Grundlagen der Programmierung

Rechnen und rechnen lassen

Ralf Hinze

Fachbereich Informatik
Technische Universität Kaiserslautern

WS 2019/2020

0. Knobelaufgabe #1

Ein Elb hat Ihnen 5 Gegenstände von unbekanntem Wert geschenkt. Die Gegenstände sollen ihrem Wert nach geordnet werden. Sie können dazu einem Orakel jeweils zwei Gegenstände vorlegen und erhalten dann Auskunft, welcher der beiden Gegenstände wertvoller ist.

Wie oft müssen Sie das Orakel *im ungünstigsten Fall* fragen?

Im günstigsten Fall muss das Orakel 4-mal befragt werden. Entwickeln Sie eine Systematik, so dass das Orakel nicht häufiger als nötig beansprucht wird.

Grundlagen der Programmierung

Teil I

Einführung

1. Das Vorlesungsteam

- Dozent
  - Prof. Dr. Ralf Hinze
- Organisation des Übungsbetriebs
  - Sebastian Schweizer
- Tutor*innen
  - Magdalena Allmann
  - Aaron Hackenberg
  - Milan Koch
  - Gianna Lisa Nicolai
  - Jonas Noglik
  - Filippo Palascino
  - Sophia Porcher
  - Roman Reimche
  - Silva Schillig
  - Julian Stieß
  - Philipp Werner
  - Felix Winkler

1. Wir helfen Ihnen ...

- Homepage der Vorlesung:

  `https://pl.cs.uni-kl.de/gdp19`

- Material (Duden: Gesamtheit von Hilfsmitteln, die für eine bestimmte Arbeit benötigt werden):
  - Vorlesungsfolien (zwei Tage vor der jeweiligen Vorlesung verfügbar)
  - Beispielprogramme aus der Vorlesung
  - Anleitungen (zum Beispiel zur Installation von Software)
  - Lösungshinweise zu Übungsblättern
  - Skript (zeitnah)
- Bei Fragen:
  - vorlesungsbegleitende Übung
  - Sprechstunden
  - Q&A System des Fachbereichs: `https://q2a.cs.uni-kl.de/`

1. Übungsbetrieb

- ein Übungstermin pro Woche (2 SWS, verpflichtend)
  - Teilnahme verpflichtend
  - erste Übung: ab dem 5. November
- Anmeldung zur Übung: mit dem Exclaim-System (siehe Homepage)
  - Anmeldefrist: Mittwoch 30. Oktober um 18:00 Uhr
  - Angabe von Präferenzen für Termine und Übungspartner
- Sprechstunden (2 SWS, freiwillig)
  - Sprechstunde zur Vorlesung: dienstags 17:15–18:45, 13-222.
    Je nach Bedarf: Repetitorium, Lösung von Knobelaufgaben, und/oder weitergehender Stoff.
  - Sprechstunden zur Übung: mehrere Termine jeweils im Terminalraum 32-411. Die Termine werden auf der Homepage bekanntgegeben. Programmieranfänger*innen empfehlen wir mindestens eine Sprechstunde pro Woche zu besuchen und die Programmieraufgaben in der Sprechstunde zu lösen.

1. Übungsblätter

☞ Programmieren lernt man nur durch Üben!

- Ein Übungsblatt pro Woche
  - Ausgabe: freitags
  - Abgabe: freitags eine Woche später um 15:00 Uhr
  - erstes Übungsblatt: diese Woche (ohne Abgabe)
  - Programmieraufgaben werden über das Online-System “Exclaim” abgegeben, andere Aufgaben können auch auf Papier abgegeben werden.
- Bearbeitung in Zweiergruppen
- Aufgabentypen
  - Pflichtaufgaben – müssen abgegeben werden, werden korrigiert und die erreichten Punkte zählen für die Klausurzulassung.
  - Trainingsaufgaben – sollten Sie bearbeiten, um mehr Übung im Programmieren zu bekommen.
  - Knobel- und Forschungsaufgaben – gehen über den Stoff der Vorlesung hinaus.
- Bearbeitungszeit: durchschnittlich 3–4 Stunden pro Übungsblatt

1. Klausurzulassung

- mindestens 60% der Übungspunkte aus den Übungsblättern
- Anwesenheitspflicht in Übungen (max. $2\times$ entschuldigtes Fehlen, darüber hinaus Attest erforderlich)
- Probeklausur: Teilnahme freiwillig, Bonuspunkte für die Zulassung, Termin: 7. Januar 2020, 17:15 Uhr in der Mensa
- Abschlussklausur: Termin wird vom zentralen Prüfungsamt festgelegt und rechtzeitig bekanntgegeben.

1. Historische Daten

- Wintersemester 2018/2019
  - Anmeldungen: 352
  - $\geqslant 5$ Übungsblätter bearbeitet: 253 ($\approx 72\%$)
  - Zulassung erreicht: 184 ($\approx 73\%$)
  - Abschlussklausur: 187 Teilnehmer (davon 22 mit alter Zulassung)

1,0	1,3	1,7	2,0	2,3	2,7	3,0	3,3	3,7	4,0	5,0
22	18	16	16	11	13	11	10	13	11	46
	21%		23%			18%			13%	25%

  - Nachklausur: 19 Teilnehmer

1,0	1,3	1,7	2,0	2,3	2,7	3,0	3,3	3,7	4,0	5,0
	1				1		1		2	14
					26%					74%

1. Die erste Woche

Diese Woche noch keine Übungstermine, dafür zusätzliche Sprechstunden.

☞ *To-do-Liste:*

- Melden Sie sich für die Übung an (siehe Homepage).
- Beantragen Sie einen Account beim „Service Center Informatik“ (SCI).
- Installieren Sie die benötigte Software (siehe Übungsblatt 0).
- Besuchen Sie eine Sprechstunde bei Problemen mit der Installation (besonderer Service in der 1. Woche).
- Bereiten Sie das Übungsblatt 0 für die Übungsstunde in der nächsten Woche vor, in der das Blatt besprochen wird.
- Besuchen Sie eine Sprechstunde, wenn Sie Fragen zu den Aufgaben des ersten Übungsblatts haben.

1. Helfen Sie uns ...

- Fehler in den Folien (mehr als tausend Folien; jede Einzelne bietet die Gelegenheit Fehler zu machen)
- Fehler in den Übungsaufgaben
- Korrekturleser für das Skript gesucht:
  - keine Vorkenntnisse
  - gute Informatikkenntnisse
  - gute Programmierkenntnisse
  - Informatik im Nebenfach

  kurze „Bewerbung“ an:

  `support@harry-hacker.org`

  Motivation? Zu welcher Gruppe oder Gruppen gehören Sie?
- Schicken Sie uns originelle oder schöne Lösungen für Aufgaben und/oder Knobelaufgaben:

  `support@harry-hacker.org`

1. Helfen Sie sich und anderen ...

- Seien sie präsent ...
  - Vorlesung und Übung *ergänzen* sich; sie ersetzen sich nicht.
- Seien Sie aktiv ...
  - Stellen Sie Fragen! Gerne auch in der Pause und/oder nach der Vorlesung.
  - Üben Sie die Formalismen.
  - Schreiben Sie Programme.
  - Nehmen Sie die Angebote wahr: Sprechstunden.
  - Nachbearbeitungszeit für die Vorlesung: durchschnittlich 3–4 Stunden pro Woche (Gesamtaufwand: 12–14 Stunden)
- Organisieren Sie sich ...
  - Zu zweit oder zu dritt lernt es sich besser.

1. Hörsaal Etikette

- Schalten Sie bitte Ihr Handy *aus*.
- Schalten Sie bitte alle anderen elektronischen Geräte in den *Flugmodus*.
- Kommen Sie bitte frühzeitig.
- Verlassen Sie die Vorlesung *bitte* nicht vorzeitig.
- Kurzum: nehmen Sie auf Ihre Kommiliton*innen Rücksicht.
- zeitlicher Ablauf
  - $\sim$ 40 min Vorlesung
  - $\sim$ 10 min Pause
  - $\sim$ 40 min Vorlesung

1. Kurzum ...

- Die “Grundlagen der Programmierung” ist *Ihre* Vorlesung.
- Studieren Sie mit Spaß und Begeisterung.
- Helfen Sie mit, dass die Vorlesung für Sie und für uns zu einem Erfolg wird.

2. Informatik

Informatik ist die Wissenschaft vom maschinellen Rechnen.

2. Die Natur des Rechnens

- Rechnen?
- Umgang mit Zahlen, Arithmetik!?
- Rechnen ist etwas Mechanisches; es folgt festen Regeln.
- Die Regeln für Addition und Multiplikation lernt man in der Grundschule. Schriftliche Addition:

$$\begin{array}{rcccc} & & 8 & 1 & 5 \\ + & 4_1 & 7 & 1 & 1 \\ \hline = & 5 & 5 & 2 & 6 \end{array}$$

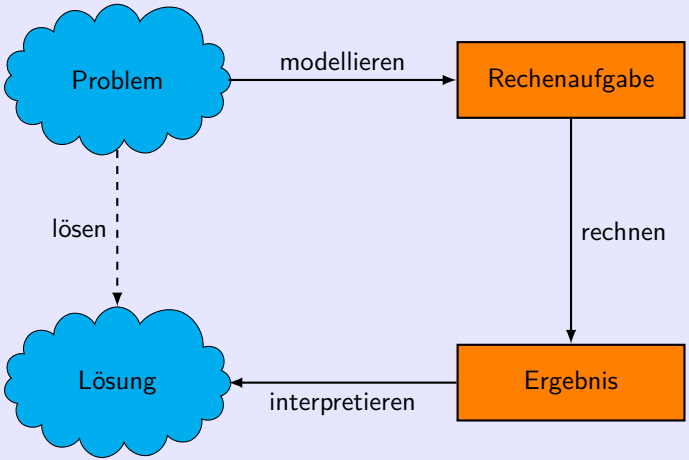
- Kreative gedankliche Leistung: arabisches Stellenwertsystem.
- Zum Vergleich: das römische System.

$$\begin{array}{rr} & \text{DCCCXV} \\ + & \text{MMMMDCCXI} \\ \hline = & ? \end{array}$$

2. Rechnen lassen

- Idee: das Rechnen Maschinen übertragen.
- Mit der Existenz von Rechenmaschinen wird es interessant, Aufgaben in Rechenaufgaben zu verwandeln, die es von Natur aus nicht sind.
- Es gilt, eine Problemstellung durch Formeln und Rechengesetze so weitgehend zu erfassen, dass wir dem Ergebnis der Rechnung eine Lösung des Problems entnehmen können.
- Informatiker*innen bilden abstrakte Modelle der konkreten Wirklichkeit.
- Die Wirklichkeitstreue dieser Modelle macht den Reiz und die Schwierigkeit dieser Aufgabe, und die Verantwortung der Informatiker*innen bei der Software-Entwicklung aus.

2. Modellbildung in der Informatik



2. Beispiel: Vom Problem zur Rechenaufgabe

Problem:

Die Klasse 2c macht einen Ausflug in den Zoo. Es fahren 27 Schülerinnen und Schüler und 3 Lehrer mit. Die Fahrtkosten betragen 2€ für Kinder und 3€ für Erwachsene. Kinder zahlen 5€ Eintritt und Erwachsene 10€. Wie teuer ist der Ausflug?

In der Grundschule lernt man, Textaufgaben in Rechenaufgaben zu verwandeln:

$$27 * (2 + 5) + 3 * (3 + 10)$$

2. Beispiel: Von der Rechenaufgabe zum Ergebnis

Das Ergebnis der Rechenaufgabe erhalten wir durch Ausrechnen:

$$\begin{aligned} 27 * (2 + 5) + 3 * (3 + 10) &= 27 * 7 + 3 * (3 + 10) \\ &= 189 + 3 * (3 + 10) \\ &= 189 + 3 * 13 \\ &= 189 + 39 \\ &= 228 \end{aligned}$$

2. Beispiel: Vom Ergebnis zur Lösung

$$27 * (2 + 5) + 3 * (3 + 10) = 228$$

Interpretieren wir das Ergebnis der Rechenaufgabe, erhalten wir die Lösung:

Lösung:

Der Ausflug kostet 228€.

3. Eine kleine Geschichte des Rechnens

- Historie des Rechnens
- Rechner und Rechnerräume

3. Geschichte — Euklid

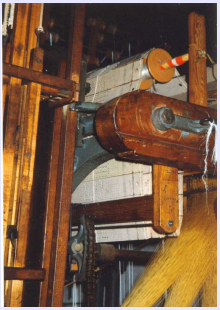
∼400–300 v.Chr.: Der griechische Mathematiker Euklid von Alexandria erfindet den ersten nicht-trivialen Algorithmus: Bestimmung des größten gemeinsamen Teilers zweier Zahlen.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{ggt}\ (m : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} = \\
\quad \textbf{\textit{if}}\quad m = 0\ \textbf{\textit{then}}\ n \\
\quad \textbf{\textit{elif}}\ n = 0\ \textbf{\textit{then}}\ m \\
\quad \textbf{\textit{elif}}\ m \geqslant n\ \textbf{\textit{then}}\ \mathit{ggt}\ (m \,\%\, n, n) \\
\quad \textbf{\textit{else}}\qquad\qquad\ \ \mathit{ggt}\ (m, n \,\%\, m)
\end{array}
$$

3. Geschichte — Abū ʻAbdallāh Muḥammad ibn Mūsā al-Khwārizmī

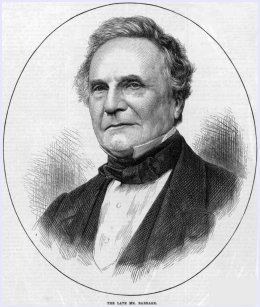
9. Jahrhundert: der Persische Mathematiker Abū ʻAbdallāh Muḥammad ibn Mūsā al-Khwārizmī macht das Dezimalsystem bekannt. Von seinem Namen leitet sich der Begriff *Algorithmus* ab.

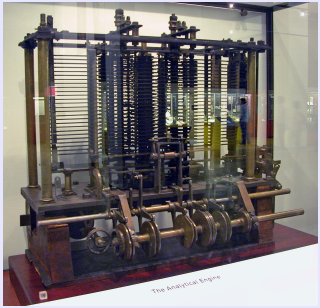
3. Geschichte — Joseph Jacquard



1801: Der Franzose Joseph Jacquard erfindet einen mechanischen Webstuhl, der mit Hilfe von Lochstreifen „programmiert“ werden kann.

3. Geschichte — Charles Babbage





1833: Der englische Mathematiker Charles Babbage konzipiert die “analytical engine”, den ersten Universalrechner. Babbage war seiner Zeit weit voraus: obwohl er keine funktionstüchtige Maschine fertigstellen konnte, gilt sein Konzept als Vorläufer des digitalen Rechners.

3. Geschichte — Ada Lovelace

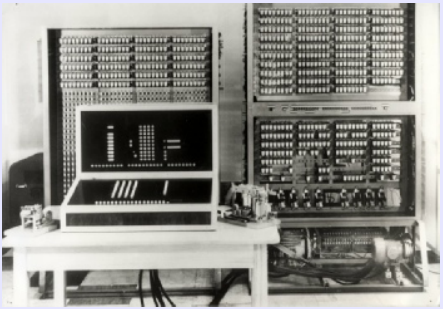
1843: Die englische Mathematikerin Ada Lovelace schreibt das erste Computerprogramm — das Programm berechnet die Bernoullizahlen auf der "analytical engine".

(Die Bernoullizahlen B_n ergeben sich als Lösungen der Gleichungen:

$$\begin{aligned} B_0 &= (1+B)^0 &&= 1 \\ B_2 &= (1+B)^2 &&= 1 \cdot B_0 + 2 \cdot B_1 + 1 \cdot B_2 \\ B_3 &= (1+B)^3 &&= 1 \cdot B_0 + 3 \cdot B_1 + 3 \cdot B_2 + 1 \cdot B_3 \\ B_4 &= (1+B)^4 &&= 1 \cdot B_0 + 4 \cdot B_1 + 6 \cdot B_2 + 4 \cdot B_3 + 1 \cdot B_4 \quad \ldots \end{aligned}$$

☞ Notationeller Trick: $B^n = B_n$.)

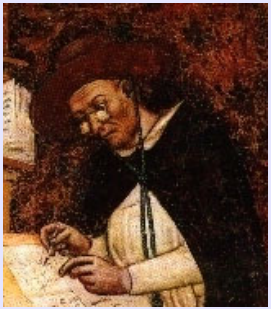
3. Geschichte — Konrad Zuse



1941: Der deutsche Ingenieur Konrad Zuse stellt die Z3 fertig, den ersten funktionsfähigen, frei programmierbaren, digitalen Rechner der Welt.

☞ Schauen wir uns weitere Rechner an ...

3. Rechner — Hugo von Saint-Cher



- Hugo von Saint-Cher (∼1200–1263) war ein französischer Mönch.
- Mit Hilfe der Mönche seines Ordens (∼500 Mönche), erstellte er die erste *Konkordanz* der Bibel.
- Eine Bibelkonkordanz ist ein alphabetisches Verzeichnis aller Wörter im Text der Bibel, mit Angabe der jeweiligen Fundstellen.
- (Parallelverarbeitung mit 500 Prozessoren!)

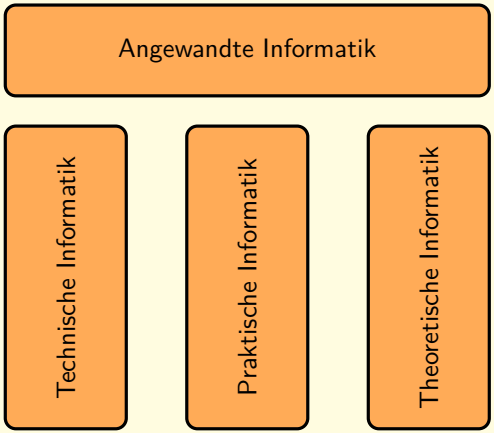
3. Rechner — Henrietta Swan Leavitt

- Henrietta Swan Leavitt (July 4, 1868–December 12, 1921) war eine US-amerikanische Astronomin.
- Sie arbeitete am Harvard College Observatory als *Computer*.
- Leavitt beobachtete und katalogisierte veränderliche Sterne.

3. Menschliche Computer

- … zeigt den „Computer Room“ der NACA (Vorgänger der NASA).
- Der Begriff „Computer“ ist seit dem 17. Jahrhundert im engl. Sprachraum in Gebrauch und bezeichnet „eine Person, die rechnet“.
- Bis ∼1950 war Computer ein Beruf wie etwa Kaufmann oder -frau.
- Gruppen von menschlichen Computern führten langwierige Berechnungen nach festgelegten Regeln durch; die Arbeit wurde aufgeteilt, so dass Berechnungen parallel erfolgen konnten.

4. Die Aufgabengebiete der Informatik



4. Technische Informatik

Die *Technische Informatik* kümmert sich um die Rechenmaschine selbst:

- Rechnerarchitektur und Rechnerentwurf,
- Speichermedien,
- Übertragungskanäle,
- Sensoren.

Ziel: immer schnellere und / oder kleinere Rechner bei fallenden Preisen.

4. Theoretische Informatik

Die *Theoretische Informatik* kümmert sich um die prinzipiellen Möglichkeiten und Grenzen des Rechnens:

- Was lässt sich berechnen? Was nicht?
- Nicht berechenbare Probleme, formal unentscheidbare Probleme!
- Wie schnell lassen sich bestimmte Aufgaben lösen?
- Laufzeit von Rechenverfahren.
- Komplexität von Problemen.

4. Praktische 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:

- Schließen der „semantischen Lücke“ durch Programmiersprachen auf immer höherer Abstraktionsstufe;
- Benutzung der Rechner: Betriebssysteme, Benutzungsoberflächen;
- Organisation großer Datenmengen: Datenbanken, Informationssysteme;
- Kommunikation der Rechner untereinander: Rechnernetze, Verteilte Systeme.

4. Angewandte Informatik

In der *Angewandten Informatik* kommt endlich der Zweck der ganzen Veranstaltung zum Zuge:

- Sie wendet
  - die schnellen Rechner,
  - die effizienten Algorithmen,
  - die höheren Programmiersprachen,
  - die Datenbanken,
  - die Rechnernetze usw.

  an, um Aufgaben jeglicher Art immer weiter und vollkommener zu lösen.
- Die Informatik dringt immer weiter in die verschiedensten Anwendungsgebiete vor.
- Obwohl die Informatik eine vergleichsweise junge Wissenschaft ist, haben sich interdisziplinäre Anwendungen als eigenständige Disziplinen von der „Kerninformatik“ abgespalten.
  - Wirtschaftsinformatik
  - Bioinformatik

5. Einordnung der Informatik in die Wissenschaftsfamilie

- Grundlagen der Informatik: aus der Mathematik;
- der reale Ausgangspunkt ihrer Entwicklung: die Konstruktion der ersten Rechner.

5. Einordnung der Informatik in die Wissenschaftsfamilie

- *Naturwissenschaften:* untersuchen Phänomene, die ihr ohne eigenes Zutun gegeben sind. Die Gesetze der Natur müssen entdeckt und erklärt werden.
- *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.

5. Einordnung der Informatik in die Wissenschaftsfamilie

☞ Die Informatik steht 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.

6. 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“.

6. Überblick über die Vorlesung — Mini–F#

Im Laufe der Vorlesung werden wir schrittweise unsere eigene Programmiersprache entwickeln:

Mini–F#

eine Sprache, in der sich Rechenregeln bequem und vor allem problemnah formulieren lassen.

6. Überblick über die Vorlesung

Die Einführung einer “eigenen” Programmiersprache 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?

☞Auf diese Weise wird die Programmiersprache selbst zum Objekt des Studiums. Wie gesagt, die Informatik steht den Ingenieurwissenschaften nahe.

6. Überblick über die Vorlesung — F#

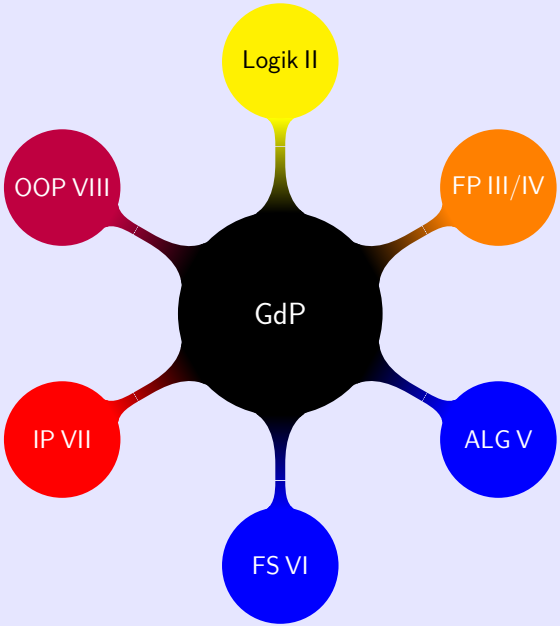
- Mini–F# ist — wie der Name andeutet — stark an die Programmiersprache F# angelehnt.
  - F# ist Mitglied der .NET Sprachfamilie (Visual Basic, C#).
  - F# ist eine Multiparadigmensprache: sie unterstützt funktionale, imperative und objektorientierte Programmierung.

