum Beispiel an, die verschiedenen Operationen einer Bank mit Hilfe eines Moduls zu einer konzeptionellen Einheit zusammenzufassen, siehe Abbildung 8.1. Das Bankinstitut „Trust Me“ wird durch ein lokales Modul realisiert, das bankspezifische, kontoübergreifende Operationen zusammenfasst. Zum Beispiel lässt sich die Gesamtzahl der eröffneten Konten abfragen. (Zur Erinnerung: ***do*** *e* ist eine Abkürzung für ***let*** $() = e$ und kennzeichnet Definitionen, die nur ihres Effektes willen eingeführt werden.) Kontospezifische Operationen werden weiterhin in der Schnittstelle *IAccount* gebündelt.

Fazit: Module und Objekte stehen in keiner wirklichen Konkurrenzsituation; die Sprachkonzepte ergänzen sich eher sinnvoll. Module sind in Mini–F# keine Konstrukte „erster Klasse“: Ein Modul kann zum Beispiel nicht an eine Funktion übergeben werden oder als Ergebnis eines Funktionsaufrufs erhalten werden. Module haben in Mini–F# einen eher statischen Charakter. (Aus diesem Grund haben wir Module für die Modellierung von Bankkonten verworfen. Ein Modul realisiert genau ein Bankkonto; eine Funktion, die ein Bankkonto sprich ein Modul generiert, lässt sich nicht programmieren.) Die Zweitklassigkeit von Modulen liegt in der Tatsache begründet, dass sie neben Wertedefinitionen auch Typdefinitionen (inbesondere Schnittstellendefinitionen) aggregieren.

Delegation Neben dem Standardkonto bietet die Bank „Trust Me“ zusätzlich ein Konto für Studierende an, das mit einer kleinen Einschränkung daherkommt: Es dürfen bei einer einzelnen Transaktion maximal 1000 € abgehoben werden. Der Konstruktor *student-account* illustriert die Programmiertechnik der *Delegation*: Dem Studierendenkonto liegt ein Standardkonto zugrunde, an das die Nachrichten *Deposit* und *Balance* delegiert werden. Die besprochene Einschränkung beim Abheben wird durch eine Alternative realisiert:

```
member self.Withdraw amount =
  if amount > limit then raise Limit
                    else basic.Withdraw amount
```

```
exception Limit
module TrustMe =
  do putline "TrustMe is founded."
  let BIC = 4711    // Bank Identifier Code
  let mutable private no = 0
  let total-no-of-accounts () = no

  // TrustMe standard account
  let account (seed : Nat) =
    do no ← no + 1
    let mutable funds = seed
    {
      new IAccount with
        member self.Deposit (amount : Nat) =
          funds ← funds + amount
        member self.Withdraw (amount : Nat) =
          funds ← monus (funds, amount)
        member self.Balance =
          funds
    }

  // TrustMe student account: maximum withdrawal given by limit
  let limit = 1000
  let student-account (seed : Nat) =
    let basic = account seed
    {
      new IAccount with
        member self.Deposit amount = basic.Deposit amount
        member self.Withdraw amount =
          if amount > limit then raise Limit
                            else basic.Withdraw amount
        member self.Balance = basic.Balance
    }
```

Abbildung 8.1.: Modellierung eines Bankinstituts (modulbasiert).

Liegt der Betrag über dem Limit, wird eine Ausnahme ausgelöst; anderenfalls wird auch die Nachricht *Withdraw* an das Standardkonto *basic* delegiert.

Der interne Zustand des Bankkontos ist in diesem Fall durch ein anderes Objekt gegeben und nicht durch eine Speicherzelle. Die kompositionale Architektur der Konten bringt viele Vorteile mit sich. Der Implementierungsaufwand wird verringert — Delegation verhindert, dass wir das Rad stets aufs Neue erfinden. Das Studierendenkonto verhält sich in den meisten Aspekten wie ein Standardkonto — Delegation stellt die Konsistenz sicher. Wird das Objekt, an das die Arbeit delegiert wird, verbessert, so profitiert auch der Auftraggeber von der Verbesserung.

Die folgende Interaktion zeigt die Operationen des Bankinstituts in Aktion.

```
Mini⟩ TrustMe.BIC
4711
Mini⟩ let hermines = TrustMe.account 8150
Mini⟩ let harrys = TrustMe.student-account 0
Mini⟩ TrustMe.total-no-of-accounts ()
2
Mini⟩ transfer (hermines, 2000, harrys)
Mini⟩ transfer (harrys, 2000, hermines)
uncaught exception : Limit
```

module Objects. Expr

Objekte versus Varianten Themenwechsel: Objekte müssen nicht zwangsläufig einen internen Zustand besitzen. Um zustandslose Objekte (engl. value objects oder immutable objects) zu illustrieren, versuchen wir uns an einer Reimplementierung arithmetischer Ausdrücke. In Abschnitt 6.4.3 haben wir Ausdrücke mit Hilfe des rekursiven Variantentyps

```
type Expr =
  | Const of Nat
  | Add   of Expr * Expr
  | Mul   of Expr * Expr
```

modelliert (siehe auch Aufgaben 4.16 und 4.17). *Lies:* Ein arithmetischer Ausdruck ist entweder eine Konstante, sprich eine natürliche Zahl, oder eine Summe bestehend aus zwei Ausdrücken oder ein Produkt.

Kommen wir zur objektbasierten Implementierung. Ein Objekt ist die Summe seines Verhaltens, so dass wir uns als Erstes fragen müssen, was wir mit einem arithmetischen Ausdruck machen wollen. Die Antwort „auswerten und anzeigen“ führt zu dem folgenden Schnittstellentyp.

```
type IExpr =
  interface
    abstract member Value : Nat
    abstract member Show  : String
  end
```

Aus den drei Datenkonstruktoren des Variantentyps werden Objektkonstruktoren des Schnittstellentyps. Wir führen Funktionen ein, die Konstanten, Summen und Produkte konstruieren.

```
let constant (n : Nat) : IExpr =
  {new IExpr with
      member self.Value = n
      member self.Show  = show n
  }
let add (expr1 : IExpr, expr2 : IExpr) : IExpr =
  {new IExpr with
      member self.Value = expr1.Value + expr2.Value
      member self.Show  = "(" ^ expr1.Show ^ " + " ^ expr2.Show ^ ")"
  }
let mul (expr1 : IExpr, expr2 : IExpr) : IExpr =
  {new IExpr with
      member self.Value = expr1.Value * expr2.Value
      member self.Show  = "(" ^ expr1.Show ^ " * " ^ expr2.Show ^ ")"
  }
```

Jeder Objektkonstruktor detailliert, wie er auf die beiden möglichen Nachrichten, *Value* und *Show*, reagiert.

Vergleichen wir die obige Implementierung mit den korrespondierenden Definitionen für den variantenbasierten Ansatz,

```
let rec Value = function
  | Const n          -> n
  | Add (expr1, expr2) -> Value expr1 + Value expr2
  | Mul (expr1, expr2) -> Value expr1 * Value expr2
let rec Show = function
  | Const n          -> show n
  | Add (expr1, expr2) -> "(" ^ Show expr1 ^ " + " ^ Show expr2 ^ ")"
  | Mul (expr1, expr2) -> "(" ^ Show expr1 ^ " * " ^ Show expr2 ^ ")"
```

sehen wir, dass der Programmcode jeweils sehr unterschiedlich organisiert ist. Im objektbasierten Ansatz wird der Code nach den Varianten gruppiert: *constant* fasst die konstantenspezifischen Anteile aller Operationen zusammen. Im variantenbasierten Ansatz ist der Code nach den Operationen gruppiert: *Value* fasst alle Fälle der Auswertungsoperation zusammen. Stellen wir Varianten und Operationen in einer Matrix

	Const	*Add*	*Mul*
Value			
Show			

dar, dann ist im objektbasierten Ansatz der Code spaltenweise organisiert und im variantenbasierten Ansatz zeilenweise.

In der Organisation spiegelt sich die unterschiedliche Philosophie von abstrakten und konkreten Datentypen wider. Objekte definieren sich durch ihr Verhalten („Was kann mit dem Objekt gemacht werden?“) und verkörpern die Idee des abstrakten Datentyps. Im Gegensatz dazu ist ein Variantentyp konkret: Er definiert sich durch die Angabe seiner Elemente. In unserem Beispiel fragen wir entweder „Was möchte ich mit einem arithmetischen Ausdruck machen?“ oder „Wie ist ein arithmetischer Ausdruck aufgebaut?“. Die jeweilige Antwort führt zu einem Schnittstellentyp bzw. einem rekursiven Variantentyp. Die jeweils andere Frage fällt natürlich nicht unter den Tisch, kann aber entspannter angegangen werden: Für den Schnittstellentyp werden peu à peu Objektkonstruktoren definiert bzw. für den Variantentyp werden peu à peu Funktionen bereitgestellt.

In der praktischen Handhabung unterscheiden sich der variantenbasierte und der objektbasierte Ansatz nur unwesentlich. Verwendet man Objekte, dann sieht eine mögliche Interaktion wie folgt aus.

Mini⟩ ***let*** $e = add\ (constant\ 4711, mul\ (constant\ 815, constant\ 2765))$
Mini⟩ $e.Show$
```
"(4711 + (815 * 2765))"
```
Mini⟩ $e.Value$
2258186
Mini⟩ $(add\ (e, e)).Value$
4516372
Mini⟩ $(add\ (e, e)).Show$
```
"((4711 + (815 * 2765)) + (4711 + (815 * 2765)))"
```

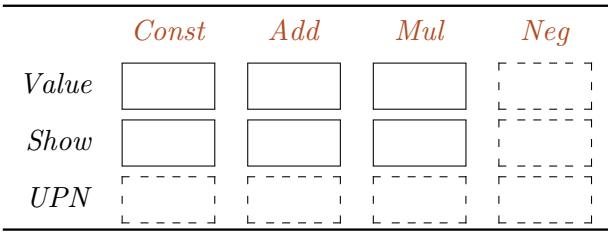
An die Stelle von Funktionsaufrufen tritt das Senden von Nachrichten mit der Punktnotation.

Unterschiede treten allerdings zutage, wenn man versucht, die jeweiligen Programme zu *erweitern*, entweder um neue Varianten (zum Beispiel Negation oder Kehrwert) oder um neue Funktionalität (zum Beispiel Übersetzung in UPN).

Im variantenbasierten Ansatz ist es einfach, neue Funktionalität hinzufügen: Es wird einfach eine neue Funktion definiert. In Gegensatz dazu ist es aufwändig, neue Varianten hinzuzufügen: Der Variantentyp muss um den neuen Fall erweitert werden und zusätzlich muss *jede* bestehende Funktion auf dem Variantentyp angepasst werden. Wir müssen uns jeweils überlegen, wie wir den neuen Fall behandeln.

Für den objektbasierten Ansatz kehren sich Vor- und Nachteile um. Es ist einfach, neue Varianten hinzuzufügen: Es wird einfach ein neuer Objektkonstruktor definiert. In Gegensatz dazu ist es aufwändig, neue Funktionalität hinzuzufügen: Der Schnittstellentyp muss erweitert werden und zusätzlich muss *jeder* bestehende Objektkonstruktor angepasst werden. Wir müssen uns jeweils überlegen, wie wir die neue Operation behandeln.

Eine der beiden Erweiterungen ist jeweils einfach, da lokal durchführbar; die andere ist aufwändig, da sie globale Änderungen nach sich zieht. Die zweidimensionale Darstellung



illustriert noch einmal Vor- und Nachteile. Ist die Matrix zeilenweise (spaltenweise) organisiert, dann ist es einfach eine neue Zeile (Spalte) hinzuzufügen, aber schwierig eine neue Spalte (Zeile) zu ergänzen.

Objekte versus abstrakte Datentypen Die Zusammenfassung von Operationen zu einer Schnittstelle kennen wir bereits von den abstrakten Datentypen. In Abschnitt 5.3 haben wir die Schnittstelle[1] eines abstrakten Datentyps mit Hilfe des *Modulsystems* realisiert. Konkret haben wir drei Implementierungen von „endlichen Abbildungen“ besprochen: Listen, Suchlisten und Suchbäume.

In diesem Abschnitt haben wir gesehen, wie man Schnittstellen mit Hilfe des *Objektsystems* realisiert. Objekte basieren ebenfalls auf der Idee des abstrakten Datentyps und entwickeln diese in gewisser Weise weiter:

- Verschiedene Implementierungen eines abstrakten Datentyps zeigen exakt das gleiche Verhalten; sie sind austauschbar: Für den Benutzer macht es bezüglich des Ein- und Ausgabeverhaltens keinen Unterschied, ob endliche Abbildungen durch Listen, Suchlisten oder Suchbäume implementiert werden.

- Charakteristik des Modulsystems: Eine Schnittstelle wird zu einem Zeitpunkt durch *eine* Implementierung realisiert.

- Objekte, die die gleiche Schnittstelle unterstützen, zeigen ein verwandtes, nicht aber notwendigerweise ein identisches Verhalten: Ein Studierenkonto verhält sich beim Abheben eines größeren Betrags anders als ein Standardkonto.

- Charakteristik des Objektsystems: Eine Schnittstelle kann zu einem Zeitpunkt durch *mehrere* Objekte realisiert werden.

Dass ein abstrakter Datentyp mit Hilfe des Modulsystem implementiert wird, ist natürlich nicht zwingend; auch das Objektsystem kann für die Realisierung herangezogen werden. Die Tatsache, dass eine Schnittstelle zu einem Zeitpunkt durch mehrere Objekte realisiert werden kann, erweist sich, wie wir im Folgenden sehen werden, dabei oft als vorteilhaft.

module
Objects.
UPNCalculator

[1] F# kennt neben *Implementierungsmodulen* (Endung `.fs`, engl. implementation files) auch *Schnittstellenmodule* (Endung `.fsi`, engl. signature files). Ein Schnittstellenmodul ist gewissermaßen der „Typ“ eines Implementierungsmoduls und enthält alle für die statische Semantik notwendigen Informationen, insbesondere Kopien der Typdefinitionen und die Signaturen von Wertedefinitionen.

Parametrisierte Schnittstellen\generische Schnittstellen Wie Record- und Variantentypen können auch Schnittstellentypen mit einem oder mehreren Typen parametrisiert werden.

```
type IStack⟨'elem⟩ =
  interface
    abstract member IsEmpty : Bool
    abstract member Push    : 'elem → Unit
    abstract member Pop     : Unit → 'elem
    abstract member Top     : 'elem
  end
```

Auf einen Stapel (engl. stack) kann mit *Push* ein Element abgelegt werden, mit *Pop* wird das oberste Element entfernt und mit *Top* wird es inspiziert, ohne den Stapel zu verändern. Wie Listen und Arrays sind auch Stacks homogene Container: Der Typ der Elemente ist beliebig; alle Elemente eines Containers müssen aber den gleichen Typ besitzen.

In gleicher Weise wie der Objektkonstruktor *account* das Modul *Account* aus Abschnitt 7.2 verallgemeinert, so verallgemeinert der Objektkonstruktor *stack* das Modul *Stack* aus Abschnitt 7.4.4.

```
exception Pop
exception Top

let stack⟨'elem⟩ () =
  let mutable stack = []
  {
    new IStack⟨'elem⟩ with
      member self.IsEmpty =
        List.isEmpty stack
      member self.Push x =
        stack ← x :: stack
      member self.Pop () =
        match stack with
        | []      → raise Pop
        | x :: xs → stack ← xs; x
      member self.Top =
        match stack with
        | []      → raise Top
        | x :: xs → x
  }
```

Der Objektkonstruktor generiert einen leeren Stack eines beliebigen Elementtyps. Mit anderen Worten, *stack* ist eine *polymorphe Funktion* des Typs $Unit \to IStack\langle'elem\rangle$.

Abbildung 8.2 zeigt den Objektkonstruktor in Aktion. Der UPN-Rechner Deluxe er-

```
let upn-calculator-deluxe () =
  let stacks : IStack ⟨IStack ⟨Nat⟩⟩ = stack ()
  try
    stacks.Push (stack ())
    while true do
      try
        match Input.query "UPN > " with
        | ""  → ()
        | "+" → stacks.Top.Push (stacks.Top.Pop + stacks.Top.Pop)
        | "*" → stacks.Top.Push (stacks.Top.Pop * stacks.Top.Pop)
        | "." → putline (show stacks.Top.Top)
        | "exit" | "halt" | "q" | "quit" | "stop" → raise EOF
        | "(" → stacks.Push (stack ())
        | ")" → stacks.Pop |> ignore
        | s   → stacks.Top.Push (read-nat s)
      with
       | Pop | Top → putline "stack is empty"
       | Read      → putline "enter a number or an operator"
    with
     | EOF → putline "bye bye"
```

Abbildung 8.2.: Ein UPN-Taschenrechner, der Hilfsrechnungen unterstützt.

möglicht Hilfsrechnungen, die mit '(' begonnen und mit ')' beendet werden. Der interne Zustand des Rechners ist ein Stack von Stacks von natürlichen Zahlen. Die Operationen '+', '*' usw. arbeiten auf dem obersten Stack, '(' legt einen leeren Stack ab und ')' entfernt den obersten Stack — mit Hilfe von *ignore* wird das Ergebnis von *Pop* ignoriert: ***let*** *ignore* _ = (). Zur Laufzeit existiert somit ein Stack vom Typ *IStack* ⟨*IStack* ⟨*Nat*⟩⟩ und beliebig viele Stacks des Typs *IStack* ⟨*Nat*⟩. Polymorphie macht's möglich.

8.2. Untertypen

module
Objects.
Subtyping

Informatikerinnen und Informatiker bilden Modelle der Wirklichkeit. Ein Modell wird in der Regel auf Grundlage von Anforderungen entwickelt, die präzisieren, was die Software leisten soll. Im Laufe der Zeit können sich diese Anforderungen ändern, etwa weil die Wirklichkeit vorangeschritten ist und die Modelle an die neuen Gegebenheiten angepasst werden müssen. Oder der Einsatz der Software war so erfolgreich, dass neue Begehrlichkeiten geweckt wurden. In diesem und in den nächsten Abschnitten werden wir uns schrittweise dem Thema „Software-Evolution" annähern. Wie kann die Änderung und insbesondere die Erweiterung von Programmen linguistisch unterstützt werden, so dass die notwendigen Eingriffe „minimal invasiv" sind? Wir sagen bewusst annähern, da das Ideal in voller Schönheit nicht erreicht wird.

Nehmen wir an, wir wollen einige Bankkonten um die Möglichkeit erweitern, einen Kontoauszug abzurufen. Zu diesem Zweck könnten wir ein entsprechendes Mitglied zur Schnittstelle *IAccount* hinzufügen. Nun haben wir im letzten Abschnitt diskutiert, dass eine solche Änderung aufwändig ist, da jeder bestehende Objektkonstruktor geändert werden muss. Eine kostengünstigere Alternative besteht darin, eine zweite Schnittstelle als *Erweiterung* von *IAccount* zu definieren.

```
type IAccountPlus =
  interface
    inherit IAccount
    abstract member Statement : Array ⟨Nat⟩
  end
```

Die Schnittstelle *IAccountPlus* bietet die Funktionalität von *IAccount*, deren Mitglieder „vererbt" werden, und offeriert darüber hinaus die Eigenschaft *Statement*. (Ein Kontoauszug ist der Einfachheit halber die Folge der letzten Kontostände — es geht uns an dieser Stelle nicht darum, das Bankenwesen möglichst wirklichkeitsgetreu zu erfassen.)

Das Objekt *ludwigs* lässt sich ohne großen Aufwand zu einem Plus-Konto erweitern. Wir ordnen lediglich die vorhandene Historie um, so dass der aktuelle Kontostand als Erstes aufgeführt wird.

```
let ludwigs : IAccountPlus =
  let size         = 10
  let history      = [| for k in 0 .. size − 1 → 0 |]
  let mutable i = 0
  let next ()      = (i + 1) % size
  {
    new IAccountPlus with
      member self.Deposit (amount : Nat) =
        history.[next ()] ← history.[i] + amount; i ← next ()
      member self.Withdraw (amount : Nat) =
        history.[next ()] ← monus (history.[i], amount); i ← next ()
      member self.Balance =
        history.[i]
      member self.Statement =
        [| for k in 0 .. size − 1 → history.[(size + i − k) % size] |]
  }
```

Wenn wir das erweiterte Konto verwenden, machen wir allerdings eine unliebsame Entdeckung. Der Aufruf

```
transfer (lisas, 1000, ludwigs)
```

wird von der statischen Semantik abgewiesen, da der Typ des formalen Parameters nicht mit dem Typ des aktuellen Parameters übereinstimmt (siehe Typregel für die Applikation (3.1) in Abschnitt 3.5).

```
transfer : IAccount ∗ Nat ∗ IAccount → Unit
ludwigs : IAccountPlus
```

Der Typ von *IAccountPlus* ist nicht *gleich* dem Typ *IAccount*. Die dynamische Semantik ist weniger wählerisch: Der Ausdruck kann problemlos ausgerechnet werden, da *ludwigs* mehr Funktionalität als gefordert anbietet — *ludwigs* ist sozusagen überqualifiziert.

Mit anderen Worten, *IAccountPlus* und *IAccount* sind nicht zwei beliebige Typen, sondern sie stehen in einer engen Beziehung: Alle Elemente von *IAccountPlus* können als Elemente *IAccount* verwendet werden. In Anlehnung an die mathematische Teilmengenbeziehung nennen wir *IAccountPlus* einen *Untertyp* von *IAccount* und umgekehrt *IAccount* einen *Obertyp* von *IAccountPlus*. In eine Formel gegossen:

$$IAccountPlus \preccurlyeq IAccount$$

Das Typsystem von Mini–F# kann mit Hilfe von *Untertypen* flexibler gemacht werden. Die grundlegende Idee ist, dass ein Element eines Untertyps überall da verwendet werden kann, wo ein Element eines Obertyps verlangt wird. Mit Hilfe einer sogenannten *Typanpassung* (engl. type cast) lässt sich ein Element eines Untertyps in ein Element eines Obertyps überführen.

transfer (*lisas*, 1000, *ludwigs* :> *IAccount*)

Das Symbol ':>' deutet an, dass die „breitere" Schnittstelle von *ludwigs* auf die „schmalere" Schnittstelle *IAccount* verengt wird.

8.2.1. Abstrakte Syntax

Wir erweitern Schnittstellentypen um ***inherit*** Klauseln.

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***type*** $U =$	erweiterte Definition eines Schnittstellentyps
interface	
inherit T_1	
inherit T_2	
abstract member $\ell : t$	
end	

Wie immer vereinfachen wir etwas: Tatsächlich dürfen beliebig viele ***inherit*** Klauseln aufgeführt werden und beliebig viele neue Mitglieder hinzukommen.

Wir erweitern Ausdrücke um Typanpassungen.

$e ::= \cdots$	*Ausdrücke:*
$\mid\ e :> t$	Typanpassung⧹Typeinschränkung

8.2.2. Statische Semantik

Gemäß dem Motto „Vertrauen ist gut, Kontrolle ist besser" überprüfen wir bei der Typanpassung $e :> t$, ob der Typ von e ein Untertyp von t ist.

$$\frac{\Sigma \vdash e : t' \qquad t' \preccurlyeq t}{\Sigma \vdash (e :> t) : t} \tag{8.1}$$

Eine Typanpassung geht in der Regel mit einem Informationsverlust einher. Wird die Schnittstelle eines Objekts verengt, so ist die Interaktion mit dem Objekt fürderhin eingeschränkt. (Ein Objekt ist die Summe seines Verhaltens.)

Die gute Nachricht ist, dass Typanpassungen auch automatisch vorgenommen werden (engl. automatic upcast).

$$\frac{\Sigma \vdash e : t \qquad t \preccurlyeq \tau'}{\Sigma \vdash e : \tau'} \tag{8.2}$$

Die sogenannte *Subsumptionsregel* fängt die Idee von Untertypen ein: ein Element eines Untertyps kann überall da verwendet werden, wo ein Element eines Obertyps verlangt wird. Die Subsumptionsregel wird zum Beispiel angewendet bei Funktionsaufrufen — wenn der Typ des aktuellen Parameters ein Untertyp des Typs des formalen Parameters ist — und bei der Zuweisung — wenn der Typ der rechten Seite ein Untertyp

des Typs der linken Seite ist. Die schlechte Nachricht ist, dass die automatische Typanpassung nicht in voller Schönheit gelingt. Sie ist gewissen technischen Einschränkungen unterworfen, die der Tatsache geschuldet sind, dass F# die Typen von Ausdrücken *inferiert*. Einschränkungen, die wir im Folgenden aber ignorieren wollen. (Die Zweige einer Alternative stellen zum Beispiel eine Problemzone dar, insbesondere wenn die Ausdrücke unterschiedliche Typen besitzen.)

Wir müssen die Relation '$\preccurlyeq$' natürlich noch mit Leben füllen. Welche prinzipiellen Eigenschaften hat die Untertypbeziehung? Wie interagieren die Typkonstrukte, die wir bisher kennengelernt haben, mit Untertypen?

Quasiordnung Zunächst einmal ist die Untertypbeziehung eine *Quasiordnung* (siehe auch Abschnitt 5.1 und Anhang A.1.4): Sie ist *reflexiv* und *transitiv*.

$$\frac{}{t \preccurlyeq t} \qquad \frac{t_1 \preccurlyeq t_2 \qquad t_2 \preccurlyeq t_3}{t_1 \preccurlyeq t_3}$$

Die Eigenschaften lassen sich im Zusammenspiel mit der Regel für die Typanpassung (8.1) motivieren. Reflexivität erlaubt eine triviale Typanpassung: Hat e den Typ t, dann ist $e :> t$ zulässig. Transitivität hilft, geschachtelte Typanpassungen zu vereinfachen: $(e :> t_2) :> t_3$ kann zu $e :> t_3$ verkürzt werden.

Schnittstellen Beziehungen zwischen Schnittstellentypen leiten sich direkt aus dem Programm ab; sie werden von der Programmierer*in *explizit* festgelegt. Aus der Typdefinition

```
type U =
  interface
    inherit T1
    inherit T2
    abstract member ℓ : t
  end
```

werden die folgenden Axiome abgeleitet:

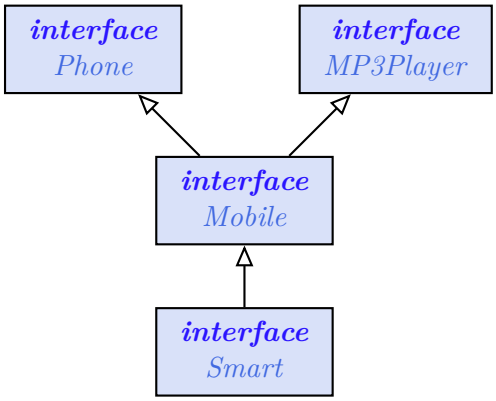
$$\frac{}{U \preccurlyeq T_1} \qquad \frac{}{U \preccurlyeq T_2}$$

Die Schnittstelle U *vereinigt die Operationen* von T_1 und T_2 und fügt eine weitere hinzu. Damit liegen die Objekte von U im *Durchschnitt der Objekte* von T_1 und T_2. (Nachdenken!) Allgemein gilt: Je mehr Operationen eine Schnittstelle umfasst, desto weniger Objekte unterstützen die Schnittstelle. Diese Beobachtung ist übrigens nicht spezifisch für Mini–F# im Besonderen oder Programmiersprachen im Allgemeinen, sondern lässt sich auch im täglichen Leben machen:

Abbildung 8.3 zeigt einen Programmausschnitt, der elektrische und elektronische Geräte modelliert: Telefone mit Wählscheibe, MP3-Spieler, Mobiltelefone und Smartphones. Bezüglich ihrer Funktionalität sind die Geräte hierarchisch organisiert.

```
type Phone =
  interface
    abstract member Dial : Nat → Unit          // Nummer wählen
  end
type MP3Player =
  interface
    abstract member Play : String → Unit       // MP3 abspielen
  end
type Mobile =
  interface
    inherit Phone
    inherit MP3Player
    abstract member Lookup : String → Nat      // Suche im Telefonbuch
    abstract member Call   : String → Unit     // Person anrufen
  end
type Smart =
  interface
    inherit Mobile
    abstract member Browse : String → Unit     // URL anzeigen
  end
let aPhone : Phone = ...
let aMobile : Mobile = ...
let aSmart : Smart = ...
```

Abbildung 8.3.: Schnittstellenhierarchie: Geräte im Wandel der Zeit.



Ein Mobiltelefon vereinigt die Eigenschaften eines gusseisernen Telefons und eines MP3–Spielers; ein Smartphone fügt dem weitere Funktionalität hinzu. (Nehmen Sie den Programmcode nicht zu ernst — er dient lediglich dem Zweck, die Phänomene an einem weiteren Beispiel zu verdeutlichen.) Mit fast jedem Gerät lässt sich telefonieren: Möchte ich einen Anruf tätigen, kann ich das mit einem altmodischen Telefon tun oder mit einem modernen Smartphone. Bezüglich dieser Funktionalität unterscheiden sich die Geräte nicht (Sichtweise des abstrakten Datentyps). Natürlich sehen die Geräte anders aus und funktionieren ganz unterschiedlich (Sichtweise des konkreten Datentyps). Die Schnittstelle mit der geringsten Funktionalität enthält also die meisten Objekte. Je mehr Funktionalität ich einfordere, desto weniger Geräte stehen zur Auswahl: Möchte ich im Internet surfen, muss ich zum Smartphone greifen.

Paare Als nächstes machen wir uns daran, '$\preccurlyeq$' auf strukturierten Typen wie zum Beispiel dem Paartyp $t_1 * t_2$ zu definieren. Die Regel für Paare legt fest, dass ein Paartyp Untertyp eines anderen Paartyps ist, wenn die jeweiligen Komponententypen in einer Untertypbeziehung stehen.

$$\frac{t_1 \preccurlyeq t_1' \qquad t_2 \preccurlyeq t_2'}{t_1 * t_2 \preccurlyeq t_1' * t_2'} \tag{8.3}$$

Warum ist das eine sinnvolle Festlegung? Um diese Frage zu beantworten ist es hilfreich, sich noch einmal ins Gedächtnis zu rufen, welche Operationen auf Paare angewendet werden können. Erhalte ich ein Paar, kann ich die Komponenten mit Hilfe der Projektionsfunktionen *fst* und *snd* extrahieren. Nach der Extraktion kann eine Typanpassung vorgenommen werden.

$$\frac{\dfrac{\dfrac{\vdots}{\Sigma \vdash e : t_1 * t_2}}{\Sigma \vdash \mathit{fst}\ e : t_1} \qquad \dfrac{\vdots}{t_1 \preccurlyeq t_1'}}{\Sigma \vdash \mathit{fst}\ e : t_1'} \qquad\qquad \frac{\dfrac{\dfrac{\vdots}{\Sigma \vdash e : t_1 * t_2}}{\Sigma \vdash \mathit{snd}\ e : t_2} \qquad \dfrac{\vdots}{t_2 \preccurlyeq t_2'}}{\Sigma \vdash \mathit{snd}\ e : t_2'}$$

Mit Hilfe der Paarregel (8.3) kann die Typanpassung alternativ zu dem Paar selbst verschoben werden. (Die Typanpassung und der Zugriff auf die Komponenten können

im Programmtext ja tatsächlich weit auseinanderliegen.)

$$\dfrac{\dfrac{\vdots}{\Sigma \vdash e : t_1 * t_2} \qquad \dfrac{\dfrac{\vdots}{t_1 \preccurlyeq t_1'} \quad \dfrac{\vdots}{t_2 \preccurlyeq t_2'}}{t_1 * t_2 \preccurlyeq t_1' * t_2'}}{\dfrac{\Sigma \vdash e : t_1' * t_2'}{\Sigma \vdash \mathit{fst}\ e : t_1'}} \qquad \dfrac{\dfrac{\vdots}{\Sigma \vdash e : t_1 * t_2} \qquad \dfrac{\dfrac{\vdots}{t_1 \preccurlyeq t_1'} \quad \dfrac{\vdots}{t_2 \preccurlyeq t_2'}}{t_1 * t_2 \preccurlyeq t_1' * t_2'}}{\dfrac{\Sigma \vdash e : t_1' * t_2'}{\Sigma \vdash \mathit{snd}\ e : t_2'}}$$

Die obige Argumentation bewegt sich innerhalb der statischen Semantik. Nun soll die statische Semantik sicherstellen, dass die Auswertung fehlerfrei verläuft. Validieren wir die Paarregel also noch einmal aus dem Blickwinkel der dynamischen Semantik. In dem folgenden Programmfragment fehlt ein Teilausdruck vom Typ *Mobile* ∗ *Mobile*.

let (*mobile1*, *mobile2*) = ☐ ***in*** *mobile1*.*Call* (`"Hannah"`); *mobile2*.*Call* (`"Lee"`)

Welcher Ausdruck kann in die Lücke eingesetzt werden, so dass die Nachricht *Call* jeweils verstanden wird? Da *Call* eine Methode der Schnittstelle *Mobile* ist, lässt sich problemlos ein Paar von Mobiltelefonen einsetzen. Wird ein Mobiltelefon durch ein Telefon ersetzt, schlägt die Auswertung fehl: Ein Telefon versteht die Nachricht *Call* nicht. Unproblematisch ist der „Upgrade“ zu einem Smartphone: Dieses bietet zusätzliche Funktionalität an, versteht aber weiterhin die Nachricht *Call*. Kurzum: beide Komponenten des Paares müssen Elemente eines Untertyps von *Mobile* sein, so wie es die Subsumptionsregel (8.2) im Zusammenspiel mit der Paarregel (8.3) vorschreibt.

Funktionen Wenden wir uns Funktionen zu. In dem folgenden Programmfragment fehlt ein Teilausdruck vom Typ *Mobile* → *Mobile* (*aMobile* ist in Abbildung 8.3 definiert).

(☐ *aMobile*).*Call* (`"Holly"`)

Wiederum die Frage: Welcher Ausdruck kann in die Lücke eingesetzt werden, ohne dass bei der Auswertung Probleme auftreten? Mit einem Ausdruck vom Typ *Mobile* → *Mobile* („das Mobiltelefon wird repariert“) sind wir auf der sicheren Seite — die Anforderung wird präzise erfüllt. Die Funktion kann aber an Stelle eines Mobiltelefons auch ein Smartphone zurückgeben; ein Ausdruck vom Typ *Mobile* → *Smart* („Upgrade“) ist zulässig. Diese Änderung betrifft den Ergebnistyp, wie sieht es mit dem Argumenttyp aus? Können wir eine Funktion des Typs *Smart* → *Mobile* einsetzen? Nein, das wird nicht gutgehen: Die Funktion fordert ein Smartphone, erhält aber nur ein Mobiltelefon — die Erwartung wird enttäuscht. Die Funktion kann aber weniger einfordern; ein Ausdruck vom Typ *Phone* → *Mobile* ist unproblematisch: Die Funktion erwartet nur ein altmodisches Telefon, erhält aber ein Mobiltelefon — die Erwartung wird übertroffen. Halten wir fest: Wird eine Funktion vom Typ $t_1' \to t_2'$ erwartet, dann können wir auch eine Funktion einsetzen, die weniger fordert (Obertyp von t_1') und mehr verspricht (Untertyp von t_2').[2]

[2] Das Phänomen ist uns bereits untergekommen, als wir Anforderungen und Garantien in Verträgen diskutiert haben (design by contract), siehe Abschnitt 5.2.4.

$$\frac{t_1' \preccurlyeq t_1 \qquad t_2 \preccurlyeq t_2'}{t_1 \to t_2 \preccurlyeq t_1' \to t_2'} \tag{8.4}$$

Im Fachjargon sagt man: Der Funktionstyp ist

- *kontravariant* im Argumenttyp ($t_1' \preccurlyeq t_1$) und
- *kovariant* im Ergebnistyp ($t_2 \preccurlyeq t_2'$).

Die Vorsilbe „kontra“ deutet an, dass sich die Richtung der Quasiordnung umkehrt. Im Gegensatz zum Funktionstyp ist der Paartyp kovariant in beiden Komponententypen.

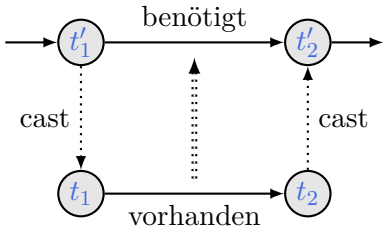
Die Varianz des Funktionstyps lässt sich auch „herleiten“, wenn man sich anschaut, wie die Typregel für die Funktionsapplikation mit der Subsumptionsregel interagiert.

$$\dfrac{\dfrac{\vdots}{\Sigma \vdash e : t_1 \to t_2} \qquad \dfrac{\dfrac{\vdots}{\Sigma \vdash e_1 : t_1'} \qquad \dfrac{\vdots}{t_1' \preccurlyeq t_1}}{\Sigma \vdash e_1 : t_1} \qquad \dfrac{\vdots}{t_2 \preccurlyeq t_2'}}{\Sigma \vdash e\ e_1 : t_2'}$$

Anstatt erst das Funktionsargument und dann das Funktionsergebnis anzupassen, kann mit Hilfe der Funktionsregel (8.4) direkt die Funktion angepasst werden.

$$\dfrac{\dfrac{\vdots}{\Sigma \vdash e : t_1 \to t_2} \qquad \dfrac{\dfrac{\vdots}{t_1' \preccurlyeq t_1} \qquad \dfrac{\vdots}{t_2 \preccurlyeq t_2'}}{t_1 \to t_2 \preccurlyeq t_1' \to t_2'}}{\Sigma \vdash e : t_1' \to t_2'} \qquad \dfrac{\vdots}{\Sigma \vdash e_1 : t_1'}$$
$$\Sigma \vdash e\ e_1 : t_2'$$

Das folgende Diagramm illustriert die Typanpassungen: Eine Funktion des Typs $t_1' \to t_2'$ kann über den Umweg $t_1 \to t_2$ konstruiert werden.



Die Richtung der Typanpassungen kehrt sich für Argument und Ergebnis um.

Wenn wir Funktionstypen schachteln, also Funktionen höherer Ordnung bilden, ergibt sich ein interessantes Bild:

$$\begin{gathered}
\mathit{Smart} \to \mathit{Smart} \preccurlyeq \mathit{Smart} \to \mathit{Phone} \\
\mathit{Phone} \to \mathit{Smart} \preccurlyeq \mathit{Smart} \to \mathit{Smart} \\
\mathit{Phone} \to \mathit{Smart} \preccurlyeq \mathit{Smart} \to \mathit{Phone} \\
(\mathit{Smart} \to \mathit{Phone}) \to \mathit{Smart} \preccurlyeq (\mathit{Phone} \to \mathit{Smart}) \to \mathit{Phone} \\
((\mathit{Phone} \to \mathit{Smart}) \to \mathit{Phone}) \to \mathit{Smart} \preccurlyeq ((\mathit{Smart} \to \mathit{Phone}) \to \mathit{Smart}) \to \mathit{Phone}
\end{gathered}$$

In dem Typ $((t_1 \to t_2) \to t_3) \to t_4$ stehen t_1 und t_3 an kontravarianten Positionen und t_2 und t_4 an kovarianten Positionen. Die Richtung der Quasiordnung wechselt mit jeder Schachtelung im Argumenttyp.

Speicherzellen Wenden wir uns dem nächsten Typ zu: den Speicherzellen. Zur Erinnerung: Eine Speicherzelle kann mit '!' gelesen und mit ':=' geschrieben werden. Starten wir die Diskussion mit einem kleinen Quiz: Gelingt die Auswertung des folgenden Ausdrucks?

```
let mobile-case : Ref⟨Mobile⟩ = ref aMobile
let phone-box   : Ref⟨Phone⟩  = mobile-case
phone-box := aPhone
(!mobile-case).Play ("Siberian Khatru")
```

Wir allokieren eine Speicherzelle vom Typ *Ref⟨Mobile⟩* und geben diese in der zweiten Zeile als Speicherzelle vom Typ *Ref⟨Phone⟩* aus. Beachten Sie, dass keine zweite Speicherzelle angelegt wird: *phone-box* ist lediglich ein *Alias*, ein anderer Name, für *mobile-case*. In der Speicherzelle *phone-box* wird sodann ein Telefon abgelegt. Schließlich nimmt das Unglück seinen Lauf: Dem Inhalt von *mobile-case* alias *phone-box* wird die Nachricht *Call* geschickt. Was läuft schief?

In den Zeilen 1, 3 und 4 wird jeweils „typkonform" gearbeitet, nur in der 2. Zeile kommen Untertypen ins Spiel. Das Programm zeigt, dass *Ref⟨Mobile⟩* *kein* Untertyp von *Ref⟨Phone⟩* ist, obwohl *Mobile* ein Untertyp von *Phone* ist! Gehen wir der Ursache auf den Grund. In dem folgenden Programmfragment fehlt ein Teilausdruck vom Typ *Ref⟨Mobile⟩*.

```
(! [          ]).Call ("Hannah")
```

Die Lücke können wir mit ***ref*** *aMobile*, aber auch mit ***ref*** *aSmart* füllen. Mit anderen Worten, der lesende Speicherzugriff ist kovariant. Wie sieht es mit dem schreibenden Speicherzugriff aus?

```
[          ] := aMobile
```

Jetzt können wir einen Ausdruck vom Typ *Ref⟨Mobile⟩*, aber auch einen vom Typ *Ref⟨Phone⟩* in die Lücke setzen. Der schreibende Speicherzugriff ist also kontravariant.

Der Lesezugriff ist kovariant; der Schreibzugriff ist kontravariant; da Speicherzellen beide Zugriffsarten unterstützen, sind sie summa summarum *invariant*.

$$\frac{t \preccurlyeq t' \qquad t' \preccurlyeq t}{Ref\langle t\rangle \preccurlyeq Ref\langle t'\rangle}$$

Der Typ $Ref\langle t\rangle$ ist ein Untertyp von $Ref\langle t'\rangle$, wenn t und t' äquivalent sind, die Typen wechselseitig austauschbar sind. In Mini–F# bedeutet dies, dass t und t' tatsächlich *identisch* sein müssen. Entsprechendes gilt für alle modifizierbaren Datenstrukturen (engl.

mutable data structures) wie zum Beispiel Arrays. Zur Erinnerung: Die Elemente eines Arrays können mit $a.[i] \leftarrow e$ überschrieben werden.

Die Varianz des lesenden und des schreibenden Speicherzugriffs lässt sich auch aus dem Zusammenspiel der Subsumptionsregel mit den Typregeln für die Dereferenzierung '!' und die Zuweisung ':=' ablesen.

$$\dfrac{\dfrac{\vdots}{\Sigma \vdash e : \mathit{Ref}\langle t\rangle} \quad \dfrac{\vdots}{t \preccurlyeq t'}}{\Sigma \vdash\ !e : t'} \qquad\qquad \dfrac{\dfrac{\vdots}{\Sigma \vdash e_1 : \mathit{Ref}\langle t\rangle} \quad \dfrac{\dfrac{\vdots}{\Sigma \vdash e_2 : t'} \quad \dfrac{\vdots}{t' \preccurlyeq t}}{\Sigma \vdash e_2 : t}}{\Sigma \vdash (e_1 := e_2) : \mathit{Unit}}$$

Beim lesenden Zugriff müssen wir $t \preccurlyeq t'$ voraussetzen, beim schreibenden Zugriff die gespiegelte Inlusion $t' \preccurlyeq t$.

Ein Gedankenexperiment: Unterschiede man mit Hilfe des Typsystems zwischen lesendem und schreibendem Zugriff — was weder Mini–F# noch F# machen —, ergäbe sich ein verfeinertes Bild.

$$\dfrac{\Sigma \vdash e : t}{\Sigma \vdash \textbf{\textit{ref}}\ e : \mathit{Ref}\langle t\rangle}$$

$$\dfrac{\Sigma \vdash e : \mathit{Read\text{-}Ref}\langle t\rangle}{\Sigma \vdash\ !e : t} \qquad\qquad \dfrac{\Sigma \vdash e_1 : \mathit{Write\text{-}Ref}\langle t\rangle \qquad \Sigma \vdash e_2 : t}{\Sigma \vdash e_1 := e_2 : \mathit{Unit}}$$

Wir verwenden drei (!) verschiedene Typen für Speicherzellen: $\mathit{Read\text{-}Ref}\langle t\rangle$ ist der Typ einer 'read-only' Speicherzelle; $\mathit{Write\text{-}Ref}\langle t\rangle$ ist der Typ einer 'write-only' Speicherzelle; und $\mathit{Ref}\langle t\rangle$ ist unverändert der Typ einer Speicherzelle, die sowohl les- als auch schreibbar ist. Da wir beide Aspekte getrennt modellieren, können wir festlegen:

$$\dfrac{t' \preccurlyeq t}{\mathit{Read\text{-}Ref}\langle t'\rangle \preccurlyeq \mathit{Read\text{-}Ref}\langle t\rangle} \qquad\qquad \dfrac{t \preccurlyeq t'}{\mathit{Write\text{-}Ref}\langle t'\rangle \preccurlyeq \mathit{Write\text{-}Ref}\langle t\rangle}$$

Der Typ $\mathit{Read\text{-}Ref}\langle t\rangle$ ist kovariant; $\mathit{Write\text{-}Ref}\langle t\rangle$ hingegen kontravariant; $\mathit{Ref}\langle t\rangle$ ist der Durchschnitt von $\mathit{Read\text{-}Ref}\langle t\rangle$ und $\mathit{Write\text{-}Ref}\langle t\rangle$.

$$\dfrac{}{\mathit{Ref}\langle t'\rangle \preccurlyeq \mathit{Read\text{-}Ref}\langle t\rangle} \qquad\qquad \dfrac{}{\mathit{Ref}\langle t'\rangle \preccurlyeq \mathit{Write\text{-}Ref}\langle t\rangle}$$

Eine les- und schreibbare Speicherzelle lässt sich zu einer 'read-only' oder einer 'write-only' Speicherzelle herabstufen. Einem Ausdruck kann auf diese Weise ein genauerer Typ zugeordnet werden, genauer und flexibler. Die Sinnhaftigkeit von 'read-only' Speicherzellen erschließt sich einem unmittelbar; unsere Bankkonten realisieren im Prinzip nichts anderes: Der Kontostand kann direkt mit *Balance* gelesen, aber nicht überschrieben werden. Aber wozu sind 'write-only' Speicherzellen gut? Denken Sie an die Implementierung der Fakultät aus Abschnitt 7.2, die ihr Ergebnis über eine Speicherzelle kommuniziert (engl. destination passing style). *Ende des Gedankenexperiments.*

8.2.3. Dynamische Semantik

Die Auswertung von Programmen ändert sich nicht.

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash (e :> t) \Downarrow \nu}$$

Typanpassungen spielen für die Abarbeitung keine Rolle.

8.2.4. Vertiefung

Untertypen sind eine Spielart der *Polymorphie*. Erinnern wir uns: Der Name Polymorphie kommt aus dem Griechischen ($\pi o \lambda \upsilon \mu o \rho \phi \iota \alpha$) und bedeutet Vielgestaltigkeit. Im Kontext von Programmiersprachen meint Polymorphie, dass ein Wert in unterschiedlichen Typkontexten verwendet werden kann. *Kurz:* ein Wert hat mehrere Typen.

Die polymorphe Funktion $length : List\,\langle 'a \rangle \to Nat$ kann zum Beispiel die Länge einer Liste eines beliebigen Elementtyps bestimmen; sie besitzt im Prinzip unendliche viele Typen.

$$\begin{aligned}
&length : List\,\langle Nat \rangle \to Nat\\
&length : List\,\langle Bool \rangle \to Nat\\
&length : List\,\langle List\,\langle Nat \rangle \rangle \to Nat\\
&length : List\,\langle List\,\langle Bool \rangle \rangle \to Nat\\
&\ldots
\end{aligned}$$

Da *length* einen heimlichen Typparameter hat, spricht man auch genauer von einer *parametrisch polymorphen Funktion* oder etwas weniger sperrig von einer *generischen Funktion* (engl. generic function).

Die Funktion *transfer* ist ebenfalls polymorph; sie kann auf Elemente eines beliebigen Untertyps von $IAccount$ angewendet werden.

$$\begin{aligned}
&transfer : IAccount * Nat * IAccount \to Unit\\
&transfer : IAccount * Nat * IAccountPlus \to Unit\\
&transfer : IAccountPlus * Nat * IAccount \to Unit\\
&transfer : IAccountPlus * Nat * IAccountPlus \to Unit\\
&\ldots
\end{aligned}$$

Die Zahl der Untertypen von $IAccount$ ist zwar endlich, aber unbegrenzt, da die Schnittstellenhierarchie zu jedem Zeitpunkt beliebig erweitert werden kann. Statt von Untertyppolymorphie spricht man aus diesem Grund auch von Schnittstellenvererbung (engl. interface inheritance).

Insgesamt unterscheidet man zwischen vier verschiedenen Arten von Polymorphie, siehe Abbildung 8.4. Zu den beiden prinzipiellen oder universellen Spielarten gesellen sich noch zwei weniger prinzipielle Formen, Überladung und Konversion, die wir im Folgenden kurz unter die Lupe nehmen wollen.

	universelle Polymorphie	ad-hoc Polymorphie
konjunktive Polymorphie	parametrische Polymorphie ($\forall$) (engl. generics) *length*	Überladung ($\wedge$) (engl. overloading) $=$, $+$
Inklusionspolymorphie	Untertypen ($\preccurlyeq$) (engl. subtyping) *transfer*	Konversion (engl. coercion) —

Abbildung 8.4.: Spielarten der Polymorphie.

Überladung Von *Überladung* (engl. overloading) spricht man, wenn der *gleiche* Bezeichner für *unterschiedliche* Funktionen bzw. allgemeiner für unterschiedliche Werte verwendet wird. In Mini–F# wird zum Beispiel '+' sowohl für die Addition von natürlichen Zahlen ($4711 + 815$), die Addition von Fließkommazahlen ($8.15 + 0.4711e2$), als auch für die Konkatenation von Strings (`"Hello, "` + `"world!"`) verwendet.

$(+) : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}$
$(+) : \mathit{float} \to \mathit{float} \to \mathit{float}$
$(+) : \mathit{String} \to \mathit{String} \to \mathit{String}$

Überladung ist die kleine Schwester der parametrischen Polymorphie. Ähnlich wie *length*, kann '+' auf Argumente unterschiedlichen Typs angewendet werden. Im Unterschied zu *length* wird die Operation aber jeweils ganz unterschiedlich implementiert; hinter den Kulissen ist jeweils eine andere Funktion am Werk. Im Gegensatz dazu wird im Fall von *length* unabhängig vom Typ immer der gleiche Programmcode ausgeführt („one size fits all").

Erinnern wir uns: Wir haben in Abschnitt 4.2.2 die natürlichen Zahlen auf zwei verschiedene Art und Weisen implementiert, einmal auf Grundlage des unären Zahlensystems (*Peano*) und ein zweites Mal auf Grundlage des binären Zahlensystems (*Leibniz*). Je nach konkreter Zahlenrepräsentation wird die Addition unterschiedlich umgesetzt. Dies gilt in gleicher Weise auch für andere Zahlentypen: Zahlen mit beschränkter Genauigkeit (Maschinenzahlen mit ihrer modularen Arithmetik), Fließkommazahlen, rationale Zahlen, komplexe Zahlen usw. Natürlich ist es naheliegend, sogar wünschenswert, jeweils das Symbol '+' zu verwenden, da allen Operationen ja gemeinsam ist, dass sie das mathematische Konzept der Addition umsetzen.

Ebenso wünschenswert wäre es natürlich, auch für numerische Konstanten, etwa die Zahl Null, jeweils das gleiche Symbol verwenden zu können. Das ist leider nicht erlaubt. Je nach Typ unterscheidet sich die lexikalische Syntax von Numeralen: 0 ist eine natürliche Zahl (*Nat*), 0.0 ist eine 64-Bit Fließkommazahl (*float*, doppelte Genauigkeit, engl. double precision). Für „ähnliche" Zahlentypen wird der Typ durch einen Suffix angezeigt: $0y$ ist eine vorzeichenbehaftete 8-Bit Zahl (*sbyte*), $0.0f$ ist eine 32-Bit Fließkommazahl

(*float32*, einfache Genauigkeit, engl. single precision).[3]

Wie bereits angesprochen, kann '+' auch für die Konkatenation von Strings verwendet werden. Eine Verwendung, die aus dem Rahmen fällt und den ad-hoc Charakter der Überladung unterstreicht. Die Addition von Zahlen und die Konkatenation von Strings haben wenig gemeinsam — beide Operationen sind assoziativ und haben ein neutrales Element (sie formen einen Monoid), aber das gilt auch für die Multiplikation von Zahlen, die Komposition von Funktionen usw. (Übrigens verwendet man in der Mathematik für die Konkatenation in der Regel kein additives, sondern ein multiplikatives Symbol: $s_1 \cdot s_2$. Auch in der Mathematik ist Überladung beliebt: Wir haben '$\cdot$' sowohl für die Konkatenation von Wörtern als auch für die Konkatenation von Sprachen verwendet.)

Die ad-hoc Natur der Überladung wird ebenfalls deutlich, wenn wir überladene Funktionen komponieren: ***fun*** $x \rightarrow x + x$. Wird hier eine Zahl verdoppelt oder ein String mit sich selbst konkateniert? Unterschiedliche Programmiersprachen reagieren unterschiedlich auf diese *Mehrdeutigkeit*. In F# wird willkürlich angenommen, dass eine ganze Zahlen verdoppelt wird: $Int \rightarrow Int$. Alternativ könnte der Ausdruck durch die statische Semantik zurückgewiesen werden, da die Mehrdeutigkeit nicht aufgelöst werden kann. Die Funktion ***fun*** $x \rightarrow x +$ `"?"` hingegen ist unproblematisch, da das zweite Argument die Mehrdeutigkeit behebt. Diese Problematik ist parametrisch polymorphen Funktionen nicht zu eigen: ***fun*** *xs* $\rightarrow$ *length xs* > 0 hat den Typ $List\,\langle'a\rangle \rightarrow Bool$ und erbt sozusagen die Flexibilität in der Anwendung von *length*. Parametrische Polymorphie ist universeller, weniger ad hoc.

Die bisherige Diskussion rankt sich um die *vordefinierte* Operation '+'; können überladene Funktionen auch selbst definiert werden? Die Antwort ist ein entschiedenes „Jein". Bezeichner als überladen zu kennzeichnen ist nicht möglich, allerdings dürfen Methodennamen überladen werden. In der folgenden Schnittstelle ist die Methode *Push* überladen.

```
type IStack<'elem> =
  interface
    abstract member IsEmpty : Bool
    abstract member Push    : 'elem → Unit
    abstract member Push    : List<'elem> → Unit
    abstract member Pop     : Unit → 'elem
    abstract member Top     : 'elem
  end
```

Mit *Push* kann sowohl ein einzelnes Element als auch alle Elemente einer Liste auf einen Stack abgelegt werden. Wie schon angesprochen, führen überladenene Bezeichner unter Umständen zu Mehrdeutigkeiten. Aus diesem Grund müssen sich die *Argumenttypen* der überladenen Methoden unterscheiden. (Zwei gleichnamige Methoden mit gleichen Argumenttypen, aber unterschiedlichen Ergebnistypen sind nicht erlaubt.)

Wir werden von diesem Feature allerdings keinen Gebrauch machen. Bei der Verwendungen von Überladung sollte man stets bedenken, dass nicht nur der Übersetzer Mehrdeutigkeiten auflösen muss, sondern auch menschliche Leser*innen eines Programms.

[3] Da F# von Haus aus keine natürlichen Zahlen kennt, bezeichnet 0 in Wirklichkeit eine ganze Zahl (*Int*); natürliche Zahlen werden mit dem Suffix N gekennzeichnet: $0N$.

Statt *Push* zu überladen, kann man im obigen Beispiel auch ohne allzu großen Komfortverlust *Push* und *PushMany* verwenden. Pointierter formuliert: Wer überladene Bezeichner einführt, ist nur zu faul, sich unterschiedliche Namen auszudenken. (Das trifft insbesondere auf den Autor des Skripts zu, der '·' sowohl für die Konkatenation von Wörtern als auch für die Konkatenation von Sprachen verwendet.)

Konversion Von *Konversion* (engl. coercion) spricht man, wenn ein Element eines Typs automatisch in ein Element eines anderen Typs umgewandelt wird. In vielen Programmiersprachen wird diese Bequemlichkeit für numerische Typen angeboten: *int*s werden zum Beispiel automatisch in *float*s überführt. Der Repräsentationswechsel kann unter Umständen mit einem Informationsverlust einhergehen. Aus diesem Grund wird Konversion in Mini–F# *nicht* unterstützt. Um eine ganze Zahl in eine Fließkommazahl zu konvertieren oder umgekehrt eine Fließkommazahl in eine ganze Zahl, muss eine Funktion aufgerufen werden — die Funktion heißt wie der Zieltyp der Umwandlung.

```
Mini⟩ float32 123456789
val it : float32 = 123456792.0f
Mini⟩ int it
val it : int = 123456792
```

Da eine 32-Bit Fließkommazahl nur $24 + 1$ Bits für die Mantisse (und 8 Bits für den Exponenten) zur Verfügung hat, kann ein 32-Bit Integer in der Regel nicht ohne Genauigkeitsverlust repräsentiert werden. Auch in der umgekehrten Richtung kann man böse Überraschungen erleben,

```
Mini⟩ int 1e10f
val it : int = -2147483648
Mini⟩ float32 it
val it : float32 = -2147483650f
```

etwa wenn eine sehr große positive Zahl plötzlich negativ wird.

Konversion ist die kleine Schwester des Untertyppolymorphismus. Eine Typumwandlung ließe sich durch eine Untertypregel einfangen.

$$\mathit{int} \preccurlyeq \mathit{float}$$

Da diese „Inklusion“ tatsächlich mit einem Wandel der Repräsentation einhergeht, müssten in der Folge sämtliche Auswertungsregeln für arithmetische Operationen angepasst werden! Aber, wie gesagt, Konversion wird in Mini–F# *nicht* unterstützt.

8.3. Klassen

module
Objects.
Bank

Ein Bankinstitut, eine Sammlung von Bankkonten, kann alternativ durch eine sogenannte *Klasse* (engl. class) modelliert werden.

```
type TrustMe (seed : Nat) =
  class
    let mutable funds = seed
    member self.Deposit (amount : Nat) =
      funds ← funds + amount
    member self.Withdraw (amount : Nat) =
      funds ← monus (funds, amount)
    member self.Balance =
      funds
  end
```

Die Definition sieht einem Objektkonstruktor sehr ähnlich: der interne Zustand, die Veränderliche *funds* wird mit Hilfe einer ***let***-Bindung eingeführt; daneben sind die uns mittlerweile schon vertrauten Methoden *Deposit*, *Withdraw* und *Balance* aufgeführt, die den internen Zustand manipulieren.

Ein Unterschied fällt allerdings ins Auge: *TrustMe* wird durch eine *Typ*definition eingeführt und nicht durch eine *Werte*definition. Klassendefinitionen sind die fünfte und letzte Spielart von Typen, die wir einführen, so dass an dieser Stelle ein kurzer Rück- bzw. Überblick angezeigt ist:

- ***type*** $T = \{\ell_1 : t_1, \ldots\}$
  führt einen Recordtyp ein;

- ***type*** $T =\ \mid C_1$ ***of*** $t_1 \mid \cdots$
  führt einen Variantentyp ein;

- ***type*** $T =$ ***interface abstract member*** $\ell_1 : t_1 \ldots$ ***end***
  führt eine Schnittstelle ein;

- ***type*** $T =$ ***class let*** $\ldots$ ***member*** $\ldots$ ***end***
  führt eine Klasse ein;

- ***type*** $T = t$
  führt ein Typsynonym ein, einen anderen Namen für t.

Kommen wir auf die Definition von *TrustMe* zurück. Etwas ungewöhnlich für eine Typdefinition hat *TrustMe* einen Werteparameter ähnlich wie eine Funktion — einen *Werte*parameter, nicht einen *Typ*parameter. Wie wir sehen werden, ist eine Klasse tatsächlich ein Zwitterwesen: *TrustMe* ist sowohl ein Typ, genauer: eine Schnittstelle, als auch ein Wert, der die Schnittstelle implementiert, ein Objektkonstruktor, der sogenannte *primäre Konstruktor* (engl. *primary constructor*). Damit ist vielleicht auch klar, dass die Typdefinition nicht unmittelbar ausgewertet wird. Selbst wenn wir die Definition als Funktion lesen, fehlt ja die Angabe des Startkapitals *seed*.

Objekterzeugung Ein Bankkonto wird eröffnet, indem wir *TrustMe* mit einem Parameter versorgen: Mit *TrustMe* 4711 wird ein neues Konto eröffnet, in das ein initialer Betrag von 4711 € eingezahlt wird. Wenn man betonen möchte, dass ein *neues* Objekt erschaffen wird, kann man auch ***new*** *TrustMe* 4711 schreiben.

```
Mini⟩ let lisas = TrustMe 4711
val lisas : TrustMe
Mini⟩ lisas.Deposit 815
()
Mini⟩ lisas.Balance
5526
Mini⟩ let ludwigs = new TrustMe 815
val ludwigs : TrustMe
```

Etwas Fachjargon: Statt „*TrustMe* 4711 ist ein Element des Typs *TrustMe*“ sagt man auch „*TrustMe* 4711 ist eine Instanz der Klasse *TrustMe*“. Klassen sind wie gesagt Zwitterwesen: *TrustMe* 4711 ist ein normaler Ausdruck und kann überall dort verwendet werden, wo Ausdrücke gefragt sind (der Ausdruck muss nicht notwendigerweise an einen Bezeichner gebunden werden); wird der Ausdruck abgearbeitet, entsteht ein Objekt des angegebenen Typs. Allgemein lässt sich eine Klassendefinition ***type*** *T* (*x*) = ***class*** . . . ***end*** als Schablone (engl. template) auffassen; mit ***new*** *T e* oder kurz *T e* wird die Schablone instantiiert; das Ergebnis ist ein Objekt vom Typ *T*.