No one's perfect.

- Unerreichtes Ideal: Mini–F# ist eine Teilmenge von F#.
- … gelegentlich vereinfachen wir etwas, sehr selten mogeln wir.
- Das kennen Sie aus der Schule: auch im Deutschunterricht wird nicht sofort die vollständige Grammatik der deutschen Sprache eingeführt.

6. Gliederung

I. Einführung⧹Rechnen und rechnen lassen
II. Grundlagen⧹Vor dem Rechnen
III. Werte⧹Elementares Rechnen
IV. Datentypen⧹Rechnen mit Daten
V. Algorithmik⧹Rechnen mit System
VI. Grammatiken⧹Konkrete Syntax
VII. Effekte⧹Effektvolles Rechnen
VIII. Objekte⧹Rechnen mit Objekten


Logik II
OOP VIII
FP III/IV
GdP
IP VII
ALG V
FS VI

6. Gastrollen ...

Lisa Lista ist die ideale Studentin: Sie ist unvoreingenommen und aufgeweckt, greift neue Gedanken auf und denkt sie in viele Richtungen weiter. Kein Schulunterricht in Informatik hat sie verdorben. Lisa Lista ist 12 Jahre alt.

Harry Hacker ist ein alter Programmierfuchs, nach dem Motto: Eine Woche Programmieren und Testen kann einen ganzen Nachmittag Nachdenken ersetzen. Er ist einfallsreich und ein gewitzter Beobachter; allerdings liegt er dabei oft haarscharf neben der Wahrheit. Zu Programmiersprachen hat er eine dezidiert subjektive Haltung: Er findet die Sprache am besten, die ihm am besten vertraut ist.

Teil II

Grundlagen

6. Knobelaufgabe #2

In einer dunklen Höhle leben Zwerge, die entweder eine weiße oder eine schwarze Mütze aufhaben. Einmal im Jahr dürfen sie die Höhle verlassen und bekommen eine Aufgabe gestellt. Lösen sie diese, sind sie frei. Misslingt die Lösung, müssen sie zurück in die Finsternis.

In diesem Jahr lautet die Aufgabe: Stellt euch nebeneinander so auf, dass die Zwerge mit einer weißen Mütze auf der linken Seite stehen und die mit schwarzer Mütze auf der rechten. Keiner der Zwerge kann die Farbe seiner eigenen Mütze sehen. Zudem dürfen die Zwerge weder miteinander reden noch sich auf sonstige Weise verständigen oder einander Hinweise geben, etwa mit der Hand oder den Augen. Auch Tricks wie die Verwendung von Spiegeln sind verboten.

Nicht verboten ist den Zwergen aber, ihren Verstand zu nutzen. Und in der Tat bekommen sie es auf Anhieb hin, sich nach der Farbe der Mützen getrennt aufzustellen. Wie haben Sie das bloß angestellt?

6. Motivation

In der Vorlesung werden wir nicht nur lernen

- *mit* der Programmiersprache Mini–F# Probleme zu lösen,

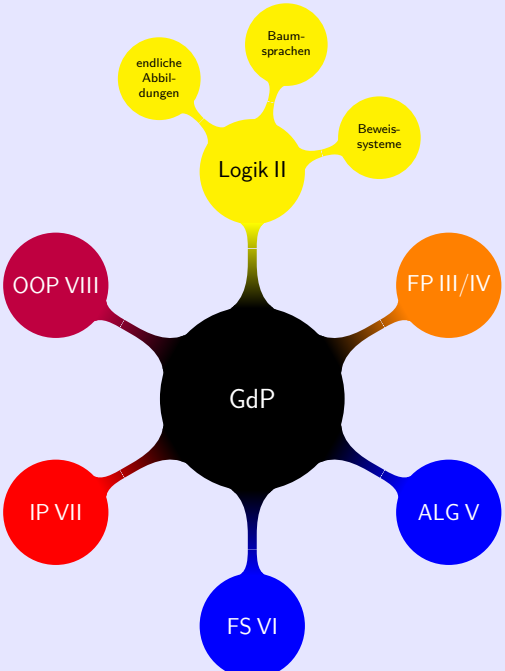
sondern auch

- *über* die Programmiersprache Mini–F# zu reden.

Mini–F# ist sowohl Subjekt als auch Objekt des Studiums.

Dazu benötigen wir ein wenig mathematisches Rüstzeug ...

6. Gliederung

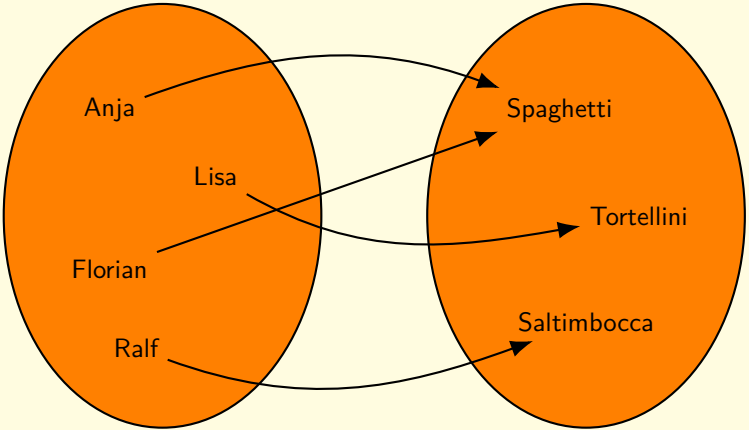

endliche Abbil-dungen
Baum-sprachen
Beweis-systeme
Logik II
OOP VIII
FP III/IV
GdP
IP VII
ALG V
FS VI

6. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- mit endlichen Abbildungen und Sequenzen umgehen können,
- die Begriffe Syntax und Semantik definieren können,
- den Unterschied zwischen konkreter und abstrakter Syntax kennen,
- Baumsprachen lesen und selbst definieren können,
- Beweissysteme lesen und selbst definieren können.

7. Endliche Abbildungen



7. Endliche Abbildungen

$$\{ \begin{array}{ll} \mathit{Anja} & \mapsto \mathit{Spaghetti}, \\ \mathit{Lisa} & \mapsto \mathit{Tortellini}, \\ \mathit{Florian} & \mapsto \mathit{Spaghetti}, \\ \mathit{Ralf} & \mapsto \mathit{Saltimbocca} \} \end{array}$$

7. Notation endlicher Abbildungen

- 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 *Definitionsbereich* von φ. Der Definitionsbereich $\mathit{dom}\ \varphi$ muss endlich sein.
- Die Anwendung einer endlichen Abbildung φ auf ein Element x notieren wir mit $\varphi(x)$.

Beispiel:

$$\varphi = \{\mathit{Anja} \mapsto \mathit{Spaghetti}, \mathit{Lisa} \mapsto \mathit{Tortellini}, \mathit{Florian} \mapsto \mathit{Spaghetti}, \mathit{Ralf} \mapsto \mathit{Saltimbocca}\}$$

$$\mathit{dom}\ \varphi = \{\mathit{Anja}, \mathit{Lisa}, \mathit{Florian}, \mathit{Ralf}\}$$

$$\varphi(\mathit{Anja}) = \mathit{Spaghetti}$$
$$\varphi(\mathit{Lisa}) = \mathit{Tortellini}$$

7. Konstruktion und Manipulation endlicher Abbildungen

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

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

7. Eigenschaften endlicher Abbildungen

- Der Kommaoperator ist assoziativ:

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

Statt $(\varphi_1, \varphi_2), \varphi_3$ schreiben wir kurz $\varphi_1, \varphi_2, \varphi_3$. Zusätzlich verwenden wir $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ als Abkürzung für $\{x_1 \mapsto y_1\}, \ldots, \{x_n \mapsto y_n\}$ (wiederholte Anwendung des Kommaoperators).

- Der Kommaoperator ist aber *nicht* kommutativ: φ_1, φ_2 ist in der Regel verschieden von φ_2, φ_1.

$$\{\mathit{Ralf} \mapsto \mathit{Pizza}\}, \{\mathit{Ralf} \mapsto \mathit{Saltimbocca}\} = \{\mathit{Ralf} \mapsto \mathit{Saltimbocca}\}$$
$$\{\mathit{Ralf} \mapsto \mathit{Saltimbocca}\}, \{\mathit{Ralf} \mapsto \mathit{Pizza}\} = \{\mathit{Ralf} \mapsto \mathit{Pizza}\}$$

☞ das zweite Argument des Kommaoperators hat Vorrang.

7. Sequenzen

Sequenzen, endliche Folgen von Elementen, lassen sich mit Hilfe endlicher Abbildungen modellieren.

Anja Florian Lisa Ralf

Die Elemente werden von links nach rechts beginnend mit 0 durchnummeriert.

$\{0 \mapsto \textit{Anja}, 1 \mapsto \textit{Florian}, 2 \mapsto \textit{Lisa}, 3 \mapsto \textit{Ralf}\}$

7. Notation von Sequenzen

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

7. Konstruktion und Manipulation von Sequenzen

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

- die *leere Sequenz* ϵ mit
  - $\mathit{dom}\ \epsilon = \mathbb{N}_0$;
- 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
  - $\mathit{dom}\ (s_1 \cdot s_2) = \mathit{dom}\ s_1 \cup \{i + \mathit{len}\ s_1 \mid i \in \mathit{dom}\ s_2\}$ und
  - $(s_1 \cdot s_2)(i) = \begin{cases} s_1(i) & \text{falls } i \in \mathit{dom}\ s_1 \\ s_2(i - \mathit{len}\ s_1) & \text{sonst} \end{cases}$
- die *n-fache Wiederholung* von s notiert s^n mit $s^0 = \epsilon$ und $s^{n+1} = s \cdot s^n$.

8. Syntax und Semantik

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

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

8. Lexikalische Syntax

Betrachten wir ein einfaches Beispielprogramm:

```
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.

8. Lexikalische Syntax: Lexeme

Als menschlicher Leser sind wir gewohnt — bzw. durch jahrelanges Training geschult — mehrere Zeichen zu einer Einheit zusammenzufassen.

4711	*	(	a11	+	815	)

☞ Nicht alle Zeichen sind für den Rechner gedacht: `(* speed *)` ist ein Kommentar, der sich an den menschlichen Leser richtet.

In der *lexikalischen Syntax* einer Programmiersprache wird festgelegt, wie Zeichen zu größeren Einheiten, sogenannten *Lexemen* (engl. tokens), zusammengefasst werden.

8. Kontextfreie Syntax

Nicht alle Folgen von Lexemen stellen ein gültiges Programm dar:

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

umfasst die gleichen Lexeme, ist aber *kein* Mini–F# Programm.

In der *kontextfreien Syntax* einer Programmiersprache wird festgelegt, welche Folgen von Lexemen gültige Programme sind und welche nicht.

8. Konkrete Syntax

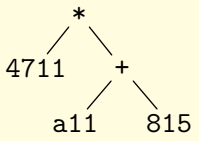
Die lexikalische und die kontextfreie Syntax bilden zusammen die *konkrete Syntax* einer Programmiersprache.

Will man eine Programmiersprache selbst zum Objekt des Studiums machen, dann ist die konkrete Syntax als Ausgangspunkt ungeeignet:

- sie ist technischen Einschränkungen unterworfen,
- sie ist das Endprodukt vieler Kompromisse und
- sie spiegelt den persönlichen Geschmack der Sprachdesigner wider.

8. Abstrakte Syntax

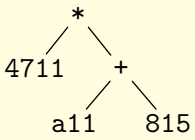
Für das Studium von Programmiersprachen ist die hierarchische Struktur eines Programms relevant.



☞ Die hierarchische Struktur verdeutlicht, dass sich der Ausdruck aus mehreren Teilausdrücken zusammensetzt. Man spricht auch von einem *Rechenbaum* oder allgemein von einem *abstrakten Syntaxbaum*.

8. Konkrete versus abstrakte Syntax

Abstrakte Syntax:



Konkrete Syntax in

- Mini–F#: `4711 * (a11 + 815)`,
- Scheme: `(* 4711 (+ a11 815))`,
- PostScript: `4711 a11 815 + *`.

8. Semantik

- Die Semantik legt die Bedeutung von Programmen fest.
- Die Bedeutung eines arithmetischen Ausdrucks ist eine Zahl.
- Die Semantik lässt sich mit Auswertungsregeln beschreiben:
  - 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 eng an der Struktur eines Programms — die Bedeutung eines Ausdrucks wird auf die Bedeutung der Teilausdrücke zurückgeführt (kompositional).

9. Baumsprachen

Den hierarchischen Aufbau von Programmen, die abstrakte Syntax, beschreiben wir mit *Baumsprachen*.

Beispiel: eine einfache Form von arithmetischen Ausdrücken.

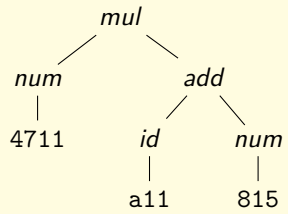
$$
\begin{array}{rcl}
e \in \mathsf{Expr} ::= & \mathit{num} & (\mathbb{N}) \\
\mid & \mathit{id} & (\mathsf{Id}) \\
\mid & \mathit{add} & (\mathsf{Expr}, \mathsf{Expr}) \\
\mid & \mathit{mul} & (\mathsf{Expr}, \mathsf{Expr})
\end{array}
$$

9. Baumsprachen: Beispiel

Beispiel für einen arithmetischen Ausdruck:

$$mul\ (num\ (\texttt{4711}), add\ (id\ (\texttt{a11}), num\ (\texttt{815})))$$

Dargestellt als Baum:



9. Baumsprachen: Aufbau

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

Die Definition führt drei verschiedene Dinge ein:

- einen Namen für die Baumsprache: Expr,
- Namen für die unterschiedlichen Knotenarten: *num*, *id*, *add* und *mul*, und
- eine *Metavariable* für Elemente der Baumsprache: e.

Die griechische Vorsilbe meta (μετά) bedeutet unter anderem „über“. Eine Metavariable ist Bestandteil der Metasprache, mit der wir *über* die Programmiersprache Mini–F# reden. Metavariablen sind von Variablen zu unterscheiden, die Bestandteil der Programmiersprache Mini–F# sind (und erst etwas später eingeführt werden).

9. Baumsprachen: umgangssprachliche Lesart

Umgangssprachlich lässt sich die Definition wie folgt lesen: ein Element $e \in \text{Expr}$ ist entweder

- von der Form $\mathit{num}\ (n)$ mit $n \in \mathbb{N}$, oder
- von der Form $\mathit{id}\ (x)$ mit $x \in \text{Id}$, oder
- von der Form $\mathit{add}\ (e_1, e_2)$ mit $e_1, e_2 \in \text{Expr}$ oder
- von der Form $\mathit{mul}\ (e_1, e_2)$ mit $e_1, e_2 \in \text{Expr}$.

☞ Wir verwenden die Metavariable e, um arithmetische Ausdrücke zu benennen (gegebenenfalls mit einem Index versehen).

9. Baumsprachen: formale Lesart

Formal ist die Menge Expr durch eine *induktive Definition* gegeben.

Die Menge Expr ist die *kleinste* Menge mit den folgenden Eigenschaften:

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

☞ 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.

9. Notationelle Freiheiten

$n \in \mathbb{N}$
$x \in \mathsf{Id}$
$e \in \mathsf{Expr} ::= n$
$\quad\quad\quad\quad\quad\quad \mid\ x$
$\quad\quad\quad\quad\quad\quad \mid\ e_1 + e_2$
$\quad\quad\quad\quad\quad\quad \mid\ e_1 * e_2$

☞ Der Unterschied zur ursprünglichen Definition ist nur ein äußerlicher: statt *mul* (*num* (4711), *add* (*id* (a11), *num* (815))) schreiben wir kurz $4711 * (\mathsf{a11} + 815)$, gemeint ist aber stets der gleiche abstrakte Syntax*baum*.

9. Beispiel: Klötzchenwelt — Baumsprachen

Ein weiteres Beispiel für eine Baumsprache: Folgen von Klötzchen.

$$
\begin{aligned}
w \in \text{Wall} ::= \; & \textit{yellow-brick} \\
\mid \; & \textit{red-brick} \\
\mid \; & \textit{sequ}\,(\text{Wall}, \text{Wall})
\end{aligned}
$$

Die gleiche Definition mit notationellen Freiheiten:

$w \in \text{Wall} ::=$ 
$\mid$ 
$\mid \; w_1 \; w_2$

10. Beweissysteme

Bei der umgangssprachlichen Beschreibung der Bedeutung arithmetischer Ausdrücke haben wir wenn-dann Aussagen formuliert:

- *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.

Wenn-dann Aussagen lassen sich mit *Beweisregeln* formalisieren.

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

Die Formulierung „wertet aus zu“ symbolisieren wir durch ‘$\Downarrow$’. Über dem Strich der Regeln stehen die Voraussetzungen (der „wenn“ Teil); unter dem Strich die Schlussfolgerung (der „dann“ Teil).

10. Beweisbäume

Beweisregeln:

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

Beweisregeln können zu *Beweisbäumen* zusammengesetzt werden.

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

10. Beweisbäume

Ohne Auslassungen:

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

10. Notationelle Freiheiten — da capo

Erlauben wir notationelle Freiheiten, gerät der Beweisbaum weniger mächtig.

$$\dfrac{\dfrac{}{4711 \Downarrow 4711} \qquad \dfrac{\dfrac{\vdots}{a11 \Downarrow 1} \qquad \dfrac{}{815 \Downarrow 815}}{a11 + 815 \Downarrow 816}}{4711 * (a11 + 815) \Downarrow 3844176}$$

10. Beweissysteme

Ist eine 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 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.

10. Beweisbäume

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

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

eine Beweisregel, dann ist

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

ein Beweisbaum mit der Wurzel ϕ.

Ist $\mathcal{P}$ ein Beweisbaum mit der Wurzel ϕ, dann sagt man auch $\mathcal{P}$ zeigt oder beweist ϕ.

10. Regelschemata

Die obigen Beweisregeln für die Auswertung arithmetischer Ausdrücke 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 — 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 *Regelschemata*.

10. Regelinstanzen

Bevor Regelschemata zu Beweisbäumen zusammengesetzt werden können, müssen die Metavariablen erst durch konkrete Konstanten bzw. Ausdrücke ersetzt werden.

Für unser laufendes Beispiel benötigen wir:

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

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

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

Das Ergebnis einer solchen Ersetzung nennt man auch *Regelinstanz* oder kurz *Instanz*.

Mit diesen Instanzen können wir wieder den Beweisbaum zusammensetzen.

10. Notationelle Freiheiten — da capo

Wir nehmen uns bei der Definition der abstrakten Syntax einige Freiheiten heraus.

Dies birgt die Gefahr von Verwechslungen.

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

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.

☞ '+' ist Syntax und '$+$' ist Semantik.

10. Beispiel: Klötzchenwelt — Beweisregeln

Formeln: $n \sim w$. *Lies:* w ist eine schöne Mauer der Ordnung n.

Beweisregeln:

$$\frac{}{0 \sim \textit{yellow-brick}} \qquad \frac{}{1 \sim \textit{red-brick}}$$

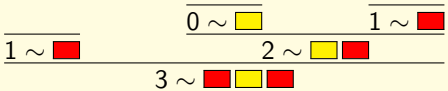
$$\frac{n \sim w_1 \qquad n+1 \sim w_2}{n+2 \sim \textit{sequ}\ (w_1, w_2)}$$

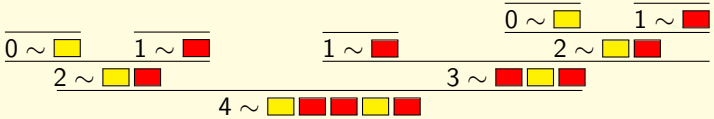
Beweisregeln mit notationellen Freiheiten:

$$\frac{}{0 \sim \square} \qquad \frac{}{1 \sim \blacksquare}$$

$$\frac{n \sim w_1 \qquad n+1 \sim w_2}{n+2 \sim w_1\, w_2}$$

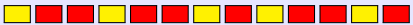
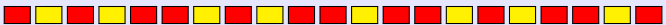
10. Beispiel: Klötzchenwelt — Beweisbäume





10. Beispiel: Klötzchenwelt — ableitbare Formeln

Ableitbare oder beweisbare Formeln:

- 0 ∼ 
- 1 ∼ 
- 2 ∼ 
- 3 ∼ 
- 4 ∼ 
- 5 ∼ 
- 6 ∼ 
- 7 ∼ 
- ...

10. Beispiel: Klötzchenwelt—nicht ableitbare Formeln

Ableitbare oder beweisbare Formeln:

- $0 \sim$ 
- $1 \sim$ 
- $2 \sim$ 
- $3 \sim$ 
- $4 \sim$ 
- ...

Eigenschaften:

- Für $n > 0$ enden die Reihen in einem roten Klötzchen.
- Es folgen nie zwei gelbe Klötzchen aufeinander.
- Es folgen nie mehr als zwei rote Klötzchen direkt hintereinander.

☞ Die Formeln $n \sim$  und $n \sim$  sind *nicht* beweisbar.

10. Zusammenfassung

Wir haben

- Notation eingeführt, um endliche Abbildungen und Sequenzen zu konstruieren und zu manipulieren,
- den Unterschied zwischen konkreter und abstrakter Syntax kennengelernt,
- Baumsprachen zur Definition abstrakter Syntax eingeführt und
- Beweissysteme zur Definition der Semantik.

Wann geht's denn endlich mit dem Programmieren los?

Nächste Woche, Harry.

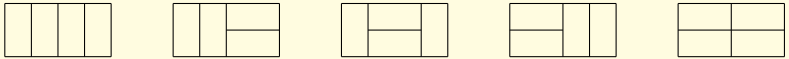
Teil III

Werte

10. Knobelaufgabe #3

Bob der Baumeister will eine 100 m breite und 2 m hohe Mauer bauen. Als Baumaterial hat er 2 m breite und 1 m hohe Quader zur Verfügung.

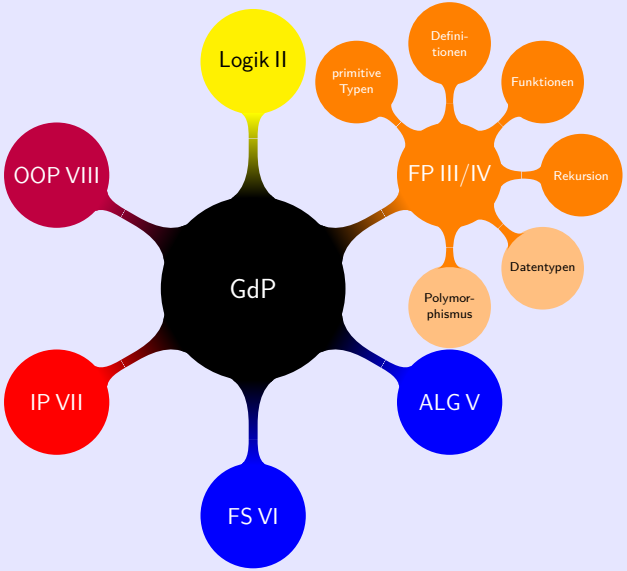
Wieviele Möglichkeiten gibt es die Mauer zu konstruieren?



Eine 4 m breite Mauer lässt sich zum Beispiel auf 5 Arten zusammensetzen.

Wie erhöht sich die Zahl der Möglichkeiten, wenn die Mauer statt 2 m sogar 3 m hoch werden soll?

10. Gliederung



10. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- den Unterschied zwischen statischer und dynamischer Semantik kennen,
- Sinn und Zweck von Typsystemen verstanden haben,
- das Konzept von Bindungen verstanden haben,
- Funktionen und Rekursion verstanden haben,
- Entwurfsmuster kennen und anwenden können,
- Mini–F# Programme lesen und selbst schreiben können.

10. 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 (1980)

10. Growing a language ...

I think you know what a man is. A woman is more or less like a man, but not of the same sex. (This may seem like a strange thing for me to start with, but soon you will see why.)

Next, I shall say that a person is a woman or a man (young or old).

To keep things short, when I say "he" I mean "he or she", and when I say "his" I mean "his or her."

A machine is a thing that can do a task with no help, or not much help, from a person.

— Growing a Language, Guy L. Steele Jr.

10. Struktur der folgenden Abschnitte

- 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

10. Statische Semantik — Sinn und Zweck

- Ein Programm in Mini–F# ist ein *Ausdruck*:
  - $4711 + 815$
  - `"Hello, world!"`
- Ausdrücke sind beliebig kombinierbar.
- Nicht alle Kombinationen machen jedoch Sinn:
  - `"Hello, world!"` $* 4711$
- Die *statische Semantik* fängt diese sinnlosen Ausdrücke ab.
- Zu diesem Zweck werden Ausdrücke mit Hilfe von *Typen* in Schubladen eingeteilt:
  - `"Hello, world!"` hat den Typ *String*
  - 4711 hat den Typ *Nat*
- Die Multiplikation arbeitet auf den natürlichen Zahlen: `"Hello, world!"` $* 4711$ ist nicht wohlgetypt.

10. Statische Semantik — Typregeln

Die *statische Semantik* legt fest, dass die Multiplikation zwei natürliche Zahlen nimmt 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 solche *Typregel* angegeben. Diese Beweisregeln spezifizieren die zweistellige Relation

$$e : t$$

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

10. Statische Semantik — Begriffe

- Die statische Semantik hat eine ordnende Funktion: fast alle Abschnitte führen *einen* neuen Typ 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.
- Wohlgetypte Ausdrücke können ausgerechnet werden.

10. Dynamische Semantik — Auswertungsregeln

Wie ein Ausdruck ausgerechnet wird, legt die *dynamische Semantik* fest. Für die Multiplikation: beide Argumente werden ausgerechnet, das Ergebnis ist das Produkt der Teilergebnisse.

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

Für jedes Konstrukt wird mindestens eine *Auswertungsregel* angegeben. Diese Regeln spezifizieren die zweistellige Relation

$$e \Downarrow \nu$$

zwischen Ausdrücken und Werten. *Lies:* „e wertet zu ν aus“.

10. Dynamische Semantik — Werte

- Was ist ein Wert?
- Ein Wert ist das Ergebnis eines Programms; der Wert eines arithmetischen Ausdrucks ist zum Beispiel eine natürliche Zahl.
- Mit jedem neu eingeführten Typ werden wir auch den Bereich der Werte erweitern.
- Ein Typ ist im Prinzip die Menge aller zugehörigen Werte.

10. Dynamische Semantik — Beweisbäume

Ein Programm wird ausgerechnet, indem die Auswertungsregeln für die Teilausdrücke zu einem Beweisbaum kombiniert werden.

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

Konstanten wie 4711 oder 815 werten zu sich selbst aus — Konstanten *sind* Werte.

11. Motivation

Nicht-triviale Programme treffen viele Entscheidungen. Im einfachsten Fall wird geprüft, ob ein bestimmter Sachverhalt wahr oder falsch ist:

- Ist das Konto überzogen?
- Ist Florian größer als Lisa?

Das Ergebnis einer solchen Überprüfung repräsentieren wir durch einen Wahrheitswert: *true* oder *false*.

11. Motivation

In Abhängigkeit von einem Wahrheitswert kann die Rechnung dann einen bestimmten Verlauf nehmen.

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

Wertet der Ausdruck e_1, die Bedingung, zu *true* aus, dann wird mit der Auswertung von e_2 fortgefahren; wertet e_1 zu *false* aus, dann wird e_3 ausgewertet.

11. Abstrakte Syntax

$e \in \text{Expr} ::=$		*Boolesche Ausdrücke:*
	false	falsch
|	*true*	wahr
|	***if*** e_1 ***then*** e_2 ***else*** e_3	Alternative

Der Teilausdruck e_1 der Alternative heißt *Bedingung*; die Teilausdrücke e_2 und e_3 heißen *Zweige* der Alternative.

11. Statische Semantik

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

Typregeln:

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

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

☞ Die Bedingung muss vom Typ *Bool* sein; die Zweige der Alternative können einen beliebigen Typ besitzen; dieser ist auch der Typ des gesamten Ausdrucks.

11. Dynamische Semantik

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

Auswertungsregeln:

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

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

11. Beispielrechnung

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

☞ Alternativen dürfen beliebig geschachtelt werden: die Bedingung der äußeren Alternative ist wiederum eine Alternative.

11. Demo