Felder, Methoden und Eigenschaften Es ist hilfreich die Klassendefinition dem entsprechenden Objektkonstruktor aus Abschnitt 8.1 gegenüberzustellen.

```
let account seed =

  let mutable funds = seed

  { new IAccount with
      member self.Deposit amount =
        funds ← funds + amount

      member self.Withdraw amount =
        funds ← monus (funds, amount)

      member self.Balance =
        funds
  }
let lisas = account 4711
lisas.Deposit 815
```

```
type TrustMe seed =
  class
    let mutable funds = seed


    member self.Deposit amount =
      funds ← funds + amount

    member self.Withdraw amount =
      funds ← monus (funds, amount)

    member self.Balance =
      funds
  end
let lisas = TrustMe 4711
lisas.Deposit 815
```

In der praktischen Handhabung unterscheiden sich die Ansätze zunächst einmal nicht. (Wir werden in Abschnitt 8.3.1 sehen, dass sich tatsächlich Unterschiede ergeben, was

```
type TrustMe (seed : Nat) =
  class
    static do putline "TrustMe is founded."
    static let mutable no = 0
    do no ← no + 1
    let mutable funds = seed
    static member BIC = 4711
    static member total-no-of-accounts = no
    member self.Deposit (amount : Nat) =
      funds ← funds + amount
    member self.Withdraw (amount : Nat) =
      funds ← monus (funds, amount)
    member self.Balance =
      funds
  end
```

Abbildung 8.5.: Modellierung eines Bankinstituts (klassenbasiert, siehe auch Abbildungen 8.1 und 8.8).

die Interaktion mit anderen Objekten anbelangt.) Der wesentliche Unterschied ist ein syntaktischer oder wenn man möchte ein organisatorischer: Der interne Zustand und die nach außen sichtbare Schnittstelle rücken in der Klassendefinition enger zusammen; sie werden unter dem Dach der ***class*** ... ***end*** Klammer zusammengefasst.

Neben Definitionen von (veränderlichen) Werten, Methoden und Eigenschaften lassen sich in eine Klassendefinition weitere Aspekte integrieren, für die wir in Abschnitt 8.1 ein Modul verwendet haben. (Wir haben schon angedeutet, dass Modul- und Klassensysteme in einer gewissen Konkurrenz stehen, so dass es vielleicht nicht verwundert, dass es eine nicht unerhebliche Überschneidung von Sprachfeatures gibt.) Das sind Aspekte, die das Bankinstitut als Ganzes betreffen, nicht nur ein einzelnes Bankkonto: Wie lautet der BIC? Wieviele Konten sind insgesamt eröffnet worden? Abbildung 8.5 zeigt, wie der modulbasierte Programmcode aus Abbildung 8.1 für Klassen umgeschrieben wird. (Für das Verständnis der folgenden Diskussion ist es ratsam, noch einmal zurückzublättern und Abbildung 8.1 zu studieren.) Im Vergleich zum modulbasierten Code wird kein Studierendenkonto (*student-account*) angeboten — darauf kommen wir in Abschnitt 8.5 noch einmal zurück.

Im modulbasierten Programm werden zwei Veränderliche eingeführt: Die Gesamtzahl der Konten wird in *no* abgelegt, der Kontostand bzw. die Kontostände in *funds*. Die Speicherzelle *no* wird bei der Abarbeitung des Moduls angelegt und ist (fast) im ge-

samten Modul sichtbar. Anders verhält es sich mit *funds*: Bei jedem Aufruf von *account* wird eine neue Speicherzelle allokiert; der Bezeichner *funds* ist nur lokal in der Funktion sichtbar. Die Position der Bezeichner im Programmtext macht deutlich, dass *no* genau einmal existiert, wohingegen *funds* einmal pro erzeugtem Objekt existiert. In der klassenbasierten Variante werden die Definitionen der Speicherzellen zusammengeführt; sowohl *no* als auch *funds* werden lokal innerhalb der Klasse definiert. Der unterschiedliche Charakter der Speicherzellen muss aus diesem Grund mit anderen linguistischen Mitteln ausgedrückt werden: Das Schlüsselwort ***static***[4] zeigt an, dass *no* nur einmal pro Klasse existiert. Im Fachjargon sagt man auch, *no* ist eine *Klassenvariable*. Bei der Definition von *funds* fehlt hingegen das Schlüsselwort: Somit existiert *funds* einmal pro mit ***new*** erzeugtem Objekt. Im Unterschied zu *no* ist *funds* eine sogenannte *Instanzvariable*.

Ähnliche Überlegungen gelten für die Abarbeitung der mit ***do*** gekennzeichneten Ausdrücke. (Zur Erinnerung: ***do*** *e* ist eine Abkürzung für ***let*** () = *e* und kennzeichnet eine Definition, die nur um ihres Effektes willen eingeführt wird.) In der modulbasierten Variante bestimmt die Position, wie oft ein Ausdruck ausgewertet wird, in der klassenbasierten Variante das Schlüsselwort ***static***. Also, ein ***static do*** Block wird einmal bei der Abarbeitung der Klassendefinition (!) ausgeführt. Fehlt das Schlüsselwort, wird der Block einmal bei jeder Erzeugung eines Objekts ausgeführt.

Die bisherige Diskussion betrifft die interne Organisation einer Klasse bzw. von Objekten einer Klasse. Kommen wir zur externen „Präsentation“, der Schnittstelle. Diese umfasst zwei „Arten“ von Methoden und Eigenschaften: solche, die spezifisch für die Klasse sind, und solche, die spezifisch für ein einzelnes Objekt sind. Zu den ersteren gehören der BIC und die Auskunft über die Gesamtzahl der Konten. In der modulbasierten Variante werden diese Bezeichner auf Modulebene definiert, also *nicht* lokal zum Objektkonstruktor *account*. Die klassenbasierte Variante kennzeichnet sie erwartungsgemäß mit dem Schlüsselwort ***static***. Man spricht auch von *Klassenmethoden* und *Klasseneigenschaften*. Beide können unabhängig von der Existenz eines Objekts sprich eines Bankkontos verwendet werden — zum Zeitpunkt der Gründung einer Bank existieren noch keine Konten. Die jeweilige Nachricht wird an die Klasse in unserem Beispiel an die Bank geschickt.

```
Mini⟩ TrustMe.BIC
4711
Mini⟩ let lisas = TrustMe 4711
val lisas : TrustMe
Mini⟩ TrustMe.total-no-of-accounts
1
```

Kommen wir zur Sichtbarkeit von Bezeichnern: Welcher Bezeichner ist wo sichtbar? Die Regeln für die klassenbasierte Variante lassen sich aus der modulbasierten Variante ableiten: Zum Beispiel ist *funds* in *Withdraw*, nicht aber in *total-no-of-accounts*

[4] Der Name ist historisch bedingt: Mit ***static*** werden in der Programmiersprache C Variablen gekennzeichnet, die statisch vor der Programmausführung und nicht dynamisch während der Programmausführung (auf dem Stack oder mit `malloc()` auf dem Heap) allokiert werden.

sichtbar. Allgemein gilt, dass Instanzvariablen und -methoden nicht in den Definitionen von Klassenvariablen und -methoden sichtbar sind. Von außen betrachtet sind mit ***let*** eingeführte Konstanten, Funktionen und Veränderliche privat, das heißt *nicht* sichtbar und damit nicht zugreifbar; mit ***member*** eingeführte Methoden und Eigenschaften sind hingegen öffentlich, das heißt sichtbar und damit zugreifbar. Letzteres kann man verhindern, wenn man Mitglieder als ***private*** kennzeichnet. Dies ist nützlich, sogar angezeigt, um Implementierungsdetails zu verbergen. Zum Beispiel würde man eine Methode verbergen, die den Charakter einer Hilfsfunktion hat: Sie arbeitet anderen Methoden zu, soll aber nach außen nicht in Erscheinung treten.

Verschaffen wir uns einen Überblick über die Komponenten einer Klassendefinition.

- ***let*** $a = e$
  führt eine Konstante ein, einen Bezeichner für einen Wert (diese Konstante kann allerdings die Adresse einer Speicherzelle sein: ***let*** *funds* $=$ ***ref*** *seed*). Die Bindung wird bei der Erzeugung eines Objekts mit ***new*** ausgewertet. Eine Konstante ist stets privat und kann nicht öffentlich gemacht werden.

- ***let*** $f\ x = e$
  führt eine Funktion ein. Eine Funktion ist stets privat.

- ***let mutable*** $s = e$
  führt eine Instanzvariable ein, auch *Feld*[5] (engl. field) genannt. Auch eine Instanzvariable ist stets privat.

- ***do*** e
  kennzeichnet einen Ausdruck, der bei der Erzeugung eines Objekts mit ***new*** ausgeführt wird.

- ***member*** $self.p$ ***with*** *get* $() = e_1$ ***and*** *set* $v = e_2$
  member $self.p$ ***with private*** *get* $() = e_1$ ***and*** *set* $v = e_2$
  führt eine Eigenschaft ein; es kann auch nur der Getter bzw. nur der Setter angegeben werden. Eigenschaften sind öffentlich; die Sichtbarkeit kann aber mit ***private*** selektiv eingeschränkt werden.

- ***member*** $self.m\ x = e$
  member private $self.m\ x = e$
  führt eine Methode ein. Methoden sind ebenfalls öffentlich, es sei denn, die Sichtbarkeit wird mit ***private*** eingeschränkt.

- ***static let*** $a = e$
  führt eine „Klassenkonstante“ ein. Die Bindung wird bei der Abarbeitung der Klassendefinition ausgewertet; in e sind nur mit ***static*** gekennzeichnete Bezeichner sichtbar. Eine Konstante ist stets privat.

[5] Der Begriff „Feld“ wird auch für Array verwendet. Das eine hat mit dem anderen nichts zu tun.

- ***static let*** $f\ x = e$
  führt eine „Klassenfunktion“ ein. Im Rumpf e sind nur mit ***static*** gekennzeichnete Bezeichner sichtbar. Eine Funktion ist stets privat.

- ***static let mutable*** $s = e$
  führt eine Klassenvariable ein. Auch eine Klassenvariable ist stets privat.

- ***static do*** e
  kennzeichnet einen Ausdruck, der bei der Abarbeitung der Klassendefinition ausgeführt wird. In e sind nur mit ***static*** gekennzeichnete Bezeichner sichtbar.

- ***static member*** p ***with*** $get\ () = e_1$ ***and*** $set\ v = e_2$
  static member p ***with private*** $get\ () = e_1$ ***and*** $set\ v = e_2$
  führt eine Klasseneigenschaft ein. Auf die Eigenschaft wird über den Klassennamen zugegriffen. Auch Klasseneigenschaften sind öffentlich; die Sichtbarkeit kann aber mit ***private*** eingeschränkt werden.

- ***static member*** $m\ x = e$
  static member private $m\ x = e$
  führt eine Klassenmethode ein. Eine Klassenmethode ist der Natur nach eine ordinäre Funktion; lediglich der Aufruf erfolgt über den Klassennamen.

8.3.1. Klassen und Schnittstellen

Eine Klasse besteht aus einer öffentlichen Schnittstelle, der Sammlung aller öffentlichen Methoden und Eigenschaften, und einer privaten Implementierung, der Sammlung aller privaten Konstanten, Funktionen, Veränderlichen, Methoden und Eigenschaften. Der interne Zustand eines Objekts, einer Instanz, ist gekapselt und damit nach außen nicht sichtbar. Wir haben schon wiederholt gesehen, dass die gleiche Schnittstelle auf sehr unterschiedliche Art und Weise implementiert werden kann. Die folgende Implementierung einer Bank merkt sich alle Kontostände.

```
type Cheap'N'Easy (seed : Nat) =
  class
    let mutable funds-log = [seed]
    member self.Deposit (amount : Nat) =
      funds-log ← (head funds-log + amount) :: funds-log
    member self.Withdraw (amount : Nat) =
      funds-log ← monus (head funds-log, amount) :: funds-log
    member self.Balance =
      head funds-log
    member self.Cancel =
      funds-log ← tail funds-log
  end
```

Die Klasse bietet zusätzlich eine aus Sicht der Finanzwelt sehr abenteuerliche Operation an, mit der die jeweils letzte Transaktion rückgängig gemacht werden kann.

In Abschnitt 8.1 haben wir die Funktion *transfer* eingeführt, die eine Banküberweisung modelliert. Wenn wir diese Funktion einsetzen wollen, erleben wir eine unliebsame Überraschung: Weder ein Objekt vom Typ *TrustMe* noch ein Objekt vom Typ *Cheap'N'Easy* kann an die Funktion *transfer* übergeben werden.

```
transfer : IAccount * Nat * IAccount → Unit
TrustMe       4711 : TrustMe
Cheap'N'Easy 815   : Cheap'N'Easy
```

Die drei Typen, *IAccount*, *TrustMe* und *Cheap'N'Easy*, sind nicht miteinander kompatibel. Diese Tatsache macht sich zunächst an den Namen fest: Alle drei Typbezeichner sind verschieden. In Anlehnung an die Unabhängigkeitserklärung der Vereinigten Staaten sagt man auch „*all types are created unequal.*“[6] Hat sich der anfängliche Ärger über die Inkompatibilität gelegt, realisiert man, dass dies tatsächlich eine sinnvolle Festlegung ist: Die Gleichheit der Methodennamen könnte rein zufällig sein. Uns geht es aber nicht um Äußerlichkeiten (Wie heißt eine Methode?), sondern um innere Werte (Wie verhält sich eine Methode?). Die Instanzen der verschiedenen Klassen und die Elemente eines Schnittstellentyps sollten nicht nur die gleichen Nachrichten verstehen, sie sollten sich auch gleich oder zumindest ähnlich verhalten. Von der Eigenschaft *Balance* erwartet man, dass sie den aktuellen Kontostand zurückgibt und nicht etwa das Gepäck im Frachtraum eines Flugzeugs gleichmäßig verteilt. (Für den Rechner ist *Balance* ein Bezeichner bestehend aus sieben Buchstaben, der frei von jeglicher Bedeutung ist; Bedeutung, die für uns als menschliche Leserinnen und Leser immer mitschwingt.)

Möchten wir Instanzen der Klassen *TrustMe* und *Cheap'N'Easy* als Bankkonten verwenden, so müssen wir unsere Intention *explizit* machen; wir müssen bekanntgeben, dass eine Klasse eine Schnittstelle implementiert. Die Bekanntgabe lässt sich auf verschiedene Art und Weise organisieren. In der folgenden Überarbeitung der Klasse *Cheap'N'Easy* werden die Methoden der Schnittstelle *IAccount* *direkt* implementiert.

[6] Von dieser Regel gibt es nur eine Ausnahme: Ein Typsynonym wie zum Beispiel ***type*** *Age* = *Int* führt einen Bezeichner für einen bestehenden Typ ein; *Age* und *Nat* sind per definitionem gleich und somit beliebig austauschbar.

```
type Cheap'N'Easy (seed : Nat) =
  class
    let mutable funds-log = [seed]
    interface IAccount with
      member self.Deposit (amount : Nat) =
        funds-log ← (head funds-log + amount) :: funds-log
      member self.Withdraw (amount : Nat) =
        funds-log ← monus (head funds-log, amount) :: funds-log
      member self.Balance =
        head funds-log
    member self.Cancel =
      funds-log ← tail funds-log
  end
```

Aus den Klassenmethoden sind Methoden der Schnittstelle *IAccount* geworden. Nach der Ankündigung ***interface*** *IAccount* ***with*** wird konkretisiert, wie die Methoden *Deposit*, *Withdraw* und *Balance* implementiert werden. *Lies: Cheap'N'Easy* implementiert die Schnittstelle *IAccount*. Damit ist *Cheap'N'Easy* ein Untertyp von *IAccount*, als Formel

$$\mathit{Cheap'N'Easy} \preccurlyeq \mathit{IAccount}$$

Jetzt gelingt die Überweisung: Beim Aufruf der Funktion *transfer* wird der Typ automatisch mit Hilfe der Subsumptionsregel (8.2) angepasst.

```
Mini⟩ let hermines = Cheap'N'Easy 4711
Mini⟩ let harrys = Cheap'N'Easy 10
Mini⟩ transfer (hermines, 100, harrys)
()
Mini⟩ (harrys :> IAccount).Balance
110
Mini⟩ harrys.Cancel
()
Mini⟩ (harrys :> IAccount).Balance
10
```

Eine explizite Typanpassung ist notwendig, wenn auf die Methoden der Schnittstelle zugegriffen wird: Von Haus aus versteht *harrys* die Nachricht *Balance* nicht. Wir sehen in Kürze, warum der „Cast“ sinnvoll ist. Im Gegensatz dazu wird die Nachricht *Cancel* direkt verstanden. Möchte man aus Gründen der Bequemlichkeit den direkten Zugriff auf die Schnittstellenmethoden ermöglichen, kann man diese *zusätzlich* als Methoden der Klasse „veröffentlichen“.

```
type Cheap'N'Easy (seed : Nat) =
  class
    ...
    // expose interface
    member self.Deposit  amount = (self :> IAccount).Deposit  amount
    member self.Withdraw amount = (self :> IAccount).Withdraw amount
    member self.Balance         = (self :> IAccount).Balance
  end
```

Die Gleichungen geben jeweils an, dass die Methode der Klasse (links: *self*.*Withdraw*) durch die Schnittstellenmethode (rechts: (*self* :> *IAccount*).*Withdraw*) definiert wird. Die Typanpassung muss zwingend vorgenommen werden; lässt man sie weg, erfolgt ein rekursiver Aufruf mit der Konsequenz der Nichtterminierung!

Man kann auch den umgekehrten Weg beschreiten und die Schnittstellenmethoden durch Methoden der Klasse definieren. Die folgende Klassendefinition illustriert diese *indirekte* Implementierung einer Schnittstelle.

```
type TrustMe (seed : Nat) =
  class
    let mutable funds = seed
    member self.Deposit (amount : Nat)  = funds ← funds + amount
    member self.Withdraw (amount : Nat) = funds ← monus (funds, amount)
    member self.Balance                 = funds
    interface IAccount with
      // these are not recursive definitions
      member self.Deposit  amount = self.Deposit  amount
      member self.Withdraw amount = self.Withdraw amount
      member self.Balance         = self.Balance
  end
```

Die Gleichungen geben jeweils an, dass die Schnittstellenmethode (links: *self*.*Withdraw*) durch die Methode der Klasse (rechts: *self*.*Withdraw*) definiert wird. Die Gleichungen sehen verdächtig nach rekursiven Definitionen aus, sind aber keine: die Nachricht *Deposit* wird an das Objekt der Klasse *TrustMe* geschickt; damit wird die weiter oben definierte Methode aufgerufen und abgearbeitet.

Nach diesen Vorarbeiten lassen sich sowohl Banküberweisungen tätigen als auch die Kontostände direkt ohne Umweg über die Schnittstelle abrufen.

```
Mini⟩ let hermines = TrustMe 4711
Mini⟩ let harrys = Cheap'N'Easy 0
transfer (hermines, 100, harrys)
()
Mini⟩ hermines.Balance
4611
Mini⟩ harrys.Balance
100
```

Der direkte Zugriff mit *Balance* ist möglich, weil entweder die Schnittstelle bekanntgegeben wurde oder die Schnittstelle mit den Methoden der Klasse implementiert wurde.

Eine Klasse kann auch mehrere Schnittstellen gleichzeitig implementieren.

```
type IPositive =
  abstract Answer : string
type INegative =
  abstract Answer : string
type Undecided () =     // don't forget '()'
  class
    member self.Answer = "maybe"
    interface IPositive with
      member self.Answer = "yup"
    interface INegative with
      member self.Answer = "nope"
  end
```

Die Methode *Answer* wird dreimal definiert: einmal als Methode der Klasse *Undecided*, ein zweites Mal als Methode der Schnittstelle *IPositive* und ein drittes Mal als Methode der Schnittstelle *INegative*. Die Schnittstellen repräsentieren unterschiedliche Stimmungen; je nach konkreter Stimmung fällt die „Antwort“ unterschiedlich aus.

```
Mini⟩ let turncoat = Undecided ()
Mini⟩ turncoat.Answer
val it : string = "maybe"
Mini⟩ (turncoat :> IPositive).Answer
val it : string = "yup"
(turncoat :> INegative).Answer
val it : string = "nope"
```

Damit wird klar, warum bei Methodenaufrufen *keine* automatischen Typanpassungen vorgenommen werden (*Undecided* $\preccurlyeq$ *IPositive* oder *Undecided* $\preccurlyeq$ *INegative*): Kein Automatismus kann die vorhandene *Mehrdeutigkeit* auflösen.

Nun ist das Beispiel etwas künstlich; ein ähnliches Szenario tritt aber durchaus auf, etwa wenn ein Objekt in verschiedene *Rollen* schlüpft: Eine Person kann in der Rolle einer Student*in oder in der Rolle einer Tutor*in auf die gleichen Nachricht unterschiedlich reagieren. (Die Methodenamen könnten auch nur zufällig übereinstimmen — neue

Namen zu erfinden ist, wie schon gesagt, schwer.) Erst die Angabe der Rolle, sprich der Schnittstelle, macht unzweideutig klar, was mit der Nachricht gemeint ist.

8.3.2. Parametrisierte Klassen⟍Generische Klassen

module
Objects.
Dictionary

Wie Record-, Varianten- und Schnittstellentypen können auch Klassentypen mit einem oder mehreren Typen parametrisiert werden. Kurzum: Typdefinitionen jeglicher Couleur können mit Typparametern versehen werden.

In Abschnitt 5.3 haben wir endliche Abbildungen mit Hilfe von Listen, Suchlisten und Suchbäumen implementiert. Zur Erinnerung ein Auszug aus der Schnittstelle:

```
type Map ⟨'key, 'value when 'key : comparison⟩
val empty : Map ⟨'key, 'value⟩
val add    : 'key * 'value → Map ⟨'key, 'value⟩ → Map ⟨'key, 'value⟩
val lookup : 'key → Map ⟨'key, 'value⟩ → 'value option
```

Die genannten Implementierungen sind *persistent*: Wird eine endliche Abbildung um ein Schlüssel-Wert Paar erweitert, so wird als Ergebnis die erweiterte Abbildung zurückgegeben — die ursprüngliche Abbildung und die neue Umgebung koexistieren friedlich und können unabhängig voneinander verwendet und weiterentwickelt werden.

Im Folgenden implementieren wir eine ephemerale Variante von endlichen Abbildungen: Beim Erweitern wird die ursprüngliche Abbildung überschrieben; sie ist nach der Operation *nicht* mehr verfügbar. Genauer: Um die ephemerale Variante zu realisieren, verwenden wir die persistente Implementierung von endlichen Abbildungen auf Basis von Suchbäumen (Abschnitt 5.3.3).

Die folgende Interaktion zeigt die Implementierung in Aktion.

```
Mini⟩ let perfume = SearchTree ⟨Nat, String⟩ ()
Mini⟩ perfume.Add (4711, "Kölnisch Wasser")
()
Mini⟩ perfume.Add (5, "Chanel")
()
Mini⟩ perfume.Lookup 5
"Chanel"
Mini⟩ perfume.[5]
"Chanel"
Mini⟩ perfume.[0] ← "Eau"
()
Mini⟩ perfume.[0]
"Eau"
Mini⟩ perfume.[0] ← "Wasser"
()
Mini⟩ perfume.[0]
"Wasser"
```

Das Wörterbuch *perfume* ordnet natürlichen Zahlen Strings zu. Mit der Methode *Add* wird das Wörterbuch um einen Eintrag erweitert; *Lookup* schlägt im Wörterbuch nach. Die ephemerale Natur wird deutlich: *Add* hat den Ergebnistyp *Unit*; zu jedem Zeitpunkt existiert nur eine Version der endlichen Abbildung *perfume*.

Für beide Operationen bieten wir syntaktischen Zucker an: $perfume.[k] \leftarrow v$ fügt ein neues Schlüssel-Wert Paar hinzu (bzw. aktualisiert einen bestehenden Eintrag für den Schlüssel k); $perfume.[k]$ schlägt den angegebenen Schlüssel nach.

```
type SearchTree⟨'key,'value when 'key : comparison⟩ () =
  class
    let mutable mapping : Map⟨'key,'value⟩ = empty
    member self.Add (key, value) =
      mapping ← add (key, value) mapping
    member self.Lookup key =
      match lookup key mapping with
      | None       → raise (KeyNotFoundException ())
      | Some value → value
    member self.Item
      with get key       = self.Lookup key
      and  set key value = self.Add (key, value)
  end
```

Ähnlich wie eine polymorphe Funktion (siehe Abschnitt 4.3.2) ist die Klasse *SearchTree* sowohl mit Typen, *'key* und *'value*, als auch mit einem Wert, dem Dummy '()', parametrisiert: *SearchTree* ist ein *polymorpher Objektkonstruktor*. Der interne Zustand ist durch die Veränderliche *mapping* gegeben, die von *Add* modifiziert und von *Lookup* gelesen wird. Zusätzlich führt *Lookup* eine Umkodierung durch: An die Stelle eines optionalen Rückgabewerts tritt eine Ausnahme.

Der syntaktische Zucker für Erweiterung und Zugriff wird mittels der Eigenschaft *Item* realisiert. Es handelt sich um eine sogenannte *indizierte Eigenschaft* (engl. indexed property). Sowohl der Getter als auch der Setter sind (zusätzlich) mit dem Schlüssel parametrisiert. So wie die Eigenschaft ℓ die Abkürzungen $e.\ell$ und $e.\ell \leftarrow e'$ ermöglicht, so erlaubt eine indizierte Eigenschaft den syntaktischen Zucker $e.\ell\ [i]$ und $e.\ell\ [i] \leftarrow e'$. Ist weiterhin $\ell = Item$, dann kann man ℓ auch auslassen und kurz $e.[i]$ und $e.[i] \leftarrow e'$ schreiben.

8.4. Aufzähler und aufzählbare Objekte

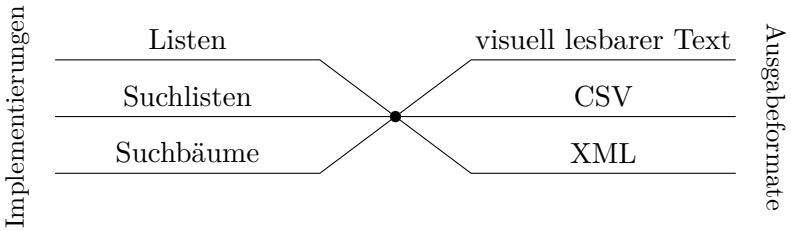
module
Objects.
Enumerator

Nehmen wir an, wir wollen das Wörterbuch aus Abschnitt 8.3.2 um die Möglichkeit erweitern, die verwalteten Schlüssel-Wert Paare auszugeben — entweder flüchtig auf dem Bildschirm oder dauerhaft in eine Datei. Eine unscheinbare Aufgabe, die aber ganz unterschiedlich angegangen werden kann. Tatsächlich eine instruktive Aufgabe, so dass wir uns verschiedene Designs für die Schnittstelle anschauen und ihre jeweiligen Vor- und Nachteile diskutieren wollen.

Möglichkeit A: Wir erweitern die Klasse um eine maßgeschneiderte Methode.

```
member Print : Unit → Unit
```

Die Erweiterung erfüllt sicherlich ihren Zweck, ist aber zu kurz gedacht. Zum Einen sind viele verschiedene Ausgabeformate denkbar: visuell lesbarer Text (engl. human readable text); durch Komma getrennte Werte (Comma-Separated Values, kurz CSV); Formate zur Darstellung hierarchisch strukturierter Daten (engl. Extensible Markup Language, abgekürzt XML). Zum Anderen kann ein Wörterbuch auf verschiedene Art und Weise implementiert werden: durch Listen, Suchlisten oder Suchbäume, siehe Abschnitt 5.3.



Der obige Ansatz führt zu einer kombinatorischen Explosion von Programmieraufgaben. Gibt es m verschiedene Implementierungen und n verschiedene Ausgabeformate, dann müssen $m \cdot n$ Methodenrümpfe mit Leben gefüllt werden, unweigerlich mit sich wiederholenden Programmfragmenten.

Möglichkeit B: Wir überführen die Daten des Wörterbuchs in ein einheitliches Zwischenformat (einen Mediator), zum Beispiel eine Liste oder ein Array von Schlüsseln, das dann von den verschiedenen Ausgabefunktionen weiterverarbeitet wird.

```
member Keys : Unit → List ⟨'key⟩
```

Ein Gewinn an Modularität: Konzeptionell wird die Verwaltung des Wörterbuchs von Funktionen zur Ein- und Ausgabe getrennt (engl. separation of concerns); praktisch reduziert sich der Implementierungsaufwand auf $m+n$ Methoden- bzw. Funktionsrümpfe. Jede Implementierung muss die Methode *Keys* umsetzen; für jedes Ausgabeformat wird eine entsprechende listenverarbeitende Funktion programmiert. Allerdings gibt es einen Nachteil: Der Ansatz benötigt zusätzlichen Speicherplatz. Die Daten, die bereits im Wörterbuch abgelegt sind, werden faktisch dupliziert, indem sie zusätzlich in einer Liste bereitgestellt werden.

Möglichkeit C: Wir ersetzen Listen durch eine alternative Darstellung von Sequenzen, die es uns erlaubt, schrittweise und *bedarfsgetrieben* ein Element nach dem anderen zu generieren.

```
member Keys : Unit → IEnumerator ⟨'key⟩
```

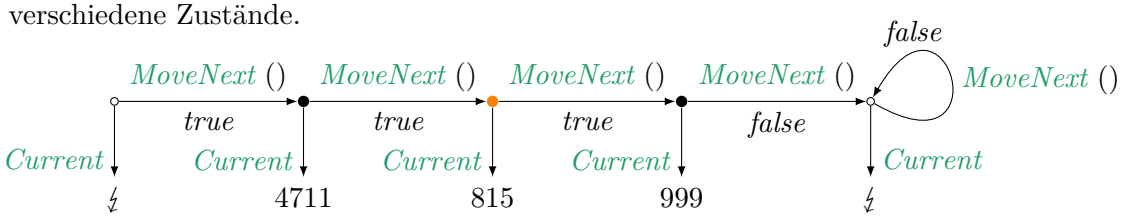
Der Sequenztyp *IEnumerator* firmiert unter vielen verschiedenen Namen — „Aufzähler“ (engl. enumerator), „Wiederholer“ (engl. iterator), „Cursor“ (engl. cursor), „faule Liste“ (engl. lazy list) und Generator — und wird vielfältig genutzt, so dass es sich lohnt, das Konzept genauer unter die Lupe zu nehmen.

Aufzähler Abstrakt gesprochen ersetzen wir eine *Datenstruktur* (eine Liste oder ein Array) durch eine *Kontrollstruktur* (einen Aufzähler). Aus einem *Datum*, das alle Elemente der repräsentierten Sequenz umfasst, wird ein *Programm*, das es erlaubt, alle Elemente nacheinander aufzuzählen. Konkret ist ein Aufzähler ein Objekt, dass zwei Nachrichten versteht.

```
type IEnumerator ⟨'elem⟩ =
  interface
    abstract member Current   : 'elem
    abstract member MoveNext : Unit → Bool
  end
```

Die Methode *Current* gibt das aktuelle Element der Aufzählung zurück; *MoveNext* () rückt zum nächsten Element vor und signalisiert über den Rückgabewert, ob ein Folgeelement tatsächlich existiert. Wird *Current* wiederholt aufgerufen, ohne zwischenzeitlich *MoveNext* aufzurufen, wird stets das identische Element zurückgegeben.

Bei der Verwendung der Schnittstelle muss man sich an ein striktes *Protokoll* halten: *vor* dem ersten Zugriff mit *Current* muss zunächst mit *MoveNext* () sichergestellt werden, dass überhaupt ein Element existiert; der Zugriff auf das erste sowie die folgenden Elemente ist nur zulässig, wenn *MoveNext* () grünes Licht gibt, also *true* zurückgibt. Ein Aufzähler der dreielementigen Sequenz 4711 815 999 durchlebt somit insgesamt 5 (!) verschiedene Zustände.



Der „Cursor“ der Aufzählung kann entweder auf ein Element zeigen oder vor dem ersten bzw. nach dem letzten Element stehen. Der Wert von *Current* ist vor dem ersten Aufruf von *MoveNext* und nach einem Aufruf von *MoveNext*, der *false* zurückgibt, undefiniert, angezeigt durch den Wurf einer Ausnahme.

```
let not-started () =
  invalidOp "Enumeration has not started. Call MoveNext."
let already-finished () =
  invalidOp "Enumeration already finished."
```

Die Funktion *invalidOp*, eine Variante von ***raise***, zeigt an, dass eine Operation ungültig ist, in unserem Fall, dass die Operation nicht gemäß des vereinbarten Protokolls verwendet wird.

Auch auf die Gefahr hin, das Offensichtliche zu benennen: Aufzähler sind zustandsbehaftet; je nach Zustand wird *Current* in der Regel verschiedene Elemente zurückgeben. Der Cursor kann nur von links nach rechts bewegt werden, aber nicht zurück; ist der Cursor ans Ende gelangt, ist die Aufzählung beendet und damit der Aufzähler praktisch

nutzlos (*Current* wirft eine Ausnahme; *MoveNext* gibt *false* zurück) oder etwas drastischer formuliert: Objektmüll.

Kommen wir zur Implementierung von Aufzählern. Der folgende Objektkonstruktor repräsentiert das Intervall *lower* .. *upper*.

```
let range (lower, upper) =
  let mutable started = false
  let mutable current = lower
  {new IEnumerator⟨Int⟩ with
    member self.Current =
      if started then
        if current ⩽ upper then current
                           else already-finished ()
      else
        not-started ()
    member self.MoveNext () =
      if started then
        current ← current + 1
      else
        started ← true
      current ⩽ upper
  }
```

Die Position des Cursors wird durch zwei Zustandsvariablen repräsentiert: *started* gibt an, ob *MoveNext* bereits aufgerufen wurde; *current* ist der aktuelle Zahlenwert im oder rechts vom Intervall. Die Implementierung ist *robust*: Falls die Benutzer*in sich nicht an das Protokoll hält, wird sie mehr oder weniger dezent darauf hingewiesen. Man sagt auch, die Funktion *range* ist *defensiv programmiert*: Annahmen und Voraussetzungen werden explizit überprüft. Der Nachteil der defensiven Herangehenweise liegt in einer gewissen Redundanz: Der Test *current* ⩽ *upper* wird zum Beispiel sowohl in *Current* als auch in *MoveNext* durchgeführt. Vertraut man hingegen darauf, dass die Klient*in sich an den Vertrag hält (Stichwort: *design by contract*), programmiert man sehr viel kürzer:

```
let mutable current = lower − 1          // „undo“ first call to MoveNext
...
member self.Current      = current
member self.MoveNext () = current ← current + 1; current ⩽ upper
```

Die Funktion *from-list*: *List* ⟨*'elem*⟩ → *IEnumerator* ⟨*'elem*⟩ zählt die Elemente einer Liste auf, siehe Abbildung 8.6. Sie implementiert somit einen *Repräsentationswechsel*: Eine Darstellung von Sequenzen, Listen, wird in eine andere überführt, Aufzählungen. Die Implementierungsdetails sind ähnlich wie im Fall von *range*: Die Veränderliche *started* gibt an, ob *MoveNext* bereits aufgerufen wurde; *suffix* enthält einen Suffix, ein Endstück, der aufzuzählenden Liste.

Die implizite Darstellung einer Sequenz durch ein Programm hat gegenüber der expliziten Darstellung durch ein Datum den Vorteil, dass sich auch *unendliche* Sequenzen

```
let non-empty = function
    | [] → false
    | _ → true
let from-list (list : List ⟨'elem⟩) : IEnumerator ⟨'elem⟩ =
    let mutable started = false
    let mutable suffix  = list
    { new IEnumerator ⟨'elem⟩ with
        member self.Current =
            if started then
                match suffix with
                | []      → already-finished ()
                | hd :: tl → hd
            else
                not-started ()
        member self.MoveNext () =
            if started then
                match suffix with
                | []      → false
                | hd :: tl → suffix ← tl
                             non-empty suffix
            else
                started ← true
                non-empty suffix
    }
```

Abbildung 8.6.: Aufzählung einer Liste.

repräsentieren lassen. Der folgende Objektkonstruktor zählt zum Beispiel *alle* natürlichen Quadratzahlen auf.

```
let squares () =
  let mutable started = false
  let mutable current = 0
  let mutable odd     = 1
  {new IEnumerator⟨Nat⟩ with
    member self.Current =
      if started then
        current
      else
        not-started ()
    member self.MoveNext () =
      if started then
        current ← current + odd
        odd ← odd + 2
      else
        started ← true
      true
  }
```

Die Methode *MoveNext* gibt immer *true* zurück; die Aufzählung terminiert nie.

Aufzählbare Objekte Kommen wir noch einmal auf den Ausgangspunkt unserer Überlegungen zurück. Möglichkeit C sieht vor, eine Methode $Keys: Unit \to IEnumerator\,\langle'key\rangle$ zum Wörterbuch hinzuzufügen. Nun sind viele „Dinge“ aufzählbar, die Elemente einer Zahlenfolge (die Primzahlen, die Catalanzahlen usw.), die Elemente eines Containers (eines Arrays, einer Liste, eines Stacks usw.), so dass es sich anbietet, die Abstraktionsleiter noch eine Stufe hinaufzusteigen und eine allgemeine Schnittstelle für *aufzählbare Objekte* zu definieren.

```
type IEnumerable⟨'elem⟩ =
  interface
    abstract member GetEnumerator : Unit → IEnumerator⟨'elem⟩
  end
```

An die Stelle des anwendungsspezifischen Bezeichners *Keys* ist der neutrale Bezeichner *GetEnumerator* getreten.

Wir haben noch nicht thematisiert, warum *Keys* bzw. *GetEnumerator* eine Funktion ist, die einen Aufzähler zurückgibt, und nicht direkt ein Aufzähler ist. Aufzählungen sind wie schon erwähnt zustandsbehaftet und damit „flüchtig“ (ephemeral): Sobald der Cursor mit *MoveNext* voranbewegt wird, kann auf das vorherige Element nicht mehr zugegriffen werden; ist der Cursor ans Ende gelangt, ist die Aufzählung beendet. Die Methode *GetEnumerator* erwartet ein Dummyargument, um bei jedem Aufruf einen

„frischen“ Aufzähler zurückgeben zu können — sie generiert sozusagen Generatoren. Das ist nützlich, ja unabdingbar, wenn zum Beispiel ein Wörterbuch in verschiedenen Formaten ausgegeben werden soll — für jedes Format muss die Liste aller Schlüssel einmal durchlaufen werden. Oder wenn sich das Wörterbuch im Laufe der Zeit ändert; *GetEnumerator* erstellt jeweils einen *Schnappschuss* der aktuell verfügbaren Schlüssel. Das ist übrigens leichter gesagt als getan. (Nachdenken!) In Kürze mehr dazu.

Die einheitliche Schnittstelle erlaubt es, die Verarbeitung von aufzählbaren Objekten mit etwas syntaktischem Zucker zu versüßen: mit Hilfe einer ***for***-Schleife können wir über alle Elemente einer Aufzählung iterieren.

```
for x in xs do
  body
```

```
let enumerator = xs.GetEnumerator ()
while enumerator.MoveNext () do
  let x = enumerator.Current
  body
```

Die ***for***-Schleife zur Linken ist eine Abkürzung für die ***while***-Schleife zur Rechten; die Aufrufe der Methoden *GetEnumerator*, *MoveNext* und *Current* werden so geschickt verborgen. Einen Wermutstropfen gibt es allerdings: ***for***-Schleifen, die über allgemeine Aufzählungen iterieren, geben nicht länger ein Terminierungsversprechen `:-(` — die Aufzählung muss ja nicht endlich sein (siehe *squares*). Eine ***for***-Schleife iteriert immer bis zum Ende einer Aufzählung; möchte man diese vorzeitig abbrechen, etwa weil ein gesuchtes Element gefunden wurde, muss an die Stelle einer ***for***-Schleife eine ***while***-Schleife mit einer entsprechenden Abbruchbedingung treten.

Die beiden Schnittstellen in Verbindung mit dem syntaktischem Zucker erlauben es, eine *generische Funktion* zu schreiben, die die Elemente einer Aufzählung addiert.

```
let sum (xs : IEnumerable⟨Nat⟩) : Nat =
  let mutable acc = 0
  for x in xs do
    acc ← acc + x
  acc
```

Das Attribut „generisch“ ist angebracht, da *sum* für beliebige aufzählbare Objekte funktioniert, insbesondere für Listen und Arrays.

Implementiert das Wörterbuch die obige Schnittstelle, so können wir die Schlüssel-Wert Paare mit Hilfe einer ***for***-Schleife ausgeben.

```
Mini⟩ let phone-book = SearchTree⟨String, Nat⟩ ()
Mini⟩ phone-book.Add ("Lisa", 815)
()
Mini⟩ phone-book.Add ("Harry", 4711)
()
Mini⟩ for key in phone-book do
          putline (key ^ "'s phone number is " ^ show phone-book.[key])
Harry's phone number is 4711
Lisa's phone number is 815
```

Der noch zu definierende Aufzähler generiert die Schlüssel in lexikographischer Reihenfolge, so dass die Einträge sortiert ausgegeben werden.

module Objects. Sequence

Sequenzausdrücke, da capo Wir haben gesehen, dass die Definition von Aufzählern des Typs *IEnumerator ⟨'elem⟩* recht mühsam ist. Die Programme der letzten Seiten haben einen dediziert *präskriptiven* Charakter: Es wird jeweils genau detailliert, *wie* das nächste Element in Abhängigkeit vom aktuellen Zustand generiert und der Zustand aktualisiert wird. Alternativ können Sequenzen *deskriptiv* programmiert werden, indem man beschreibt, *was* die Elemente der Sequenz sind. Zu diesem Zweck können wir *Sequenzbeschreibungen* verwenden. Die Syntax kennen wir schon von Listen- und Arraybeschreibungen, siehe Abschnitt 7.3.1. Der einzige syntaktische Unterschied besteht in der umschließenden Klammer: An die Stelle von [. . .] bzw. [| . . . |] tritt *seq* { . . . }. Zum Beispiel korrespondiert *seq* {*lower* .. *upper*} zu dem Aufzähler *range* (*lower*, *upper*) bzw. genauer zu dem aufzählbaren Objekt *Range* (*lower*, *upper*).

```
let Range (lower : Int, upper : Int) =
  {new IEnumerable⟨Int⟩ with
     member self.GetEnumerator () = range (lower, upper)
  }
```

Intervalle eignen sich übrigens prima, um die Unterschiede zwischen Sequenzen als Daten (*List ⟨'a⟩* oder *Array ⟨'a⟩*) und Sequenzen als Programme (*IEnumerable ⟨'a⟩*) zu illustrieren. Stellt man dem Mini–F# Interpreter die Anfrage [1 . . 100000000], tritt eine längere Stille ein, da hinter den Kulissen eine Liste mit einhundert Millionen Elementen aufgebaut wird. Die korrespondierende Anfrage *seq* {1 . . 100000000} wird hingegen in Windeseile ausgerechnet: Das Ergebnis ist ein Programm, das die einhundert Millionen Elemente bedarfsgetrieben peu à peu generiert.

Die Folge aller Quadratzahlen lässt sich mit Hilfe von Sequenzbeschreibungen wie folgt definieren.

```
let squares =
  let rec generate (current, odd) =
    seq {yield current; yield! generate (current + odd, odd + 2)}
  generate (0, 1)
```

Aus dem internen Zustand eines Objekts (siehe Seite 435) sind Parameter einer Hilfsfunktion geworden. Der Sequenzausdruck ***yield*** *x*; ***yield!*** *xs* entspricht der Listenkonstruktion *x* :: *xs*. Zur Erinnerung: ***yield!*** *e* ist eine Abkürzung für ***for*** *x* ***in*** *e* ***do yield*** *x* und zählt die Elemente der Sequenz *e* auf.

Tatsächlich lässt sich *squares* noch direkter und eleganter formulieren:

```
let squares = seq {for n in nats do yield n * n}
```

wobei *nats* die Folge aller natürlichen Zahlen ist. Die Folge der natürlichen Zahlen selbst lässt sich entweder präskriptiv mit einem Objektausdruck (siehe Aufgabe 8.4) oder deskriptiv mit Hilfe einer rekursiven *Werte*definition programmieren.

```
let rec nats =
  seq {yield 0
       for n in nats do
         yield n + 1}
```

Die Definition fängt Giuseppe Peanos Beschreibung der natürlichen Zahlen ein: Eine natürliche Zahl ist entweder 0 oder der Nachfolger $n + 1$ einer natürlichen Zahl n.

Kommen wir zur Implementierung der Klasse *SearchTree*; die Definition des Aufzählers für die Schlüssel des Wörterbuchs steht noch aus. Da das Wörterbuch durch einen binären Suchbaum gegeben ist, müssen wir im Wesentlichen den Inorder-Durchlauf aus Abschnitt 5.3.4 adaptieren.

```
let rec inorder = function
  | Leaf           -> Seq.empty
  | Node (l, x, r) -> seq {yield! inorder l; yield x; yield! inorder r}
```

Im Fall eines Blatts, geben wir die leere Sequenz zurück. (Leider ist *seq { }* *nicht* zulässig, so dass wir auf die Bibliotheksfunktion *Seq.empty* zurückgreifen müssen.) Anderenfalls zählen wir die Elemente des linken Teilbaums, das Wurzelelement und die Elemente des rechten Teilbaums auf.

Die vollständige Implementierung des Wörterbuchs ist in Abbildung 8.7 aufgeführt. Die Hilfsfunktion *enumerate-keys* entspricht im Wesentlichen *inorder*, zählt aber die Schlüssel eines Suchbaums auf, nicht die Elemente. (Aus historischen Gründen müssen zwei Schnittstellen implementiert werden: die getypte Schnittstelle *IEnumerable ⟨'key⟩* und die ungetypte Schnittstelle *IEnumerable*. Letztere wird auf Erstere zurückgeführt.)

Wir haben schon angesprochen, dass *GetEnumerator* einen *Schnappschuss* der Schlüsselfolge zum Zeitpunkt des Aufrufs erstellt. In unserem Fall ist dies einfach zu bewerkstelligen, da die zugrundeliegende Implementierung von Binärbäumen *persistent* ist. Die Knoten eines Binärbaums werden niemals überschrieben, so dass der Aufzähler ohne Probleme über den jeweils aktuellen Binärbaum iterieren kann — zwar wird die Veränderliche *mapping* durch *Add* modifiziert, nicht aber der unterliegende Binärbaum.

Würden wir die persistente Implementierung von Binärbäumen durch eine ephemerale ersetzen (analog zu den ephemeralen Listen aus Abschnitt 7.2.5), dann ließe sich die „Schnappschuss-Semantik“ nicht länger mit vertretbarem Aufwand realisieren. (Nachdenken!) Aus diesem Grund greift man zu einer drastischen Alternative: Wird während der Aufzählung der Schlüssel das zugrundeliegende Wörterbuch verändert, wird die Rechnung unmittelbar abgebrochen und eine Ausnahme geworfen (engl. fail-fast behaviour)!

8.4.1. Abstrakte Syntax

Wir erweitern Ausdrücke um Sequenzbeschreibungen.

$e \in \mathsf{Expr} ::=$	*Ausdrücke:*
$\quad \mid \; seq\,\{\,se\,\}$	Sequenzbeschreibungen

```
let enumerate-keys (Rep tree) =
  let rec inorder = function
    | Leaf           → Seq.empty
    | Node (l, x, r) → seq { yield! inorder l; yield x.key; yield! inorder r }
  inorder tree
type SearchTree⟨'key,'value when 'key : comparison⟩ () =
  class
    let mutable mapping : Map⟨'key,'value⟩ = empty
    member self.Add (key, value) =
      mapping ← add (key, value) mapping
    member self.Lookup key =
      match lookup key mapping with
      | None       → raise (KeyNotFoundException ())
      | Some value → value
    member self.Item
      with get key       = self.Lookup key
      and set key value = self.Add (key, value)
    interface IEnumerable⟨'key⟩ with
      member self.GetEnumerator () =
                  (enumerate-keys mapping).GetEnumerator ()
    interface IEnumerable with     // needed for historical reasons
      member self.GetEnumerator () =
                  (self :> IEnumerable⟨'key⟩).GetEnumerator () :> IEnumerator
  end
```

Abbildung 8.7.: Implementierung eines Wörterbuchs.

Innerhalb der Klammern steht ein *Sequenzausdruck*, dessen Syntax in Abschnitt 7.3.1 definiert ist.

8.4.2. Statische Semantik

Sequenzbeschreibungen besitzen den Typ $seq\,\langle t\rangle$, ein Alias für den Schnittstellentyp $IEnumerable\,\langle t\rangle$ bzw. genauer $System.Collections.Generic.IEnumerable\,\langle t\rangle$.

$t \in \mathsf{Type} ::=$		*Typen:*
	$\mid\ seq\,\langle t\rangle$	Sequenztyp

Listen und Arrays implementieren die Schnittstelle $IEnumerable\,\langle t\rangle$ alias $seq\,\langle t\rangle$ und sind damit Untertypen von $seq\,\langle t\rangle$.

$$\frac{}{List\,\langle t\rangle \preccurlyeq seq\,\langle t\rangle} \qquad \frac{}{Array\,\langle t\rangle \preccurlyeq seq\,\langle t\rangle}$$

Die Klammer $seq\,\{\ldots\}$ überführt einen Sequenzausdruck in einen Aufzähler.

$$\frac{\Sigma \vdash^* se : t}{\Sigma \vdash seq\,\{se\} : seq\,\langle t\rangle}$$

Die Typregeln für Sequenzausdrücke, $\Sigma \vdash^* se : t$, sind in Abschnitt 7.3.1 aufgeführt.

8.4.3. Dynamische Semantik

Bei der Festlegung der dynamischen Semantik beschreiten wir im Allgemeinen zwei unterschiedliche Wege. Die Bedeutung von Sprachkonstrukten der „Kernsprache“, wie zum Beispiel der Alternative $\textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3$, wird durch Beweisregeln festgelegt. Die Bedeutung von „syntaktischem Zucker“, also Sprachkonstrukten, die zwar bequem aber nicht lebensnotwendig sind, wird mittels Übersetzung in die Kernsprache festgelegt. Die bequeme Formulierung wird erklärt, indem sie auf die unbequeme Variante der Kernsprache zurückgeführt wird.

Sequenzbeschreibungen sind ein gutes Beispiel für den zweiten Ansatz. Sequenzbeschreibungen verwenden die gleiche Syntax wie Listenbeschreibungen. Letztere haben wir in Abschnitt 7.3.1 auf vordefinierte Listenfunktionen abgebildet. Listen repräsentieren Sequenzen; da aufzählbare Objekte eine andere Repräsentation von Sequenzen sind, müssen wir in den semantischen Gleichungen lediglich die Listenfunktionen durch korrespondierende Funktionen auf dem Typ seq ersetzen:

Die Sequenzbeschreibung $seq\,\{se\}$ wird in den Ausdruck $[\![se]\!]$ übersetzt, wobei $[\![se]\!]$ wie folgt definiert ist.

$$\begin{aligned}
[\![\textbf{\textit{yield}}\ e]\!] &= Seq.singleton\ e \\
[\![\textbf{\textit{yield!}}\ e]\!] &= e \\
[\![d\ \textbf{\textit{in}}\ se]\!] &= d\ \textbf{\textit{in}}\ [\![se]\!] \\
[\![se_1; se_2]\!] &= Seq.append\ ([\![se_1]\!])\ ([\![se_2]\!]) \\
[\![\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ se]\!] &= \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ [\![se]\!]\ \textbf{\textit{else}}\ Seq.empty \\
[\![\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ se_1\ \textbf{\textit{else}}\ se_2]\!] &= \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ [\![se_1]\!]\ \textbf{\textit{else}}\ [\![se_2]\!] \\
[\![\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e\ \textbf{\textit{do}}\ se]\!] &= Seq.collect\ (\textbf{\textit{fun}}\ x \to [\![se]\!])\ e
\end{aligned}$$

Die Implementierung der Bibliotheksfunktionen *Seq.empty* usw. ist zwar kein Hexenwerk, aber sehr mühsam — der syntaktische Zucker steht ja nicht zur Verfügung, die aufzählbaren Objekte müssen präskriptiv mit zustandsbehafteten Objektausdrücken realisiert werden —, so dass wir uns auf ein illustratives Beispiel beschränken wollen: die Konkatenation *Seq.append*.

Wir sind in diesem Abschnitt die Abstraktionsleiter zwei Stufen hinaufgestiegen und haben zwei Schnittstellentypen definiert: Aufzähler, *IEnumerator*, und aufzählbare Objekte, *IEnumerable*. Entsprechend müssen wir für *append* zwei Objekte definieren. Entweder durch geschachtelte Objektausdrücke oder etwas modularer mit Hilfe zweier Objektkonstruktoren wie unten (siehe auch die Definition von *range* und *Range*).

```
let append-enumerators (first : IEnumerator ⟨'elem⟩) (second : IEnumerator ⟨'elem⟩) =
  let mutable first-active = true
  {new IEnumerator ⟨'elem⟩ with
    member self.Current =
      if first-active then
        first.Current
      else
        second.Current
    member self.MoveNext () =
      if first-active then
        first-active ← first.MoveNext ()
        first-active || second.MoveNext ()
      else
        second.MoveNext ()
  }
```

Wir merken uns in einer Veränderlichen, ob der erste Aufzähler aktiv ist. Wenn nicht, werden die Nachrichten an den zweiten Aufzähler delegiert.

```
let append (first : IEnumerable ⟨'elem⟩) (second : IEnumerable ⟨'elem⟩) =
  {new IEnumerable ⟨'elem⟩ with
    member self.GetEnumerator () =
      append-enumerators (first.GetEnumerator ()) (second.GetEnumerator ())
  }
```

Noch ein abschließendes Wort zur Laufzeit: aufzählbare Objekte können in konstanter Zeit konkateniert werden: *Seq.append* (*seq* {1 .. 1000000}) (*seq* {**yield** 0}) wird in wenigen Schritten ausgerechnet. Die Laufzeit hängt insbesondere *nicht* von der Länge der beteiligten Sequenzen ab, sondern von der Schachtelungstiefe der Generatoren! Tief geschachtelte Aufzähler führen zu langen Nachrichtenketten: Die Nachricht *Current* zum Beispiel wird von übergeordneten Aufzählern nach „unten“ weitergereicht, so dass einige Zeit vergehen kann, bis das erste Element tatsächlich generiert wird: Der Aufruf *inorder* (*left-skewed* (1.000.000, `"Hello, world!"`)) zeigt das gleiche quadratische Verhalten, wie die listenbasierte Variante aus Abschnitt 5.3.4 (siehe auch Aufgabe 8.5).

module
Objects.
Permutations

8.4.4. Vertiefung

Testen Harry Hacker hat einen neuen Sortieralgorithmus entwickelt: *lightning-sort*. Lisa Lista, die mit einer gesunden Skepsis ausgestattet ist, schlägt vor, das Programm vor dem produktiven Einsatz systematisch zu testen. Sequenzbeschreibungen eignen sich wunderbar für die Generierung geeigneter Testdaten.

```
for n in 0 .. 10 do
  for xs in permutations [1 .. n] do
    if lightning-sort xs <> List.sort xs then
      raise (Panic "Harry!")
```

Die Funktion *permutations* : $List\,\langle'a\rangle \to seq\,\langle List\,\langle'a\rangle\rangle$ generiert systematisch alle Permutationen der angegebenen Liste. Zur Erinnerung: Zu einer Liste der Länge n existieren n Fakultät verschiedene Permutationen. Um nicht gigantische Mengen an Speicherplatz zu verbrauchen, ist es essentiell, dass eine *Sequenz* von Listen und nicht etwa eine *Liste* von Listen zurückgegeben wird. Auf diese Weise wird Harrys Programm erschöpfend für alle Eingabelisten der Länge $\leqslant 10$ getestet. Summa summarum werden 4.037.914 Tests durchgeführt. Wie lassen sich die Permutationen systematisch generieren?

Wenden wir das verallgemeinerte Peano Entwurfsmuster auf das Permutationsproblem an. Um die Problemgröße zu reduzieren, müssen wir ein Element der zu permutierenden Folge zur Seite legen. *Zwei* kanonische Ansätze drängen sich auf: Wir entfernen das erste Element der Eingabe *oder* wir wählen das erste Element der Ausgabe. Zwei Möglichkeiten, zwei Permutationsverfahren:

Permutieren durch Einfügen:

- Lege das erste Element zur Seite,
- bilde alle Permutationen der restlichen Elemente,
- *füge* das erste Element in jede Permutation nacheinander an allen möglichen Positionen *ein*.

Permutieren durch Auswählen:

- *Wähle* nacheinander alle Elemente *aus*,
- bilde jeweils alle Permutationen der restlichen Elemente,
- setze das entfernte Element vor jede korrespondierende Permutation.

Die beiden Verfahren sind im gewissen Sinne „dual“ zueinander. Beim Permutieren durch Einfügen ist der erste Schritt einfach und der letzte Schritt aufwändig; beim Permutieren durch Auswählen ist es genau umgekehrt.

Nehmen wir an, wir wollen alle Ziffern der Zahl 1234 permutieren. Permutieren durch Einfügen entfernt die erste Ziffer, die ①, und berechnet die $3! = 6$ Permutationen der restlichen Ziffern: 234, 324, 342, 243, 423, 432. In jede dieser Permutationen müssen wir ① an allen 4 Positionen einfügen. Insgesamt erhalten wir $3! \cdot 4 = 4! = 24$ Permutationen, die auf der linken Seite aufgeführt sind.

①234	2①34	23①4	234①
①324	3①24	32①4	324①
①342	3①42	34①2	342①
①243	2①43	24①3	243①
①423	4①23	42①3	423①
①432	4①32	43①2	432①

①234	①243	①324	①342	①423	①432
②134	②143	②314	②341	②413	②431
③214	③241	③124	③142	③421	③412
④231	④213	④321	④312	④123	④132

Beim Permutieren durch Auswählen wird nacheinander die erste Ziffer der Ausgabe ausgewählt: zuerst ①, dann ②, dann ③ und schließlich ④. Für jede Auswahl werden die $3! = 6$ Permutationen der restlichen Ziffern bestimmt und jeweils an die erste Ziffer angehängt. Wiederum erhalten wir insgesamt $4 \cdot 3! = 4! = 24$ Permutationen, die auf der rechten Seite aufgeführt sind.

Das Einfügen an allen Positionen und die Generierung aller Permutationen selbst lassen sich bequem mit Sequenzbeschreibungen formulieren.

```
let rec insertions x = function
  | []              → seq {yield [x]}
  | xs & (y :: ys) → seq {yield (x :: xs); for zs in insertions x ys do yield y :: zs}
let rec ipermutations = function
  | []      → seq {yield []}
  | x :: xs → seq {for ys in ipermutations xs do
                     for zs in insertions x ys do yield zs}
```

Wie *ipermutations* ist auch die Hilfsfunktion *insertions* streng nach dem Struktur Entwurfsmuster für Listen gestrickt. Ist die Liste, in die x einzufügen ist, leer, dann gibt es nur ein Ergebnis: die einelementige Liste $[x]$. Anderenfalls setzen wir x entweder an die erste Position, $x :: xs$, oder wir fügen x rekursiv in die Restliste ein.

```
Mini⟩ ipermutations [1..4]
seq {[1;2;3;4];[2;1;3;4];[2;3;1;4];[2;3;4;1];
     [1;3;2;4];[3;1;2;4];[3;2;1;4];[3;2;4;1];
     [1;3;4;2];[3;1;4;2];[3;4;1;2];[3;4;2;1];
     [1;2;4;3];[2;1;4;3];[2;4;1;3];[2;4;3;1];
     [1;4;2;3];[4;1;2;3];[4;2;1;3];[4;2;3;1];
     [1;4;3;2];[4;1;3;2];[4;3;1;2];[4;3;2;1]}
```

Kommen wir zum Permutieren durch Auswählen. Die Hilfsfunktion *deletions* gibt neben dem ausgewählten Element zusätzlich die Liste der restlichen Elemente zurück.

```
let rec deletions = function
  | []      → Seq.empty
  | x :: xs → seq {yield (x, xs); for (y, ys) in deletions xs do yield (y, x :: ys)}
let rec dpermutations = function
  | [] → seq {yield []}
  | xs → seq {for (y, ys) in deletions xs do
                for zs in dpermutations ys do yield y :: zs}
```

Die Dualität der beiden Verfahren lässt sich auch am Programmcode festmachen. Im Rekursionsfall wird hier *erst* ein Element ausgewählt, *deletions xs*, *dann* erfolgt der rekursive Aufruf *dpermutations ys*. Beim Permutieren durch Einfügen vertauscht sich die Reihenfolge: *Erst* erfolgt der rekursive Aufruf, *ipermutations xs*, *dann* wird das Kopfelement eingefügt, *insertions x ys*.

```
Mini⟩ dpermutations [1 .. 4]
seq {[1; 2; 3; 4]; [1; 2; 4; 3]; [1; 3; 2; 4]; [1; 3; 4; 2]; [1; 4; 2; 3]; [1; 4; 3; 2];
     [2; 1; 3; 4]; [2; 1; 4; 3]; [2; 3; 1; 4]; [2; 3; 4; 1]; [2; 4; 1; 3]; [2; 4; 3; 1];
     [3; 1; 2; 4]; [3; 1; 4; 2]; [3; 2; 1; 4]; [3; 2; 4; 1]; [3; 4; 1; 2]; [3; 4; 2; 1];
     [4; 1; 2; 3]; [4; 1; 3; 2]; [4; 2; 1; 3]; [4; 2; 3; 1]; [4; 3; 1; 2]; [4; 3; 2; 1]}
```

Ein Problem, zwei algorithmische Lösungen. Welches Verfahren ist vorzuziehen? Bezüglich der Laufzeit gibt es keine Unterschiede. Trotzdem lässt sich ein klarer Gewinner ausmachen: Permutieren durch Auswählen, weil es die Permutationen in *lexikographischer Reihenfolge* generiert.