```
Mini⟩ false
false
Mini⟩ true
true
Mini⟩ if false then "yes" else "no"
"no"
Mini⟩ if true then "yes" else "no"
"yes"
Mini⟩ if 4711 < 815 then "yes" else "no"
"no"
Mini⟩ if (if 4711 < 815 then false else true) then "yes" else "no"
"yes"
Mini⟩ if 4711 < 815 then true else 7 > 1
true
Mini⟩ if 4711 < 815 then 7 > 1 else false
false
```

11. Vertiefung

Ein Boolescher Ausdruck modelliert einen Sachverhalt oder eine Aussage.

Wir sind gewohnt, einfache Aussagen zu komplexen Aussagen zusammenzusetzen:

- *Negation*: Der Kunde ist *nicht* kreditwürdig.
- *Konjunktion*: Das Konto ist überzogen *und* der Kunde ist nicht kreditwürdig.
- *Disjunktion*: Das Netzteil ist defekt *oder* die Leitung ist unterbrochen.

11. Vertiefung

- *Negation* von e:

  if e ***then*** *false* ***else*** *true*

- *Konjunktion* von e_1 und e_2:

  if e_1 ***then*** e_2 ***else*** *false*

- *Disjunktion* von e_1 und e_2:

  if e_1 ***then*** *true* ***else*** e_2

☞ Wir kürzen die Negation von e mit *not* e, die Konjunktion von e_1 und e_2 mit e_1 && e_2 und die Disjunktion mit e_1 || e_2 ab.

11. George Boole (1815–1864)

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,	$\therefore y = 1$

12. Motivation

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

- Den Begriff Rechnen werden die meisten mit Zahlen in Verbindung bringen.
- In diesem Abschnitt führen wir einige elementare Konstrukte zum Rechnen mit *natürlichen* Zahlen ein.

12. Abstrakte Syntax

$n \in \mathbb{N} ::= 0 \mid 1 \mid 2 \mid \cdots$	*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

☞ Die Notation $e ::= \cdots$ soll andeuten, dass wir die Kategorie der Ausdrücke um arithmetische Operatoren und Vergleichsoperatoren *erweitern*.

12. Statische Semantik

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

Typregeln:

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

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

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

12. Dynamische Semantik

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

Auswertungsregeln:

$$\frac{}{n \Downarrow n}$$

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

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

12. Dynamische Semantik

Ist doch alles ziemlich offensichtlich, oder? Ich meine, es ist doch klar, dass '$*$' beide Argumente ausrechnet und die Ergebnisse dann multipliziert.

Was ist denn, wenn eins der Argumente 0 ist?

Na ja, dann ist das Ergebnis halt auch 0. Ich sehe nicht, worauf Du hinaus willst.

Wenn das erste Argument schon 0 ist, müssen wir das zweite Argument ja gar nicht mehr ausrechnen!

$$\frac{e_1 \Downarrow 0}{e_1 * e_2 \Downarrow 0}$$

Ok, ist vielleicht schneller, aber 0 kommt immer noch raus.

12. Dynamische Semantik — Subtraktion

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

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

☞ Mini–F# arbeitet auf den natürlichen und nicht auf den ganzen Zahlen. (F# kennt von Haus aus keine natürlichen Zahlen, dafür aber ganze Zahlen und Fließkommazahlen. In Kürze mehr dazu.)

Der Operator $\dot{-}$ hört auch auf den Namen „monus“.

12. Dynamische Semantik — Division

Die Operatoren '$\div$' 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 Nebenbedingungen legen die Metavariablen $q, r \in \mathbb{N}$ eindeutig fest.

☞ 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 $b > 0$.

12. Dynamische Semantik — Division

- Die Ausdrücke $e \div 0$ und $e \,\%\, 0$ sind undefiniert; die dynamische Semantik ordnet ihnen keinen Wert zu: es gibt kein r mit $r < 0$.
- Das ist unbefriedigend — die statische Semantik sollte ja gerade derartige Programme herausfiltern.
- Eine Lösung für dieses Problem stellen wir erst *sehr* viel später vor (in Teil VII).

12. Dynamische Semantik

Ich habe noch mal über die Multiplikation nachgedacht. Wenn wir die Regel

$$\frac{e_1 \Downarrow 0}{e_1 * e_2 \Downarrow 0}$$

hinzunehmen, dann können wir Programme auswerten, die sonst keinen Wert haben.

Ähem, wie meinst Du das? Die statische Semantik, oder wie das heißt, lässt sowas doch gar nicht zu.

Hast Du nicht aufgepasst ;-)? Der Ausdruck $0 \div 0$ hat zum Beispiel keinen Wert.

Ja ...

Na, dann hat $0 * (0 \div 0)$ auch keinen Wert. Aber *mit* meiner Regel können wir den Ausdruck zu 0 auswerten.

12. Dynamische Semantik — Vergleichsoperatoren

Für jeden Vergleichsoperator gibt es zwei Auswertungsregeln.

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

usw.

12. Demo

Mini⟩ $4711 * 2 + 815$
10237
Mini⟩ $11 * 11$
121
Mini⟩ $111 * 111$
12321
Mini⟩ $111111111 * 111111111$
12345678987654321

☞ Die Ergebnisse sind stets exakt. (Die Genauigkeit ist *nicht* auf die native Genauigkeit von Rechnern, 32 oder 64 Bit, eingeschränkt.)

12. Über den Tellerrand: F#

F# kennt eine Reihe verschiedener Zahlentypen (*Nat* ist leider nicht vordefiniert):

- *byte*: natürliche Zahlen von 0 bis $2^8 - 1 = 255$;
- *sbyte*: ganze Zahlen von $-2^7 = -128$ bis $2^7 - 1 = 127$ (signed byte);
- *int16*: ganze Zahlen von $-2^{15} = -32768$ bis $2^{15} - 1 = 32767$;
- *uint16*: natürliche Zahlen von 0 bis $2^{16} - 1 = 65535$ (unsigned int16);
- *int* und *uint32*: ditto mit 32 Bit;
- *int64* und *uint64*: ditto mit 64 Bit;
- *float32*: 32-bit Fließkommazahlen;
- *float*: 64-bit Fließkommazahlen;
- *bigint*: ganze Zahlen.

☞ Rechnen mit beschränkter Genauigkeit hat seine Tücken: $255uy + 1uy = 0uy$ und $2147483647 + 1 = -2147483648$.

13. Knobelaufgabe #4

Ein Ausdruck besteht aus einer Folge von Zeichen; ein Ausdruck vom Typ *String* wertet zu einer Zeichenfolge aus. Schreiben Sie ein Programm, das zu seinem eigenen Programmtext auswertet.

Easy!

```
Mini⟩ "done"
"done"
```

Nicht ganz Harry: das Programm `"done"` besteht aus 6 Zeichen, sein Wert aus 4.

??? … Ach so, die Anführungsstriche!!! Schnell repariert:

```
Mini⟩ show "done"
"\"done\""
```

Ich sehe, Du hast die Funktion *show* entdeckt, die einen Wert in seine externe Darstellung überführt. Jetzt besteht Dein Programm aus 11 Zeichen, der Wert aus 6.
Ein Tipp: wenn Du nach der Auswertung des Ausdrucks *putline it* eintippst, musst Du Zeichen für Zeichen den Programmtext erhalten.

13. Motivation

- Rechnungen sind in der Regel nicht linear, sie enthalten Zwischen- oder Hilfsrechnungen.
- Beispiel: Berechnung des Flächeninhalts eines Quadrats:
  - zunächst: Seitenlänge ausrechnen,
  - dann: Ergebnis der Zwischenrechnung mit sich selbst multiplizieren.
- Um das Ergebnis einer Zwischenrechnung gegebenenfalls mehrfach verwenden zu können, geben wir ihm einen Namen :

$$\textbf{\textit{let}}\ s = 4711 + 815$$

Das Konstrukt ist eine *Wertedefinition*: der Bezeichner links wird an den Wert des Ausdrucks rechts gebunden. *Lies:* sei s gleich $4711 + 815$.

- Ein Bezeichner ist ein Ausdruck und somit auch:

$$s * s$$

Der obige Ausdruck berechnet den gewünschten Flächeninhalt.

13. Motivation

- Welcher Bezeichner kann wo verwendet werden?
- Wie werden Wertedefinitionen und Ausdrücke verknüpft?
- Der Zusammenhang wird durch einen ***in***-Ausdruck hergestellt.

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

  - Vor ***in*** steht eine Wertedefinition.
  - Nach ***in*** steht ein Ausdruck.
  - Das gesamte Konstrukt ist wiederum ein Ausdruck.
  - Der Bezeichner s ist nur in $s * s$ *sichtbar*.

13. Beispiel: Ausflug in den Zoo

Problem:

Die Klasse 2c macht einen Ausflug in den Zoo. Es fahren 27 Schülerinnen und Schüler und 3 Lehrer mit. Die Fahrtkosten betragen 2€ für Kinder und 3€ für Erwachsene. Kinder zahlen 5€ Eintritt und Erwachsene 10€. Wie teuer ist der Ausflug?

Berechnung der Kosten:

$$
\begin{array}{lll}
\textbf{\textit{let}}\ \mathit{Schüler} & = 27 & \textbf{\textit{in}} \\
\textbf{\textit{let}}\ \mathit{Lehrer} & = 3 & \textbf{\textit{in}} \\
\textbf{\textit{let}}\ \mathit{Fahrtkosten} & = 2 * \mathit{Schüler} + 3 * \mathit{Lehrer} & \textbf{\textit{in}} \\
\textbf{\textit{let}}\ \mathit{Eintritt} & = 5 * \mathit{Schüler} + 10 * \mathit{Lehrer} & \textbf{\textit{in}} \\
\mathit{Fahrtkosten} + \mathit{Eintritt} & &
\end{array}
$$

☞ Benennung von Teilrechnungen.

13. Wahl der Bezeichner

- Der Bezeichner in einer Wertedefinition kann frei gewählt werden:

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

  ist gleichwertig zu

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

- Für das Ausrechnen spielen Namen keine Rolle, wohl aber für den menschlichen Betrachter eines Programms. *Deshalb:* möglichst aussagekräftige Bezeichner vergeben.
- *Allgemein gilt:* je größer der Sichtbarkeitsbereich eines Namens, desto mehr Sorgfalt sollte man bei der Namenswahl walten lassen.

13. Abstrakte Syntax — Definitionen

Wir führen eine neue syntaktische Kategorie ein: *Deklarationen*.

$x \in \mathsf{Id}$	*Bezeichner*
$d \in \mathsf{Decl} ::=$	*Deklarationen:*
$\mid$ ***let*** $x = e$	Wertedefinition

☞ Der Bereich der Bezeichner Id wird in Teil VI genau festgelegt.

Für's erste: ein Bezeichner fängt mit einem Buchstaben an. Danach können weitere Buchstaben, Ziffern, und Sonderzeichen wie ein Unterstrich oder ein Apostroph folgen.

13. Abstrakte Syntax — Ausdrücke

Ein ***in***-Ausdruck verknüpft eine Definition mit einem Ausdruck.

$e ::= \cdots$	*lokale Definitionen:*
$\mid\ x$	Bezeichner
$\mid\ d$ ***in*** e	lokale Definition

Der Teilausdruck e heißt Rumpf des ***in***-Ausdrucks.

13. Statische Semantik

- Wie werden ***in***-Ausdrücke typisiert?

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

  Wenn wir den Teilausdruck $s * s$ typisieren, woher kennen wir den Typ von s?
- Ein Bezeichner kann in unterschiedlichen Kontexten einen unterschiedlichen Typ besitzen.

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

13. Statische Semantik — Signatur

- Wir merken uns die Typen von Bezeichnern mit Hilfe einer sogenannten Signatur.

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

Eine Signatur ist eine endliche Abbildung von Bezeichnern auf Typen.

- Wir erweitern die Typregeln um Signaturen. Die Regeln spezifizieren die nunmehr *dreistellige* Relation

$$\Sigma \vdash e : t$$

zwischen Signaturen, Ausdrücken und Typen. *Lies:* „bezüglich der Signatur Σ hat e den Typ t“.

13. Statische Semantik — Definitionen

Die statische Semantik ordnet

- einem Ausdruck einen Typ und
- einer Definition eine Signatur

zu.

Typregel:

$$\frac{\Sigma \vdash e : t}{\Sigma \vdash (\textbf{\textit{let}}\ x = e) : \{x \mapsto t\}}$$

☞ Eine Signatur repräsentiert — ähnlich wie ein Typ — das, was wir über eine Definition statisch wissen.

13. Statische Semantik — Ausdrücke

Typregeln:

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

☞ Zu jedem Bezeichner muss eine definierende Bindung existieren.

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

☞ Der Teilausdruck e wird bezüglich der Signatur Σ, Σ' typisiert. Der Kommaoperator erweitert Σ um Σ' und regelt Überschneidungen.

13. Statische Semantik — Beispiel

Beispiel für einen wohlgetypten Ausdruck:

$$\dfrac{\dfrac{\overline{\varnothing \vdash 47 : \mathit{Nat}}}{\varnothing \vdash (\textbf{\textit{let}}\ s = 47) : \{s \mapsto \mathit{Nat}\}} \quad \dfrac{\overline{\{s \mapsto \mathit{Nat}\} \vdash s : \mathit{Nat}} \quad \overline{\{s \mapsto \mathit{Nat}\} \vdash s : \mathit{Nat}}}{\{s \mapsto \mathit{Nat}\} \vdash s * s : \mathit{Nat}}}{\varnothing \vdash (\textbf{\textit{let}}\ s = 47\ \textbf{\textit{in}}\ s * s) : \mathit{Nat}}$$

Beispiel für einen *nicht* wohlgetypten Ausdruck:

$$\dfrac{\dfrac{\overline{\varnothing \vdash 47 : \mathit{Nat}}}{\varnothing \vdash (\textbf{\textit{let}}\ s = 47) : \{s \mapsto \mathit{Nat}\}} \quad \dfrac{\overline{\{s \mapsto \mathit{Nat}\} \vdash s : \mathit{Nat}} \quad \{s \mapsto \mathit{Nat}\} \vdash a : \mathit{Nat}?}{\{s \mapsto \mathit{Nat}\} \vdash s * a : \mathit{Nat}}}{\varnothing \vdash (\textbf{\textit{let}}\ s = 47\ \textbf{\textit{in}}\ s * a) : \mathit{Nat}}$$

☞ Die Formel $\{s \mapsto \mathit{Nat}\} \vdash a : \mathit{Nat}$ lässt sich nicht ableiten.

13. Dynamische Semantik

- Wie werden ***in***-Ausdrücke ausgerechnet?

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

  Wenn wir den Teilausdruck $s * s$ ausrechnen, woher kennen wir den Wert von s?
- Ein Bezeichner kann in unterschiedlichen Kontexten einen unterschiedlichen Wert besitzen.

  (***let*** $s = true$ ***in*** s) && (***let*** $s = 815$ ***in*** $s * s > 4711$)

13. Dynamische Semantik — Umgebung

- Wir merken uns die Werte von Bezeichnern mit Hilfe einer sogenannten Umgebung.

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

Wie eine Signatur ist eine Umgebung eine endliche Abbildung; im Unterschied zur Signatur bildet sie Bezeichner auf *Werte* ab.

- Wir erweitern die Auswertungsregeln um Umgebungen. Die Regeln spezifizieren die nunmehr *dreistellige* Relation

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

zwischen Umgebungen, Ausdrücken und Werten. *Lies:* „bezüglich der Umgebung δ wertet e zu ν aus“.

13. Dynamische Semantik — Definitionen

Die dynamische Semantik ordnet

- einem Ausdruck einen Wert und
- einer Definition eine Umgebung

zu.

Auswertungsregel:

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash (\textbf{\textit{let}}\ x = e) \Downarrow \{x \mapsto \nu\}}$$

13. Dynamische Semantik — Ausdrücke

Auswertungsregeln:

$$\frac{}{\delta \vdash x \Downarrow \delta(x)} \qquad x \in \mathit{dom}\ \delta$$

☞ Zu jedem Bezeichner muss eine definierende Bindung existieren.

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

☞ Der Teilausdruck e wird bezüglich der Umgebung δ, δ' ausgewertet. Der Kommaoperator erweitert δ um δ' und regelt Überschneidungen.

13. Dynamische Semantik — Beispiel

$$\dfrac{\dfrac{}{\varnothing \vdash 4711 + 815 \Downarrow 5526} \quad \dfrac{\dfrac{}{\delta \vdash s \Downarrow 5526} \quad \dfrac{}{\delta \vdash s \Downarrow 5526}}{\delta \vdash s * s \Downarrow 30536676}}{\varnothing \vdash \textit{let}\ s = 4711 + 815\ \textit{in}\ s * s \Downarrow 30536676}$$

wobei $\delta = \{s \mapsto 5526\}$

☞ Der Rumpf $s * s$ wird bezüglich der Umgebung $\{s \mapsto 5526\}$ ausgerechnet.

☞ Aus Gründen der Übersichtlichkeit kürzen wir einfache Teilrechnungen ab.

13. Demo

Mini⟩ *let* $s = 4711 + 815$ *in* $s * s$
30536676
Mini⟩ *let* *size* $= 4711 + 815$ *in* *size* $*$ *size*
30536676
Mini⟩ $(\textit{let}\ s = 4711 + 815\ \textit{in}\ s * s) + 1$
30536677
Mini⟩ $(\textit{let}\ s = 4711\ \textit{in}\ s * s) + (\textit{let}\ s = 815\ \textit{in}\ s * s)$
22857746
Mini⟩ *let* $s = 4711$ *in let* $a = s * s$ *in* $a + s$
22198232

13. Freie Bezeichner

Ein Bezeichner, der nicht im Geltungsbereich einer Definition liegt, heißt *frei*.

Ausdruck	freie Bezeichner
$s * s$	$\{s\}$
$s * a$	$\{a, s\}$
let $s = 4711$ ***in*** $s * s$	$\emptyset$
let $s = 4711$ ***in*** $s * a$	$\{a\}$

☞ In dem Ausdruck *d* ***in*** *e* werden die in *d* definierten Bezeichner von den freien Bezeichner in *e* abgezogen.

Ein Ausdruck, der keine freien Bezeichner enthält, heißt *geschlossen*.

13. Invarianten der Semantik

Invariante der statischen Semantik:

Die statische Semantik typisiert nur Ausdrücke, 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.

Invariante der dynamischen Semantik:

Die dynamische Semantik legt nur die Bedeutung von Ausdrücken 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.

13. Redefinition

Was passiert, wenn man den gleichen Bezeichner mehrfach definiert?

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash 4 \Downarrow 4}}{\varnothing \vdash \textit{let } s = 4 \Downarrow \{s \mapsto 4\}} \quad \dfrac{\dfrac{\dfrac{}{\{s \mapsto 4\} \vdash 7 \Downarrow 7}}{\{s \mapsto 4\} \vdash \textit{let } s = 7 \Downarrow \{s \mapsto 7\}} \quad \dfrac{\dfrac{}{\{s \mapsto 7\} \vdash s \Downarrow 7}}{\star\{s \mapsto 7\} \vdash s * s \Downarrow 49} \; \vdots}{\{s \mapsto 4\} \vdash \textit{let } s = 7 \textit{ in } s * s \Downarrow 49}}{\varnothing \vdash \textit{let } s = 4 \textit{ in let } s = 7 \textit{ in } s * s \Downarrow 49}$$

$\star$ Der Kommaoperator räumt der "neuen" Definition Vorrang ein:
$\{s \mapsto 4\}, \{s \mapsto 7\} = \{s \mapsto 7\}$.

13. Redefinition

Sollte man den überhaupt Bezeichner redefinieren? Ich meine, ist das nicht schlechter Programmierstil?

Man soll in der Tat darauf achten, Bezeichner nicht zu redefinieren. *Allerdings:* neue Namen zu erfinden ist schwer.

13. Umbenennungen

Die Festlegung, “neuen” Definitionen Vorrang einzuräumen, ist sinnvoll. Sie ist insbesondere kompatibel zum Umbenennen von Bezeichnern.

Der Bezeichner *s* in

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

kann zu *size* umbenannt werden,

let $size = 815$ ***in*** $size * size$

ohne die Bedeutung des Programms zu ändern.

13. Umbenennungen

Die Bedeutung bleibt ebenso unverändert, wenn der Ausdruck *Teil* eines größeren Programms ist.

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.

13. Demo

Mini⟩ $(\textit{let } s = 4711 \textit{ in } s * s) + (\textit{let } s = 815 \textit{ in } s * s)$
22857746
Mini⟩ $\textit{let } s = 4711 \textit{ in let } s = 815 \textit{ in } s * s$
664225
Mini⟩ $\textit{let } s_1 = 4711 \textit{ in let } s_2 = 815 \textit{ in } s_2 * s_2$
664225
Mini⟩ $\textit{let } s = 4711 \textit{ in let } a = s * s \textit{ in } a + a$
44387042
Mini⟩ $\textit{let } a = \textit{let } s = 4711 \textit{ in } s * s \textit{ in } a + a$
44387042

13. Über den Tellerrand: F#

Wollen wir mehrere Definitionen in einem Ausdruck verwenden, müssen wir ***in***-Ausdrücke aneinanderreihen.