8.5. Vererbung

> *I fear the new object-oriented systems may suffer the fate of LISP, in that they can do many things, but the complexity of the class hierarchies may cause them to collapse under their own weight.*
>
> — Bill Joy

In ähnlicher Weise wie Schnittstellen durch „Vererbung“ verbreitert werden können, so lassen sich auch Klassen um zusätzliche Funktionalität erweitern. Die folgende Klassendefinition erweitert die uns wohlbekannte Klasse *TrustMe* um eine Methode, mit der sich ein Konto „leerräumen“ lässt.

```
type TrustMeGold (seed : Nat) =
  class
    inherit TrustMe (seed)
    member self.Clear () : Nat =
      let amount = self.Balance
      self.Withdraw amount
      amount
  end
```

Die ***inherit*** Klausel muss als erste Deklaration im Rumpf der Klassendefinition aufgeführt werden. Sie macht *TrustMeGold* zu einer *Unterklasse* (engl. subclass) der Klasse *TrustMe*; umgekehrt wird *TrustMe* zu einer *Oberklasse* (engl. superclass) der Klasse *TrustMeGold* bzw. zu deren *Basisklasse* (engl. base class). Die neu definierte Klasse erbt die *Implementierung* der Methoden der Oberklasse und fügt diesen eine weitere hinzu: *Clear*. Aus diesem Grund spricht man auch von *Implementierungsvererbung* (engl.

implementation inheritance), im Gegensatz zur reinen *Schnittstellenvererbung* (engl. interface inheritance). Während eine Schnittstelle *mehrere* andere Schnittstellen beerben kann — die Schnittstelle *Mobile* erweitert sowohl *Phone* als auch *MP3Player* —, darf eine Klasse nur von *einer* Oberklasse erben. Im Fachjargon: Schnittstellen unterstützen *Mehrfachvererbung*, wohingegen Klassen lediglich *Einfachvererbung* zulassen. Eine Klasse kann allerdings mehrere, auch voneinander unabhängige Schnittstellen implementieren. Auf diesen Aspekt kommen wir später noch einmal zurück.

Klassen sind wie schon mehrfach angesprochen Zwitterwesen, sowohl Schnittstelle als auch Implementierung. Lesen wir die Klassendefinition mit der „Schnittstellenbrille“, dann etabliert die ***inherit***-Klausel eine Untertypbeziehung:

$$\mathit{TrustMeGold} \preccurlyeq \mathit{TrustMe}$$

Setzen wir die „Implementierungsbrille“ auf, erkennen wir eine Aufrufstruktur: Wird mit ***new*** ein Objekt der Klasse *TrustMeGold* erzeugt, so wird zunächst der Objektkonstruktor *TrustMe* aufgerufen. Der Parameter *seed* des Konstruktors wird unverändert weitergereicht — dies ist aber nicht zwingend.

8.5.1. Redefinition⧹Overriding

Das Bankinstitut „Trust Me“ bietet neben dem Standardkonto zusätzlich ein Konto für Studierende an, das wir bisher noch nicht realisiert haben. In der objektbasierten Implementierung, siehe Abbildung 8.1, haben wir das Studierendenkonto mit Hilfe von *Delegation* umgesetzt. In der klassenbasierten Implementierung ist dies in gleicher Weise möglich (siehe Aufgabe 8.7). Alternativ können wir die zweite Kontoart mit Hilfe einer Unterklasse realisieren.

```
type TrustMeStudent (seed : Nat, limit : Nat) =
  class
    inherit TrustMe (seed)
    let limit = min limit 1000
    override self.Withdraw (amount : Nat) =
      if amount > limit then raise Limit
                        else base.Withdraw amount
  end
```

Streng genommen ist die abgeleitete Klasse keine Erweiterung von *TrustMe*, da keine neue Funktionalität hinzukommt. Stattdessen wird die in der Basisklasse eingeführte Methode *Withdraw* *redefiniert*. Das Schlüsselwort ***override*** zeigt an, dass sich über die bereits bestehende Definition hinweggesetzt wird (engl. override). In der Redefinition von *Withdraw* wird allerdings auf die ursprüngliche Definition mit ***base***.*Withdraw* zugegriffen. Der Zusatz ***base*** ist notwendig, da nunmehr zwei unterschiedliche Definitionen der Methode existieren: Mit *self*.*Withdraw* erfolgt ein *rekursiver* Aufruf der Methode der abgeleiteten Klasse, ***base***.*Withdraw* ruft hingegen die Methode der Basisklasse auf. (Der formale Parameter *limit* des Objektkonstruktors wird übrigens auch redefiniert:

Die Bindung ***let*** *limit* = *min limit* 1000 verwendet den Parameter und verschattet ihn dann — so wird das Transaktionslimit auf 1000 € oder einen kleineren Betrag gesetzt.)

Die Basisklasse muss allerdings Reimplementierungen zulassen — die bisherige Klassendefinition verbietet diese, so dass eine Änderungen notwendig wird (die vollständige Implementierung des Bankinstitut „Trust Me“ ist in Abbildung 8.8 aufgeführt):

```
type TrustMe (seed : Nat) =
  class
    ...
    abstract member Withdraw : Nat → Unit
    default self.Withdraw (amount : Nat) =
      funds ← monus (funds, amount)
    ...
  end
```

Wie in einer Schnittstellendefinition spezifizieren wir die Signatur von *Withdraw*. Zusätzlich wird eine *Standarddefinition* (engl. default definition) angegeben. Wie wir später sehen werden, ist die mit dem Schlüsselwort ***default*** gekennzeichnete Definition optional. Eine abstrakte Methode mit einer Standarddefinition heißt auch *virtuelle Methode* (engl. virtual method).

8.5.2. Klassen und Schnittstellen

Eine Klasse kann eine oder mehrere, auch voneinander unabhängige Schnittstellen implementieren. Die Klassenhierarchie kann dabei unabhängig von der Schnittstellenhierarchie entwickelt werden. Die Organisation der Implementierung kann die Schnittstellenhierarchie widerspiegeln, siehe Abbildung 8.9. Dies ist aber nicht zwingend: Eine Klasse kann zum Beispiel eine Unterschnittstelle implementieren, ohne eine Unterklasse zu sein, siehe Abbildung 8.10. Die Definition der Klasse *OnlineBank* in Abbildung 8.10 illustriert übrigens ein weiteres Sprachfeature. Die Ersteinzahlung wird mittels eines ***do***-Blocks vorgenommen, in dem die Methode *Deposit* aufgerufen wird. Der für die Selbstnachricht benötigte Bezeichner *self* wird mit Hilfe einer ***as***-Klausel auf der linken Seite der Typdefinition eingeführt. Wie immer ist der Bezeichner frei wählbar und steht für das erzeugte Objekt selbst.

Für den Fall, dass Sie den Überblick verloren haben: Die folgende Grafik fasst die bisher eingeführten Schnittstellen und Klassen in einem sogenannten *Klassendiagramm* zusammen.

```
type TrustMe (seed : Nat) =
  class
    static do putline "TrustMe is founded."
    static let mutable no = 0
    do no ← no + 1
    let mutable funds = seed
    static member BIC = 4711
    static member total-no-of-accounts = no
    member self.Deposit (amount : Nat) =
      funds ← funds + amount
    abstract member Withdraw : Nat → Unit
    default self.Withdraw (amount : Nat) =
      funds ← monus (funds, amount)
    member self.Balance =
      funds
    interface IAccount with
      // these are not recursive definitions
      member self.Deposit  amount = self.Deposit  amount
      member self.Withdraw amount = self.Withdraw amount
      member self.Balance         = self.Balance
  end
type TrustMeStudent (seed : Nat) =
  class
    inherit TrustMe (seed)
    let limit = 1000
    override self.Withdraw (amount : Nat) =
      if amount > limit then raise Limit
                        else base.Withdraw amount
  end
```

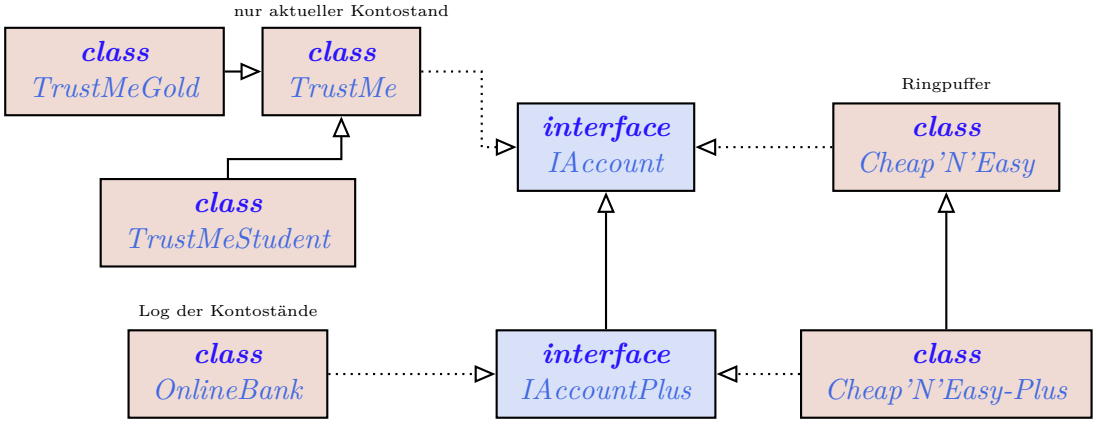
Abbildung 8.8.: Modellierung eines Bankinstituts (klassenbasiert, mit Vererbung).

```
type Cheap'N'Easy (seed : Nat) =
  class
    let mutable funds-log = [seed]
    static member BIC = 1234
    member internal self.Log = funds-log
    interface IAccount with
      member self.Deposit (amount : Nat) =
        funds-log ← (head funds-log + amount) :: funds-log
      member self.Withdraw (amount : Nat) =
        funds-log ← monus (head funds-log, amount) :: funds-log
      member self.Balance =
        List.head funds-log
    member self.Cancel =
      funds-log ← List.tail funds-log
    // expose interface
    member self.Deposit  amount = (self :> IAccount).Deposit  amount
    member self.Withdraw amount = (self :> IAccount).Withdraw amount
    member self.Balance         = (self :> IAccount).Balance
  end
type Cheap'N'Easy-Plus (seed : Nat) =
  class
    inherit Cheap'N'Easy (seed)
    interface IAccountPlus with
      member self.Statement =
        [| for x in take 10 self.Log → x |]
    // expose interface
    member self.Statement = (self :> IAccountPlus).Statement
  end
```

Abbildung 8.9.: Parallele Schnittstellen- und Klassenhierarchie.

```
type OnlineBank (seed : Nat, size : Int) as self =      // introduce name for object
  class
    let size         = min size 365
    let history      = [| for k in 0 .. size − 1 → 0 |]
    let mutable i = 0
    let next ()      = (i + 1) % size
    do (self :> IAccount).Deposit seed                  // initial deposit
    static member BIC = 0815
    interface IAccount with
      member self.Deposit (amount : Nat) =
        history.[next ()] ← history.[i] + amount; i ← next ()
      member self.Withdraw (amount : Nat) =
        history.[next ()] ← monus (history.[i], amount); i ← next ()
      member self.Balance =
        history.[i]
    interface IAccountPlus with
      member self.Statement =
        [| for k in 0 .. size − 1 → history.[(size + i − k) % size] |]
  end
```

Abbildung 8.10.: Eine Klasse implementiert eine Schnittstellenhierarchie.



Durchgezogene Pfeile zeigen an, dass eine Schnittstelle (Klasse) von einer anderen Schnittstelle (Klasse) abgeleitet wird bzw. erbt. Gepunktete Pfeile halten fest, dass eine Klasse eine Schnittstelle implementiert. Klassendiagramme sind übrigens Bestandteil der „Unified Modeling Language“ (kurz UML), einer Modellierungssprache für Software. Zu diesem Thema erfahren Sie im weiteren Verlauf des Studiums mehr aus der Abteilung „Software Engineering“.

8.5.3. Abstrakte Klassen

Bei der Definition einer Klasse können eine oder mehrere Methoden als abstrakt markiert werden (siehe zum Beispiel Abbildung 8.8). Gibt man keine ***default***-Implementierung für eine dieser Methoden an, so wird die Klasse selbst *abstrakt* (engl. abstract class): Es können keine Instanzen der Klasse erzeugt werden. Ein solches Objekt wäre ja nicht in der Lage, auf eine entsprechende Nachricht zu reagieren, da nur die Absicht erklärt wurde, eine Methode bereitzustellen. Erst wenn man dieser Absicht nachkommt und in einer Unterklasse der abstrakten Klasse die Methode(n) realisiert, dann können Objekte mit ***new*** erzeugt werden. In diesem Abschnitt schauen wir uns zwei Beispiele für abstrakte Klassen an, die allgemeine objektorientierte Programmiertechniken illustrieren: das sogenannte Beobachter-Entwurfsmuster (engl. observer design pattern, auch listener design pattern) und das Schablonen-Entwurfsmuster (engl. template design pattern).

module
Objects.
Observer

Beobachter-Entwurfsmuster In Abschnitt 7.2.4 haben wir mit Hilfe einer Speicherzelle einen einfachen Schrittzähler realisiert, den wir verwendet haben, um den Aufwand für die Berechnung der Fibonacci-Funktion abzuschätzen. Im Folgenden reimplementieren wir den Zähler, allerdings mit einem zusätzlichen Pfiff: Veränderungen des Zählers sollen beobachtbar sein, so dass wir auf Änderungen frühzeitig reagieren können. Das folgende Beispiel illustriert die Idee.

```
let steps = counter ()
do steps.Attach (fun n → if n % 1000 = 1 then putchar '*')
let rec fibonacci (n : Nat) : Nat =
  steps.State ← steps.State + 1
  if n ⩽ 1 then n
           else fibonacci (n - 1) + fibonacci (n - 2)
```

Mit *counter* () erzeugen wir einen neuen Zähler, an den ein Beobachter angeheftet wird, der eine einfache „Fortschrittsanzeige“ (auch Fortschrittsbalken) realisiert: Nach jeweils 1000 Schritten wird ein Asteriskus ‘*’ ausgegeben.

```
Mini⟩  steps.State ← 0
val it : Unit = ()
Mini⟩  fibonacci 20
**********************val it = 987
Mini⟩  steps.State
val it = 21891
```

Ein *Beobachter* ist eine Prozedur, die einen Zustand verarbeitet. Aus Gründen der Allgemeinheit parametrisieren wir Beobachter mit dem Typ des Zustands.

```
type Observer ⟨'state⟩ = 'state → Unit
```

Das Subjekt, das wir beobachten wollen, wird durch eine abstrakte Klasse realisiert. Diese kümmert sich um die Verwaltung von Beobachtern und deren Benachrichtigung — wir erlauben, dass sich mehrere Beobachter registrieren. Die Verwaltung des eigentlichen Zustands bleibt hingegen abstrakt.

```
[< AbstractClass >]
type Subject ⟨'state⟩ () =
  class
    let mutable observers = []
    abstract member State : 'state with get, set
    member self.Attach (observer : Observer ⟨'state⟩) =
      observers ← observer :: observers
      observer self.State       // notify observer
    member internal self.Notify () =
      for observer in observers do
        observer self.State
  end
```

Die Programmierer*in muss ihre Absicht, eine abstrakte Klasse zu definieren, *explizit* machen: Das sogenannte *Attribut* (engl. attribute) [< *AbstractClass* >] kennzeichnet *Subject* als abstrakt. Auf diese Weise wird sichergestellt, dass nicht aus Versehen die Bereitstellung einer ***default***-Methode versäumt wird. Die interne Methode *Notify*, mit der

Nachrichten an die Beobachter versendet werden, ist nicht für den Endbenutzer gedacht, sondern wird lediglich Unterklassen zur Verfügung gestellt. (Grob gesprochen schränkt ***internal*** die Sichtbarkeit der Methode auf das gleiche Modul bzw. die gleiche Bibliothek ein. Andere Programmiersprachen bieten den Zugriffsmodifizierer ***protected*** an, der die Sichtbarkeit gezielt auf die Klasse und deren Unterklassen einschränkt. Dieser würde an dieser Stelle benötigt, ist in F# aber leider nicht verfügbar.)

Kommen wir zur Implementierung des eigentlichen Zählers. Eine abstrakte Klasse kann entweder durch eine Unterklasse konkretisiert werden oder „on the fly“ durch einen Objektausdruck. Letztere verwendet man typischerweise, wenn die notwendigen Ergänzungen wie in unserem Fall nicht allzu umfangreich sind und/oder wenn viele verschiedene Konkretisierungen vorgenommen werden sollen (für jede Unterklasse müsste man sich ja einen Namen ausdenken — neue Namen zu erfinden ist, wie schon mehrfach erwähnt, schwer).

```
let counter () =
  let mutable n = 0
  { new Subject⟨Nat⟩ () with
      member self.State
        with set k = n ← k
                     self.Notify ()
        and  get () = n
  }
```

Der Zähler ist ein Subjekt vom Typ *Nat*, der die Eigenschaft *State* mit entsprechenden Getter und Setter-Methoden konkretisiert. Wir haben uns mehr oder weniger willkürlich dafür entschieden, nur schreibende Zugriffe zu beobachten (via *Notify*), nicht aber lesende.

module Objects. Template

Schablonen-Entwurfsmuster Beim Beobachter-Entwurfsmuster wird der zu beobachtende Zustand zu einer abstrakten Eigenschaft; alle anderen Mitglieder der Klasse *Subject* sind konkret. Im nächsten Beispiel verkehrt sich die Situation: Alle Methoden sind abstrakt, mit einer Ausnahme, die das Skelett eines Algorithmus definiert.

```
[< AbstractClass >]
type TwoPlayerGame () =
  class
    abstract member Finished : Bool
    abstract member Position : Unit → Unit
    abstract member Move     : Bool → Unit
    abstract member Winner   : Bool → Unit
    member self.Play () =
      try
        let mutable turn = false
        while not self.Finished do
          self.Position ()
          turn ← not turn
          self.Move turn
        self.Winner turn
      with
       | EOF → putline "\nbye bye"
  end
```

Die Klasse modelliert ein sogenanntes *neutrales Zwei-Personen-Spiel* (engl. impartial two-player game). Neutral meint, dass die Zugmöglichkeiten in einer Position unabhängig davon sind, welcher Spieler zieht. (Schach ist nicht neutral, da ein Spieler die weißen und der andere Spieler die schwarzen Figuren kontrolliert.) Der Spielablauf lässt sich in eine *Schablone* pressen: Die Spieler sind abwechselnd am Zug; ein Spieler verliert, wenn keine Zugmöglichkeiten mehr existieren. Im Schablonen-Entwurfsmuster werden die variablen Bestandteile des Algorithmus (Ist die Endposition erreicht? Gebe die aktuelle Position am Bildschirm aus. Mache einen Zug. Präsentiere den Gewinner.) mit Hilfe abstrakter Methoden modelliert. Der Algorithmus selbst ist eine konkrete Methode, die aber von den abstrakten Bestandteilen abhängt.

Das wohl bekannteste neutrale Zwei-Personen-Spiel ist *Nim*: Von einem Haufen Streichhölzer werden in einem Spielzug 1–3 Hölzer entfernt. Die folgende Interaktion zeigt einen typischen Spielverlauf (Mensch versus Rechner).

```
Mini⟩ (Nim 10).Play ()
||||| |||||
number of matches: 1
||||| ||||
I take 1
||||| |||
number of matches: 3
|||||
I take 1
||||
number of matches: 2
||
I take 2
I win
```

Der menschliche Spieler eröffnet den Reigen und entfernt 1 Streichholz. Nach drei Runden gewinnt, wie zu erwarten (?), der Rechner („`I win`“).

Kommen wir zur Implementierung von Nim. Wir definieren zunächst zwei Hilfsfunktionen, um Streichholzhaufen übersichtlich in Fünferpäckchen ausgeben zu können.

```
let bars (i : Nat)    = String.replicate (int i) "|"
let (^^) x (i : Nat) = List.replicate (int i) x
```

Der Aufruf *bars* 5 wertet zu einem Fünferpäckchen aus: `"|||||"`; *bars* 5 `^^` 3 erzeugt eine Liste mit 3 Fünferpäckchen: `["|||||";"|||||";"|||||"]`. Zur Erinnerung: Um einen Infixoperator zu definieren, muss dieser in runde Klammern eingeschlossen werden: *let* (^^) *x i* = ...

Das Spiel selbst spezialisiert die Klasse *TwoPlayerGame*.

```
type Nim (n : Nat) =
  class
    inherit TwoPlayerGame ()
    let mutable matches = n
    let announce i = putline ("I take " ^ show i); i
    override self.Finished =
      matches = 0
    override self.Position () =
      putline (String.concat " " (((bars 5) ^^ (matches ÷ 5)) @ [bars (matches % 5)]))
    override self.Move (human : Bool) : Unit =
      let i =
        if human then
          checked-query ("number of matches",
                         both (is-nat, both (is-greater 0, is-less 4)))
        else
          announce (max 1 (matches % 4))
      matches ← matches ÷ i
    override self.Winner human =
      putline ((if human then "you" else "I") ^ " win")
  end
```

Die abstrakten Methoden, die Schritte des Algorithmus, werden jeweils mit Leben gefüllt. Die Methode *Move* verwendet die validierende Eingabefunktion *checked-query* aus Abschnitt 7.1.4, wenn der menschliche Spieler am Zug ist. Anderenfalls werden *max* 1 (*matches* % 4) Hölzer entfernt. Wie beurteilen Sie die Spiellogik des Rechners?

Kritik Von den unzähligen Sprachfeatures, die wir im Laufe der Vorlesung eingeführt haben, ist Vererbung das wohl kontroverseste. Wir werden im folgenden und letzten Abschnitt sehen, dass Vererbung in einem sehr angespannten Verhältnis zur Kapselung, einem der Grundpfeiler der Objektorientierung, steht. In vielen Fällen lässt sich Vererbung durch einfachere und bewährte Sprachfeatures ersetzen. Nehmen wir das Schablonen-Entwurfsmuster. Eine Schablone ist im Prinzip nichts anderes als eine parametrisierte Funktion — eine *Funktion höherer Ordnung*, da die Parameter, die Schritte des Algorithmus, selbst Funktionen sind. (Erinnern Sie sich an das Ratespiel aus Abschnitt 3.6.)

Bündeln wir die abstrakten Schritte in einer Schnittstelle,

```
type ITwoPlayerGame =
  interface
    abstract member Finished : Bool
    abstract member Position : Unit → Unit
    abstract member Move     : Bool → Unit
    abstract member Winner   : Bool → Unit
  end
```

dann lässt sich das Schablonen-Entwurfsmuster alternativ mit Hilfe einer Funktion umsetzen, die mit einem Spielobjekt parametrisiert ist.

```
let play (game : ITwoPlayerGame) =
  try
    let mutable turn = false
    while not game.Finished do
      game.Position ()
      turn ← not turn
      game.Move turn
    game.Winner turn
  with
  | EOF → putline "\nbye bye"
```

Die Unterschiede sind marginal: Die Nachrichten *Finished* etc. werden nicht länger an *self* geschickt, sondern an den Parameter *game*. Konkrete Spiele, Instanzen der Schnittstelle *ITwoPlayerGame*, können dann entweder mit Objektausdrücken oder mit Hilfe von Klassen definiert werden (siehe Aufgabe 8.8).

Lässt sich Vererbung auch im Fall des Beobachter-Entwurfsmusters umgehen? Die Verflechtung zwischen der abstrakten Klasse und dem sie konkretisierenden Objekt sind hier enger: Das Objekt ruft *Notify* auf; im Rumpf von *Notify* wird wieder auf das Objekt selbst zurückgegriffen.

Als ersten Schritt definieren wir zunächst eine Schnittstelle für „Zustände“.

```
type IState ⟨'state⟩ =
  interface
    abstract member State : 'state with get, set
  end
```

Eine „low cost“ Version des beobachtbaren Zählers lässt sich sehr einfach umsetzen. Wenn es nur einen Beobachter gibt, können wir diesen dem Objektkonstruktor unmittelbar mit auf den Weg geben.

```
let counter (observer : Nat → Unit) =
  let mutable n = 0
  { new IState ⟨Nat⟩ with
      member self.State
        with set k  = n ← k
                      observer n
        and  get () = n
  }
let steps = counter (fun n → if n % 1000 = 1 then putchar '*')
```

Der Objektkonstruktor *counter* ist nunmehr eine *Funktion höherer Ordnung*: Er nimmt eine Funktion als Argument und gibt ein Objekt, eine Sammlung von Methoden, als Ergebnis zurück.

Diese Umsetzung des Entwurfsmuster ist natürlich sehr speziell. Im Allgemeinen möchten wir Beobachter auch verwalten können: neue Beobachter hinzufügen, alte Beobachter entfernen. Denken Sie etwa an ein Zeitschriftenabonnement oder einen Newsletter, einen elektronischen Informationsbrief — das Entwurfsmuster wird auch unter dem Namen „publish-subscribe pattern“ geführt. In diesem Fall müssen wir die Parametrisierung weiter vorantreiben! Der Zähler wird jetzt mit der Benachrichtigungsfunktion parametrisiert (der vollständige Programmcode ist in Abbildung 8.11 aufgeführt).

```
let counter (notify : Unit → Unit) = ...
```

Die Verwaltung von Beobachtern wird von der Klasse *Observable* übernommen, die wiederum mit dem Subjekt der Beobachtungen parametrisiert ist (*counter* hat den passenden Typ $(Unit \to Unit) \to IState\,\langle Nat\rangle$).

```
type Observable ⟨'state⟩ (subject : (Unit → Unit) → IState ⟨'state⟩) as self =
  class
    let state = subject self.Notify        // delegatee
    ...
    member private self.Notify () = ...
    ...
  end
```

Der Objektkonstruktor ist sozusagen ein *Objektkonstruktor höherer Ordnung*: Er nimmt einen Objektkonstruktor als Argument und gibt ein Objekt als Ergebnis zurück. Im Rumpf der Klasse wird der Parameter, das Subjekt, mit der Benachrichtigungsfunktion versorgt (die jetzt privat ist); an das resultierende Zustandsobjekt *state* werden dann die Get- und Set-Nachrichten *delegiert* (siehe Abbildung 8.11). Einen beobachtbaren Zähler erhalten wir schließlich mit

```
let steps = Observable ⟨Nat⟩ counter
do steps.Attach (fun n → if n % 1000 = 1 then putchar '*')
```

Im Vergleich zur ursprünglichen Version des Beobachter-Entwurfsmusters sind jetzt die Abhängigkeiten zwischen den einzelnen Komponenten *manifest* (lat. offenbar, offenkundig). Das ist ein Segen und ein Fluch. Ein Fluch, weil die Versorgung mit Parametern von Hand vorgenommen werden muss. Diese Klempnerarbeiten (engl. plumbing) werden in der ursprünglichen Version hinter den Kulissen vom Mechanismus der Vererbung erledigt. Ein Segen, weil die Versorgung mit Parametern von Hand vorgenommen werden kann. Die Komponenten (*counter* und *Observable*) sind entkoppelt und lassen sich getrennt programmieren und testen und können auch einfach ausgetauscht werden. Wie so oft haben Automatismen sowohl Vor- als auch Nachteile.

8.5.4. Delegation versus Vererbung

module Objects. StepCounter

Um Objekte kompostional zu definieren, haben wir zwei grundlegende Techniken kennengelernt: *Delegation* in Abschnitt 8.1 und *Unterklassen* (Implementierungsvererbung)

```
type IState⟨'state⟩ =
  interface
    abstract member State : 'state with get, set
  end
let counter (notify : Unit → Unit) =
  let mutable n = 0
  {
    new IState⟨Nat⟩ with
      member self.State
        with set k = n ← k
                    notify ()
        and  get () = n
  }
type Observable⟨'state⟩ (subject : (Unit → Unit) → IState⟨'state⟩) as self =
  class
    let state = subject self.Notify        // delegatee
    let mutable observers = []
    member self.Attach (observer : Observer⟨'state⟩) =
      observers ← observer :: observers
      observer state.State                 // notify observer
    member private self.Notify () =
      for observer in observers do
        observer state.State
    member self.State                      // delegate calls
      with get () = state.State
      and  set v = state.State ← v
    interface IState⟨'state⟩ with          // implement interface
      member self.State
        with get () = self.State
        and  set v = self.State ← v
  end
let steps = Observable⟨Nat⟩ counter
do steps.Attach (fun n → if n % 1000 = 1 then putchar '*')
let rec fibonacci (n : Nat) : Nat =
  steps.State ← steps.State + 1
  if n ⩽ 1 then n
           else fibonacci (n - 1) + fibonacci (n - 2)
```

Abbildung 8.11.: Beoachter-Entwurfsmuster mit Delegation.

in diesem Abschnitt, Abschnitt 8.5. Wir haben schon anklingen lassen, dass im Allgemeinen Delegation die bessere Wahl ist. Im Folgenden wollen wir den Gründen für diese Präferenz nachgehen.

Zunächst einmal ist es wichtig, den Mechanismus des Nachrichtensendens im Zusammenspiel von Klassen und Unterklassen genau zu verstehen. Zu diesem Zweck haben wir eine Variante des Schrittzählers mit Kontrollausgaben versehen (ignorieren Sie zunächst den Code auf der rechten Seite).

```
type StepCounter () =
  class
    let mutable n = 0
    abstract Inc : Unit → Unit
    default self.Inc () =
      putline "Inc: base class"
      n ← n + 1
    abstract Step : Int → Unit
    default self.Step k =
      putline "Step: base class"
      for i in 1 .. k do
        self.Inc ()
  end
```

```
type StepCounter () =
  class
    let mutable n = 0
    abstract Inc : Unit → Unit
    default self.Inc () =
      putline "Inc: base class"
      n ← n + 1
    abstract Step : Int → Unit
    default self.Step k =
      putline "Step: base class"
      n ← n + k

  end
```

Die Nachrichten *Inc* und *Step* sind *virtuell* — sie sind abstrakt, aber mit einer Standarddefinition versehen. Insbesondere lassen wir zu, dass eine Unterklasse die Methoden redefiniert, so geschehen im folgenden Programmstück (ignorieren Sie wiederum den Code auf der rechten Seite).

```
type ParityCounter () =
  class
    inherit StepCounter ()
    let mutable even = true
    override self.Inc () =
      putline "Inc: subclass"
      base.Inc ()
      even ← not even
    override self.Step k =
      putline "Step: subclass"
      base.Step k
      even ← even = (k % 2 = 0)
    member self.Parity = even
  end
```

```
type ParityCounter () =
  class
    inherit StepCounter ()
    let mutable even = true
    override self.Inc () =
      putline "Inc: subclass"
      base.Inc ()
      even ← not even

    member self.Parity = even
  end
```

Der Paritätszähler hält nach, ob die Gesamtzahl der Schritte gerade oder ungerade ist.[7] Zu diesem Zweck wird die Veränderliche *even* verwendet; die Methoden *Inc* und *Step* werden entsprechend redefiniert, um Änderungen nachhalten zu können. Dazu wird ausgenutzt, dass die Parität der Summe $i+j$ sich aus der Parität der Summanden ableiten lässt: gerade + gerade = gerade, gerade + ungerade = ungerade, ungerade + gerade = ungerade und ungerade + ungerade = gerade, oder als Formel,

$$((i+j)\,\%\,2 = 0) \quad = \quad (i\,\%\,2 = 0) = (j\,\%\,2 = 0)$$

Bevor die Parität nachgehalten wird, werden jeweils die Methoden der Basisklasse aufgerufen, die die eigentliche Arbeit verrichten.

Senden wir einem Paritätszähler die Nachricht *Step* 3, ergibt sich eine interessante Nachrichtenkaskade.

```
Mini⟩ let counter = ParityCounter ()
Mini⟩ counter.Step 3
Step: subclass
Step: baseclass
Inc: subclass
Inc: baseclass
Inc: subclass
Inc: baseclass
Inc: subclass
Inc: baseclass
Mini⟩ counter.Parity
true
```

Die Nachricht *Step* 3 wird an einen Paritätszähler gesendet; die *Step*-Methode der Unterklasse ruft dann die gleichnamige Methode der Basisklasse auf (***base***.*Step* k); in der Basisklasse sendet das Objekt sich selbst eine Nachricht: *self*.*Inc* (). Da das Objekt ein Paritätszähler ist (!), ist die redefinierte *Inc*-Methode der Unterklasse gemeint, die ihrerseits die gleichnamige Methode der Basisklasse aufruft (***base***.*Inc* ()). Mit anderen Worten, virtuelle Methoden werden stets zur Laufzeit in der zu dem Objekt gehörigen Methodentabelle nachgeschlagen (dynamic dispatch), auch und inbesondere für Selbstnachrichten.

Für unsere Anwendung hat die Nachrichtenkaskade einen unerwünschten Effekt: Jede Erhöhung wird doppelt gezählt, so dass 3 als gerade klassifiziert wird (*counter*.*Parity* ergibt *true*). Autsch! Was ist zu tun? Wir dürfen *Step* nicht redefinieren, so wie im Code oben auf der rechten Seite. Problem gelöst? Nicht ganz … Einige Zeit später im Zuge einer Coderevision optimiert Harry Hacker die Klasse *StepCounter*: Er ersetzt die

[7] Das Beispiel ist bewusst einfach gehalten, steht aber stellvertretend für Anwendungen, in denen aufwändig zu berechnende Eigenschaften mittels eines Caches in konstanter Zeit bereitgestellt werden. Zum Beispiel: Um die Größe einer Liste oder eines Stacks zu bestimmen, benötigt man lineare Laufzeit. Merkt man sich die Länge und trägt bei Modifikationen des Stacks, *Push* oder *Pop*, jeweils die Längenänderung nach, +1 oder −1, steht die Eigenschaft in konstanter Zeit zur Verfügung.

Schleife ***for*** *i* ***in*** 1 . . *k* ***do*** *self*.*Inc* () im Rumpf der Methode *Step* durch die Zuweisung $n \leftarrow n + k$, siehe Code ganz oben auf der rechten Seite. Wiederholen wir jetzt die obige Interaktion, ergibt sich das folgende Bild.

```
Mini⟩ let counter = ParityCounter ()
Mini⟩ counter.Step 3
Step: baseclass
Mini⟩ counter.Parity
true
```

Da *Step* nicht redefiniert wurde, wird die Nachricht direkt von der Basisklasse behandelt mit dem Ergebnis, dass keine Aktualisierung der Parität erfolgt (*counter*.*Parity* ergibt den ursprünglichen Wert, nämlich *true*).

Wie man's macht, macht man's falsch.
Und macht man's falsch, ist's auch nicht richtig.
— Klaus Klages (1938)

Die Änderung der Basisklasse *StepCounter* zieht also eine Änderung der Unterklasse *ParityCounter* nach sich. Im Zuge der Coderevision hätte die Unterklasse ebenfalls revidiert werden müssen: Die Methode *Step* muss redefiniert werden, so wie im Code oben links. (Das Zusammenspiel der Klassen funktioniert nur „über Kreuz".) Das ist kein großes Problem, wenn die Unterklasse im gleichen Modul definiert wird und von der gleichen Programmierer*in gepflegt wird. Nun kann eine Klasse aber viele Unterklassen besitzen und diese können über verschiedene Module und Anwendungen verstreut sein. Problematisch ist, dass die Änderung der Basisklasse nicht lokal ist; sie zieht nicht-lokale Folgeänderungen nach sich (das Programm ist fragil). Umgekehrt betrachtet, lässt sich die Unterklasse *ParityCounter* nicht ohne intime Kenntnis der Oberklasse *StepCounter* korrekt implementieren. Die Kapselung, die eine Klasse vornimmt, wird durch virtuelle Methoden und Vererbung aufgeweicht: *inheritance breaks abstraction*.

Komponieren wir die beiden Zählertypen mit Hilfe von Delegation, tritt, wie wir sehen werden, das Problem nicht auf. Delegation kann mit Klassen realisiert werden (siehe Abbildung 8.11) oder mit Schnittstellen und Objektausdrücken. Wir ziehen die zweite Option und definieren zunächst eine Schnittstelle für Schrittzähler.

```
type IStepCounter =
  interface
    abstract member Inc  : Unit → Unit
    abstract member Step : Int  → Unit
  end
```

Ähnlich wie in der Variante mit Klassen und Unterklassen realisieren wir den Objektkonstruktor *step-counter* auf zwei Weisen: Links ruft *Step* die Methode *Inc* auf, die Zustandsänderung erfolgt also indirekt; rechts wird der Zustand direkt mit einer Zuweisung aktualisiert.

```
let step-counter () =
  let mutable n = 0
  {new IStepCounter with
      member self.Inc () =
        putline "Inc: delegatee"
        n ← n + 1
      member self.Step k =
        putline "Step: delegatee"
        for i in 1 .. k do
          self.Inc ()
  }
```

```
let step-counter () =
  let mutable n = 0
  {new IStepCounter with
      member self.Inc () =
        putline "Inc: delegatee"
        n ← n + 1
      member self.Step k =
        putline "Step: delegatee"
        n ← n + k

  }
```

Die Implementierung des Paritätszählers erfolgt *unabhängig* davon, wie der Schrittzähler implementiert ist. Wir müssen nur wissen, *was* der Schrittzähler macht, aber nicht *wie* das Verhalten detailliert umgesetzt wird. Wie üblich definieren wir zunächst einen Untertyp von *IStepCounter*.

```
type IParityCounter =
  interface
    inherit IStepCounter
    abstract member Parity : Bool
  end
```

Der Paritätszähler delegiert die Hauptarbeit an einen Schrittzähler; zusätzlich werden wie vorher die Änderungen der Priorität nachgehalten.[8]

```
let parity-counter () =
  let basic          = step-counter ()      // delegatee
  let mutable even = true
  {new IParityCounter with
      member self.Inc () =
        putline "Inc: delegator"
        basic.Inc ()
        even ← not even
      member self.Step k =
        putline "Step: delegator"
        basic.Step k
        even ← even = (k % 2 = 0)
      member self.Parity = even
  }
```

[8] Im Englischen ist der „delegator“ die- oder derjenige, der Arbeit delegiert, und „delegatee“ die- oder derjenige, an den Arbeit delegiert wird. Die Endungen „-or“ und „-ee“ finden sich auch in anderen Wortpaaren wieder: Ein „advisor“ erteilt Ratschläge, die ein „advisee“ entgegennimmt.

Technisch gesehen ergeben sich im Vergleich zur Realisierung mit Klassen und Unterklassen zwei Änderungen: Die ***inherit***-Klausel der Unterklasse ist einer ***let***-Bindung gewichen, die einen Namen für den zugrundeliegenden Schrittzähler vergibt. (Warum ist *basis* kein ***mutable***?) Entsprechend sind die Aufrufe der Basisklasse, ***base***.*Inc* (), durch Aufrufe des „Basisobjekts“ , *basic*.*Inc* (), ersetzt worden.

Je nachdem, welche Implementierung des Schrittzählers zugrundeliegt, ergeben sich unterschiedliche Kontrollausgaben. Die Parität wird aber unabhängig von der Umsetzung des Schrittzählers korrekt berechnet.

```
Mini⟩ let counter = parity-counter ()
Mini⟩ counter.Step 3
Step: delegator
Step: delegatee
Inc: delegatee
Inc: delegatee
Inc: delegatee
Mini⟩ counter.Parity
false
```

```
Mini⟩ let counter = parity-counter ()
Mini⟩ counter.Step 3
Step: delegator
Step: delegatee
Mini⟩ counter.Parity
false
```

Der Paritätszähler delegiert die Arbeit an den Schrittzähler, dann arbeitet dieser: links durch wiederholte Aufrufe von *Inc*; rechts direkt mit einer Zuweisung.

Wie erklären sich die Unterschiede zwischen den beiden Ansätzen? In der ursprünglichen Realisierung existiert genau *ein* Objekt, der Paritätszähler, der aber Verhalten von der Basisklasse erbt. Mit dem Akt der Vererbung werden die Methoden kompliziert miteinander verflochten, via Nachrichten an die Basisklasse und via Selbstnachrichten. Im Gegensatz dazu existieren hier *zwei* Objekte, der Paritätszähler und der Schrittzähler, die in nachvollziehbarer Weise miteinander interagieren: Einer delegiert Arbeit an den anderen.

Damit soll das Konzept von Unterklassen nicht prinzipiell verteufelt werden, aber als Fazit können wir festhalten, dass virtuelle Methoden und Unterklassen mit Bedacht eingesetzt werden sollten.

Übungen.

1. Implementieren Sie die Schnittstelle *IStack* aus Abschnitt 8.1.4 mit Hilfe von Arrays. Verwenden Sie einen „Wasserstandsanzeiger“, der auf die erste freie Position verweist; *Push* erhöht den Wasserstand; *Pop* verringert ihn. Ein array-basierter Stack hat eine Kapazitätsgrenze, die durch die Größe des Arrays gegeben ist. Wird diese überschritten, wird eine Ausnahme geworfen (der bekannte und gefürchtete „Stack Overflow“). *Tipp:* Das Interface ist generisch; wenn Sie Probleme haben, ein leeres Array zu erzeugen, verwenden Sie optionale Elemente: *Array* ⟨*Option* ⟨*'elem*⟩⟩.

2. Re-implementieren Sie Aufgaben 4.16 und 4.17 mit Hilfe von Objekten. Lassen Sie sich von der in Abschnitt 8.1.4 vorgestellten Darstellung (*IExpr*) inspirieren.

3. Definieren Sie eine Schnittstelle für ephemerale Wörterbücher, siehe Abschnitt 8.3.2, und ändern Sie die Definition der Klasse *SearchTree*, so dass die Schnittstelle implementiert wird.

4. Definieren Sie einen Aufzähler für natürliche Zahlen, präskriptiv mit Objektausdrücken, das heißt, ohne Sequenzbeschreibungen zu verwenden.

5. Wir haben in Abschnitt 8.4.3 bemerkt, dass die Laufzeit von *Seq.append* nicht von der Länge der zu konkatenierenden Sequenzen abhängt, sondern von der Schachtelungstiefe: Je tiefer *Seq.append* Aufrufe geschachtelt werden, desto länger werden die Nachrichtenketten. Explorieren wir die Konsequenzen:

(a) Testen Sie die naive Definition von *inorder* mit links- und rechtsentarteten Binärbäumen. Wie lassen sich die Ergebnisse erklären?

(b) Übertragen Sie die Definition von *inorder-append* aus Abschnitt 5.3.4 auf aufzählbare Objekte von Typ $seq\,\langle'a\rangle$. Wiederholen Sie die Laufzeittests.

(c) Implementieren Sie *inorder* ohne syntaktischen Zucker präskriptiv mit Objektausdrücken. *Hinweis:* Orientieren Sie sich an der Definition von *from-list* aus Abschnitt 8.4; merken Sie sich in einer Liste die noch zu traversierenden Bäume. Führen Sie die Laufzeittests ein drittes Mal durch.

6. Aufzählbare Objekte vom Typ $IEnumerator\,\langle'a\rangle$ werden auch als *externere Iteratoren* bezeichnet. Extern, weil aus Sicht des Objekts die Benutzer*in die Kontrolle ausübt. Alternativ kann das Objekt selbst die Iteration durchführen.

```
type Iterator ⟨'a⟩ =
  interface
    member Foreach : ('a → Unit) → Unit
  end
```

Eine Instanz des Typs $Iterator\,\langle'a\rangle$ ist ein sogenannter *interner Iterator*: Die übergebene Funktion wird über alle Elemente der repräsentierten Sequenz iteriert. Definieren sie elementare Iteratoren: für die leere Sequenz, für eine einelementige Sequenz, für Intervalle, Listen und Arrays. Zeigen Sie, wie Iteratoren hintereinander geschaltet (*append*) und geschachtelt (*collect*) werden können. In Abschnitt 8.4 haben wir gesehen, dass ein interner Iterator (eine ***for***-Schleife) mit Hilfe eines externen Iterators ($IEnumerator\,\langle'a\rangle$) definiert werden kann. Gilt auch die Umkehrung?

7. In Abschnitt 8.5.1 haben wir Studierendenkonten mittels Vererbung von Standardkonten abgeleitet. Reimplementieren Sie die Klasse *TrustMeStudent* mit Hilfe von *Delegation*. Bietet Delegation in diesem Fall konkrete Vorteile?

8. Komplettieren Sie die alternative Implementierung des Schablonen-Entwurfsmusters, indem Sie die Schnittstelle *ITwoPlayerGame* mit Leben füllen: Definieren Sie konkrete Spiele (zum Beispiel Nim oder Hackenbush) mit Objektausdrücken oder mit Hilfe von Klassen.

9. Die Klasse *Subject* erlaubt es, neue Beobachter hinzufügen (etwa wenn ein Zeitschriftenabonnement abgeschlossen wird), aber nicht einen Beobachter zu entfernen (etwa wenn das Abonnement ausläuft oder gekündigt wird). Tatsächlich kann aus prinzipiellen Gründen keine *Remove* Methode definiert werden, da die Gleichheit von Funktionen nicht formal entscheidbar ist. (Zwei Funktionen f und g sind gleich, wenn sie für gleiche Argumente gleiche Ergebnisse liefern: $f\ x = g\ x$, für alle x. Wäre die Gleichheit von Funktionen entscheidbar, dann könnte man Übungsblätter korrigieren, indem man die Abgaben mit den Musterlösungen vergleicht, zum Beispiel, *student-sort* = *trusted-sort*. Leider gelingt das nicht in voller Schönheit — wäre auch zu schön gewesen.) Was ist zu tun? Eine Möglichkeit besteht darin, die Beobachterfunktion als Objekt einzukleiden.

```
type Observer⟨'state⟩ =
  interface
    abstract Update : 'state → Unit
  end
```

Warum löst dieser Kniff das Problem der unentscheidbaren Gleichheit? Bei Objekten wird nicht die *Wertegleichheit*, sondern die *Verweisgleichheit* zugrundegelegt, siehe auch Aufgabe 7.4. Genau wie Speicherzellen haben Objekte eine eindeutige Identität („*all objects are created unequal*“). Führen Sie die notwendigen Änderungen durch und fügen Sie eine Methode *Remove* hinzu, die es ermöglicht, einen Beobachter zu entfernen.

A. Anhang

A.1. Kompendium Mathematik

Dieser Anhang fasst die wichtigsten mathematischen Begriffe und Notationen zusammen, die im Skript als bekannt vorausgesetzt werden. Die Übersicht ist als „Nachschlagewerk“ konzipiert. — Vielleicht ist der Anhang hilfreich, verschüttetes mathematisches Grundwissen aufzufrischen; auf Grund der Kürze der Darstellung wird der Text allerdings weniger geeignet sein, in die Themengebiete einzuführen. Für eine Einführung in die Mathematik sei auf die einschlägigen Lehrbücher verwiesen und auf den Online Mathematik Brückenkurs (insbesondere das Zusatzmodul „Überblick: Logik und Mengenlehre“):

https://www.mathematik.uni-kl.de/studium/brueckenkurse/omb/

Mit einem Stern ⋆ markierte Abschnitte enthalten weiterführenden Stoff und richten sich in erster Linie an mathematisch interessierte Leser*innen.

A.1.1. Logik und Algebra

Aussagenlogik Eine *Aussage* ist ein Satz, für den es sinnvoll ist zu fragen, ob er falsch oder wahr ist. Je nach Teilgebiet der Mathematik oder Informatik werden die *Wahrheitswerte* unterschiedlich notiert:

- „falsch“ unter anderem als 0, $\bot$, f, F oder *false*;
- „wahr“ als 1, $\top$, w, t, T oder *true*.

Aussagen werden mit Hilfe von „nicht“, „und“, „oder“, „wenn, dann“, „genau dann, wenn“ usw. zu komplexen Aussagen verknüpft. Auch die Bezeichnungen für diese *aussagenlogischen Verknüpfungen* sind nicht einheitlich und variieren von Lehrbuch zu Lehrbuch:

- die *Negation* von a wird unter anderem durch $\neg a$, $\bar{a}$, a' oder *not* a bezeichnet;
- die *Konjunktion* von a und b durch $a \wedge b$, $a \cdot b$, ab oder a `&&` b;
- die *Disjunktion*[1] von a und b durch $a \vee b$, $a + b$ oder a `||` b;
- die *Implikation* (Subjunktion) von a und b durch $a \Rightarrow b$, $a \to b$ oder $a \leqslant b$;

[1] Der Begriff „Disjunktion“ ist etwas unglücklich gewählt, da er die falsche Assoziation weckt, dass sich die beiden Teilaussagen ausschließen müssen („entweder … oder“). Das ist nicht der Fall.

- die *Äquivalenz* (Bijunktion) von a und b durch $a \Leftrightarrow b$, $a \leftrightarrow b$, $a \equiv b$ oder $a = b$.

Die Bedeutung der Verknüpfungen wird mit Hilfe von *Wahrheitstabellen* festgelegt.

a	$\neg a$
0	1
1	0

a	b	$a \wedge b$	$a \vee b$	$a \Rightarrow b$	$a \Leftrightarrow b$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Zwischen den Verknüpfungen gelten die folgenden Beziehungen, die sich mit Hilfe der Wahrheitstabellen nachprüfen lassen.

$$\begin{aligned}
\neg a &= a \Rightarrow 0 \\
a \wedge b &= \neg(a \Rightarrow \neg b) \\
a \vee b &= \neg a \Rightarrow b \\
a \Rightarrow b &= \neg a \vee b \\
a \Leftrightarrow b &= (a \Rightarrow b) \wedge (b \Rightarrow a)
\end{aligned}$$

Viele mathematische Sätze, Lemmata und Theoreme, haben die Form einer Implikation $a \Longrightarrow b$. Die Aussage a heißt *Voraussetzung* oder *Annahme* und b *Behauptung* oder *Folgerung*. Hat die Annahme die Form einer Konjunktion $a_1 \wedge \cdots \wedge a_n$, so spricht man auch von Annahmen (Plural). Um $a \Longrightarrow b$ zu beweisen, müssen wir unter der Annahme, dass a wahr ist, die Behauptung b zeigen.

Boolesche Algebra Die beiden Wahrheitswerte bilden zusammen mit den Verknüpfungen „nicht“, „und“, und „oder“ eine Boolesche Algebra, eine mathematische Struktur, die die Gesetze in Abbildung A.1 erfüllt. Es lohnt sich, die Gesetze und deren Namen einzuprägen. Drei Gesetze verdienen besondere Beachtung:

Das *Assoziativgesetz* $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ erlaubt es, mehrere Aussagen mit „und“ zu verknüpfen, ohne Klammern setzen zu müssen: $a \wedge b \wedge c$ — eine wichtige Schreiberleichterung.

Das *Kommutativgesetz* $a \wedge b = b \wedge a$ besagt, dass weiterhin die Reihenfolge der Einzelaussagen keine Rolle spielt. Die Konjunktion von drei Aussagen kann zum Beispiel auf sechs Weisen formuliert werden (warum sechs?):

$$a \wedge b \wedge c = a \wedge c \wedge b = b \wedge a \wedge c = b \wedge c \wedge a = c \wedge a \wedge b = c \wedge b \wedge a$$

Das *Distributivgesetz* $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ regelt schließlich das Zusammenspiel von Konjunktion und Disjunktion. Von links nach rechts gelesen wird die Formel „ausmultipliziert“; von rechts nach links gelesen wird ein gemeinsamer „Faktor“ herausgezogen. (Auch die ganzen Zahlen mit Addition und Multiplikation erfüllen das Distributivgesetz: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.)

Axiome der Booleschen Algebra.

(Assoziativgesetze)	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	$(a \vee b) \vee c = a \vee (b \vee c)$
(Kommutativgesetze)	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
(Idempotenzgesetze)	$a \wedge a = a$	$a \vee a = a$
(Absorptionsgesetze)	$a \vee (a \wedge b) = a$	$a \wedge (a \vee b) = a$
(Distributivgesetze)	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
(Neutralitätsgesetze)	$a \wedge 1 = a$	$a \vee 0 = a$
(Extremalgesetze)	$a \wedge 0 = 0$	$a \vee 1 = 1$
(Komplementärgesetze)	$a \wedge \neg a = 0$	$a \vee \neg a = 1$

Die Gesetze sind nicht voneinander unabhängig. So folgen etwa (Asso), (Idem), (Abso) und (Extr) aus den vier Huntington-Axiomen: (Komm), (Dist), (Neut) und (Komp).

Folgerungen aus den Axiomen.

(Dualitätsgesetze)	$\neg 0 = 1$	$\neg 1 = 0$
(Involution)	$\neg(\neg a) = a$	$\neg(\neg a) = a$
(De Morgansche Gesetze)	$\neg(a \wedge b) = \neg a \vee \neg b$	$\neg(a \vee b) = \neg a \wedge \neg b$

Abbildung A.1.: Axiome der Booleschen Algebra und Folgerungen aus den Axiomen.

Die Anordnung der Gesetze in zwei Spalten veranschaulicht das *Dualitätsprinzip*: Ersetze ich in einer Formel systematisch 0 durch 1, 1 durch 0, $\wedge$ durch $\vee$ und $\vee$ durch $\wedge$, erhalte ich die *duale* Formel aus der anderen Spalte — die Involution $\neg(\neg a) = a$ ist zu sich selbst dual. Allgemein gilt, dass ein Gesetz genau dann wahr ist, wenn das duale Gesetz wahr ist („Buy one get one free!“). Die obigen Ausführungen zur Assoziativität, Kommutativität und Distributivität treffen also in gleicher Weise auf die Disjunktion zu. (Das Dualitätsprinzip gilt *nicht* für die ganzen Zahlen mit Addition und Multiplikation: Zum Beispiel distributiert die Multiplikation über die Addition, $a \cdot (b+c) = (a \cdot b) + (a \cdot c)$, aber *nicht* umgekehrt, $a + (b \cdot c) = (a + b) \cdot (a + c)$.)

Prädikatenlogik Mit den bisherigen Sprachmitteln lassen sich nur endlich viele Aussagen miteinander verknüpfen. Oft trifft man aber Aussagen über alle Elemente einer Menge. Die starke Goldbachsche Vermutung besagt zum Beispiel, dass „jede gerade Zahl, die größer als 2 ist, sich als Summe zweier Primzahlen schreiben lässt.“ Wenn wir die Menge der natürlichen Zahlen mit $\mathbb{N}$ und die Menge der Primzahlen mit $\mathbb{P}$ bezeichnen, dann lässt sich die Aussage mit Hilfe sogenannter *Quantoren* wie folgt formalisieren:

$$\forall\, n \in \mathbb{N} \;.\; \exists\, p_1 \in \mathbb{P} \;.\; \exists\, p_2 \in \mathbb{P} \;.\; n = p_1 + p_2$$

Hier bedeutet $n \in \mathbb{N}$, dass n ein Element der Menge $\mathbb{N}$ ist, also eine natürliche Zahl. Das gespiegelte „A“ und das gespiegelte „E“ sind Quantoren:

- *Existenzquantor*: es existiert ein $x \in X$, so dass die Aussage $P(x)$ gilt;

  $$\exists\, x \in X \;.\; P(x)$$

- *Allquantor*: für alle $x \in X$ gilt die Aussage $P(x)$.

  $$\forall\, x \in X \;.\; P(x)$$

Dabei ist $P(x)$ eine sogenannte *Aussagenfunktion*, eine Aussage, die eine oder mehrere Variablen enthält und zu einer Aussage wird, wenn die Variablen durch Elemente der Menge X ersetzt werden.

Der Existenzquantor verallgemeinert die Disjunktion und wird aus diesem Grund manchmal auch mit $\bigvee x \in X$ notiert; enstprechend verallgemeinert der Allquantor die Konjunktion und wird mit $\bigwedge x \in X$ notiert. Besteht die Menge X nur aus endlich vielen Elementen, zum Beispiel $x_1, \ldots, x_n$, dann gilt somit:

$$\begin{aligned} \exists\, x \in X \;.\; P(x) &= P(x_1) \vee \cdots \vee P(x_n) \\ \forall\, x \in X \;.\; P(x) &= P(x_1) \wedge \cdots \wedge P(x_n) \end{aligned}$$

Ist die Menge leer, dann ist die Existenzaussage falsch und die Allaussage wahr!

Für das Arbeiten mit Quantoren sind die folgenden Gesetze nützlich — die Auflistung erhebt aber keinen Anspruch auf Vollständigkeit. Sei $a \in X$, dann

$$\begin{aligned}
P(a) &\implies \exists\, x \in X \,.\, P(x) \\
\forall\, x \in X \,.\, P(x) &\implies P(a) \\
\neg(\exists\, x \in X \,.\, P(x)) &\iff \forall\, x \in X \,.\, \neg P(x) \\
\neg(\forall\, x \in X \,.\, P(x)) &\iff \exists\, x \in X \,.\, \neg P(x) \\
p \wedge (\exists\, x \in X \,.\, Q(x)) &\iff \exists\, x \in X \,.\, p \wedge Q(x) \\
p \vee (\forall\, x \in X \,.\, Q(x)) &\iff \forall\, x \in X \,.\, p \vee Q(x) \\
p \Rightarrow (\forall\, x \in X \,.\, Q(x)) &\iff \forall\, x \in X \,.\, p \Rightarrow Q(x)
\end{aligned}$$

A.1.2. Mengenlehre

Eine Menge ist eine Zusammenfassung von Elementen zu einer Einheit. Dabei spielt weder die Reihenfolge der Elemente noch deren Vielfachheit eine Rolle. Wir schreiben $x \in X$ um auszudrücken, dass x ein Element der Menge X ist, und $x \notin X$, wenn das nicht der Fall ist. Die leere Menge wird mit $\emptyset$ bezeichnet. Da die leere Menge keine Elemente enthält, ist $x \in \emptyset$ stets falsch. Mit $|X|$ bezeichnen wir die Kardinalität der Menge X, also die Gesamtzahl ihrer Elemente, falls diese Anzahl endlich ist.

Die folgenden Bezeichnungen sind gebräuchlich:

- $\mathbb{B}$ ist die Menge der Wahrheitswerte (falsch und wahr);
- $\mathbb{N}$ ist die Menge der natürlichen Zahlen $(0, 1, 2, \ldots)$;
- $\mathbb{Z}$ ist die Menge der ganzen Zahlen $(\ldots, -2, -1, 0, 1, 2, \ldots)$;
- $\mathbb{Q}$ ist die Menge der rationalen Zahlen (Bruchzahlen);
- $\mathbb{R}$ ist die Menge der reellen Zahlen.

Eine endliche Menge kann durch Aufzählung ihrer Elemente definiert werden:

$$\{2, 3, 5, 7\}$$

ist zum Beispiel die Menge der ersten vier Primzahlen. Da die Reihenfolge der Elemente irrelevant ist, kann die Menge alternativ durch $\{7, 5, 3, 2\}$ oder durch $\{3, 7, 5, 2\}$ angegeben werden. Auch $\{2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 7, 7, 7, 7, 7, 7, 7\}$ bezeichnet die gleiche Menge, da irrelevant ist, wie häufig ein Element aufgeführt wird.

Eine endliche oder unendliche Menge kann durch eine sogenannte „Mengenbeschreibung“ oder kurz „Beschreibung“ (engl. set comprehension) definiert werden:

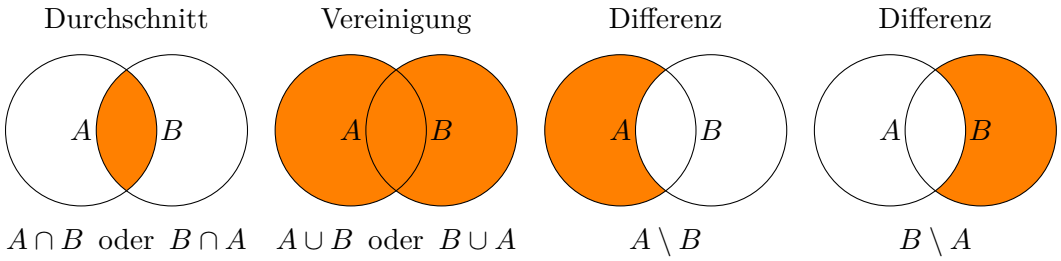
$$\{\, x^2 \mid x \in \mathbb{N} \,\}$$

ist zum Beispiel die Menge der Quadratzahlen. *Lies:* „die Menge aller x^2, so dass x eine natürliche Zahl ist“.

Mengenoperationen Aus zwei gegebenen Mengen lassen sich mit Hilfe der folgenden *Mengenoperationen* neue Mengen bilden.[2]

(Durchschnitt)	$A \cap B := \{\, x \mid x \in A \wedge x \in B \,\}$
(Vereinigung)	$A \cup B := \{\, x \mid x \in A \vee x \in B \,\}$
(Differenz)	$A \setminus B := \{\, x \mid x \in A \wedge x \notin B \,\}$

Die Operationen kann man sich durch sogenannte Venn-Diagramme veranschaulichen:

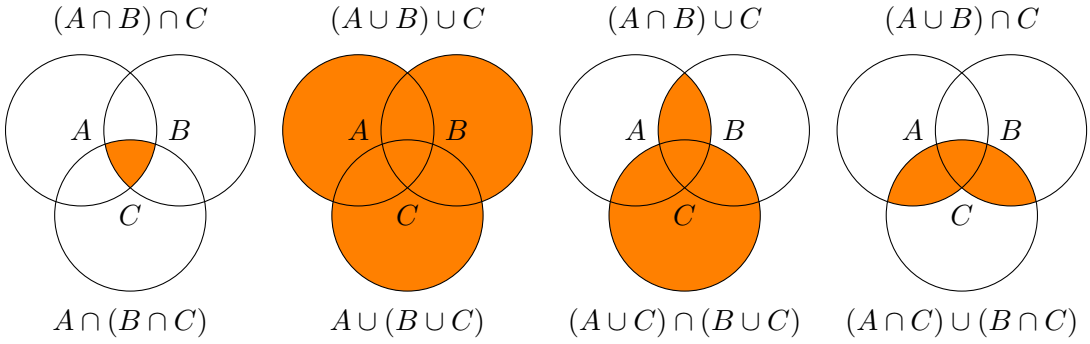


Nach Definition des Durchschnitts gilt $x \in A \cap B \iff x \in A \wedge x \in B$. Somit ziehen die Eigenschaften der Konjunktion entsprechende Eigenschaften des Mengendurchschnitts nach sich: Der Durchschnitt ist assoziativ, kommutativ und idempotent. Entsprechende Überlegungen gelten für die Mengenvereinigung: Auch sie ist assoziativ, kommutativ und idempotent. Die Menge der ersten vier Primzahlen kann zum Beispiel alternativ als Vereinigung einelementiger Mengen geschrieben werden:

$$\{2\} \cup \{3\} \cup \{5\} \cup \{7\} = \{3\} \cup \{7\} \cup \{5\} \cup \{2\} = \{2\} \cup \{3\} \cup \{2\} \cup \{7\} \cup \{5\} \cup \{3\}$$

Da die Mengenvereinigung assoziativ ist, müssen wir geschachtelte Vereinigungen nicht klammern; da sie kommutativ ist, spielt die Reihenfolge der Grundmengen keine Rolle; da sie idempotent ist, spielt keine Rolle, wie oft eine Grundmenge aufgeführt wird. Im Unterschied zum Durchschnitt und zur Vereinigung ist die Mengendifferenz weder assoziativ, noch kommutativ (siehe obige Venn-Diagramme), noch idempotent.

Auch das Zusammenspiel von Durchschnitt und Vereinigung lässt sich mit Hilfe von Venn-Diagrammen illustrieren.