```
let a = 4711
      + 815
in let b = a * a
   in a + b
```

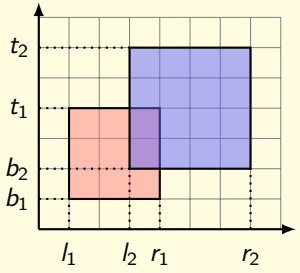
F# erlaubt alternativ das Ende einer lokalen Definition und den Anfang des Rumpfes mit Hilfe des *Layouts* festzulegen.

```
let a = 4711
      + 815
let b = a * a
a + b
```

Abseitsregel: die Einrückung bestimmt die Zugehörigkeit von Teilausdrücken. Einrückung: der „aktuelle“ Ausdruck wird weitergeführt; keine Einrückung: eine neue Definition oder der Rumpfausdruck wird begonnen.

13. Beispiel: Berechnung der Wohnfläche

Die Fläche einer Wohnung mit dem unten skizzierten Grundriss soll berechnet werden.



(Das ist eine Vereinfachung, das Ergebnis des schon angesprochenen Abstraktionsprozesses, mit dem wir Aufgaben in Rechenaufgaben verwandeln.)

13. Beispiel: Berechnung der Wohnfläche

Erfassung der Daten:

$$
\begin{aligned}
&\textbf{\textit{let}}\ l_1 = 1\\
&\textbf{\textit{let}}\ r_1 = 4\\
&\textbf{\textit{let}}\ b_1 = 1\\
&\textbf{\textit{let}}\ t_1 = 4\\
&\textbf{\textit{let}}\ l_2 = 3\\
&\textbf{\textit{let}}\ r_2 = 7\\
&\textbf{\textit{let}}\ b_2 = 2\\
&\textbf{\textit{let}}\ t_2 = 6
\end{aligned}
$$

13. Beispiel: Berechnung der Wohnfläche

Gesamtfläche: Summe der beiden Quadratflächen minus der Schnittfläche.

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

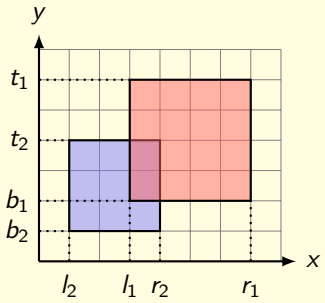
☞ Vermeidung von Mehrfachrechnungen: $s_1 * s_1$ statt $(r_1 \dot{-} l_1) * (r_1 \dot{-} l_1)$.

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

Die Wohnfläche beträgt somit $23\,m^2$.

13. Beispiel: Berechnung der Wohnfläche

Ändern wir die Daten, indem wir 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 $t_2 = 4$

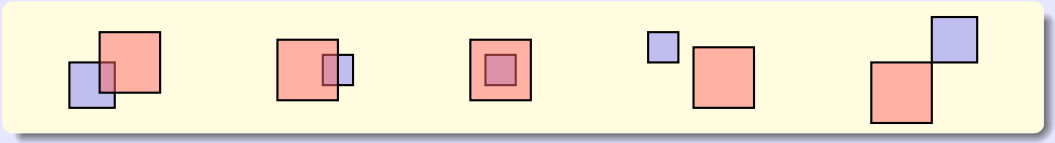
erleben wir eine unangenehme Überraschung:

Mini⟩ ***let*** $l_1 = 3$
Mini⟩ ...
Mini⟩ $s_1 * s_1 + s_2 * s_2 \doteq w * h$
0

13. Beispiel: Berechnung der Wohnfläche

☞ Das obige 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. Viele unterschiedliche Konstellationen sind denkbar:

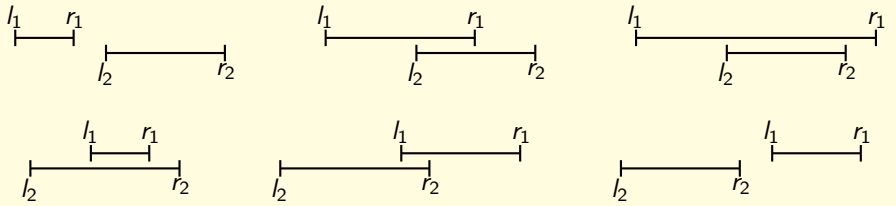


Wie werden wir Herr oder Frau der Lage?

☞ *Entscheidende Einsicht:* Überschneidungen können *getrennt* für jede der beiden Koordinatenachsen ausgerechnet werden. Auf diese Weise wird ein 2-dimensionales auf ein 1-dimensionales Problem zurückgeführt.

13. Beispiel: Berechnung der Wohnfläche

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



Länge der Überschneidung: $min\ r_1\ r_2 \dot{-} max\ l_1\ l_2$.

13. Intermezzo: Minimum und Maximum

- *Minimum* von e_1 und e_2:

  if $e_1 \leqslant e_2$ ***then*** e_1 ***else*** e_2

- *Maximum* von e_1 und e_2:

  if $e_1 \leqslant e_2$ ***then*** e_2 ***else*** e_1

☞ Wir kürzen das Minimum von e_1 und e_2 mit *min* e_1 e_2 und das Maximum mit *max* e_1 e_2 ab.

13. Intermezzo: Minimum und Maximum

☞ Ist die Berechnung von e_1 oder e_2 aufwändig, formuliert man besser:

- *Minimum* von e_1 und e_2:

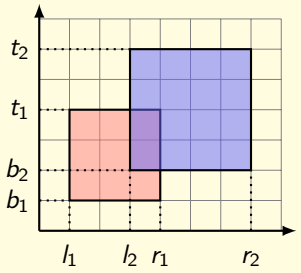
  let $a_1 = e_1$
  let $a_2 = e_2$
  if $a_1 \leqslant a_2$ ***then*** a_1 ***else*** a_2

- *Maximum* von e_1 und e_2:

  let $a_1 = e_1$
  let $a_2 = e_2$
  if $a_1 \leqslant a_2$ ***then*** a_2 ***else*** a_1

Warum?

13. Beispiel: Berechnung der Wohnfläche



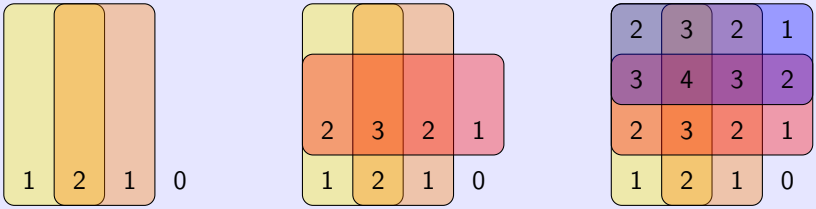
let $l_1 = 1$
let $r_1 = 4$
let $b_1 = 1$
let $t_1 = 4$
let $l_2 = 3$
let $r_2 = 7$
let $b_2 = 2$
let $t_2 = 6$

Lösung:

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

14. Knobelaufgabe #5

Mit zwei Rechtecken lassen sich $2^2 = 4$ „verschiedene“ Flächen konstruieren; mit drei Rechtecken $2^3 = 8$ und mit vier Rechtecken $2^4 = 16$. Zwei Flächen gelten als „gleich“, wenn sie von den gleichen Rechtecken überdeckt werden. (Die Zahlen in den Beispielen geben jeweils die Gesamtzahl der Überdeckungen an.)



Wieviele verschiedene Flächen lassen sich mit fünf Rechtecken erzeugen? Und mit sechs?

14. Motivation

- Zur Erinnerung:

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

  berechnet den Flächeninhalt eines Quadrats der Seitenlänge $4711 + 815$.
- Wir können das Programm verallgemeinern, indem wir von der Seitenlänge *abstrahieren*: aus $s * s$ wird eine Funktion in s.

  let *area* $(s : Nat) : Nat = s * s$

  - *area* ist der Name der Funktion,
  - s vom Typ *Nat* ist der *formale Parameter* und
  - $s * s$ vom Typ *Nat* heißt *Rumpf* der Funktion.

14. Motivation

- Um den Flächeninhalt für eine gegebene Seitenlänge zu berechnen, wenden wir die Funktion an:

$$\mathit{area}\ (4711 + 815)$$

$4711 + 815$ ist das Argument oder der *aktuelle Parameter* der Funktion.

- Der Funktionsaufruf wird ausgerechnet, indem der formale Parameter an den aktuellen Parameter gebunden wird und in dieser Umgebung der Rumpf ausgewertet wird.

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

14. Motivation

- Der Vorteil einer Funktionsdefinition ist, dass die Funktion mehrfach angewendet werden kann:

$$\mathit{area}\ (4711) + \mathit{area}\ (815)$$

- Ohne Funktionen im Repertoire müssten wir formulieren:

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

☞ Der Ausdruck wird verdoppelt, nicht nur die Rechnung!

- Funktionsdefinitionen sind Definitionen und können wie diese in ***in***-Ausdrücken verwendet werden.

$$\begin{array}{l}\textbf{\textit{let}}\ \mathit{area}\ (s : \mathit{Nat}) : \mathit{Nat} = s * s \\ \textbf{\textit{in}}\ \ \mathit{area}\ (4711) + \mathit{area}\ (815)\end{array}$$

Sichtbarkeit: area ist im ***in***-Ausdruck sichtbar; der formale Parameter *s* hingegen nur im Funktionsrumpf.

14. Abstrakte Syntax

$f \in \mathsf{Id}$

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

☞ t_1 ist der Argumenttyp von f und t_2 der Ergebnistyp.

☞ Funktionsapplikation ist der vornehme Name für Funktionsaufruf oder Funktionsanwendung.

14. Statische Semantik

Eine Funktion mit dem 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

Typregeln:

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

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

☞ Der Rumpf einer Funktion wird in der um den formalen Parameter erweiterten Signatur getypt (‘,’ ist der Kommaoperator).

14. Dynamische Semantik

- Was ist der Wert einer Funktion?
- Können wir Funktionen überhaupt auswerten?
- Man könnte eine Funktion tabellieren: für jedes Argument wird der Funktionswert ausgerechnet. Beispiel:

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

wird ausgewertet zu

$$\{\mathit{false} \mapsto \mathit{true}, \mathit{true} \mapsto \mathit{false}\}$$

☞ Nicht machbar, wenn das Argument eine natürliche Zahl ist.

- *Idee:* die Auswertung einer Funktion wird verzögert oder "eingefroren", bis der aktuelle Parameter bekannt ist.

14. Verzögerte Auswertung

Was passiert, wenn der Funktionsrumpf freie Bezeichner enthält?

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

☞ Wenn wir die Auswertung “einfrieren” und später fortsetzen (“auftauen”) wollen, müssen wir uns die Funktionsdefinition *und* die aktuelle Umgebung merken.

14. Dynamische Semantik

Der Bereich der Werte wird um *Funktionsabschlüsse* erweitert (engl. closures).

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

Für das obige Beispiel, die Funktion *next*, erhalten wir:

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

Die Funktionsdefinition, die Zuordnung von n zu $n + d$, wird festgehalten; die Umgebung $\{ d \mapsto 2 \}$ legt die Bedeutung des *freien* Bezeichners d fest.

14. Dynamische Semantik

Auswertungsregel:

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

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

Die Kombination aus Umgebung und Ausdruck, $\delta \vdash e$, kann als Konfiguration oder Zustand eines Rechners aufgefasst werden; die Auswertungsregeln legen die Arbeitsweise des Rechners fest; eine "closure" speichert im wesentlichen eine Konfiguration.

14. Dynamische Semantik

Auswertungsregel:

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

☞ Jetzt da der aktuelle Parameter bekannt ist, kann die verzögerte Auswertung wiederaufgenommen werden.

14. Dynamische Semantik — Beispielrechnung

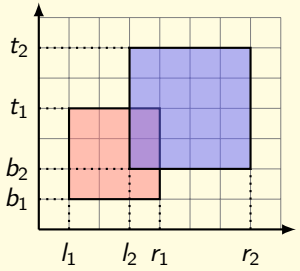
$$\dfrac{\dfrac{}{\varnothing \vdash \textit{let area}\ (s) = s * s \Downarrow \delta} \qquad \dfrac{\dfrac{}{\delta \vdash 4711 + 815 \Downarrow 5526} \quad \dfrac{}{\{s \mapsto 5526\} \vdash s * s \Downarrow 30536676}}{\delta \vdash \textit{area}\ (4711 + 815) \Downarrow 30536676}}{\varnothing \vdash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

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

☞ Der Bezeichner *area* wird an den Funktionsabschluss $\langle \varnothing,\ s,\ s * s \rangle$ gebunden.

14. Beispiel: Berechnung der Wohnfläche — da capo

Die Fläche einer Wohnung mit dem unten skizzierten Grundriss soll berechnet werden.



14. Beispiel: Berechnung der Wohnfläche — da capo

Alte Lösung:

$$
\begin{array}{l}
\textbf{let}\ l_1 = 1 \\
\textbf{let}\ r_1 = 4 \\
\textbf{let}\ b_1 = 1 \\
\textbf{let}\ t_1 = 4 \\
\\
\textbf{let}\ l_2 = 3 \\
\textbf{let}\ r_2 = 7 \\
\textbf{let}\ b_2 = 2 \\
\textbf{let}\ t_2 = 6 \\
\\
\textbf{let}\ s_1 = r_1 \dot{-} l_1 \\
\textbf{let}\ s_2 = r_2 \dot{-} l_2 \\
\textbf{let}\ w = \mathit{min}\ r_1\ r_2 \dot{-} \mathit{max}\ l_1\ \ l_2 \\
\textbf{let}\ h = \mathit{min}\ t_1\ t_2 \dot{-} \mathit{max}\ b_1\ b_2 \\
s_1 * s_1 + s_2 * s_2 \dot{-} w * h
\end{array}
$$

Neue Lösung:

$$
\begin{array}{l}
\textbf{let}\ \mathit{total\text{-}area}\ (l_1, r_1, b_1, t_1, l_2, r_2, b_2, t_2) : \mathit{Nat} = \\
\quad \textbf{let}\ s_1 = r_1 \dot{-} l_1 \\
\quad \textbf{let}\ s_2 = r_2 \dot{-} l_2 \\
\quad \textbf{let}\ w = \mathit{min}\ r_1\ r_2 \dot{-} \mathit{max}\ l_1\ \ l_2 \\
\quad \textbf{let}\ h = \mathit{min}\ t_1\ t_2 \dot{-} \mathit{max}\ b_1\ b_2 \\
\quad s_1 * s_1 + s_2 * s_2 \dot{-} w * h \\
\\
\textbf{let}\ \mathit{Wohnfläche} = \\
\quad \mathit{total\text{-}area}\ (1, 4, 1, 4, 3, 7, 2, 6)
\end{array}
$$

14. Demo

Mini⟩ *total-area* $(1, 4, 1, 4, 3, 7, 2, 6)$
23
Mini⟩ *total-area* $(3, 7, 2, 6, 1, 4, 1, 4)$
23
Mini⟩ *total-area* $(0, 4, 0, 4, 1, 2, 1, 2)$
16
Mini⟩ *total-area* $(0, 4, 0, 4, 4, 7, 4, 7)$
25

14. Vertiefung

Den Booleschen Verknüpfungen können wir jetzt einen Namen geben.

- *Negation*:

```
let not (a : Bool) : Bool = if a then false else true
```

- *Konjunktion*:

```
let and-also (a : Bool, b : Bool) : Bool = if a then b else false
```

- *Disjunktion*:

```
let or-else (a : Bool, b : Bool) : Bool = if a then true else b
```

☞ *and-also* und *or-else* haben zwei Parameter.

Funktionsabschluss? Wozu soll das gut sein?
Bachelorabschluss wär mir lieber.

Jedes Programm hat seinen Wert: der Abschluss $\langle \delta, x, e \rangle$ ist der Wert einer Funktion. Der Wert umfasst alles, was wir über eine Funktion wissen müssen: wie das Funktionsergebnis e in Abhängigkeit vom Funktionsargument x berechnet werden kann und welche Bedeutung die freien Variablen haben.

Müssen wir eigentlich Funktionen immer einen Namen geben? Das machen wir beim Rechnen mit natürlichen Zahlen ja auch nicht—jeder Zahl einen Namen geben, meine ich. Warum haben wir keine Ausdrücke, die zu Funktionen auswerten? Wenn Funktionen Werte sind ...

Gute Idee, Lisa!

15. Abstrakte Syntax

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

☞ ***fun*** $(x : t) \to e$ ist eine *anonyme* Funktion mit dem formalen Parameter x und dem Rumpf e.

☞ Funktionsaufrufe werden ebenfalls verallgemeinert: da Funktionen berechnet werden können, verallgemeinert sich $f\ e_1$ zu $e\ e_1$.

Aus der Schule ist man gewohnt, Applikationen zu klammern: $f\ (e)$ statt $f\ e$. In der konkreten Syntax sind Klammern nur nötig, wenn das Argument nicht „atomar“ ist: $f\ x$ und $f\ 4711$, aber $f\ (x + 4711)$. Später mehr dazu.

15. Statische Semantik

Typregeln:

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

$$\frac{\Sigma \vdash e : t_1 \to t_2 \qquad \Sigma \vdash e_1 : t_1}{\Sigma \vdash e\ e_1 : t_2}$$

15. Dynamische Semantik

Auswertungsregeln:

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

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

☞ Die Auswertung der Funktionsapplikation ist jetzt etwas komplizierter, da zusätzlich e ausgerechnet werden muss — vorher stand an dieser Stelle ein Bezeichner.

Die Auswertung einer Applikation vollzieht sich somit in drei Schritten:

1. die Funktion wird ausgerechnet;
2. das Argument wird ausgerechnet;
3. der Rumpf der Funktion wird ausgerechnet.

15. Vertiefung

- Auch mit Funktionen lässt sich vorzüglich rechnen.
- Die folgenden Definitionen setzen die Addition und die Multiplikation auf Funktionen fort.

$$\textbf{\textit{let}}\ add\ (f : Nat \to Nat, g : Nat \to Nat) : Nat \to Nat =$$
$$\quad \textbf{\textit{fun}}\ (x : Nat) \to f\ x + g\ x$$

$$\textbf{\textit{let}}\ mul\ (f : Nat \to Nat, g : Nat \to Nat) : Nat \to Nat =$$
$$\quad \textbf{\textit{fun}}\ (x : Nat) \to f\ x * g\ x$$

- *add* und *mul* arbeiten auf Funktionen des Typs $Nat \to Nat$.
- Mit Hilfe von *id* und *constant* lassen sich Funktionen des Typs $Nat \to Nat$ konstruieren.

$$\textbf{\textit{let}}\ id : Nat \to Nat = \textbf{\textit{fun}}\ (x : Nat) \to x$$

$$\textbf{\textit{let}}\ constant\ (n : Nat) : Nat \to Nat =$$
$$\quad \textbf{\textit{fun}}\ (x : Nat) \to n$$

15. Demo

```
Mini⟩ add (id, constant 4711)
val it : Nat → Nat
Mini⟩ (add (id, constant 4711)) 815
5526
Mini⟩ add (id, constant 4711) 815
5526
Mini⟩ let square = mul (id, id)
val square : Nat → Nat
Mini⟩ square 4711
22193521
Mini⟩ let twice (f : Nat → Nat) : Nat → Nat = fun x → f (f x)
val twice : (Nat → Nat) → (Nat → Nat)
Mini⟩ twice square 2
16
Mini⟩ twice (twice square) 2
65536
```

Wenn ich das richtig sehe, dann sind Funktionsdefinitionen gar nicht mehr notwendig: die Funktionsdefinition

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

entspricht exakt der Wertedefinition

$$\textbf{\textit{let}}\ f = \textbf{\textit{fun}}\ (x : t_1) \to e$$

Hmm, Du meinst wohl, anonyme Funktionen sind überflüssig. Statt

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

schreibe ich immer

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

Also Harry, ich erkenne Dich gar nicht wieder:
28 Tastendrücke statt 17. Sonst bist Du doch so tippfaul.

16. Knobelaufgabe #6

Harry Hacker hat mit dem folgenden Ausdruck den „International Obfuscated F# Code Contest“ gewonnen.

$$\textbf{\textit{let}}\ a = \textbf{\textit{fun}}\ b\ c \rightarrow b\ (b\ c)\ \textbf{\textit{in}}\ a\ a\ a\ a\ (\textbf{\textit{fun}}\ d \rightarrow d * d * d)\ 9$$

Zu welcher natürlichen Zahl wertet der Ausdruck aus?

16. Motivation

- Wieviele Möglichkeiten gibt es, n verschiedene Objekte in einer Reihe zu arrangieren?
  - Für die erste Position gibt es n Möglichkeiten,
  - für die zweite Position gibt es $n - 1$ Möglichkeiten,
  - ...
  - für die vorletzte Position gibt es 2 Möglichkeiten,
  - für die letzte Position gibt es 1 Möglichkeit.
- Insgesamt gibt es $1 * 2 * \cdots * (n - 1) * n$ Möglichkeiten.
- Diese Zahl heißt auch n *Fakultät*, notiert $n!$.

16. Motivation

Wie können wir die Fakultätsfunktion in Mini–F# programmieren?

Ein erster Versuch:

```
let factorial (n : Nat) : Nat =
  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 ...
```

☞ Der Wert von 0 Fakultät ist 1, da das leere Produkt von Zahlen vereinbarungsgemäß 1 ist.

16. Motivation

Beobachtung: in jedem der Fälle $n > 0$ ist der letzte Faktor die Zahl n selbst.

```
let factorial (n : Nat) : Nat =
  if n = 0 then 1
  else    if n = 1 then n
     else if n = 2 then 1 * n
     else if n = 3 then 1 * 2 * n
     else ...
```

16. Motivation

☞ Der Faktor n kann aus den Zweigen der Alternativen „herausgezogen“ werden.

```
let factorial (n : Nat) : Nat =
  if n = 0 then 1
  else (  if n = 1 then 1
    else if n = 2 then 1
    else if n = 3 then 1 * 2
    else ...) * n
```

16. Motivation

Der Ausdruck in Klammern sieht dem ursprünglichen Funktionsrumpf sehr ähnlich. Die Konstanten können in Übereinstimmung gebracht bringen, indem wir links und rechts jeweils 1 subtrahieren.

```
let factorial (n : Nat) : Nat =
  if n = 0 then 1
  else (  if n - 1 = 0 then 1
    else if n - 1 = 1 then 1
    else if n - 1 = 2 then 1 * 2
    else  ...) * n
```

☞ Der Ausdruck in Klammern ist gleich dem Aufruf *factorial* $(n - 1)$.

16. Motivation

Erlauben wir bei der Definition einer Funktion den Rückgriff auf die definierte Funktion selbst (!), so erhalten wir:

let rec *factorial* $(n : Nat) : Nat =$
  if $n = 0$ ***then*** 1 ***else*** *factorial* $(n \dot{-} 1) * n$

☞ Greift man bei der Definition auf das definierte Objekt selbst zurück, spricht man von einer *rekursiven* Definition.

☞ Rekursive Definitionen werden mit dem Schlüsselwort ***rec*** gekennzeichnet. (Warum?)

16. Abstrakte Syntax

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

☞ ***let rec*** $f\ (x) = e$ definiert genau wie ***let*** $f\ (x) = e$ eine Funktion, nur dass in e zusätzlich der Bezeichner f sichtbar ist.

16. Statische Semantik

Typregeln:

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

Zum Vergleich die Regel für nicht-rekursive Definitionen:

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

16. Dynamische Semantik

Der Bereich der Werte wird um rekursive Funktionsabschlüsse erweitert.

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

Auswertungsregeln:

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

$$\frac{\delta \vdash e \Downarrow \nu \qquad \delta \vdash e_1 \Downarrow \nu_1 \qquad \delta', \{f \mapsto \nu, x_1 \mapsto \nu_1\} \vdash e' \Downarrow \nu'}{\delta \vdash e\ e_1 \Downarrow \nu'}$$

mit $\nu = \langle \delta', f, x_1, e' \rangle$

☞ f wird an den rekursiven Funktionsabschluss gebunden, in dem f selbst aufgeführt wird. (Aus der Rekursion wird ein zyklisches Geflecht.)

16. Demo

Die Fakultät wächst sehr schnell, wie die folgenden Aufrufe 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.

17. Vertiefung

- Der Schritt von den nicht-rekursiven zu den rekursiven Funktionen ist ein gewaltiger.

  Mini–F# ist berechnungsuniversell.

  Mit Mini–F# 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. Im folgenden:
  - Potenzfunktion,
  - Quadratwurzel,
  - ein Ratespiel,
  - und mehr.

17. Vertiefung

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

Die Definition basiert auf der Tatsache, dass eine natürliche Zahl entweder 0 oder größer als 0 ist.

Problemlösungsbrille:

- *Rekursionsbasis*: die Lösung muss unmittelbar angeben werden.
- *Rekursionsschritt*: rekursiv wird eine Lösung für $n \dot{-} 1$ bestimmt, dann wird die Teillösung zu einer Gesamtlösung für n erweitert.

17. Potenzfunktion

Wenden wir das Problemlösungsschema auf die Potenzfunktion x^n an.

let rec *power* $(x : Nat, n : Nat) : Nat =$
  if $n = 0$ ***then*** $\ldots$
    else $\ldots$ *power* $(x, n \dot{-} 1) \ldots$

☞Wir rekurrieren über n, nicht über x. (Warum?)

17. Potenzfunktion

- *Rekursionsbasis*: $x^0 = 1$.

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

- *Rekursionsschritt*: $x^n = x^{n-1} * x$.

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

17. Multiplikation

Die Potenzfunktion wird auf wiederholte Multiplikation zurückgeführt. Ebenso lässt sich die Multiplikation auf wiederholte Addition zurückführen.

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

17. Multiplikation

- *Rekursionsbasis*: $0 * n = 0$.

```
let rec mul (m : Nat, n : Nat) : Nat =
  if m = 0 then 0
           else  ... mul (m ∸ 1, n) ...
```

- *Rekursionsschritt*: $m * n = (m - 1) * n + n$.

```
let rec mul (m : Nat, n : Nat) : Nat =
  if m = 0 then 0
           else  mul (m ∸ 1, n) + n
```

17. Addition

Und die Addition lässt sich auf die Nachfolgerfunktion zurückführen.

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

17. Addition

- *Rekursionsbasis*: $0 + n = n$.

```
let rec add (m : Nat, n : Nat) : Nat =
  if m = 0 then n
           else  ... add (m ∸ 1, n) ...
```

- *Rekursionsschritt*: $m + n = (m - 1) + n + 1$.

```
let rec add (m : Nat, n : Nat) : Nat =
  if m = 0 then n
           else  add (m ∸ 1, n) + 1
```

17. Zwischenfazit

Die Beispielprogramme zeigen, dass wir theoretisch mit einigen wenigen vordefinierten Funktionen auskommen:

- der Zahl 0,
- dem Test „gleich 0“,
- der Nachfolgerfunktion und
- der Vorgängerfunktion.

☞ Minimalistische Sprachen sind von Vorteil, wenn man Aussagen über *alle* Programme einer Sprache machen möchte. Mehr zu diesem Thema später aus der Abteilung der Theoretischen Informatik.

17. Peano Entwurfsmuster

Haben wir die Aufgabe eine Funktion $f : Nat \to t$ zu erstellen, dann sieht ein erster Entwurf folgendermaßen aus.

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

Die Ellipsen müssen mit Leben gefüllt werden:

- *Rekursionsbasis*: ein Ausdruck des Typs t.
- *Rekursionsschritt*: ein Ausdruck, der die Teillösung $f\ (n \dot{-} 1)$ vom Typ t zu einer Gesamtlösung vom Typ t erweitert.

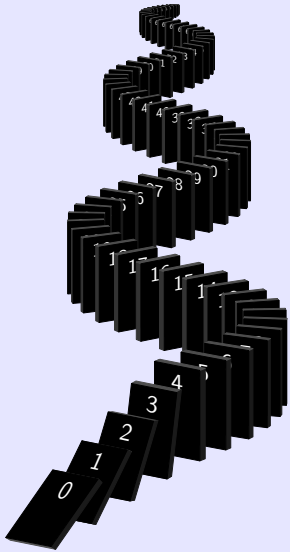
17. Giuseppe Peano (1858–1932)

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

17. Induktion: das Domino Prinzip



17. Quadratwurzel

Aufgabe: square-root n soll die Quadratwurzel der Zahl *n* bestimmen. Genauer: gesucht wird die *größte* Zahl *r*, so dass $r * r \leqslant n$.

```
Mini⟩ square-root 4711
68
Mini⟩ square 68
4624
Mini⟩ square 69
4761
```

In eine Formel gegossen suchen wir $\lfloor \sqrt{n} \rfloor$.

17. Quadratwurzel

Mit dem Peano Entwurfsmuster erhalten wir:

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

17. Quadratwurzel

- *Rekursionsbasis*: $\lfloor\sqrt{0}\rfloor = 0$.

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

- *Rekursionsschritt*: Wie lässt sich aus $\lfloor\sqrt{n-1}\rfloor$ eine Lösung für $\lfloor\sqrt{n}\rfloor$ herleiten? Es gilt: $0 \leqslant \lfloor\sqrt{n}\rfloor - \lfloor\sqrt{n-1}\rfloor \leqslant 1$ wenn $n > 0$. Also,
  - $\lfloor\sqrt{n}\rfloor$ ist entweder identisch zu $\lfloor\sqrt{n-1}\rfloor$ oder
  - $\lfloor\sqrt{n}\rfloor$ ist um eins größer.

  Wir testen 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
```

17. Peano Entwurfsmuster — programmiert

Was ist im Rekursionsschritt zu tun?

- *Rekursionsschritt*: ein Ausdruck, der die Teillösung $f\ (n \dot{-} 1)$ vom Typ t zu einer Gesamtlösung vom Typ t erweitert.

Einsicht: die Erweiterung der Teillösung zu einer Gesamtlösung ist der Natur nach eine Funktion.

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))
```

17. Peano Entwurfsmuster — programmiert

Abstrahieren wir von *zero* und *succ*, erhalten wir eine *Implementierung* des Peano Entwurfsmusters.

```
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
```

17. Peano Entwurfsmuster — programmiert

Mit Hilfe von *peano-pattern* können wir *power* usw. kürzer aufschreiben.

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

☞ Man sieht sehr schön, wie jeweils die Teillösung s zu einer Gesamtlösung erweitert wird.

17. Peano Entwurfsmuster — programmiert

Ganz perfekt ist die Umsetzung des Entwurfsmusters noch nicht.

- Der Typ ist zu speziell:

  $$\textbf{\textit{let}}\ \mathit{peano\text{-}pattern}\ (\mathit{zero} : \mathit{Nat}, \mathit{succ} : \mathit{Nat} \to \mathit{Nat}) : \mathit{Nat} \to \mathit{Nat} =$$

- Die Fakultätsfunktion und die Quadratwurzel lassen sich damit nicht definieren.

17. Knobelaufgabe #7

Harry Hacker hat die Fakultätsfunktion auf die ganzen Zahlen portiert.

```
let rec factorial (n : int) : int =    // natural numbers are
  if n = 0 then 1                      // for quiche eaters
           else factorial (n − 1) ∗ n
```

Die ersten Tests sind vielversprechend ...

```
Mini⟩ factorial 5
120
Mini⟩ factorial 10
3628800
```

...aber irgendwo muss sich ein Fehler eingeschlichen haben:

```
Mini⟩ factorial 17
− 288522240
Mini⟩ factorial 34
0
```

17. Leibniz Entwurfsmuster

- Beim Peano Entwurfsmuster wird das Problem für n auf das Problem für $n \div 1$ zurückgeführt.
- Wir können alternativ versuchen, das Problem für n auf das Problem für $n \div 2$ zurückzuführen.
- Zur Erinnerung:

$$a = (a \div 2) * 2 + (a \,\%\, 2)$$

☞ Jede Zahl a lässt sich eindeutig in einen Quotienten und in einen Rest zerlegen.

17. Potenzfunktion — da capo

Programmieren wir die Potenzfunktion neu.

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

17. Potenzfunktion — da capo

- *Rekursionsbasis*: $x^0 = 1$.

```
let rec power (x : Nat, n : Nat) : Nat =
  if n = 0 then 1
           else ... power (x, n ÷ 2) ...
```

- *Rekursionsschritt*: $x^n = x^{(n \div 2)*2+(n\%2)} = (x^{n \div 2})^2 * x^{n\%2}$.

```
let rec power (x : Nat, n : Nat) : Nat =
  if n = 0 then 1
           else if n % 2 = 0 then square (power (x, n ÷ 2))
                             else square (power (x, n ÷ 2)) * x
```

17. Potenzfunktion — Laufzeit

- Wieviele rekursive Aufrufe benötigt *power* (x, n) jetzt? (Vorher waren es n.)
- In jedem Schritt wird n halbiert; das können wir insgesamt $\lg n = \log_2 n$ mal machen.
- *power* hat eine *logarithmische* Laufzeit.
- Die erste Version hat eine *lineare* Laufzeit.
- Programme mit logarithmischer Laufzeit haben einen erheblichen Geschwindigkeitsvorteil gegenüber solchen mit linearer Laufzeit.

n		$\lg n$
1.000	$\approx$	10
1.000.000	$\approx$	20
1.000.000.000	$\approx$	30

17. Multiplikation — da capo

Mit dem gleichen Entwurfsmuster lässt sich auch die Multiplikation verbessern.

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

17. Multiplikation — da capo

- *Rekursionsbasis*: $0 * n = 0$.

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

- *Rekursionsschritt*: $m * n = ((m \div 2) * 2 + (m \,\%\, 2)) * n = 2\ ((m \div 2) * n) + (m \,\%\, 2) * n$.

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

Also, etwas merkwürdig ist das schon.

$$2 * mul\ (m \div 2, n)$$

Die Multiplikation soll doch implementiert werden, dafür kann ich doch die Multiplikation '$*$' selbst nicht verwenden.

Verdopplung ist ja ein ziemlich einfaches Produkt: *statt* $2 * x$ kannst Du ja $x + x$ rechnen.

Ok, dann schreibe ich das mal um ...

$$mul\ (m \div 2, n) + mul\ (m \div 2, n)$$

Ich glaube, das ist keine gute Idee. Du willst doch das Ergebnis verdoppeln, nicht die Rechnung.

$$\textbf{\textit{let}}\ p = mul\ (m \div 2, n)\ \textbf{\textit{in}}\ p + p$$

Um noch einmal auf Harrys Ausgangspunkt zurückzukommen. Der Punkt ist, dass die Operationen $e * 2$, $e \div 2$, $e \,\%\, 2$ sehr leicht in Hardware zu implementieren sind.

$2 * x$ ist einfacher als $x + x$?

Ja, so wie in unserem Dezimalsystem $10 * x$ sehr viel einfacher ist als $x + x + x + x + x + x + x + x + x + x$.

Ja, klar! Rechner verwenden das Dualsystem: $2 * x$ hängt hinten eine 0 dran, $x \div 2$ streicht das letzte Bit und $x \,\%\, 2$ ist das letzte Bit.

17. Lösung Knobelaufgabe #7

Der Effekt lässt sich einfacher vorführen, wenn man an Stelle des Typs *int* (vorzeichenbehaftete 32-Bit Zahl) den Typ *sbyte* (vorzeichenbehaftete 8-Bit Zahl) betrachtet.

```
Mini⟩ 1y * 2y * 3y * 4y * 5y
120y
Mini⟩ 1y * 2y * 3y * 4y * 5y * 6y
-48y
Mini⟩ 1y * 2y * 3y * 4y * 5y * 6y * 7y * 8y * 9y * 10y
0y
```

Warum $0y$? Die Multiplikation $2y * x$ „hängt hinten eine 0 dran“. Wenn man sich die Primfaktorenzerlegung des Produkts anschaut,

```
Mini⟩ 1y * 2y * 3y * (2y * 2y) * 5y * (2y * 3y) * 7y * (2y * 2y * 2y) *
        (3y * 3y) * (2y * 5y)
```

sieht man, dass insgesamt 8-mal mit $2y$ multipliziert wird: alle 8 Positionen einer 8-Bit Zahl sind somit 0.

17. Leibniz Entwurfsmuster

Haben wir die Aufgabe eine Funktion $f : Nat \to t$ zu erstellen, dann sieht ein zweiter Entwurf folgendermaßen aus.

let rec $f\ (n : Nat) : t =$	*Leibniz Entwurfsmuster:*
if $n = 0$ ***then*** $\ldots$	*Rekursionsbasis*
else $\ldots f\ (n \div 2) \ldots$	*Rekursionsschritt*

Die Ellipsen müssen mit Leben gefüllt werden:

- *Rekursionsbasis*: ein Ausdruck des Typs t.
- *Rekursionsschritt*: ein Ausdruck, der die Teillösung $f\ (n \div 2)$ vom Typ t zu einer Gesamtlösung vom Typ t erweitert.

17. Gottfried Wilhelm Leibniz (1646 – 1716)

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

17. Quadratwurzel — da capo

Versuchen wir das Leibniz Entwurfsmuster auf die Quadratwurzel anzuwenden.

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

17. Quadratwurzel — da capo

- *Rekursionsbasis*: $\lfloor\sqrt{0}\rfloor = 0$.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then 0
           else ... square-root (n ÷ 2) ...
```

- *Rekursionsschritt*: Wie lässt sich aus $\lfloor\sqrt{n \div 2}\rfloor$ eine Lösung für $\lfloor\sqrt{n}\rfloor$ herleiten?

17. Quadratwurzel — da capo

☞ Es ist nicht zwingend durch zwei zu dividieren. Für unser Problem ist eine Quadratzahl, zum Beispiel vier, eine geschicktere Wahl.

$$\begin{array}{l}\textbf{\textit{let rec}}\ \textit{square-root}\ (n : \textit{Nat}) : \textit{Nat} = \\ \quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ \ldots \\ \qquad\qquad \textbf{\textit{else}}\ \ \ldots \textit{square-root}\ (n \div 4) \ldots\end{array}$$

17. Quadratwurzel — da capo

- *Rekursionsbasis*: $\lfloor\sqrt{0}\rfloor = 0$.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then 0
           else ... square-root (n ÷ 4) ...
```

- *Rekursionsschritt*: Da $2\lfloor\sqrt{n/4}\rfloor$ und $\lfloor\sqrt{n}\rfloor$ höchstens um eins differieren, können wir die erste Version 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
```

17. Demo

Mini⟩ *power* $(2, 10)$
1024
Mini⟩ *power* $(2, 16)$
65536
Mini⟩ *square-root it*
256
Mini⟩ *power* $(2, 8)$
256
Mini⟩ *square-root* (*factorial* 10)
1904
Mini⟩ *square it* $\leqslant$ *factorial* 10 && *factorial* $10 <$ *square* $(it + 1)$
true

17. Vertiefung: 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.

17. Schnittstelle

- Spieler A: muss zu einem gegebenen n Auskunft geben, ob die gesuchte Zahl gleich oder kleiner als n ist.

  $$\mathit{player\text{-}A} : \mathit{Nat} \to \mathit{Bool}$$

  ☞ Eine solche Funktion heißt auch *Orakel*.
- Spielerin B: ihr machen wir Spieler A bekannt.

  $$\mathit{player\text{-}B} : (\mathit{Nat} \to \mathit{Bool}) \to \mathit{Nat}$$

17. Spieler A

Beispielimplementierung von Spieler A.

let *player-A* (*guess* : *Nat*) : *Bool* =
$4711 \leqslant$ *guess*

Die Repräsentation des Orakels stellt sicher, dass keine der beteiligten Parteien mogelt:

- die gesuchte Zahl ist fest verdrahtet — Spieler A kann sie nicht nachträglich ändern,
- die gesuchte Zahl kann aber auch nicht eingesehen werden — Spielerin B kann das Orakel nur auf eine Zahl anwenden und aus dem Ergebnis ihre Schlüsse ziehen.

17. Spielerin B

Logik von Spielerin B:

```
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
```

Das Programm ist aber nicht nach einem Entwurfsmuster gestrickt: das Problem für n wird auf das Problem für $n+1$ zurückgeführt.

Wieso, Hauptsache es klappt:

Mini⟩ *player-B player-A*
4711

Und was ist, wenn Spieler A sich gar keine Zahl ausgedacht hat?

Mini⟩ *player-B* (***fun*** $k \to$ *false*)

[wartet] Komisch, tut sich nichts. Wohl mal wieder ein schnellerer Prozzi fällig.

17. Terminierung

- Der Aufruf *player-B* (***fun*** $k \rightarrow$ *false*) *terminiert nicht.*
  - *search* 0 ruft *search* 1 auf.
  - *search* 1 ruft *search* 2 auf.
  - ...
- Die Semantik ordnet dem Aufruf keinen Wert zu.
  - Um einen Beweisbaum für *search* $0 \Downarrow \nu_0$ zu konstruieren, wird ein Beweisbaum für *search* $1 \Downarrow \nu_1$ benötigt.
  - Um einen Beweisbaum für *search* $1 \Downarrow \nu_1$ zu konstruieren, wird ein Beweisbaum für *search* $2 \Downarrow \nu_2$ benötigt.
  - ...

17. Terminierung

Das Peano Entwurfsmuster 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.

17. Spielerin B

Laut Aufgabenstellung ist die Zahl höchstens sechsstellig.

Wir parametrisieren *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
```

☞ Ist die Terminierung sichergestellt?

17. Spielerin B

Beherzigen wir das Peano Entwurfsmuster und rekurrieren nicht über den Ratekandidaten selbst, 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 ersten Version steht aber noch aus.

Ich hab das mal nachgerechnet. Beide Versionen sind gleich! Pass auf: die Parameter der zwei Versionen von *search* stehen in der folgenden Beziehung:

$$n + d = \mathit{upper}$$

Beim ersten Aufruf von *search* wird die Beziehung hergestellt:

$$\mathit{lower} + (\mathit{upper} \dot{-} \mathit{lower}) = \mathit{upper}$$

Der rekursive Aufruf erhält die Beziehung:

$$(n + 1) + (d \dot{-} 1) = \mathit{upper}$$

Alle anderen korrespondierenden Ausdrücke ergeben genau die gleichen Werte. Zum Beispiel

$$n = \mathit{upper} \quad \text{und} \quad d = 0.$$

Ich bin beeindruckt! Aber irgendwo muss sich doch ein Fehler eingeschlichen haben.

Mini⟩ *player-B* (***fun*** $k \to$ *false*$, 9, 0$)

[wartet] Terminiert nicht, das gibt's doch nicht!

Der Ansatz war schon goldrichtig! Auch ein gescheiterter Beweis kann wertvoll sein. Wo läuft es schief: Der Zusammenhang $a + (b \dot{-} a) = b$ gilt nur, wenn $b \geqslant a$.

Argh, '$\dot{-}$' ist ja die Subtraktion auf den natürlichen Zahlen!

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

17. Spielerin B — korrigierte Fassung

Korrigierte Fassung der ersten Version: '$\geqslant$' statt '$=$'.