[2] Mit $u := k$ oder $k =: u$ ist gemeint, dass u durch k definiert wird. Der Doppelpunkt steht jeweils auf der Seite der definierten Größe.

Die ersten beiden Diagramme veranschaulichen die Assoziativität von Durchschnitt und Vereinigung. Die letzten beiden Diagramme „zeigen“, dass die Vereinigung über den Durchschnitt distributiert und umgekehrt der Durchschnitt über die Vereinigung.

Konstruktionen auf Mengen Die Menge A heißt *Teilmenge* oder *Untermenge* von B genau dann, wenn jedes Element aus A in B enthalten ist.

(Teilmenge/Untermenge)	$A \subseteq B$	$:\Longleftrightarrow$	$\forall x \;.\; x \in A \Longrightarrow x \in B$
(Obermenge)	$A \supseteq B$	$:\Longleftrightarrow$	$\forall x \;.\; x \in A \Longleftarrow x \in B$
(Mengengleichheit)	$A = B$	$:\Longleftrightarrow$	$\forall x \;.\; x \in A \Longleftrightarrow x \in B$

Die Teilmengenbeziehung ist reflexiv, transitiv und antisymmetrisch.

(Reflexivität)	$A \subseteq A$			
(Transitivität)	$A \subseteq B$	$\wedge$ $B \subseteq C$	$\Longrightarrow$	$A \subseteq C$
(Antisymmetrie)	$A \subseteq B$	$\wedge$ $B \subseteq A$	$\Longrightarrow$	$A = B$

Sei U eine beliebige Grundmenge, ein sogenanntes *Universum*. Die Gesamtheit aller Teilmengen von U heißt *Potenzmenge* $\mathcal{P}(U)$ von U.

(Potenzmenge) $$\mathcal{P}(U) := \{\, X \mid X \subseteq U \,\}$$

Ist das Universum U endlich, dann gilt für die Anzahl der Teilmengen: $|\mathcal{P}(U)| = 2^{|U|}$. (Für jedes Element des Universums können wir entscheiden, ob es in einer bestimmten Teilmenge enthalten ist oder nicht.) Die Potenzmenge $\mathcal{P}(U)$ mit den Mengenoperationen bildet eine Boolesche Algebra: $(\mathcal{P}(U), \emptyset, U, \neg, \cap, \cup)$ wobei $\neg A := U \setminus A$ das relative *Komplement* bezüglich U ist. Mit anderen Worten, die in Abbildung A.1 aufgeführten Gesetze gelten in gleicher Weise für die Mengenoperationen.

Das *kartesische Produkt* $A \times B$ von A und B ist die Menge aller Paare, deren erste Komponente aus A und deren zweite Komponente aus B stammt.

(Kartesisches Produkt) $$A \times B := \{\, (a, b) \mid a \in A \wedge b \in B \,\}$$

Entsprechend wird das kartesisches Produkt von endlich vielen Mengen als Menge aller n-Tupel definiert. Sind die Grundmengen A und B endlich, dann gilt: $|A \times B| = |A| \cdot |B|$.

Relationen und Abbildungen Eine binäre *Relation* R zwischen zwei Mengen A und B ist eine Teilmenge des kartesischen Produkts: $R \subseteq A \times B$. Eine Relation $f \subseteq A \times B$ mit der Eigenschaft, dass es zu jedem $x \in A$ genau ein $y \in B$ mit $(x, y) \in f$ gibt, heißt *Abbildung* oder *Funktion*. Dabei ist A der *Definitionsbereich* und B der *Bild-* oder *Wertebereich* der Funktion, notiert $f\colon A \to B$. Das eindeutige Element y heißt *Funktionswert* von x, notiert $y = f(x)$.

A.1.3. Induktion

Auf Giuseppe Peano (1858–1932) geht das folgende Axiomensystem zur Einführung der natürlichen Zahlen zurück, siehe auch Abbildung 3.4.

- 0 ist eine natürliche Zahl.

  $$0 \in \mathbb{N}$$

- Für alle n gilt, dass, wenn n eine natürliche Zahl ist, auch die auf n folgende Zahl eine natürliche Zahl ist.

  $$\forall\, n \;.\; n \in \mathbb{N} \Longrightarrow n + 1 \in \mathbb{N}$$

- Wenn auf zwei Zahlen dieselbe Zahl folgt, sind sie identisch.

  $$\forall\, m, n \;.\; m + 1 = n + 1 \Longrightarrow m = n$$

- 0 kann nicht auf eine natürliche Zahl folgen.

  $$\neg(\exists\, n \;.\; n \in \mathbb{N} \wedge 0 = n + 1) \quad \text{oder} \quad \forall\, n \;.\; n \in \mathbb{N} \Longrightarrow 0 \neq n + 1$$

- Das *Induktionsaxiom:* Wenn 0 eine Eigenschaft hat, und wenn jede auf eine natürliche Zahl folgende Zahl die Eigenschaft besitzt, sofern die Zahl selbst die Eigenschaft hat, dann haben alle natürlichen Zahlen die betreffende Eigenschaft.

  $$P(0) \;\wedge\; (\forall\, n \;.\; P(n) \Rightarrow P(n+1)) \quad\Longrightarrow\quad \forall n \;.\; P(n)$$

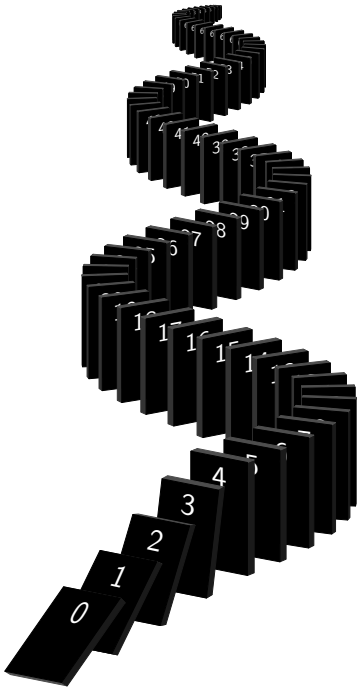
Das Induktionsaxiom ist die Grundlage der Beweismethode durch *vollständige Induktion*. (Die Beweismethode hat somit ihre Wurzeln in den Eigenschaften der natürlichen Zahlen und ist nicht etwa formallogisch begründet, gottgegeben oder gar ein Naturgesetz.)

Die Methode lässt sich etwas übersichtlicher mit Hilfe einer Deduktionsregel aufschreiben.

$$\frac{P(0) \qquad \forall\, n \;.\; P(n) \Longrightarrow P(n+1)}{\forall n \;.\; P(n)}$$

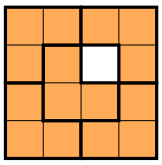
Um eine Aussage zu zeigen, die für alle natürlichen Zahlen gilt — die Schlussfolgerung unter dem Strich —, müssen zwei Beweise geführt werden — die Voraussetzungen über dem Strich:

- *Induktionsbasis* oder *-verankerung*: Man muss zeigen, dass die Aussage für 0 gilt.
- *Induktionsschritt*: Für alle natürlichen Zahlen n muss man unter der Annahme, dass die Aussage für n gilt, zeigen, dass sie für $n+1$ gilt. Die Annahme nennt man *Induktionsannahme* oder *-hypothese*.



Die Beweismethode der vollständigen Induktion wird auch manchmal als *Dominoprinzip* bezeichnet. Um eine lange Reihe hochkant aufgestellter Dominosteine umzuwerfen, muss man den ersten Stein antippen (Induktionsbasis) und man muss sicherstellen, dass der n-te Stein beim Fallen den $(n+1)$-ten Stein trifft und diesen zum Umfallen bringt (Induktionsschritt). Um die Domino-Kettenreaktion zu garantieren, muss letzteres für alle n gewährleistet sein.

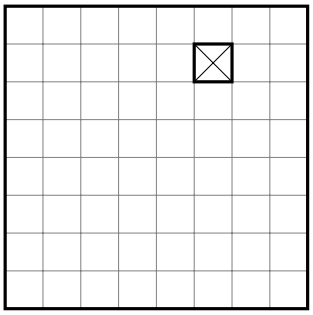
In der Schule und in den meisten mathematischen Lehrbüchern wird das Induktionsprinzip mit Beispielen aus der Arithmetik illustriert und eingeübt. Wir wollen uns an dieser Stelle ein geometrisches Problem vornehmen und dieses induktiv lösen. Es gilt ein Schachbrett mit sogenannten *L-Trominos* zu kacheln. Ähnlich wie ein Dominostein aus zwei gleichgroßen Quadraten zusammengesetzt wird, besteht ein Tromino aus drei Quadraten. Da die Steine beim Kacheln beliebig gedreht werden dürfen, gibt es lediglich zwei unterschiedliche Trominos: das I-Tromino  und das L-Tromino  . Ziel ist es, ein $2^n \times 2^n$ großes Schachbrett möglichst vollständig zu kacheln, ohne dass sich die Steine überlappen oder aus dem Brett herausragen.



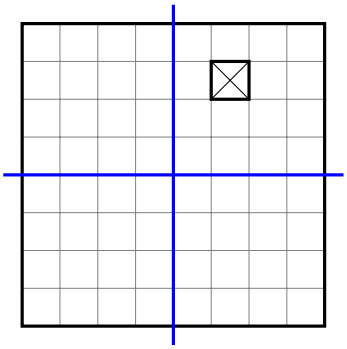
Da $2^n \times 2^n = 4^n$ kein Vielfaches von 3 ist, lässt sich das Brett nicht vollständig kacheln. Aber wir können alle Felder mit einer einzigen, *beliebigen* Ausnahme bedecken. Wenn man möchte, kann man das Problem als Spiel auffassen: Der Opponent wählt eine Brettgröße aus und markiert ein beliebiges Feld, das freibleiben soll. Wir müssen dann das restliche Spielfeld mit L-Trominos kacheln. Wir führen den Beweis, dass dies immer gelingt, durch vollständige Induktion über n.

Induktionsbasis: Ein $2^0 \times 2^0$ Schachbrett besteht nur aus einem einzigen Feld. Der Opponent hat keine andere Möglichkeit, als dieses Feld auszuwählen. Das restliche Feld ist leer und damit auch vollständig gekachelt.

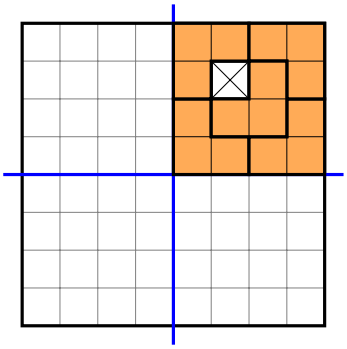
Induktionsschritt: Wir müssen ein $2^{n+1} \times 2^{n+1}$ Schachbrett kacheln und dürfen annehmen, dass wir Schachbretter der Größe $2^n \times 2^n$ bereits kacheln können (*Induktionsannahme*). Schauen wir uns ein konkretes Beispiel an, um eine Idee für den Lösungsansatz zu bekommen (n ist 3 und das ausgewählte Feld ist f7).



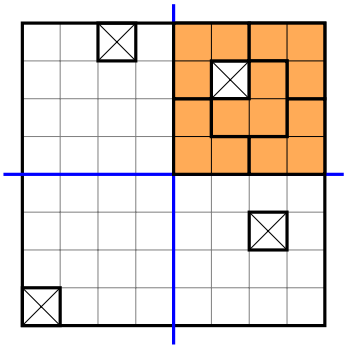
Wir müssen versuchen, die Induktionsannahme anzuwenden. Eine vielleicht naheliegende Idee ist, dass Feld in vier Quadranten der Größe $2^n \times 2^n$ aufzuteilen. (Das ist *nicht* die einzige Möglichkeit — später mehr dazu.)



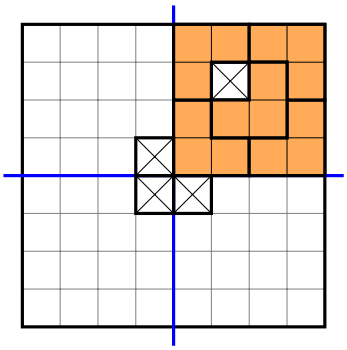
Damit lässt sich die Induktionsannahme auf den rechten, oberen Quadranten anwenden. (Da es in dem Beweis nur um die Existenz einer Kachelung geht, interessiert uns eigentlich nicht, wie diese konkret aussieht.)



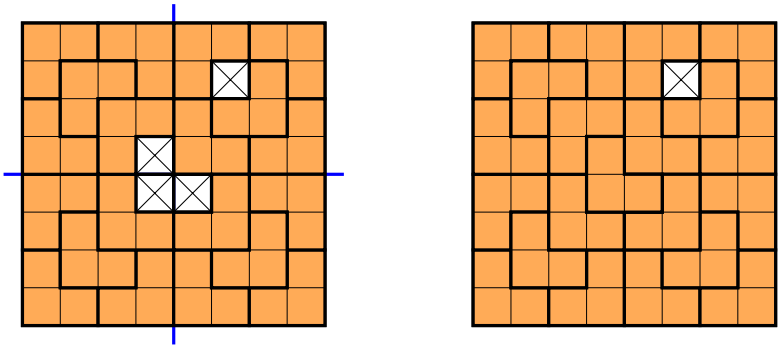
Auf die übrigen drei Quadranten ist die Induktionsannahme leider *nicht* anwendbar, da keine Felder markiert sind. Was ist zu tun? Es gibt mindestens zwei Auswege aus dem Dilemma. Wir haben im Prinzip ein neues Problem identifiziert, die Kachelung eines großen L-Trominos, das wir ebenfalls induktiv lösen könnten. Versuchen Sie es! Alternativ können wir einfach in jedem der Quadranten ein Feld markieren.



Jetzt lässt sich die Induktionsannahme auf die drei Quadranten anwenden. Leider ist nicht viel gewonnen, da wir insgesamt eine Kachelung mit vier freien Feldern erhalten. Gewonnen haben wir aber vielleicht die Einsicht, dass wir die freien Felder so wählen müssen, dass wir sie mit einem L-Tromino bedecken können. Da bleibt uns nur eine Wahl.



Jetzt lässt sich die Induktionsannahme auf alle vier Quadranten anwenden *und* wir können die drei Felder in der Mitte mit einem L-Tromino bedecken.



Da die Vorgehensweise unabhängig von der Größe des Schachbretts und der Position des freien Feldes ist, haben wir das Problem gelöst.

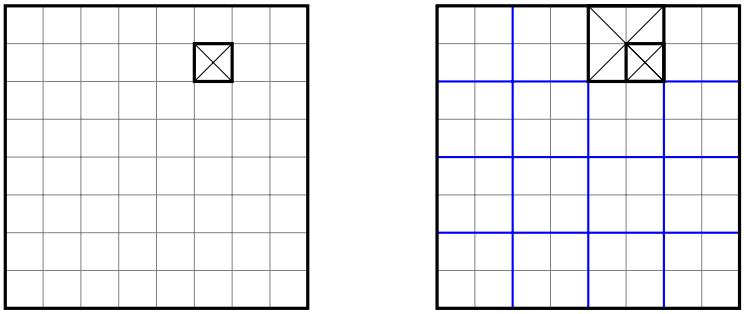
Was lassen sich für Erkenntnisse aus dem Beweis ziehen? Was haben wir gelernt? Fasst man Beweisen als eine zielgerichtete Tätigkeit auf, so gibt es im Induktionsschritt zwei klar identifizierbare Teilziele bzw. Teilprobleme („divide et impera“):

- Die Anwendbarkeit der Induktionsannahme ist nicht immer offensichtlich. Wir müssen *kreativ* sein und uns überlegen, wie wir das Problem für $n + 1$ auf ein oder mehrere Teilprobleme für n reduzieren können („teile“).

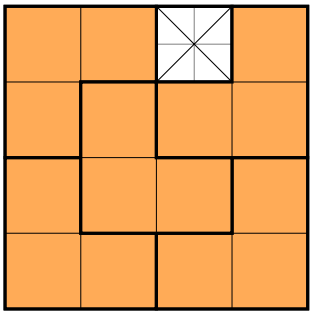
- Wenn wir die Teillösungen für die Teilprobleme durch Anwendung der Induktionsannahme in der Hand halten, müssen wir überlegen, wie sich diese zu einer Gesamtlösung für das ursprüngliche Problem zusammenführen lassen („herrsche“). Auch das ist Teil der *kreativen* Leistung beim induktiven Beweisen.

Um die beiden Punkte noch einmal zu betonen, lassen Sie uns das Kachelungsproblem bzw. den Induktionsschritt ein zweites Mal lösen.

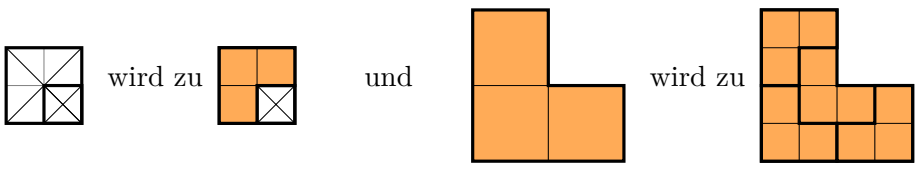
Wir können ein $2^{n+1} \times 2^{n+1}$ Schachbrett alternativ auf *ein* $2^n \times 2^n$ Schachbrett zurückführen, wenn wir die Felder vergrößern.



Jedes Feld besteht jetzt aus 2×2 Feldern des ursprünglichen Schachbretts; aus dem markierten Feld f7 wird das 2×2 Feld c4. Haben wir vorher die Induktionsannahme viermal angewendet, so müssen wir das jetzt nur noch einmal tun.



Allerdings erhalten wir eine Teillösung mit großen L-Trominos und einem großen freien Feld. Um die Teillösung in eine Lösung für das ursprüngliche Problem zu überführen, müssen wir uns überlegen, wie wir das große „Loch“ in das ursprüngliche kleine „Loch“ verwandeln und wie sich ein großes L-Tromino mit kleinen L-Trominos kacheln lässt. (Eine geometrische Figur, die mit kleinen Kopien ihrer selbst gekachelt werden kann, nennt man im Englischen übrigens *rep-tile*.)



Führt man die entsprechenden Ersetzungen durch, erhält man vielleicht überraschenderweise die gleiche Lösung wie beim ersten Ansatz. Beide Beweise sind *konstruktiv*:

Der induktive Beweis lässt sich jeweils in ein rekursives Programm überführen, dass eine konkrete Kachelung berechnet. (Wenn Sie bereits über etwas Programmiererfahrung verfügen, ist das vielleicht eine reizvolle Aufgabe.)

Das Kachelungsproblem zeigt geometrisch, dass $4^n \bmod 3 = 1$ — wenn wir ein Schachbrett der Größe $2^n \times 2^n$ mit Steinen der Größe 3 kacheln, bleibt notwendigerweise ein Feld frei. (Die Operation „mod“ berechnet den Rest der ganzzahligen Division.) Schließen wir mit einem induktiven Beweis von $4^n \equiv 1$ wobei $a \equiv b :\Longleftrightarrow a \bmod 3 = b \bmod 3$.

Induktionsbasis:

$$
\begin{aligned}
& 4^0 \\
\equiv \quad & \{ \text{ Potenzgesetze } \} \\
& 1
\end{aligned}
$$

Induktionsschritt:

$$
\begin{aligned}
& 4^{n+1} \\
\equiv \quad & \{ \text{ Potenzgesetze } \} \\
& 4 \cdot 4^n \\
\equiv \quad & \{ 4 \equiv 1 \text{ und Induktionsannahme } 4^n \equiv 1 \} \\
& 1 \cdot 1 \\
\equiv \quad & \{ \text{ Arithmetik } \} \\
& 1
\end{aligned}
$$

Im zweiten Schritt verwenden wir ein Gesetz der modularen Arithmetik (Rechnen mit Restklassen): Aus $a \equiv a'$ und $b \equiv b'$ folgt die Äquivalenz $a \cdot b \equiv a' \cdot b'$. (Wie das obige Beispiel illustriert, lässt sich das Beweisformat für beliebige *transitive* Relationen verwenden. Als Äquivalenzrelation ist „$\equiv$“ insbesondere transitiv.)

A.1.4. Ordnungen und Verbände

Order! Order! Order!
— John Simon Bercow (1963)

Ordnungen Eine Menge P mit einer Relation $R \subseteq P \times P$ heißt *Quasiordnung*, wenn R *reflexiv* und *transitiv* ist. Ist die Ordnungsrelation *total*, so spricht man entsprechend von einer *totalen Quasiordnung*, anderenfalls von einer *partiellen Quasiordnung*. Eine *Ordnung* liegt vor, wenn R zusätzlich *antisymmetrisch* ist. Eine *totale Ordnung* besitzt alle vier Eigenschaften. Eine Ordnungsrelation wird üblicherweise mit dem Vergleichszeichen $\leqslant$ notiert, das in vielen verschiedenen Variationen existiert: $\leq$, $\leqslant$, $\preceq$, $\preccurlyeq$, $\subseteq$.

(Reflexivität)	$a \leqslant a$
(Transitivität)	$a \leqslant b \ \wedge \ b \leqslant c \quad \Longrightarrow \quad a \leqslant c$
(Totalität)	$a \leqslant b \ \vee \ b \leqslant a$
(Antisymmetrie)	$a \leqslant b \ \wedge \ b \leqslant a \quad \Longrightarrow \quad a = b$

Ist a kleiner gleich b und b kleiner gleich c, so notiert man kurz $a \leqslant b \leqslant c$.[3]

Aus einer gegebenen Quasiordnung lassen sich weitere Relationen ableiten: die *duale* Ordnung $\geqslant$, die die Quasiordnung „auf den Kopf stellt“; die *strikten* Ordnungen $<$ und $>$; und die *Äquivalenzrelation* $\sim$.

$$\begin{aligned} a \geqslant b \quad &:\Longleftrightarrow \quad b \leqslant a \\ a < b \quad &:\Longleftrightarrow \quad a \leqslant b \ \wedge \ \neg(b \leqslant a) \\ a \sim b \quad &:\Longleftrightarrow \quad a \leqslant b \ \wedge \ b \leqslant a \\ a > b \quad &:\Longleftrightarrow \quad \neg(a \leqslant b) \ \wedge \ b \leqslant a \end{aligned}$$

Eine strikte Ordnung ist *irreflexiv* ($a < a$ gilt nicht) und transitiv; eine Äquivalenzrelation ist reflexiv, *symmetrisch* (aus $a \sim b$ folgt $b \sim a$) und transitiv.

Die Wahrheitswerte $\mathbb{B}$ bilden mit der Implikation $\Rightarrow$ eine totale Ordnung, „falsch“ ist kleiner als „wahr“ : $0 \Rightarrow 1$. Die Potenzmenge $\mathcal{P}(U)$ bildet mit der Mengeninklusion $\subseteq$ eine partielle Ordnung. Auf den natürlichen Zahlen $\mathbb{N}$ wird mit

$$a \leq b \quad :\Longleftrightarrow \quad \exists\, n \in \mathbb{N} \ . \ a + n = b$$

eine totale Ordnung festgelegt.

Um die Gleichheit bzw. Ungleichheit von Elementen nachzuweisen, gibt es zwei populäre Beweistechniken: den Ping-Pong Beweis und den indirekten Beweis.

Der *Ping-Pong Beweis* leitet sich direkt aus der Antisymmetrie ab.

$$a = b \quad :\Longleftrightarrow \quad a \leqslant b \ \wedge \ b \leqslant a$$

Ich zeige, dass das eine Element von dem anderen dominiert wird und umgekehrt.

Der *indirekte Beweis* basiert auf der folgenden Einsicht: Zwei Elemente sind gleich, wenn sie sich zu allen anderen Elementen gleich verhalten.

$$\begin{aligned} a = b \quad &\Longleftrightarrow \quad \forall\, x \in P \ . \ a \leqslant x \Longleftrightarrow b \leqslant x \\ a = b \quad &\Longleftrightarrow \quad \forall\, x \in P \ . \ x \leqslant a \Longleftrightarrow x \leqslant b \\ a \leqslant b \quad &\Longleftrightarrow \quad \forall\, x \in P \ . \ a \leqslant x \Longleftarrow b \leqslant x \\ a \leqslant b \quad &\Longleftrightarrow \quad \forall\, x \in P \ . \ x \leqslant a \Longrightarrow x \leqslant b \end{aligned}$$

Die Technik der Indirektion kommt auch bei Definitionen zum Einsatz, etwa wenn ein Element durch seine Beziehungen zu anderen Elementen festgelegt wird. Die folgenden Konzepte illustrieren „indirekte Definitionen“.

Die Ordnung P besitzt genau dann ein *kleinstes Element* $\bot$, wenn für alle $b \in P$ gilt:

$$\text{(kleinstes Element)} \qquad \bot \leqslant b \qquad \text{(A.1)}$$

[3] Geschachtelte Ausdrücke kennen wir von Summen und Produkten. Die Schachtelung meint hier aber etwas völlig anderes: $a+b+c$ ist *assoziativ* zu lesen, $(a+b)+c$ oder $a+(b+c)$, und wird üblicherweise bei assoziativen Operationen verwendet; $a \leqslant b \leqslant c$ ist hingegen *konjunktiv* zu lesen, $a \leqslant b \wedge b \leqslant c$, und wird üblicherweise bei transitiven Relationen verwendet.

Entsprechend besitzt P ein *größtes Element* $\top$, wenn für alle $a \in P$ gilt:

$$\text{(größtes Element)} \qquad a \leqslant \top \tag{A.2}$$

Sei P eine Ordnung und $A, B \subseteq P$. Die Menge A besitzt genau dann eine *kleinste obere Schranke* $\bigsqcup A$, wenn für alle $b \in P$ gilt:

$$\text{(kleinste obere Schranke)} \qquad \bigsqcup A \leqslant b \iff (\forall\, a \in A \;.\; a \leqslant b) \tag{A.3}$$

Ein Element b, das die Formel auf der rechten Seite erfüllt, heißt *obere Schranke* von A: b ist größer als alle Elemente in A. Von links nach rechts gelesen besagt die obige Äquivalenz, dass $\bigsqcup A$ eine obere Schranke von A ist — ersetze b durch $\bigsqcup A$. Von rechts nach links gelesen wird $\bigsqcup A$ als kleinste der oberen Schranken identifiziert. Entsprechend existiert die *größte untere Schranke* $\bigsqcap B$ von B, wenn für alle $a \in P$ gilt:

$$\text{(größte untere Schranke)} \qquad (\forall\, b \in B \;.\; a \leqslant b) \iff a \leqslant \bigsqcap B \tag{A.4}$$

Ist P eine totale Ordnung, so wird $\bigsqcap B$ auch *Minimum* von B genannt und $\bigsqcup A$ *Maximum* von A.

Verbände⋆ Wenden wir uns den algebraischen Eigenschaften der kleinsten oberen und der größten unteren Schranke zu und nehmen die Gelegenheit wahr, einige Beweise zu führen. Dazu vereinbaren wir Abkürzungen für die binären Instanzen:

- *Supremum* $a \sqcup b := \bigsqcup\{a, b\}$ (engl. join) und
- *Infimum* $a \sqcap b := \bigsqcap\{a, b\}$ (engl. meet).

Abbildung A.2 fasst die Gesetze zusammen, die wir im Folgenden Schritt für Schritt einführen und diskutieren.

Eine Ordnung P heißt *Verband*, wenn $a \sqcup b$ und $a \sqcap b$ für alle $a, b \in P$ existieren. Der Verband ist *vollständig*, wenn $\bigsqcup A$ und $\bigsqcap B$ für beliebige Teilmengen $A, B \subseteq P$ existieren. Ein vollständiger Verband besitzt stets ein kleinstes und ein größtes Element und es gilt:

$$\bigsqcup \emptyset = \bot = \bigsqcap P \quad \text{und} \quad \bigsqcap \emptyset = \top = \bigsqcup P$$

Statt Supremum und Infimum von der kleinsten oberen bzw. der größten unteren Schranke abzuleiten, können wir sie alternativ mittels der folgenden Äquivalenzen definieren — (A.5) und (A.6) sind (A.3) bzw. (A.4) spezialisiert auf 2-elementige Mengen.

$$\text{(Supremum)} \qquad a_1 \sqcup a_2 \leqslant b \iff a_1 \leqslant b \;\wedge\; a_2 \leqslant b \tag{A.5}$$

$$\text{(Infimum)} \qquad a \leqslant b_1 \;\wedge\; a \leqslant b_2 \iff a \leqslant b_1 \sqcap b_2 \tag{A.6}$$

Konzentrieren wir uns auf das Supremum — die folgenden Erörterungen gelten entsprechend für das Infimum. Überlegen wir, welche Folgerungen sich aus (A.5) ziehen lassen.

Wir können zum Beispiel die linke Seite der Äquivalenz wahr machen, indem wir b durch $a_1 \sqcup a_2$ ersetzen. Somit gilt auch die rechte Seite und wir erhalten:

$$\text{(obere Schranke)} \qquad a_1 \leqslant a_1 \sqcup a_2 \quad \wedge \quad a_2 \leqslant a_1 \sqcup a_2 \tag{A.7a}$$

Die Formel besagt, dass das Supremum $a_1 \sqcup a_2$ tatsächlich eine obere Schranke von a_1 und a_2 ist. (Welche Formel erhalten wir, wenn wir umgekehrt die rechte Seite von (A.5) wahr machen?)

Das Supremum „verträgt“ sich mit der zugrundeliegenden Ordnung: $\sqcup$ ist *ordnungserhaltend* oder *monoton*.

$$\text{(Monotonie)} \qquad a_1 \leqslant b_1 \quad \wedge \quad a_2 \leqslant b_2 \quad \Longrightarrow \quad a_1 \sqcup a_2 \leqslant b_1 \sqcup b_2 \tag{A.7b}$$

Der Beweis basiert im Wesentlichen auf der Transitivität der Ordnungsrelation.

$$\begin{aligned}
& a_1 \sqcup a_2 \leqslant b_1 \sqcup b_2 \\
\Longleftrightarrow \quad & \{ \text{ Supremum (A.5) } \} \\
& a_1 \leqslant b_1 \sqcup b_2 \quad \wedge \quad a_2 \leqslant b_1 \sqcup b_2 \\
\Longleftarrow \quad & \{ \text{ obere Schranken } b_1 \leqslant b_1 \sqcup b_2 \text{ und } b_2 \leqslant b_1 \sqcup b_2 \text{ (A.7a) und Transitivität } \} \\
& a_1 \leqslant b_1 \quad \wedge \quad a_2 \leqslant b_2
\end{aligned}$$

Das Supremum ist weiterhin assoziativ, kommutativ und idempotent. Da die Operation indirekt definiert worden ist, führt man die Beweise ebenfalls indirekt.

$$\begin{aligned}
& a \sqcup b \leqslant x \\
\Longleftrightarrow \quad & \{ \text{ Supremum (A.5) } \} \\
& a \leqslant x \quad \wedge \quad b \leqslant x \\
\Longleftrightarrow \quad & \{ \text{ Logik: } \wedge \text{ ist kommutativ } \} \\
& b \leqslant x \quad \wedge \quad a \leqslant x \\
\Longleftrightarrow \quad & \{ \text{ Supremum (A.5) } \} \\
& b \sqcup a \leqslant x
\end{aligned}$$

Die Kommutativität von $\sqcup$ folgt somit aus der entsprechenden Eigenschaft der Konjunktion. Gleiches gilt für Assoziativität und Idempotenz.