```
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
```

17. Demo

Mini⟩ *player-B* (*player-A*, 0, 999999)
4711
Mini⟩ *player-B* (***fun*** $k \to 815 \leqslant k, 0, 999$)
815
Mini⟩ *player-B* (***fun*** $k \to 815 < square\ (k + 1), 0, 999$)
28
Mini⟩ *square it* $\leqslant 815$ && $815 < square\ (it + 1)$
true

☞ Auf diese Weise lässt sich auch die Quadratwurzel einer Zahl bestimmen: die gedachte Zahl ist durch eine Formel gegeben.

17. Quadratwurzel — da capo

Eine neue Version von *square-root*:

$$\textbf{\textit{let}}\ \textit{square-root}\ (n : \textit{Nat}) : \textit{Nat} =$$
$$\quad \textit{linear-search}\ (\textbf{\textit{fun}}\ k \to n < \textit{square}\ (k+1), 0, n)$$

☞ Da aus der Implementierung eines Spiels ein Programmstück von allgemeinem Nutzen geworden ist, haben wir *player-B* in *linear-search* umbenannt.

17. Spielerin B — da capo

Können wir die Laufzeit mit Hilfe des Leibniz Entwurfsmusters verbessern? Ja! *Idee:* das Suchintervall in jedem Schritt halbieren.

$$
\begin{array}{l}
\textbf{let}\ \mathit{binary\text{-}search}\ (\mathit{oracle} : \mathit{Nat} \to \mathit{Bool}, \\
\qquad\qquad\qquad\qquad \mathit{lower} : \mathit{Nat}, \mathit{upper} : \mathit{Nat}) : \mathit{Nat} = \\
\quad \textbf{let rec}\ \mathit{search}\ (l : \mathit{Nat}, u : \mathit{Nat}) : \mathit{Nat} = \\
\qquad \textbf{if}\ l \geqslant u\ \textbf{then}\ u \\
\qquad \textbf{else let}\ m = (l + u) \div 2 \\
\qquad\quad \textbf{in}\ \ \textbf{if}\ \mathit{oracle}\ m\ \textbf{then}\ \mathit{search}\ (l, \qquad m) \\
\qquad\qquad\qquad\qquad\ \ \textbf{else}\ \ \mathit{search}\ (m + 1, u) \\
\quad \textbf{in}\ \mathit{search}\ (\mathit{lower}, \mathit{upper})
\end{array}
$$

17. Demo

Es ist instruktiv, sich die Abfolge der Rateversuche anzuschauen. Zu diesem Zweck lassen wir Spieler A die geratenen Zahlen am Bildschirm ausgeben.

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

17. Demo

```
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 4711 wird in 20 Runden systematisch eingekreist.

17. Quadratwurzel — da capo

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

$$\textbf{\textit{let}}\ \mathit{square\text{-}root}\ (n : \mathit{Nat}) : \mathit{Nat} =$$
$$\quad \mathit{binary\text{-}search}\ (\textbf{\textit{fun}}\ k \to n < \mathit{square}\ (k+1), 0, n)$$

17. Zusammenfassung

Wir haben

- die grundlegenden Bestandteile von Mini–F# kennengelernt:
  - Boolesche Werte,
  - natürliche Zahlen,
  - Wertedefinitionen,
  - nicht-rekursive und rekursive Funktionen.
- alle Konzepte rigoros definiert:
  - abstrakte Syntax: Baumsprachen,
  - statische Semantik: Typregeln,
  - dynamische Semantik: Auswertungsregeln.
- Entwurfsmuster kennengelernt, die bei der systematischen Programmentwicklung hilfreich sind, und die die Terminierung sicherstellen,
- gesehen, dass das gleiche Problem sich oft auf verschiedene Art und Weisen und unterschiedlich effizient lösen lässt.

Teil IV

Datentypen

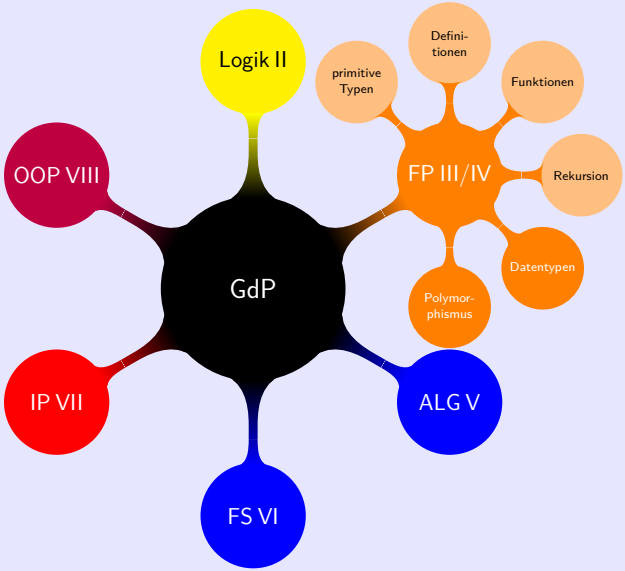
17. Knobelaufgabe #8

Ein *Fleißiger Biber* ist ein Mini–F# Ausdruck, der zu der größten Zahl auswertet — unter allen Ausdrücken der gleichen textuellen Länge.

Länge	Fleißiger Biber / Wert
1	`9` / 9
2	`99` / 99
⋮	⋮
50	`let rec f n:Nat=if n=0 then 9 else n*f(n-1)in f 99` / 8399359389954973741352931497064003044164437143794 3459321733667505695839993906924048047317578540866 4576283281287445013824260666898251776000000000000 0000000000
⋮	⋮

Füllen Sie die fehlenden Einträge (korrigieren Sie ggf. die obigen)! Wann werden die Werte zu groß? Lässt sich die Funktion, die jeder Länge den Wert des Fleißigen Bibers zuordnet, in Mini–F# programmieren?

17. Gliederung



17. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- die beiden grundsätzlichen Strukturierungselemente für Daten kennen: Tupel bzw. Records und Varianten,
- Mini–F# Sprachkonstrukte für Tupel bzw. Records und Varianten kennen und verwenden können,
- mit der Verwendung von Mustern vertraut sein,
- rekursive und parametrisierte Typdefinitionen lesen und selbst definieren können,
- einfache Datenstrukturen wie Listen kennen,
- das Konzept der Polymorphie verstanden haben,
- Arrays verwenden können.

17. Überblick

- Informatiker*innen bilden Modelle der Wirklichkeit.
- Einen wesentlichen Teil dieser Modelle machen Daten aus.
- *Bisher:* bescheidenes Repertoire an Datentypen:
  - Boolesche Werte: *Bool*,
  - natürliche Zahlen: *Nat*,
  - Funktionen: $t_1 \to t_2$.
- Was uns fehlt, sind Möglichkeiten
  - mehrere Daten zu einem Datum 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.

Schau mal, Lisa. Ich habe zwei Funktionen programmiert, die von zwei Zahlen die kleinere bzw. die größere bestimmen.

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

Ja?

Wenn ich jetzt sowohl die kleinere als auch die größere Zahl brauche, dann muss ich beide Funktionen aufrufen. Aber dann wird der Vergleich $a \leqslant b$ zweimal durchgeführt. Unnötigerweise.

Du willst die beiden Zahlen also sortieren?

Genau!

Wie wär's, wenn Du einfach ein Paar zurückgibst?

$$\textbf{let}\ sort2\ (a : Nat, b : Nat) : Nat * Nat = \textbf{if}\ a \leqslant b\ \textbf{then}\ (a, b)\ \textbf{else}\ (b, a)$$

Cool! Darauf hätte ich auch kommen können. Aber wie kriege ich das Paar wieder auseinanderklamüsert?

Stimmt, wir brauchen noch zwei zusätzliche Konstrukte. Wie wär's mit *fst e* und *snd e*, um an die erste bzw. zweite Komponente zu kommen?

Dann müsste ich also schreiben:

$$\textbf{let}\ x = sort2\ (\ldots, \ldots)\ \textbf{in}\ \ldots fst\ x \ldots snd\ x \ldots$$

18. Motivation

- Paare erlauben es, Daten zu aggregieren; zwei verschiedene Daten als Einheit zu behandeln.
- Die zwei Komponenten eines Paares müssen nicht den gleichen Typ besitzen:
  - $(\texttt{"Lisa"}, 9)$
  - $(7, \textbf{\textit{fun}}\ (i : \mathit{Nat}) \to i)$
- Eigentlich sind Paare kein neues Konzept; die Funktion *minimum* nimmt ein Paar als *Argument*.
  - *Bisherige Sichtweise: minimum* hat zwei Argumente,
  - *Jetzt: minimum* hat *ein* Argument, nämlich ein Paar.

18. Abstrakte Syntax

Wir erweitern Ausdrücke um Sprachkonstrukte, die Paare konstruieren bzw. 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).

18. Statische Semantik

Der Typ eines Paares ist ein Paar von Typen, das sogenannte kartesische Produkt der Typen.

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

Typregeln:

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

$$\frac{\Sigma \vdash e : t_1 * t_2}{\Sigma \vdash \mathit{fst}\ e : t_1} \qquad \frac{\Sigma \vdash e : t_1 * t_2}{\Sigma \vdash \mathit{snd}\ e : t_2}$$

18. Dynamische Semantik

Wir erweitern den Bereich der Werte um Paare von Werten.

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

Auswertungsregeln:

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

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

18. Tupel

Alle Konstrukte verallgemeinern sich in natürlicher Weise auf *Tupel*, Aggregationen von n verschiedenen Komponenten.

- $n = 0$:
  - keine Komponente, keine Projektionsfunktion;
  - der sogenannte *Unit* Typ umfasst genau ein Element, nämlich ();
  - später: nützlich als „Dummytyp“.
- $n = 1$:
  - eine Komponente, eine Projektionsfunktion;
  - wenig sinnvoll, da anstelle des 1-Tupels stets die einzige Komponente treten kann;
  - wird von der konkreten Syntax *nicht* unterstützt, da ‘(e)’ zur Gruppierung von Ausdrücken dient. (Mehr dazu in Teil VI.)
- $n = 3$:
  - drei Komponenten, drei Projektionsfunktionen.
- ...

18. 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 Implementierung ist *optimal*: Drei Zahlen können auf 3 Fakultät Arten angeordnet werden (als Formel: $3! = 6$). Mit zwei ineinander geschachtelten Alternativen können aber nur $2^2 = 4$ Fälle unterschieden werden.

18. Vertiefung

Die Funktion *sort3* lässt sich etwas kompakter aufschreiben, indem wir auf *sort2* zurückgreifen.

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

☞ Diese Version macht die Vorgehensweise deutlich: zunächst werden a und b geordnet, dann wird die Position von c bestimmt.

19. Motivation

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.

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

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

19. Abstrakte Syntax

Bezeichner in Bindungspositionen werden verallgemeinert zu sogenannten *Mustern* (engl. patterns).

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***let*** $p = e$	verallgemeinerte Wertedefinition

Muster:

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

☞ Statische Semantik, siehe Skript/zur Übung.

19. Dynamische Semantik

Beispiele: wir nehmen an, dass der Ausdruck $e = \mathit{sort2}\ (e_1, e_2)$ zu dem Wert $\nu = (\nu_1, \nu_2)$ auswertet.

Definition	Umgebung
let $\_ = e$	$\varnothing$
let $(\mathit{min}, \_) = e$	$\{\, \mathit{min} \mapsto \nu_1 \,\}$
let $(\mathit{min}, \mathit{max}) = e$	$\{\, \mathit{min} \mapsto \nu_1, \mathit{max} \mapsto \nu_2 \,\}$
let x & $(\mathit{min}, \_) = e$	$\{\, x \mapsto \nu, \mathit{min} \mapsto \nu_1 \,\}$
let x & $(\mathit{min}, \mathit{max}) = e$	$\{\, x \mapsto \nu, \mathit{min} \mapsto \nu_1, \mathit{max} \mapsto \nu_2 \,\}$

☞ Eine verallgemeinerte Wertedefinition bindet mehrere Bezeichner oder auch keine.

☞ Dynamische Semantik, siehe Skript/zur Übung.

19. Lösung Knobelaufgabe #1

Mit wievielen Vergleichen lassen sich 5 Zahlen sortieren?

- Mit weniger als 7 Vergleichen geht es *nicht*:
  - 5 Zahlen lassen sich auf $5! = 120$ Weisen anordnen.
  - Mit 6 geschachtelten Vergleichen können nur $2^6 = 64$ Fälle unterschieden werden.
  - Mit 7 Vergleichen können $2^7 = 128$ Fälle unterschieden werden.
  - 7 ist die sogenannte informationstheoretische Schranke.
- Aber, ist es auch möglich, mit 7 Vergleichen auszukommen? Ja!

19. Lösung Knobelaufgabe #1

```
let sort5 (a : Nat, b : Nat, c : Nat, d : Nat, e : Nat) : Nat * Nat * Nat * Nat * Nat =
  let (a, b, c, d) =
      if a ⩽ b then
        if c ⩽ d then if b ⩽ d then (a, b, c, d) else (c, d, a, b)
                 else if b ⩽ c then (a, b, d, c) else (d, c, a, b)
      else
        if c ⩽ d then if a ⩽ d then (b, a, c, d) else (c, d, b, a)
                 else if a ⩽ c then (b, a, d, c) else (d, c, b, a)
    // a ⩽ b ⩽ d und c ⩽ d
  in if b ⩽ e then
      if d ⩽ e then    // a ⩽ b ⩽ d ⩽ e
        if b ⩽ c then (a, b, c, d, e)
                 else if a ⩽ c then (a, c, b, d, e) else (c, a, b, d, e)
      else             // a ⩽ b ⩽ e ⩽ d
        if b ⩽ c then if c ⩽ e then (a, b, c, e, d) else (a, b, e, c, d)
                 else if a ⩽ c then (a, c, b, e, d) else (c, a, b, e, d)
    else
      if a ⩽ e then    // a ⩽ e ⩽ b ⩽ d
        if c ⩽ e then if a ⩽ c then (a, c, e, b, d) else (c, a, e, b, d)
                 else if b ⩽ c then (a, e, b, c, d) else (a, e, c, b, d)
      else             // e ⩽ a ⩽ b ⩽ d
        if a ⩽ c then if b ⩽ c then (e, a, b, c, d) else (e, a, c, b, d)
                 else if c ⩽ e then (c, e, a, b, d) else (e, c, a, b, d)
```

19. Lösung Knobelaufgabe #1

Vorgehensweise:

- Zunächst werden die ersten beiden Zahlen sortiert (1 Vergleich).
- Dann die zweiten beiden (1 Vergleich).
- Dann die beiden größeren Zahlen aus den ersten beiden Runden (1 Vergleich).
- Situation: $a \leqslant b \leqslant d$ und $c \leqslant d$.
- Dann wird e in die sortierte Folge $a \leqslant b \leqslant d$ eingefügt (2 Vergleiche).
- Schließlich wird c eingefügt (1 oder 2 Vergleiche).

☞ Siehe Donald E. Knuth, TAOCP, Band 3, Seite 183f.

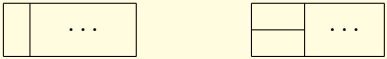
19. Lösung Knobelaufgabe #3

Wieviele Möglichkeiten gibt es, eine Mauer der Breite n zu konstruieren?

- $n = 0$: eine Möglichkeit, die leere Mauer.

- $n = 1$: eine Möglichkeit, ein einzelner Quader.

- $n \geqslant 2$: ergibt sich als Summe aus der Anzahl der Möglichkeiten für $n - 1$ und der Anzahl der Möglichkeiten für $n - 2$.



19. Lösung Knobelaufgabe #3 — Programm

Als Mini–F# Programm:

```
let rec bob (w : Nat) : Nat =
  if w ⩽ 1 then 1
           else bob (w ∸ 1) + bob (w ∸ 2)
```

☞ Das Programm folgt *nicht* dem Peano Entwurfsmuster.

19. Lösung Knobelaufgabe #3 — Demo

```
Mini⟩ bob 4
5
Mini⟩ bob 10
89
Mini⟩ bob 20
10946
Mini⟩ bob 30
1346269
Mini⟩ bob 100
...
```

☞ Um *bob n* auszurechnen, werden mehr als *bob n* Funktionsaufrufe benötigt. (Nachdenken!)

19. Lösung Knobelaufgabe #3 — Programm

Beobachtung: im Rekursionsschritt benötigen wir *bob* $(w \dot{-} 1)$ *und bob* $(w \dot{-} 2)$.

Idee: wir definieren eine Funktion, die beides auf einen Schlag berechnet:
two-bob $w = (bob\ w, bob\ (w + 1))$.

```
let rec two-bob (w : Nat) : Nat * Nat =
  if w = 0 then (1, 1)
           else let
                  (a, b) = two-bob (w ∸ 1)
                in
                  (b, a + b)
let fast-bob (w : Nat) : Nat = fst (two-bob w)
```

☞ Das Programm *folgt* dem Peano Entwurfsmuster.

19. Lösung Knobelaufgabe #3 — Demo

```
Mini⟩ two-bob 10
(89, 144)
Mini⟩ two-bob 20
(10946, 17711)
Mini⟩ two-bob 30
(1346269, 2178309)
Mini⟩ fast-bob 100
573147844013817084101
```

☞ Um *fast-bob n* auszurechnen, werden ungefähr *n* Funktionsaufrufe benötigt.

Ich habe mir überlegt, wie man die Fakultät mit Hilfe von *peano-pattern* programmieren kann.

$$\textbf{\textit{let}}\ \mathit{n\text{-}and\text{-}factorial} : \mathit{Nat} \to \mathit{Nat} * \mathit{Nat} =$$
$$\quad \mathit{peano\text{-}pattern}\ ((0,1), \textbf{\textit{fun}}\ (n,s) \to (n+1, s*(n+1)))$$

Wir konstruieren ein Paar: 1. Komponente: aktueller Wert von n, 2. Komponente: Fakultät n.

Das geht aber nicht durch den Typechecker!

```
This expression was expected
to have type 'Nat'
but here has type 'Nat * Nat'
```

Ich weiß. Das hatten wir doch schon besprochen: der Typ von *peano-pattern* ist zu speziell.

20. Motivation

- Bei Paaren und allgemein bei Tupeln spielt die Reihenfolge der Komponenten eine Rolle:
  - $(12, 1, 2018)$ versus $(1, 12, 2018)$;
  - `("Stefan", "Thomas")` versus `("Thomas", "Stefan")`.

  ☞ Die Rolle der Komponenten ist nur implizit festgelegt: Programmierkonvention.
- Alternative: *Records* statt Tupel.
  - $\{day = 12; month = 1; year = 2018\}$;
  - $\{forename = \texttt{"Stefan"}; surname = \texttt{"Thomas"}\}$.

  ☞ Die sogenannten *Labels* machen die Rolle der verschiedenen Komponenten explizit.
- Die Reihenfolge, in der die benannten Komponenten aufgeschrieben werden, ist irrelevant.
  - $\{month = 1; day = 12; year = 2018\}$;
  - $\{surname = \texttt{"Thomas"}; forename = \texttt{"Stefan"}\}$.
- Mit Hilfe der Labels können Komponenten auch extrahiert werden: *date*.*year* oder *person*.*surname*.

20. Motivation

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

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

Eine Recordtypdefinition führt zwei verschiedene Dinge ein:

- einen Namen für den Recordtyp: *Date* und *Name*,
- Namen um Komponenten des Recordtyps zu extrahieren: *day*, *month*, *year*, *forename* und *surname*. Diese Bezeichner heißen auch *Recordlabels* oder kurz *Labels*.

20. Motivation

Ein Label ähnelt einer Funktion. Der Typ nach dem Label korrespondiert zum Ergebnistyp, der deklarierte Recordtyp korrespondiert zum Argumenttyp:

- *year* hat im Prinzip den Typ $Date \to Nat$ und
- *surname* den Typ $Name \to String$.

☞ 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: *date*.*year* oder *person*.*surname*.

20. Abstrakte Syntax

☞ Der Übersichtlichkeit halber formalisieren wir nur Records mit genau 2 Komponenten.

Ein Recordtyp wird durch eine Definition eingeführt.

$T \in \mathrm{TyId}$	*Typbezeichner*
$\ell \in \mathrm{Lab}$	*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.

20. Abstrakte Syntax

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

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

20. Statische Semantik: Vorüberlegungen⋆

- *Zur Erinnerung:* ein Bezeichner kann redefiniert werden.

  let $s = \mathit{false}$
  let $s = 4711$

  ☞ Die zweite Definition verschattet die erste.
- Sollen wir zulassen, dass auch Typen redefiniert können?

  type $\mathit{Oh} = \{\mathit{je} : \mathit{Bool}\}$
  type $\mathit{Oh} = \{\mathit{je} : \mathit{Nat}\ \}$

  ☞ Die zweite Definition verschattet die erste.
- Aber, was passiert, wenn der Typbezeichner in Typangaben verwendet wird?

20. Statische Semantik: Vorüberlegungen⋆

```
type Oh = {je : Bool}
let na-und (oh : Oh) : Bool = not (oh.je)
type Oh = {je : Nat  }
let egal = na-und {je = 4711}
```

- Der Typ *Oh* wird definiert.
- Die Funktion *na-und* erhält den Typ $Oh \to Bool$.
- Der Typ *Oh* wird redefiniert.
- Die Funktion wird mit einem Element des neuen Typs aufgerufen: *na-und* : $Oh \to Bool$ und $\{je = 4711\} : Oh$ ...
- ...und das Unglück nimmt seinen Lauf.
- *Konsequenz:* Typen dürfen *nicht* redefiniert werden. Aus ähnlichen Gründen sind keine lokalen Typdefinitionen erlaubt.

20. Statische Semantik

Die folgenden Typregeln setzen voraus, dass die Typdefinition

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

bekannt ist.

Typregeln:

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

$$\frac{\Sigma \vdash e : T}{\Sigma \vdash e.\ell_i : t_i}$$

☞ Ähnlich den Regeln für Paare: an die Stelle des anonymen Typs $t_1 * t_2$ tritt der benannte Typ T.

20. Dynamische Semantik

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

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

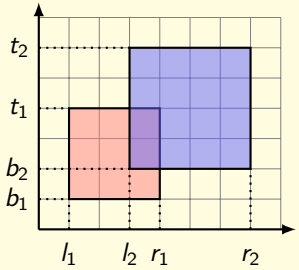
Auswertungsregeln:

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

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

20. Beispiel: Berechnung der Wohnfläche — da capo

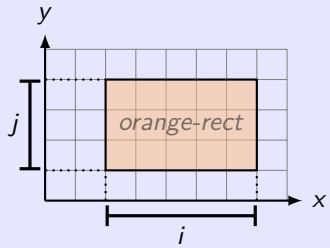
Problem: Die Fläche einer Wohnung soll berechnet werden.