Die folgenden Äquivalenzen, das sogenannte *Verbindungslemma* (engl. connection lemma), setzen Supremum $\sqcup$ und Infimum $\sqcap$ zueinander und mit der Ordnungsrelation $\leqslant$ in Beziehung.

$$a \sqcup b = b \quad \Longleftrightarrow \quad a \leqslant b \quad \Longleftrightarrow \quad a \sqcap b = a \tag{A.8}$$

Wir zeigen die erste Äquivalenz — ein analoger Beweis etabliert die zweite.

$$
\begin{aligned}
& a \sqcup b = b \\
\iff & \quad \{ \text{ Ping-Pong Beweis } \} \\
& a \sqcup b \leqslant b \ \wedge \ b \leqslant a \sqcup b \\
\iff & \quad \{ \, a \sqcup b \text{ ist eine obere Schranke von } a \text{ und } b \text{ (A.7a) } \} \\
& a \sqcup b \leqslant b \\
\iff & \quad \{ \text{ Supremum (A.5) } \} \\
& a \leqslant b \ \wedge \ b \leqslant b \\
\iff & \quad \{ \text{ Reflexivität } \} \\
& a \leqslant b
\end{aligned}
$$

Aus dem Verbindungslemma lassen sich interessante Schlüsse ziehen, indem man die Ungleichung mit bekannten Fakten instantiiert, zum Beispiel (A.7a) und (A.9a):

$$
a \sqcap (a \sqcup b) = a \quad \iff \quad a \leqslant a \sqcup b
$$

$$
a \sqcap b \leqslant b \quad \iff \quad (a \sqcap b) \sqcup b = b
$$

Wir erhalten die Absorptionsgesetze, die das Zusammenspiel von $\sqcup$ und $\sqcap$ regeln.

Als einstweiliges Fazit halten wir fest, dass aus der ordnungstheoretischen Definition des Verbandes die algebraischen Eigenschaften der Operationen folgen, die wir schon im Zusammenhang mit Booleschen Algebren kennengelernt haben, siehe Abbildung A.1.

$$
\begin{aligned}
& (\text{Supremum}) \ \wedge \ (\text{Infimum}) \quad \iff \\
& \qquad (\text{Assoziativität}) \ \wedge \ (\text{Kommutativität}) \ \wedge \ (\text{Idempotenz}) \ \wedge \ (\text{Absorption})
\end{aligned}
$$

Auch die Umkehrung gilt: Eine algebraische Struktur, die die rechts aufgeführten Gesetze erfüllt, definiert einen Verband. Für den Beweis konzentrieren wir uns zunächst auf eine einzelne Operation: Ist $\oplus$ eine assoziative, kommutative und idempotente Operation, so wird mit

$$
a \leqslant b \quad :\iff \quad a \oplus b = b
$$

eine partielle Ordnung definiert — eine Hälfte des Verbindungslemmas gilt damit *per definitionem*. Die Reflexivität von $\leqslant$ folgt aus der Idempotenz von $\oplus$, die Transitivität aus der Assoziativität und die Antisymmetrie aus der Kommutativität. Weiterhin ist $\oplus$ das Supremum. Die Gültigkeit von (A.5) zeigen wir mit einem Ping-Pong Beweis.

„$\Longrightarrow$": Wir nehmen an, dass $a_1 \oplus a_2 \leqslant b$ bzw. $a_1 \oplus a_2 \oplus b = b$ gilt, und zeigen $a_1 \leqslant b$ bzw. $a_1 \oplus b = b$ und $a_2 \leqslant b$ bzw. $a_2 \oplus b = b$:

$$\begin{aligned} & a_1 \oplus b \\ = \quad & \{ \text{ Annahme } \} \\ & a_1 \oplus a_1 \oplus a_2 \oplus b \\ = \quad & \{ \text{ Idempotenz } \} \\ & a_1 \oplus a_2 \oplus b \\ = \quad & \{ \text{ Annahme } \} \\ & b \end{aligned}$$

Ein analoger Beweis zeigt $a_2 \oplus b = b$.

„$\Longleftarrow$": Wir nehmen an, dass $a_1 \leqslant b$ bzw. $a_1 \oplus b = b$ und $a_2 \leqslant b$ bzw. $a_2 \oplus b = b$ gelten, und zeigen $a_1 \oplus a_2 \leqslant b$ bzw. $a_1 \oplus a_2 \oplus b = b$:

$$\begin{aligned} & a_1 \oplus a_2 \oplus b \\ = \quad & \{ \text{ Annahme } \} \\ & a_1 \oplus b \\ = \quad & \{ \text{ Annahme } \} \\ & b \end{aligned}$$

Wir haben einen sogenannten Halbverband vor uns — eine Ordnung P heißt *Halbverband*, wenn das Supremum stets existiert.

Zwei Halbverbände über der gleichen Grundmenge formen einen Verband, wenn die beiden Operationen $\oplus$ und $\otimes$ das Absorptionsgesetz erfüllen. Dann gilt das Verbindungslemma (A.8), $a \oplus b = b \iff a \otimes b = a$, wie der folgende Ping-Pong Beweis zeigt.

„$\Longrightarrow$": Wir nehmen an, dass $a \oplus b = b$ gilt, und zeigen $a \otimes b = a$:

$$\begin{aligned} & a \otimes b \\ = \quad & \{ \text{ Annahme } \} \\ & a \otimes (a \oplus b) \\ = \quad & \{ \text{ Absorption } \} \\ & a \end{aligned}$$

„$\Longleftarrow$": Wir nehmen an, dass $a \otimes b = a$ gilt, und zeigen $a \oplus b = b$:

$$\begin{aligned} & a \oplus b \\ = \quad & \{ \text{ Annahme } \} \\ & (a \otimes b) \oplus b \\ = \quad & \{ \text{ Absorption } \} \\ & b \end{aligned}$$

Verbände mit kleinstem und größtem Element$\star$ Besitzt ein Verband ein kleinstes bzw. ein größtes Element, so sind diese neutrale bzw. absorbierende Elemente des Supremums bzw. Infimums — eine weitere Konsequenz aus dem Verbindungslemma.

$$\begin{aligned} \bot \sqcap b = \bot \quad &\iff \quad \bot \leqslant b \quad \iff \quad \bot \sqcup b = b \\ a \sqcap \top = a \quad &\iff \quad a \leqslant \top \quad \iff \quad a \sqcup \top = \top \end{aligned}$$

Aus der Definition der Ordnung $a \leqslant b \iff a \oplus b = b$ bzw. $a \leqslant b \iff a \otimes b = a$ folgt auch die Umkehrung und somit:

$$\text{(kleinstes Element)} \;\wedge\; \text{(größtes Element)} \iff \text{(Neutralitätsgesetze)} \;\wedge\; \text{(Extremalgesetze)}$$

Axiome der Booleschen Verbände.

(Supremum)	$a_1 \sqcup a_2 \leqslant b$	$\iff$	$a_1 \leqslant b \ \wedge \ a_2 \leqslant b$	
(Infimum)	$a \leqslant b_1 \ \wedge \ a \leqslant b_2$	$\iff$	$a \leqslant b_1 \sqcap b_2$	
(kleinstes Element)	$\bot \leqslant b$	$\iff$	*true*	
(größtes Element)	*true*	$\iff$	$a \leqslant \top$	
(Komplement)	$a \sqcap p \leqslant b$	$\iff$	$a \leqslant b \sqcup p'$	
(Komplement')	$a \sqcap p' \leqslant b$	$\iff$	$a \leqslant b \sqcup p$	

Das Axiome (Komp) und (Komp') folgen jeweils aus den übrigen Axiomen.

Folgende Axiompaare sind jeweils dual zueinander: (Supremum) zu (Infimum), (kleinstes Element) zu (größtes Element) und (Komplement) zu (Komplement').

Folgerungen aus den Axiomen.

(obere Schranke)	$a_1 \leqslant a_1 \sqcup a_2$		$a_2 \leqslant a_1 \sqcup a_2$	
(untere Schranke)	$b_1 \sqcap b_2 \leqslant b_1$		$b_1 \sqcap b_2 \leqslant b_2$	(A.9a)
(Monotonie)	$a_1 \leqslant b_1 \ \wedge \ a_2 \leqslant b_2$	$\implies$	$a_1 \sqcup a_2 \leqslant b_1 \sqcup b_2$	
(Monotonie)	$a_1 \leqslant b_1 \ \wedge \ a_2 \leqslant b_2$	$\implies$	$a_1 \sqcap a_2 \leqslant b_1 \sqcap b_2$	(A.9b)
(Komplementärgesetze)	$p' \sqcap p \leqslant \bot$		$\top \leqslant p \sqcup p'$	
(Antitonie)	$a \leqslant b$	$\implies$	$b' \leqslant a'$	

Abbildung A.2.: Axiome der Booleschen Verbände und Folgerungen aus den Axiomen.

Boolesche Verbände⋆ Sei P ein Verband und $p \in P$. Das Element p besitzt genau dann ein *Komplement* p', wenn für alle $a, b \in P$ gilt:

$$\text{(Komplement)} \qquad a \sqcap p \leqslant b \iff a \leqslant b \sqcup p' \tag{A.10a}$$

$$\text{(Komplement')} \qquad a \sqcap p' \leqslant b \iff a \leqslant b \sqcup p \tag{A.10b}$$

Ein Verband mit $\bot$ und $\top$ heißt *Boolescher Verband*, wenn jedes Element ein Komplement besitzt.

Aus (A.10a) folgen die Komplementärgesetze der Booleschen Algebra.

$$\text{(Komplementärgesetze)} \qquad p' \sqcap p \leqslant \bot \qquad\qquad \top \leqslant p \sqcup p' \tag{A.11}$$

Die linke Ungleichung erhalten wir, wenn wir in (A.10a) a durch p' und b durch $\bot$ ersetzen; die rechte, wenn wir a durch $\top$ und b durch p ersetzen. (Beachte, dass $a \leqslant \bot$ äquivalent zu $a = \bot$ ist und $\top \leqslant b$ zu $\top = b$ — warum?)

In einem Verband regeln die Absorptionsgesetze bzw. das Verbindungslemma das Zusammenspiel von Supremum und Infimum. In einem Booleschen Verband gelten sogar die Distributivgesetze, wie die folgenden indirekten Beweise zeigen.

$$\begin{array}{ll}
 & (a \sqcup b) \sqcap c \leqslant x \\
\iff & \quad \{ \text{ Komplement (A.10a) } \} \\
 & a \sqcup b \leqslant x \sqcup c' \\
\iff & \quad \{ \text{ Supremum (A.3) } \} \\
 & a \leqslant x \sqcup c' \ \wedge \ b \leqslant x \sqcup c' \\
\iff & \quad \{ \text{ Komplement (A.10a) } \} \\
 & a \sqcap c \leqslant x \ \wedge \ b \sqcap c \leqslant x \\
\iff & \quad \{ \text{ Supremum (A.3) } \} \\
 & (a \sqcap c) \sqcup (b \sqcap c) \leqslant x
\end{array}$$

$$\begin{array}{ll}
 & x \leqslant (a \sqcap b) \sqcup c \\
\iff & \quad \{ \text{ Komplement' (A.10b) } \} \\
 & x \sqcap c' \leqslant a \sqcap b \\
\iff & \quad \{ \text{ Infimum (A.6) } \} \\
 & x \sqcap c' \leqslant a \ \wedge \ x \sqcap c' \leqslant b \\
\iff & \quad \{ \text{ Komplement' (A.10b) } \} \\
 & x \leqslant a \sqcup c \ \wedge \ x \leqslant b \sqcup c \\
\iff & \quad \{ \text{ Infimum (A.6) } \} \\
 & x \leqslant (a \sqcup c) \sqcap (b \sqcup c)
\end{array}$$

Der linke Beweis ist dual zum rechten Beweis.

Die Umkehrung gilt ebenfalls: Aus dem Distributivgesetz und den Komplementärgesetzen folgen (A.10a) und (A.10b). Wir zeigen (A.10a) mit einem Ping-Pong Beweis.

Wir nehmen an, dass $a \leqslant b \sqcup p'$ gilt, und zeigen $a \sqcap p \leqslant b$.

$$
\begin{aligned}
& a \sqcap p \\
\leqslant \quad & \{ \text{ Annahme u. Monotonie (A.9b) } \} \\
& (b \sqcup p') \sqcap p \\
\leqslant \quad & \{ \text{ Distributivität } \} \\
& (b \sqcap p) \sqcup (p' \sqcap p) \\
\leqslant \quad & \{ \text{ Komplementärgesetz (A.11) } \} \\
& (b \sqcap p) \sqcup \bot \\
\leqslant \quad & \{ \bot \text{ neutrales Element } \} \\
& b \sqcap p \\
\leqslant \quad & \{ \text{ untere Schranke (A.9a) } \} \\
& b
\end{aligned}
$$

Wir nehmen an, dass $a \sqcap p \leqslant b$ gilt, und zeigen $a \leqslant b \sqcup p'$.

$$
\begin{aligned}
& a \\
\leqslant \quad & \{ \text{ obere Schranke (A.7a) } \} \\
& a \sqcup p' \\
\leqslant \quad & \{ \top \text{ neutrales Element } \} \\
& (a \sqcup p') \sqcap \top \\
\leqslant \quad & \{ \text{ Komplementärgesetz (A.11) } \} \\
& (a \sqcup p') \sqcap (p \sqcup p') \\
\leqslant \quad & \{ \text{ Distributivität } \} \\
& (a \sqcap p) \sqcup p' \\
\leqslant \quad & \{ \text{ Annahme u. Monotonie (A.7b) } \} \\
& b \sqcup p'
\end{aligned}
$$

Wiederum ist der linke Beweis dual zum rechten Beweis, der am besten von unten nach oben gelesen wird.

Das Komplement lässt sich sowohl ordnungstheoretisch als auch algebraisch definieren.

$$\text{(Komplement)} \iff \text{(Distributivität)} \ \wedge \ \text{(Komplementärgesetze)}$$

Summa summarum folgen aus den Gesetzen der Booleschen Verbände die Axiome der Booleschen Algebra und umgekehrt. Boolesche Algebren und Boolesche Verbände definieren die gleiche mathematische Struktur, einmal aus algebraischer und einmal aus ordnungstheoretischer Sicht. Beide Perspektiven sind erhellend und hilfreich.

$$\text{(Boolesche Algebra)} \iff \text{(Boolescher Verband)}$$

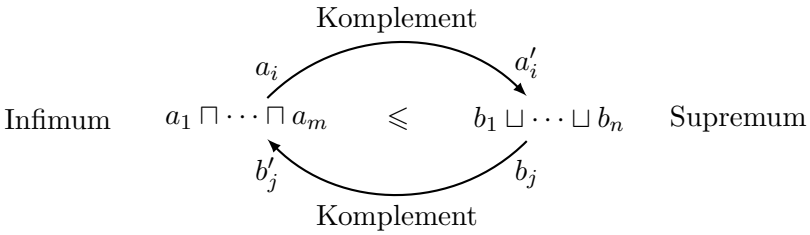
Abbildung A.2 fasst die Axiome Boolescher Verbände zusammen. Es sei noch einmal an das *Dualitätsprinzip* erinnert: Durch systematisches Ersetzen, $\leqslant \leftrightarrow \geqslant$, $\bot \leftrightarrow \top$ und $\sqcup \leftrightarrow \sqcap$, erhält man zu einer Formel die *duale* Formel. In einem Verband ist eine Formel genau dann wahr, wenn die duale Formel wahr ist.

Schauen wir uns zum Schluß an, wie die Axiome für das Komplement systematisch in Beweisen verwendet werden. Wir halten zunächst fest, dass in einem Verband mit einem kleinsten und einem größten Element jede *endliche* Teilmenge eine kleinste obere und eine größte untere Schranke hat.

$$\bigsqcap\{a_1, \ldots, a_m\} = a_1 \sqcap \cdots \sqcap a_m \qquad \bigsqcup\{b_1, \ldots, b_n\} = b_1 \sqcup \cdots \sqcup b_n$$

Ist $m = 0$, dann gilt vereinbarungsgemäß $a_1 \sqcap \cdots \sqcap a_m = \top$; ist $n = 0$, entsprechend $b_1 \sqcup \cdots \sqcup b_n = \bot$. Die Axiome (A.10a) und (A.10b) erlauben es, in einer Formel der

folgenden Form[4] Elemente von der einen auf die andere Seite zu transferieren.



Beim Transfer kommt entweder ein „Strich“ hinzu oder ein „Strich“ wird entfernt. Aus den Axiomen folgt, dass das Komplement *ordnungsumkehrend* oder *antiton* ist.

$$\text{(Antitonie)} \qquad a \leqslant b \implies b' \leqslant a' \tag{A.12}$$

Wir zeigen eine stärkere Aussage: Die beiden Ungleichungen sind tatsächlich äquivalent.

$$\begin{aligned}
& a \leqslant b \\
\iff & \quad \{\ \top \text{ neutrales Element } \} \\
& \top \sqcap a \leqslant b \\
\iff & \quad \{\ \text{Komplement (A.10a)} \ \} \\
& \top \leqslant b \sqcup a' \\
\iff & \quad \{\ \sqcup \text{ kommutativ } \} \\
& \top \leqslant a' \sqcup b \\
\iff & \quad \{\ \text{Komplement' (A.10b)} \ \} \\
& \top \sqcap b' \leqslant a' \\
\iff & \quad \{\ \top \text{ neutrales Element } \} \\
& b' \leqslant a'
\end{aligned}$$

Man sieht, wir mischen ordnungstheoretische und algebraische Eigenschaften. (Wenn man etwas Erfahrung beim Beweisen gewonnen hat, lässt man Schritte, die Assoziativität, Kommutativität oder Neutralität verwenden, typischerweise aus.)

Die Folgerungen aus den Axiomen der Booleschen Algebra, siehe Abbildung A.1, lassen sich nach diesen Vorüberlegungen leicht mittels indirekter Beweise zeigen. Zunächst einmal ist $\top$ das Komplement von $\bot$ und umgekehrt: $\bot' = \top$ und $\top' = \bot$.

$$\begin{aligned}
& \bot' \leqslant x \\
\iff & \quad \{\ \text{Komplement' (A.10b)} \ \} \\
& \top \leqslant x \sqcup \bot \\
\iff & \quad \{\ \bot \text{ neutrales Element } \} \\
& \top \leqslant x
\end{aligned}
\qquad\qquad
\begin{aligned}
& x \leqslant \top' \\
\iff & \quad \{\ \text{Komplement (A.10a)} \ \} \\
& x \sqcap \top \leqslant \bot \\
\iff & \quad \{\ \top \text{ neutrales Element } \} \\
& x \leqslant \bot
\end{aligned}$$

[4] In der Logik heißt eine Formel der Form $a_1 \wedge \cdots \wedge a_m \implies b_1 \vee \cdots \vee b_n$ auch *Klausel*.

Die Beweise sind dual zueinander—auf Grund des Dualitätsprinzips muss tatsächlich nur einer geführt werden.

Das Komplement ist eine sogenannte *Involution*. Wird die Operation zweimal in Folge angewendet, so erhält man das ursprüngliche Element: $a'' = a$.

$$
\begin{array}{ll}
 & a'' \leqslant x \\
\iff & \quad \{\ \text{Komplement'}\ (\text{A.10b})\ \} \\
 & \top \leqslant x \sqcup a' \\
\iff & \quad \{\ \text{Komplement}\ (\text{A.10a})\ \} \\
 & a \leqslant x
\end{array}
$$

Schließlich gelten die *De Morganschen Gesetze*: $(a \sqcup b)' = a' \sqcap b'$ und $(a \sqcap b)' = a' \sqcup b'$.

$$
\begin{array}{ll}
 & (a \sqcup b)' \leqslant x \\
\iff & \quad \{\ \text{Komplement'}\ (\text{A.10b})\ \} \\
 & \top \leqslant x \sqcup a \sqcup b \\
\iff & \quad \{\ \text{Komplement'}\ (\text{A.10b})\ \} \\
 & a' \leqslant x \sqcup b \\
\iff & \quad \{\ \text{Komplement'}\ (\text{A.10b})\ \} \\
 & a' \sqcap b' \leqslant x
\end{array}
\qquad
\begin{array}{ll}
 & x \leqslant (a \sqcap b)' \\
\iff & \quad \{\ \text{Komplement}\ (\text{A.10a})\ \} \\
 & x \sqcap a \sqcap b \leqslant \bot \\
\iff & \quad \{\ \text{Komplement}\ (\text{A.10a})\ \} \\
 & x \sqcap b \leqslant a' \\
\iff & \quad \{\ \text{Komplement}\ (\text{A.10a})\ \} \\
 & x \leqslant a' \sqcup b'
\end{array}
$$

Zuerst wird $a \sqcup b$ auf die rechte Seite transportiert; dann nacheinander a auf die linke und b auf die linke. Die Beweise sind wiederum dual zueinander.

A.2. Wunsch und Wirklichkeit: Mini–F# versus F#

Unterschiede in der lexikalischen Syntax Die Lehrsprache Mini–F# ist eng angelehnt an die Programmiersprache F#. Idealerweise ist jedes Mini–F# Programm ein gültiges F# Programm. Aus didaktischen Gründen sowie aus Gründen der Lesbarkeit und Bequemlichkeit weicht allerdings die *lexikalische Syntax* von Mini–F# in einigen Details von F# ab. Einige wenige Punkte sind zu beachten, wenn Ausdrücke oder Programme aus dem Skript oder den Vorlesungsfolien übernommen werden. Die folgende Gegenüberstellung illustriert zwei wichtige Unterschiede.

```
let rec ggt (m : Nat, n : Nat) : Nat =
  if   m = 0 then n
  elif n = 0 then m
  elif m >= n then ggt (m % n, n)
  else             ggt (m, n % m)
```

```
let rec ggt(m: Nat, n: Nat): Nat =
    if   m =  0N then n
    elif n =  0N then m
    elif m >= n  then ggt(m % n, n)
    else              ggt(m, n % m)
```

Der wichtigste Unterschied: Natürliche Zahlen müssen in F# mit dem Suffix `N` gekennzeichnet werden, also `0N` statt 0 und `4711N` statt 4711. Das liegt daran, dass F# im

Unterschied zu Mini–F# viele verschiedene Zahlentypen kennt (`47uy` ist zum Beispiel ein vorzeichenloses „Byte“, eine natürliche Zahl aus dem Bereich von 0 bis $2^8 - 1 = 255$).

Operatoren und Bezeichner werden in Mini–F# aufgehübscht. Die Vergleichsoperation $\geqslant$ (in Mini–F# ein Zeichen) muss in F# mit `>=` (zwei Zeichen) notiert werden.

Mini–F#	F#	Mini–F#	F#
$a < b$	`a < b`	$a + b$	`a + b`
$a \leqslant b$	`a =< b`	$a \dot{-} b$	`a - b`
$a = b$	`a = b`	$a * b$	`a * b`
$a <> b$	`a <> b`	$a \div b$	`a / b`
$a \geqslant b$	`a >= b`	$a \,\%\, b$	`a % b`
$a > b$	`a > b`		

Die konkrete Syntax für die natürliche Subtraktion $\dot{-}$ ist `-`. Die Subtraktion auf den natürlichen Zahlen „$\dot{-}$“ und die Subtraktion auf den reellen Zahlen „$-$“ haben sehr unterschiedliche Eigenschaften. Aus diesem Grund ist es hilfreich, sie einfach im Text unterscheiden zu können. Die natürliche Division $\div$ wird in F# mit `/` notiert. Auch hier ist der Grund für die Diskrepanz ein didaktischer: Die Notation „$\div$“ erinnert stets daran, dass die Division auf den natürlichen Zahlen (und auch auf den ganzen Zahlen) sich stark von der Division auf den reellen Zahlen „/“ unterscheidet.

In Mini–F# dürfen Bezeichner Bindestriche enthalten; in F# ist das nicht erlaubt: Aus *total-area* wird in F# zum Beispiel `totalArea`. In Mini–F# nehmen wir uns weiterhin die Freiheit, Bezeichner mit einem Index zu versehen; in F# ist das nicht erlaubt: Aus s_1 wird in F# `s1`. (Hier spiegeln sich meine persönlichen Vorlieben wider, für die Sie mich gerne kritisieren können — sehen Sie es als Herausforderung, geistig flexibel zu bleiben.)

Interpreter \Read-Eval-Print-Loop Programme können mit dem Interpreter, der sogenannten Read-Eval-Print-Loop (REPL), einfach ausgeführt, ausprobiert und getestet werden. Der *virtuelle* Mini–F# Interpreter und der *reale* F# Interpreter unterscheiden sich ebenfalls in einigen Details.

Mini⟩ $47 * 11$
517
Mini⟩ $it * it$
267289
Mini⟩ *reduce* $(*)$ $[1\,..\,10]$
3628800
Mini⟩ *ggt* $(2*2*3*7, 2*5*7)$
14

```
> 47N * 11N ;;
val it : Nat = 517N
> it * it ;;
val it : Nat = 267289N
> List.reduce (*) [1..10] ;;
val it : int = 3628800
> let rec ggt(m: Nat, n: Nat): Nat =
-     if   m =  0N then n
-     elif n =  0N then m
-     elif m >= n  then ggt(m % n, n)
-     else              ggt(m, n % m) ;;
val ggt : m:Nat * n:Nat -> Nat
> ggt (2N*2N*3N*7N, 2N*5N*7N) ;;
val it : Nat = 14N
```

Das sogenannte *Prompt*, die Eingabeaufforderung, ist jeweils unterschiedlich. Mini–F# verwendet „Mini⟩ “; F# unterscheidet zwei Aufforderungszeichen: „`>` “ kennzeichnet den Start einer Eingabe; eine noch unvollständige Eingabe wird durch „`-` “ angezeigt. Da sich Eingaben in F# über mehrere Zeilen erstrecken können, muss das Eingabeende explizit durch `;;` angezeigt werden. Die Antwort auf eine Eingabe fällt in F# etwas ausführlicher aus: Mit ***val*** $it : t = \nu$ wird neben dem Wert ν auch sein Typ t ausgegeben. Der Wert wird an den Bezeichner *it* gebunden und steht für weitere Rechnungen zur Verfügung.

Das Modulsystem von F# F# verfügt über ein einfaches *Modulsystem*, mit dessen Hilfe Ausdrücke und Deklarationen zu größeren konzeptionellen Einheiten zusammengefasst werden können. In der Vorlesung selbst wird das Modulsystem nicht thematisiert (siehe aber Kapitel 8); für die praktischen Übungen sind Module zwar nicht unverzichtbar, jedoch sehr bequem. In der Regel werden Sie für jede Programmieraufgabe entweder ein „Modulskelett mit Leben füllen“ oder ein eigenes Modul erstellen müssen. Bei der Softwareentwicklung hilft das Modulsystem größere oder große Softwaresysteme zu organisieren; in ähnlicher Weise wird die Organisation der Übungen und deren Lösung unterstützt: Es wird sichergestellt, dass sich Lösungen für verschiedene Aufgaben „nicht ins Gehege kommen“ und Sie können, wenn die Aufgaben aufeinander aufbauen, Lösungen wieder- und weiterverwenden.

module Introduction. Arithmetic

Schauen wir uns ein überschaubares Beispiel an.

$$
\begin{array}{l}
\textbf{\textit{module}}\ \mathit{Introduction.Arithmetic} \\
\textbf{\textit{let rec}}\ \mathit{ggt}\ (m : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} = \\
\quad \textbf{\textit{if}}\quad m = 0\ \textbf{\textit{then}}\ n \\
\quad \textbf{\textit{elif}}\ n\ = 0\ \textbf{\textit{then}}\ m \\
\quad \textbf{\textit{elif}}\ m \geqslant n\ \textbf{\textit{then}}\ \mathit{ggt}\ (m \,\%\, n, n) \\
\quad \textbf{\textit{else}}\qquad\qquad\ \ \mathit{ggt}\ (m, n \,\%\, m) \\
\textbf{\textit{let}}\ \mathit{kgv}\ (m : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} = \\
\quad (m * n) \div \mathit{ggt}\ (m, n)
\end{array}
$$

```
module Introduction.Arithmetic
let rec ggt(m: Nat, n: Nat): Nat =
    if   m =  0N then n
    elif n =  0N then m
    elif m >= n  then ggt(m % n, n)
    else              ggt(m, n % m)
let kgv(m: Nat, n: Nat): Nat =
    (m * n) / ggt (m, n)
```

Nach dem Schlüsselwort ***module*** wird der Name des Moduls angegeben —für jedes Modul müssen Sie sich einen Namen ausdenken bzw. geben wir in der Übung einen Namen vor. Die Beispielprogramme aus der Vorlesung sind der Übersichtlichkeit halber auf mehrere Ordner verteilt: Der Ordner `Introduction` enthält zum Beispiel die Programme aus Kapitel 1; das obige Programm befindet sich in diesem Ordner in der Datei `Arithmetic`. (Hinweise der Form ***module*** *Introduction.Arithmetic* im Seitenrand weisen den Weg.) Das Modul kann man entweder im Interpreter mit dem Befehl `#load` „laden“

```
$ fsi Mini.fs
Microsoft (R) F# Interactive version 10.7.0.0 for F# 4.7
...
> #load "Introduction/Arithmetic.fs" ;;
```

```
> Introduction.Arithmetic.ggt(84N, 70N) ;;
val it : Nat = 14N
```

oder dem Interpreter beim Aufruf mit auf den Weg geben.

```
$ fsi Mini.fs Introduction/Arithmetic.fs
Microsoft (R) F# Interactive version 10.7.0.0 for F# 4.7
...
> Introduction.Arithmetic.ggt(84N, 70N) ;;
val it : Nat = 14N
```

F# kennt von Haus aus keine natürlichen Zahlen; das Modul `Mini.fs` bringt F# die natürlichen Zahlen und einige andere Dinge bei. Das von uns zur Verfügung gestellte Modul sollte beim Aufruf des Interpreters stets als erstes Argument angegeben werden.

Jedes Modul spannt einen eigenen Namensraum auf. In unterschiedlichen Modulen können die gleichen Bezeichner definiert werden, ohne dass sich diese ins Gehege kommen. Innerhalb eines Moduls gelten die üblichen Sichtbarkeitsregeln: *kgv* sieht zum Beispiel *ggt* und kann sich auf diese Funktion abstützen. Außerhalb eines Moduls muss man dem Bezeichner den Modulnamen voranstellen: aus *ggt* wird der sogenannte *qualifizierte* Bezeichner *Introduction.Arithmetic.ggt* (siehe obige Interaktion). Alternativ lässt sich ein Modul „öffnen“.

```
> open Introduction.Arithmetic ;;
> ggt(84N, 70N) ;;
val it : Nat = 14N
```

Jetzt wird der Modulpräfix automatisch ergänzt. (Das Modul *Mini* wird übrigens immer automatisch geöffnet, so dass keine qualifizierten Bezeichner verwendet werden müssen.)

Literaturverzeichnis

[Kap00] Robert Kaplan. *Die Geschichte der Null*. Campus Verlag, 2000. ISBN 3593364271.

[KCR98] Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Publishing Company, 3rd edition, 1997.

Index

Kursiv gesetzte Seitenzahlen verweisen auf die Definition des betreffenden Sachworts, mit einem Stern markierte Seitenzahlen referenzieren in Aufgaben. Die mathematischen Symbole sind zu Beginn aufgeführt und nach Themen geordnet, alphabetische Symbole bilden die zweite Gruppe und sind lexikalisch angeordnet.

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

Y

Z

Inhaltsverzeichnis