Abstraktes Problem: Gesamtfläche zweier gegebener Rechtecke berechnen.

Lösung: Die Gesamtfläche zweier Rechtecke ist die Summe der Einzelflächen minus der Fläche des Durchschnitts. *Ziel:* Vokabular einführen, um die umgangssprachliche Lösung möglichst direkt umzusetzen.

20. Repräsentation der Daten




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

Repräsentation von Intervallen:

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

Repräsentation von Rechtecken:

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

☞ Viele alternative Darstellungen von Rechtecken sind denkbar. Welche? Welche Vor- und Nachteile haben die verschiedenen Darstellungen?

20. Rechnen mit Intervallen

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

Länge eines Intervalls:

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

Durchschnitt zweier Intervalle:

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

☞ Lassen sich Intervalle auch vereinigen?

20. Rechnen mit Rechtecken

$$\textbf{\textit{type}}\ \mathit{Rectangle} = \{x : \mathit{Interval}; y : \mathit{Interval}\}$$

Flächeninhalt eines Rechtecks:

$$\textbf{\textit{let}}\ \mathit{area}\ (r : \mathit{Rectangle}) = \mathit{length}\ r.x * \mathit{length}\ r.y$$

Durchschnitt zweier Rechtecke:

$$\begin{aligned}&\textbf{\textit{let}}\ \mathit{Intersection}\ (r : \mathit{Rectangle}, s : \mathit{Rectangle}) = \\ &\quad \{x = \mathit{intersection}\ (r.x, s.x); y = \mathit{intersection}\ (r.y, s.y)\}\\ &\textbf{\textit{let}}\ (\hat{}\&\&\hat{}\,)\ (r : \mathit{Rectangle})\ (s : \mathit{Rectangle}) = \\ &\quad \{x = \mathit{intersection}\ (r.x, s.x); y = \mathit{intersection}\ (r.y, s.y)\}\end{aligned}$$

☞ F# erlaubt es, eigene Infixoperatoren zu definieren: statt *Intersection* (*red-rect*, *blue-rect*) können wir kurz *red-rect* ^&&^ *blue-rect* schreiben.

20. Berechnung der Gesamtfläche

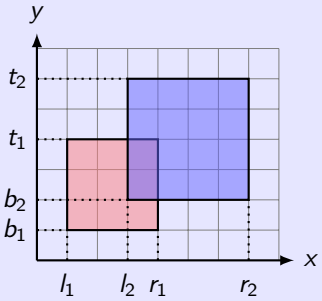
Jetzt haben wir das Vokabular zusammen, um die umgangssprachliche Lösung in Mini–F# zu transliterieren.

Gesamtfläche zweier Rechtecke:

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

☞ Man erkennt die Fortschritte, die wir erzielt haben, wenn man das obige Programm mit dem Code aus Teil III vergleicht. *Modularer Aufbau:* Das Programm besteht aus vielen kleinen Bausteinen, die separat ausprobiert, verstanden, getestet und verifiziert werden können.

20. Beispiel




```
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}
```

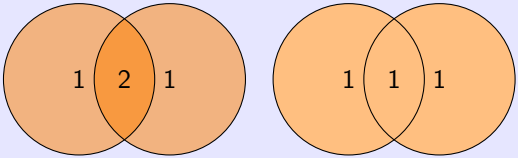
Der Durchschnitt der Quadrate ergibt das kleine, violette Rechteck in der Mitte.

```
Mini⟩ 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
```

20. Gesamtfläche von drei Rechtecken

Werden wir etwas ambitionierter: Wie können wir die Gesamtfläche von *drei* Rechtecken bestimmen?

Dem Programm *area2* liegt das *Prinzip der Einschließung und Ausschließung* zugrunde.

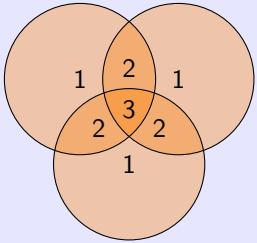


$\|A\| + \|B\|$ $\quad \ldots - \|A \cap B\|$

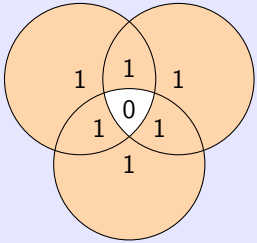
☞ $\|A\|$ ist die Fläche der Punktmenge A.

20. Gesamtfläche von drei Rechtecken

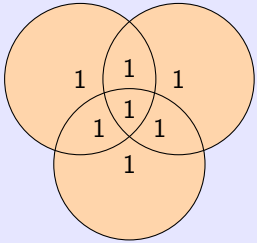
Wenn wir drei Flächen addieren, dann werden die Schnitte von zwei Flächen doppelt, der Schnitt von allen drei Flächen wird dreifach gezählt.



$\|A\| + \|B\| + \|C\|$



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



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

☞ Die Flächen werden alternierend ein- und ausgeschlossen, daher der Name des Prinzips.

20. Gesamtfläche von drei Rechtecken

Damit ergibt sich das folgende Mini–F# Programm.

```
let area3 (r : Rectangle, s : Rectangle, t : Rectangle) =
  area r + area s + area t
  - area (r ^&&^ s) - area (r ^&&^ t) - area (s ^&&^ t)
  + area (r ^&&^ s ^&&^ t)
```

☞ (Die Infixnotation bietet Vorteile, wenn drei oder mehr Rechtecke geschnitten werden. Welche?)

☞ Das Prinzip der Ein- und Ausschließung lässt sich auch anwenden, um die Gesamtfläche von vier oder mehr Rechtecken auszurechnen. Aber ist das auch eine gute Idee?

20. Fallstudie: Ganze Zahlen

Idee: wir repräsentieren eine ganze Zahl durch einen positiven und einen negativen Summanden.

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

☞ Die Bedeutung von $\{pos = p; neg = n\}$ ist $p - n$. (Hier meint '$-$' die *mathematische* Subtraktion auf den ganzen Zahlen.)

20. Ganze Zahlen — Normalisierung

Eine ganze Zahl hat viele verschiedene Repräsentationen: -4 wird zum Beispiel durch $0-4$, $5-9$ oder $4711-4715$ dargestellt.

Normalisierung einer ganzen Zahl:

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{normalize}\ (i : \mathit{Int}) : \mathit{Int} = \\
\quad \textbf{\textit{if}}\ i.\mathit{pos} \geqslant i.\mathit{neg}\ \textbf{\textit{then}} \\
\qquad \{\mathit{pos} = i.\mathit{pos} \dot{-} i.\mathit{neg};\ \mathit{neg} = 0\} \\
\quad \textbf{\textit{else}} \\
\qquad \{\mathit{pos} = 0;\ \mathit{neg} = i.\mathit{neg} \dot{-} i.\mathit{pos}\}
\end{array}
$$

Alternative Definition:

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{normalize}\ (i : \mathit{Int}) : \mathit{Int} = \\
\quad \{\mathit{pos} = i.\mathit{pos} \dot{-} i.\mathit{neg};\ \mathit{neg} = i.\mathit{neg} \dot{-} i.\mathit{pos}\}
\end{array}
$$

☞ Hier bezeichnet '$\dot{-}$' die Subtraktion auf den natürlichen Zahlen.

20. Ganze Zahlen — Klassifikation

```
let is-negative (i : Int) : Bool =
  i.pos < i.neg
let is-zero (i : Int) : Bool =
  i.pos = i.neg
let is-positive (i : Int) : Bool =
  i.pos > i.neg
```

☞ $i.pos - i.neg > 0$ gdw. $i.pos > i.neg$.

20. Ganze Zahlen — arithmetische Operationen

$$
\begin{array}{l}
\textbf{let}\ \mathit{negate}\ (i : \mathit{Int}) : \mathit{Int} = \\
\quad \{\mathit{pos} = i.\mathit{neg}; \mathit{neg} = i.\mathit{pos}\} \\
\textbf{let}\ \mathit{add}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Int} = \\
\quad \mathit{normalize}\ \{\mathit{pos} = i.\mathit{pos} + j.\mathit{pos}; \mathit{neg} = i.\mathit{neg} + j.\mathit{neg}\} \\
\textbf{let}\ \mathit{sub}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Int} = \\
\quad \mathit{normalize}\ \{\mathit{pos} = i.\mathit{pos} + j.\mathit{neg}; \mathit{neg} = i.\mathit{neg} + j.\mathit{pos}\} \\
\textbf{let}\ \mathit{mul}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Int} = \\
\quad \mathit{normalize}\ \{\mathit{pos} = i.\mathit{pos} * j.\mathit{pos} + i.\mathit{neg} * j.\mathit{neg}; \\
\qquad\qquad \mathit{neg} = i.\mathit{pos} * j.\mathit{neg} + i.\mathit{neg} * j.\mathit{pos}\}
\end{array}
$$

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

☞ *div* und *mod* zur Übung.

20. Ganze Zahlen — Konversion und Betrag

```
let int (n : Nat) : Int =
  {pos = n; neg = 0}
let abs (i : Int) : Nat =
  if i.pos < i.neg then
    i.neg ∸ i.pos
  else
    i.pos ∸ i.neg
```

Alternative Definition:

```
let abs (i : Int) : Nat =
  (i.neg ∸ i.pos) + (i.pos ∸ i.neg)
```

☞Ist die Definition wirklich korrekt?

20. Ganze Zahlen — Vergleichsoperationen

$$\textbf{let}\ \mathit{less}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = i.\mathit{pos} + j.\mathit{neg} < i.\mathit{neg} + j.\mathit{pos}$$

$$\textbf{let}\ \mathit{less\text{-}equal}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = i.\mathit{pos} + j.\mathit{neg} \leqslant i.\mathit{neg} + j.\mathit{pos}$$

$$\textbf{let}\ \mathit{equal}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = i.\mathit{pos} + j.\mathit{neg} = i.\mathit{neg} + j.\mathit{pos}$$

$$\textbf{let}\ \mathit{not\text{-}equal}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = i.\mathit{pos} + j.\mathit{neg} <> i.\mathit{neg} + j.\mathit{pos}$$

$$\textbf{let}\ \mathit{greater\text{-}equal}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = i.\mathit{pos} + j.\mathit{neg} \geqslant i.\mathit{neg} + j.\mathit{pos}$$

$$\textbf{let}\ \mathit{greater}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = i.\mathit{pos} + j.\mathit{neg} > i.\mathit{neg} + j.\mathit{pos}$$

☞ Echt kleiner: $i.\mathit{pos} - i.\mathit{neg} < j.\mathit{pos} - j.\mathit{neg}$ gdw. $i.\mathit{pos} + j.\mathit{neg} < i.\mathit{neg} + j.\mathit{pos}$.

20. Ganze Zahlen — Demo

Mini⟩ *int* 4711
{*pos* = 4711; *neg* = 0}
Mini⟩ *add* (*negate* (*int* 4711), *int* 815)
{*pos* = 0; *neg* = 3896}
Mini⟩ *add* (*int* 4711, *negate* (*int* 4711))
{*pos* = 0; *neg* = 0}
Mini⟩ *is-zero* (*add* (*int* 4711, *negate* (*int* 4711)))
true
Mini⟩ *mul* (*neg* (*int* 2), *negate* (*int* 3))
{*pos* = 6; *neg* = 0}
Mini⟩ *abs* (*mul* (*negate* (*int* 2), *int* 3))
6
Mini⟩ *div* (*negate* (*int* 4), *int* 3)
{*pos* = 0; *neg* = 2}
Mini⟩ *mod* (*negate* (*int* 4), *int* 3)
{*pos* = 2; *neg* = 0}

☞ Für $b \neq 0$ gilt weiterhin $a = (a \div b) * b + (a \,\%\, b)$. *Aber:* diese Eigenschaft legt $\div$ und % nicht länger eindeutig fest!

21. Knobelaufgabe #9: Die Magie der Zahlen

Sie sind Kandidat*in in der Spielshow „Die Magie der Zahlen“. In der ersten Runde wird ihnen eine Reihe von nummerierten Schachteln präsentiert, die jeweils eine für Sie nicht sichtbare ganze Zahl enthalten. Sie müssen mit möglichst wenigen Versuchen eine *magische Schachtel* finden, eine Schachtel, die ihre eigene Hausnummer enthält.

Die versteckten Zahlen sind alle unterschiedlich. Schachteln mit größeren Hausnummern enthalten größere Zahlen. Entwickeln Sie eine Strategie, um möglichst wenige Schachteln zu öffnen.

- Zum Beispiel:

1	2	3	4	5	6	7	8	9
−10	−5	0	4	7	11	27	65	99

- Es gibt nur eine magische Schachtel: #4.
- Gibt es immer eine magische Schachtel?

21. Motivation

Eine Person ist entweder weiblich oder männlich; eine männliche Person hat Attribute, die eine weibliche nicht hat (und vielleicht umgekehrt — aber nicht modelliert).

$$\textbf{\textit{type}}\ \mathit{Woman} = \{\mathit{name} : \mathit{String}\}$$
$$\textbf{\textit{type}}\ \mathit{Man} \quad = \{\mathit{name} : \mathit{String}; \mathit{bald} : \mathit{Bool}\}$$

Daten, die unterschiedliche Ausprägungen besitzen, können wir in Mini–F# mit sogenannten *Varianten* modellieren.

$$\begin{array}{l}\textbf{\textit{type}}\ \mathit{Person} = \\ \quad \mid \mathit{Female}\ \textbf{\textit{of}}\ \mathit{Woman} \\ \quad \mid \mathit{Male} \quad \textbf{\textit{of}}\ \mathit{Man}\end{array}$$

21. Motivation

```
type Person = | Female of Woman
              | Male   of Man
```

Eine Variantentypdefinition führt zwei verschiedene Dinge ein:

- einen Namen für den Variantentyp: *Person*,
- Namen um Elemente des Variantentyps zu konstruieren: *Female* und *Male*. Diese Bezeichner heißen auch *Datenkonstruktoren* oder kurz *Konstruktoren*.

21. Motivation

$$\textbf{\textit{type}}\ \mathit{Person} = \mid \mathit{Female}\ \textbf{\textit{of}}\ \mathit{Woman} \mid \mathit{Male}\ \ \textbf{\textit{of}}\ \mathit{Man}$$

Umgangssprachlich lässt sich die Definition wie folgt lesen: ein Element $p : \mathit{Person}$ ist entweder

- von der Form $\mathit{Female}\ e$ mit $e : \mathit{Woman}$ *oder*
- von der Form $\mathit{Male}\ e$ mit $e : \mathit{Man}$.

Ein Konstruktor ähnelt einer Funktion. Der Typ nach dem Konstruktor korrespondiert zum Argumenttyp, der deklarierte Variantentyp korrespondiert zum Ergebnistyp:

- Female hat im Prinzip den Typ $\mathit{Woman} \to \mathit{Person}$ und
- Male den Typ $\mathit{Man} \to \mathit{Person}$.

☞ Im Unterschied zu einer Funktion hat ein Konstruktor aber keine Definition; er steht sozusagen für sich selbst.

21. Motivation

Beispiele:

$$\mathit{Female}\ \{\mathit{name} = \texttt{"Lisa"}\}$$
$$\mathit{Male}\ \{\mathit{name} = \texttt{"Florian"};\ \mathit{bald} = \mathit{false}\}$$

☞ Beide Ausdrücke sind vom Typ *Person*.

Ausdrücke können wie immer an Bezeichner gebunden werden.

$$\begin{array}{lll}
\textbf{\textit{let}}\ \mathit{ralf} & = \mathit{Male} & \{\mathit{name} = \texttt{"Ralf"};\ \ \mathit{bald} = \mathit{true}\} \\
\textbf{\textit{let}}\ \mathit{melanie} & = \mathit{Female} & \{\mathit{name} = \texttt{"Melanie"}\} \\
\textbf{\textit{let}}\ \mathit{julia} & = \mathit{Female} & \{\mathit{name} = \texttt{"Julia"}\} \\
\textbf{\textit{let}}\ \mathit{andres} & = \mathit{Male} & \{\mathit{name} = \texttt{"Andres"};\ \mathit{bald} = \mathit{false}\}
\end{array}$$

21. Motivation

Ein Variantentyp beschreibt Alternativen; mit Hilfe der *Fallunterscheidung* ***match*** können wir feststellen, welche konkrete Alternative vorliegt.

```
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.

21. Variantentypen versus Baumsprachen

Variantentypen ähneln der Notation, mit der wir die abstrakte Syntax unserer Programmiersprache beschreiben.

- Baumsprachen sind Bestandteil der Sprache, mit der wir *über* die Sprache Mini–F# reden.

$$
\begin{array}{rcl}
e \in \text{Expr} & ::= & \mathit{false} \\
& | & \mathit{true} \\
& | & \mathit{num}\ (\mathbb{N}) \\
& | & \ldots
\end{array}
$$

- Variantentypen sind Bestandteil von Mini–F#.

```
type Expr = | False
            | True
            | Num of Nat
            | ...
```

☞ Fachjargon: Variantentypen *internalisieren* Baumsprachen.

21. Abstrakte Syntax

Ein Variantentyp (engl. union type) wird durch eine *Definition* eingeführt.

$C \in \text{Con}$	*Datenkonstruktoren*
$d ::= \cdots$	*Deklarationen:*
$\mid$ ***type*** $T = \mid C_1$ ***of*** t_1	Variantentypdefinition ($C_1 \neq C_2$)
$\mid C_2$ ***of*** t_2	

☞ Der Bezeichner T wird durch die Definition neu eingeführt, ebenso die Konstruktoren C_1 und C_2.

☞ Wie Recordtypen dürfen auch Variantentypen weder redefiniert noch lokal definiert werden.

☞ Der Bereich der Konstruktoren Con wird in Teil VI festgelegt.

Für's erste: Ein Konstruktor fängt mit einem *großen* Buchstaben an. Danach können weitere Buchstaben, kleine und große, Ziffern, und Sonderzeichen wie ein Unterstrich oder ein Apostroph folgen.

21. Abstrakte Syntax

Wir erweitern Ausdrücke um Sprachkonstrukte, die Varianten konstruieren bzw. analysieren.

$e ::= \cdots$	*Ausdrücke:*
$\mid\ C\ e$	Konstruktion
$\mid\ \textbf{\textit{match}}\ e\ \textbf{\textit{with}}\ \mid C_1\ x_1 \to e_1$	Fallunterscheidung ($C_1 \neq C_2$)
$\mid C_2\ x_2 \to e_2$	

☞ 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.

21. Statische Semantik

Die folgenden Typregeln setzen 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.

Typregeln:

$$\frac{\Sigma \vdash e : t_i}{\Sigma \vdash C_i\ e : T}$$

$$\frac{\Sigma \vdash e : T \qquad \Sigma, \{x_1 \mapsto t_1\} \vdash e_1 : t \qquad \Sigma, \{x_2 \mapsto t_2\} \vdash e_2 : t}{\Sigma \vdash (\textbf{\textit{match}}\ e\ \textbf{\textit{with}} \mid C_1\ x_1 \to e_1 \mid C_2\ x_2 \to e_2) : t}$$

☞ Alle Zweige der Fallunterscheidung müssen den gleichen Typ besitzen; dieser ist auch der Typ des gesamten Ausdrucks.

21. Dynamische Semantik

Konstruktoren konstruieren Werte, entsprechend müssen wir den Bereich der Werte erweitern.

$\nu ::= \cdots$	*Werte:*
$\mid\ C\ \nu$	Konstruktion$\diagdown$Injektion in einen Variantentyp

Auswertungsregeln:

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash C\ e \Downarrow C\ \nu}$$

$$\frac{\delta \vdash e \Downarrow C_i\ \nu_i \qquad \delta, \{x_i \mapsto \nu_i\} \vdash e_i \Downarrow \nu}{\delta \vdash (\textbf{\textit{match}}\ e\ \textbf{\textit{with}} \mid C_1\ x_1 \to e_1 \mid C_2\ x_2 \to e_2) \Downarrow \nu}$$

21. Varianten mit n Alternativen

Alle Konstrukte verallgemeinern sich in natürlicher Weise auf Varianten mit n Alternativen.

- $n = 0$:
  - keine Alternative: leerer Typ;

    type *Empty* = |

  - Fallunterscheidung ohne Zweige: ***match*** *e* ***with***;
  - die leere Fallunterscheidung signalisiert *toten Code*:

    you-cannot-call-me (*x* : *Empty*) : *Nat* = ***match*** *x* ***with***

  ☞ F# kennt keine leeren Varianten.

21. Varianten mit n Alternativen

- $n = 1$:
  - eine Alternative:

```
type Price    = | Cent of Nat
type Postcode = | Code of Nat
```

  - Fallunterscheidung hat genau einen Zweig:

```
let double (price : Price) : Price =
  match price with | Price n → Price (2 ∗ n)
```

  - 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.
  - Auch lässt sich ein Preis p nicht mit $2 * p$ verdoppeln. Zu diesem Zweck muss *double* verwendet werden.
  - Der Gewinn an Sicherheit wird mit einem Verlust an Bequemlichkeit erkauft.
- $n = 3$:
  - drei Alternativen und Fallunterscheidungen;
- ...

21. Vertiefung

Der Typ *Bool* kann durch einen Variantentyp implementiert werden.

type *Bool* = | *False* ***of*** *Unit* | *True* ***of*** *Unit*

- *false* wird durch *False* () repräsentiert,
- *true* wird durch *True* () repräsentiert,
- ***if*** e_1 ***then*** e_2 ***else*** e_3 wird durch die Fallunterscheidung ***match*** e_1 ***with*** | *False* () $\rightarrow e_3$ | *True* () $\rightarrow e_2$ realisiert.

„Nullstellige“ Konstruktoren wie *False* oder *True* sind relativ häufig. Aus diesem Grund erlauben wir, das Dummyargument auch wegzulassen.

21. Fallstudie: Ganze Zahlen — da capo

Idee: eine ganze Zahl wird durch das Vorzeichen und den Betrag repräsentiert.

```
type Int = | Neg of Nat | Pos of Nat
```

☞ *Neg n* bezeichnet die Zahl $-n$; *Pos n* entsprechend die Zahl $+n$.

☞ *Invariante:* 0 wird durch *Pos* 0 repräsentiert; für *Neg n* gilt stets $n > 0$.

Cleverer Konstruktor:

```
let neg (n : Nat) : Int =
  if n = 0 then Pos 0 else Neg n
```

☞ *neg* etabliert die Invariante.

21. Ganze Zahlen — Klassifikation

```
let is-negative (i : Int) : Bool =
  match i with
    | Neg n → true
    | Pos n → false
let is-zero (i : Int) : Bool =
  match i with
    | Neg n → false
    | Pos n → n = 0
let is-positive (i : Int) : Bool =
  match i with
    | Neg n → false
    | Pos n → true
```

21. Ganze Zahlen — arithmetische Operationen

```
let negate (i : Int) : Int =
  match i with
    | Neg n → Pos n
    | Pos n → neg n

let add (i : Int, j : Int) : Int =
  match (i, j) with
    | (Neg m, Neg n) → Neg (m + n)
    | (Neg m, Pos n) → if m ⩽ n then Pos (n ∸ m)
                                else Neg (m ∸ n)
    | (Pos m, Neg n) → if m < n then Neg (n ∸ m)
                                else Pos (m ∸ n)
    | (Pos m, Pos n) → Pos (m + n)

let sub (i : Int, j : Int) : Int =
  add (i, negate j)
```

21. Ganze Zahlen — arithmetische Operationen

```
let mul (i : Int, j : Int) : Int =
  match (i, j) with
      | (Neg m, Neg n) → Pos (m * n)
      | (Neg m, Pos n) → neg (m * n)
      | (Pos m, Neg n) → neg (m * n)
      | (Pos m, Pos n) → Pos (m * n)
```

☞ *mul* implementiert die Vorzeichenregel: $- * - = +$ etc.

☞ *div* und *mod* zur Übung.

21. Ganze Zahlen — Konversion und Betrag

```
let int (n : Nat) : Int = Pos n

let abs (i : Int) : Nat =
  match i with
    | Neg n → n
    | Pos n → n
```

21. Ganze Zahlen — Vergleichsoperationen

```
let less (i : Int, j : Int) : Bool =
  match (i, j) with
    | (Neg m, Neg n) → m > n
    | (Neg m, Pos n) → true
    | (Pos m, Neg n) → false
    | (Pos m, Pos n) → m < n

let less-equal (i : Int, j : Int) : Bool =
  match (i, j) with
    | (Neg m, Neg n) → m ⩾ n
    | (Neg m, Pos n) → true
    | (Pos m, Neg n) → false
    | (Pos m, Pos n) → m ⩽ n

let equal (i : Int, j : Int) : Bool =
  match (i, j) with
    | (Neg m, Neg n) → m = n
    | (Neg m, Pos n) → false
    | (Pos m, Neg n) → false
    | (Pos m, Pos n) → m = n
```

21. Ganze Zahlen — Vergleichsoperationen

```
let not-equal (i : Int, j : Int) : Bool =
  match (i, j) with
    | (Neg m, Neg n) → m <> n
    | (Neg m, Pos n) → true
    | (Pos m, Neg n) → true
    | (Pos m, Pos n) → m <> n

let greater-equal (i : Int, j : Int) : Bool =
  match (i, j) with
    | (Neg m, Neg n) → m ⩽ n
    | (Neg m, Pos n) → false
    | (Pos m, Neg n) → true
    | (Pos m, Pos n) → m ⩾ n

let greater (i : Int, j : Int) : Bool =
  match (i, j) with
    | (Neg m, Neg n) → m < n
    | (Neg m, Pos n) → false
    | (Pos m, Neg n) → true
    | (Pos m, Pos n) → m > n
```

21. Ganze Zahlen — Demo

```
Mini⟩ int 4711
Pos 4711
Mini⟩ add (negate (int 4711), int 815)
Neg 3896
Mini⟩ add (int 4711, negate (int 4711))
Pos 0
Mini⟩ is-zero (add (int 4711, negate (int 4711)))
true
Mini⟩ mul (negate (int 2), negate (int 3))
Pos 6
Mini⟩ abs (mul (negate (int 2), int 3))
6
Mini⟩ div (negate (int 4), int 3)
Neg 2
Mini⟩ mod (negate (int 4), int 3)
Pos 2
```

☞ Für $b \neq 0$ gilt weiterhin $a = (a \div b) * b + (a \,\%\, b)$. *Aber:* die Eigenschaft legt $\div$ und % nicht länger eindeutig fest, siehe Übung.

21. Summen und Produkte

Ist t ein endlicher Typ, so bezeichnet $|t|$ die Anzahl der Elemente von t, die *Kardinalität* von t.

1. Der Paartyp $t_1 * t_2$ korrespondiert zu einem *Produkt*, da

   $$|t_1 * t_2| = |t_1| * |t_2|$$

2. Der Typ *Unit* korrespondiert zu der 1, da

   $$|\mathit{Unit}| = 1$$

3. Der Variantentyp T mit ***type*** $T =$ ∣ C_1 ***of*** t_1 ∣ C_2 ***of*** t_2 korrespondiert zu einer *Summe*, da

   $$|T| = |t_1| + |t_2|$$

4. Der Variantentyp *Empty* mit ***type*** *Empty* $=$ ∣ korrespondiert zu der 0, da

   $$|\mathit{Empty}| = 0$$

☞ Und Funktionen?

21. Summen und Produkte — Eigenschaften

Für Paar- und Variantentypen gelten ähnliche Gesetze wie für die natürlichen Zahlen. Zum Beispiel:

$$0 + t \cong t \cong t + 0$$

$$0 \times t \cong 0 \cong t \times 0$$

$$1 \times t \cong t \cong t \times 1$$

$$t_1 \times (t_2 + \tau_3) \cong (t_1 \times t_2) + (t_1 \times \tau_3)$$

$$(t_1 + t_2) \times \tau_3 \cong (t_1 \times \tau_3) + (t_2 \times \tau_3)$$

☞ Im Unterschied zu den natürlichen Zahlen sind die beiden oder die drei Seiten nicht gleich, sondern nur *isomorph*: $t_1 \cong t_2$ bedeutet, dass sich jedes Element aus t_1 eineindeutig einem Element aus t_2 zuordnen lässt.

☞ Und Funktionen?

21. Rechnen mit Typen

Der Typ *Person* in arithmetischer Notation:

$$\mathit{Person} \cong \mathit{String} + (\mathit{String} \times \mathit{Bool})$$

Jetzt können wir rechnen:

$$\begin{aligned} & \mathit{String} + (\mathit{String} \times \mathit{Bool}) \\ \cong \quad & \{\ 1 \text{ ist das neutrale Element von } `\times'\ \} \\ & (\mathit{String} \times 1) + (\mathit{String} \times \mathit{Bool}) \\ \cong \quad & \{\ \text{Distributivgesetz}\ \} \\ & \mathit{String} \times (1 + \mathit{Bool}) \end{aligned}$$

☞ Der Typ $\mathit{String} \times (1 + \mathit{Bool})$ ist eine alternative Implementierung von *Person*.

☞ Für das Rechnen mit Typen (!) ist die arithmetische Notation sehr bequem — es ist aber nur eine Notation!

21. Rechnen mit Typen

Übersetzen wir den Typ $String \times (1 + Bool)$ in Mini–F# Notation, so erhalten wir

$$\textbf{\textit{type}}\ Person' = \{name : String; gender : Gender\}$$
$$\textbf{\textit{type}}\ Gender = Female' \mid Male'\ \textbf{\textit{of}}\ \{bald : Bool\}$$

☞ Das Geschlecht umfasst die trennenden Merkmale, die gemeinsamen sind in $Person'$ zusammengefasst.

22. Motivation

- 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 Typen sind ungeeignet um 6, 8, 127 oder 129 Daten aufzunehmen.
- Zum Zeitpunkt des Programmierens kennt man häufig die genaue Anzahl von Daten nicht:
  - Wieviele Personen immatrikulieren sich im WS 2019/2020?
  - Wieviele Unternehmen sind an der Börse notiert?
  - Wieviele Mitarbeiter*innen hat eine Abteilung?
  - usw.
- Wie können wir eine beliebige Anzahl von Daten aggregieren?

22. Motivation

Konkret: wie können wir eine Folge von natürlichen Zahlen repräsentieren?

Ein erster Versuch (in arithmetischer Notation):

$$
\begin{aligned}
\mathit{Nats} \cong\ & 1 \\
& + \mathit{Nat} \\
& + \mathit{Nat} \times \mathit{Nat} \\
& + \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \\
& + \ldots
\end{aligned}
$$

☞ Eine Folge von natürlichen Zahlen ist entweder die leere Folge (ein 0-Tupel), oder eine einelementige Folge (ein „1-Tupel“), oder eine zweielementige Folge (ein 2-Tupel) usw.

22. Motivation

Beobachtung: alle Alternativen bis auf die erste haben eine *Nat* Komponente. Der gemeinsame Faktor kann „herausgezogen“ werden.

$$
\begin{aligned}
\mathit{Nats} \cong\ & 1 \\
& + \mathit{Nat} \times (\ 1 \\
& \qquad + \mathit{Nat} \\
& \qquad + \mathit{Nat} \times \mathit{Nat} \\
& \qquad + \ldots)
\end{aligned}
$$

22. Motivation

Der Typausdruck in Klammern entspricht der ursprünglichen Definition von *Nats*.

$$Nats \cong 1 + Nat \times Nats$$

Erlauben wir bei der Definition eines Typs den Rückgriff auf den definierten Typ selbst (!), erhalten wir (in Mini–F# Notation):

$$\textbf{\textit{type}}\ Nats = \ |\ Nil\ |\ Cons\ \textbf{\textit{of}}\ Nat * Nats$$

☞ *Zur Erinnerung:* Greift man bei der Definition auf das definierte Objekt selbst zurück, spricht man von einer *rekursiven* Definition.

22. Listen

type *Nats* = | *Nil* | *Cons* ***of*** *Nat* ∗ *Nats*

☞ 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.

- *Nil* ist eine Verkürzung des lateinischen Wortes *nihil* für „nichts“.
- *Cons* kürzt das englische Wort *construct* ab.
- Statt von einer Folge von natürlichen Zahlen spricht man auch kurz von einer *Liste*:
  - n ist das *Kopfelement* der Liste *Cons* (n, ns),
  - ns ist die *Restliste* der Liste *Cons* (n, ns).
- Bei Listen ist wie bei Tupeln die Reihenfolge der Elemente signifikant.
- Listen sind die erste und einfachste *Datenstruktur*, die uns begegnet. Eine Datenstruktur verwaltet Daten und unterstützt Zugriff und Manipulation dieser Daten.

22. Listen — Programmierung

Wie gehen wir mit einem rekursiven Variantentyp um? Wir konstruieren und analysieren rekursive Varianten mit Hilfe rekursiver Funktionen!

Beispiel: Sortieren von Listen. Der Variantentyp gibt das folgende Skelett für *sort* vor.

```
let rec sort (nats : Nats) : Nats =
  match nats with
    | Nil          → ...
    | Cons (n, ns) → ...
```

☞ Wie füllen wir den *Cons* Zweig mit Leben?

22. Sortieren

Getreu dem Motto „rekursive Funktionen für rekursive Typen“ erlauben wir, *sort* auf die Restliste *ns* anzuwenden.

```
let rec sort (nats : Nats) : Nats =
  match nats with
    | Nil          → ...
    | Cons (n, ns) → ... sort ns ...
```

22. Sortieren

- *Rekursionsbasis*: die leere Liste ist bereits geordnet.

```
let rec sort (nats : Nats) : Nats =
  match nats with
    | Nil          → Nil
    | Cons (n, ns) → ... sort ns ...
```

- *Rekursionsschritt*: *sort ns* ist eine geordnete Liste; wir müssen das Element *n* an die richtige Stelle einordnen. Wir geben dieser Teilaufgabe einen Namen: *insert*.

```
let rec sort (nats : Nats) : Nats =
  match nats with
    | Nil          → Nil
    | Cons (n, ns) → insert (n, sort ns)
```

☞ Wenn sich ein Problem nicht mit dem bisherigen Vokabular lösen lässt, müssen wir das Vokabular erweitern. An dieser Stelle ist Kreativität gefragt!

22. Einfügen in eine geordnete Liste

Die Definition von *insert* gehen wir auf die gleiche Art und Weise an.

```
let rec insert (nat : Nat, nats : Nats) : Nats =
  match nats with
    | Nil          → ...
    | Cons (n, ns) → ... insert (nat, ns) ...
```

Vorbedingung: nats ist bereits geordnet. Das Typsystem stellt diese Eigenschaft nicht sicher, darum müssen wir uns kümmern.

22. Einfügen in eine geordnete Liste

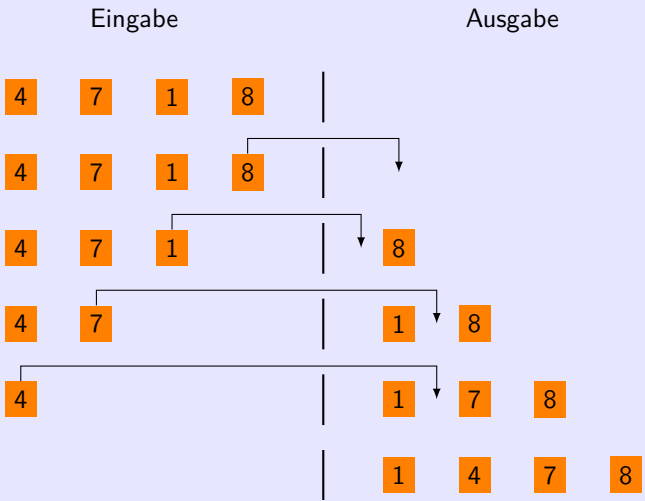
- *Rekursionsbasis*: Ist die Liste leer, so geben wir die einelementige Liste $Cons\ (nat, Nil)$ zurück.

```
let rec insert (nat : Nat, nats : Nats) : Nats =
  match nats with
    | Nil          → Cons (nat, Nil)
    | Cons (n, ns) → ...
```

- *Rekursionsschritt*: gilt $nat \leqslant n$, so stellen wir nat an den Anfang der Liste; anderenfalls wird nat in die Restliste ns eingefügt.

```
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))
```

22. Sortieren durch Einfügen



22. Sortieren — Verallgemeinerung

Die Funktion *sort* sortiert eine Liste aufsteigend. Was machen wir, wenn wir die Liste absteigend ordnen wollen?

- Programmcode duplizieren und '$\leqslant$' systematisch durch '$\geqslant$' ersetzen. Unökonomisch!
- Aufsteigend sortieren und das Ergebnis umdrehen, siehe Skript.
- Wir verallgemeinern die Aufgabenstellung und abstrahieren von einer speziellen Ordnungsrelation.

$$\mathit{sort\text{-}by}\ (\mathit{less\text{-}equal} : \mathit{Nat} * \mathit{Nat} \to \mathit{Bool}) : \mathit{Nats} \to \mathit{Nats}$$

22. Sortieren — Verallgemeinerung

Sortieren durch Einfügen

```
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
```

☞ die Definition von *sort-by* verwendet Layout (Abseitsregel).

22. Sortieren — Verallgemeinerung

Die ursprünglichen Sortierfunktionen sind jetzt hausbackene Spezialfälle:

$$\textbf{\textit{let}}\ \textit{increasing-sort} = \textit{sort-by}\ (\textbf{\textit{fun}}\ (m, n) \to m \leqslant n)$$
$$\textbf{\textit{let}}\ \textit{decreasing-sort} = \textit{sort-by}\ (\textbf{\textit{fun}}\ (m, n) \to m \geqslant n)$$

☞ 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.

22. Struktur Entwurfsmuster für Listen

Haben wir die Aufgabe eine Funktion $f : Nats \to t$ zu erstellen, dann sieht ein erster Entwurf folgendermaßen aus.

```
let rec f (nats : Nats) : t =                    Struktur Entwurfsmuster
  match nats with
    | Nil          → ...                         Rekursionsbasis
    | Cons (n, ns) → ... n ... f ns ...          Rekursionsschritt
```

Die Ellipsen müssen mit Leben gefüllt werden:

- *Rekursionsbasis*: ein Ausdruck des Typs t.
- *Rekursionsschritt*: ein Ausdruck, der die Teillösung $f\ ns$ vom Typ t zu einer Gesamtlösung vom Typ t erweitert.

22. Summe einer Liste von Zahlen

Aufgabe: sum nats soll die Elemente der Liste *nats* aufaddieren (die Verallgemeinerung von $+$ auf Listen).

```
let rec sum (nats : Nats) : Nat =
  match nats with
    | Nil          → ...
    | Cons (n, ns) → ... sum ns ...
```

22. Summe einer Liste von Zahlen

- *Rekursionsbasis*: $sum\ Nil = 0$. Warum?

```
let rec sum (nats : Nats) : Nat =
  match nats with
    | Nil          → 0
    | Cons (n, ns) → ... sum ns ...
```

- *Rekursionsschritt*: Wir addieren n zur Summe der Restliste.

```
let rec sum (nats : Nats) : Nat =
  match nats with
    | Nil          → 0
    | Cons (n, ns) → n + sum ns
```

22. Produkt einer Liste von Zahlen

Aufgabe: product nats soll die Elemente der Liste *nats* miteinander multiplizieren (die Verallgemeinerung von $*$ auf Listen).

```
let rec product (nats : Nats) : Nat =
  match nats with
    | Nil          → ...
    | Cons (n, ns) → ... product ns ...
```

22. Produkt einer Liste von Zahlen

- *Rekursionsbasis*: $product\ Nil = 1$. Warum?

```
let rec product (nats : Nats) : Nat =
  match nats with
    | Nil          → 1
    | Cons (n, ns) → ... product ns ...
```

- *Rekursionsschritt*: Wir multiplizieren n mit dem Produkt der Restliste.

```
let rec product (nats : Nats) : Nat =
  match nats with
    | Nil          → 1
    | Cons (n, ns) → n * product ns
```

22. Konstruktion von Listen

Die bisherigen Funktionen verarbeiten Listen; wie können wir Listen erzeugen?

Beispiel: between (l, u) erzeugt die Liste aller Elemente in dem gegebenen Intervall.

Wir wenden das Peano Entwurfsmuster auf die Intervallgröße an.

```
let rec between (l : Nat, u : Nat) : Nats =
  if l > u then Nil
           else Cons (l, between (l + 1, u))
```

☞ Im Basisfall geben wir die leere Liste zurück; im Rekursionsfall setzen wir die linke Intervallgrenze vor die rekursiv erzeugte Liste.

22. Alte Funktionen neu

Wir haben unser Vokabular beträchtlich erweitert. Mit den neuen Vokabeln können wir zum Beispiel *factorial* kürzer definieren.

let *factorial* $(n : \mathit{Nat}) : \mathit{Nat} = \mathit{product}\ (\mathit{between}\ (1, n))$

☞ Wie lässt sich *power* auf *product* zurückführen?

22. Knobelaufgabe #10

Ministerpräsident Sörkus Mader möchte im Rahmen seines Raumfahrtprogramms „Failaria One“ einen Roboter auf dem Mars absetzen. Dazu wird der Roboter aus der Trägerrakete auf den Planeten fallen gelassen. Hierbei ist die Fallhöhe von entscheidender Bedeutung: Wird sie zu hoch angesetzt, dann zerschellt der Roboter an der Oberfläche des Planeten. Eine zu niedrige Höhe birgt das Risiko, dass die Trägerrakete dem Planeten zu nahe kommt und explodiert.

In der Forschungsstation „Dahoam“ wurde ein Turm errichtet, der die Fallbedingungen exakt nachstellt. Es stehen zwei Imitate des Roboters zur Verfügung, die die gleichen Aufpralleigenschaften wie der echte Roboter aufweisen. Das Roboter-Imitat wird jeweils aus einem der n Stockwerke des Versuch-Turms fallen gelassen, um das höchste Stockwerk zu ermitteln, bei dem der Roboter unversehrt am Boden ankommt. Da jeder Versuch mit hohen Kosten für den Steuerzahler verbunden ist, soll die Anzahl der Fall-Versuche minimiert werden. Allerdings stehen auch nur die genannten beiden Imitate zur Verfügung — beim Fall aus zu großer Höhe werden sie zerstört und sind unbrauchbar.

Helfen Sie Sörkus eine Strategie zu entwickeln, um das höchste Stockwerk, aus dem der Roboter unversehrt fallen gelassen werden kann, exakt zu ermitteln bei minimaler Anzahl von Fall-Versuchen.

22. Vertiefung

Der Typ *Bool* kann mit einem Variantentyp implementiert werden; *Nat* lässt sich mit einem rekursiven Variantentyp implementieren!

Eine natürliche Zahl ist entweder 0 oder der Nachfolger einer natürlichen Zahl $(n + 1)$.

Denken wir uns Namen für die 0 und die Nachfolgerfunktion aus.

```
type Peano =
    | Zero
    | Succ of Peano
```

22. Natürliche Zahlen — Peano

Wir zählen

Zero
Succ Zero
Succ (*Succ Zero*)
Succ (*Succ* (*Succ Zero*))
Succ (*Succ* (*Succ* (*Succ Zero*)))
...

☞ Die Zahlendarstellung unserer Urahnen: für jedes erlegte Bison ein Strich.

22. Addition — Peano

Die Addition lässt sich auf die Nachfolgerfunktion zurückführen.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{add}\ (m : \mathit{Peano}, n : \mathit{Peano}) : \mathit{Peano} = \\
\quad \textbf{\textit{match}}\ m\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Zero} \quad\ \to \ldots \\
\qquad \mid \mathit{Succ}\ m' \to \ldots \mathit{add}\ (m', n) \ldots
\end{array}
$$

22. Addition — Peano

- *Rekursionsbasis*: $0 + n = n$.

```
let rec add (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → n
    | Succ m' → ... add (m', n) ...
```

- *Rekursionsschritt*: $(m' + 1) + n = (m' + n) + 1$.

```
let rec add (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → n
    | Succ m' → Succ (add (m', n))
```

22. Multiplikation — Peano

Die Multiplikation lässt sich auf die Addition zurückführen.

```
let rec mul (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → ...
    | Succ m' → ... mul (m', n) ...
```

22. Multiplikation — Peano

- *Rekursionsbasis*: $0 * n = 0$.

```
let rec mul (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → Zero
    | Succ m' → ... mul (m', n) ...
```

- *Rekursionsschritt*: $(m' + 1) * n = m' * n + n$.

```
let rec mul (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → Zero
    | Succ m' → add (mul (m', n), n)
```

22. Peano Entwurfsmuster — da capo

Das Peano Entwurfsmuster entspricht dem Struktur Entwurfsmuster für den Typ *Peano*.

Peano Entwurfsmuster:

```
let rec f (n : Nat) : t =          Peano Entwurfsmuster:
  if n = 0
    then ...                       Rekursionsbasis
    else ... f (n ÷ 1) ...         Rekursionsschritt
```

Struktur Entwurfsmuster für den Typ *Peano*:

```
let rec f (n : Peano) : t =        Struktur Entwurfsmuster:
  match n with
    | Zero    → ...                Rekursionsbasis
    | Succ n' → ... f n' ...       Rekursionsschritt
```

22. Natürliche Zahlen — Leibniz

Eine natürliche Zahl ist entweder 0, gerade $(2n)$ oder ungerade $(2n+1)$.

Die Fälle sind aber nicht exklusiv: 0 ist auch eine gerade Zahl.

Na ja, wie ich das verstanden habe, brauchen wir die 0 als Rekursionsbasis.

Ich will nicht die 0 streichen, sondern gerade Zahlen auf $n \geqslant 2$ beschränken $(2n+2)$.

Vom höchsten Ordnungssinn ist es nur ein Schritt zur Pedanterie. — Eine natürliche Zahl ist entweder 0, ungerade $(2n+1)$ oder positiv gerade $(2n+2)$.

Nur als Warnung: wir weichen an dieser Stelle vom Skript ab.

22. Natürliche Zahlen — Leibniz

Denken wir uns Namen für die 0, ungerade Zahlen $(2n + 1)$ und gerade Zahlen größer als 0 $(2n + 2)$ aus.

```
type Leibniz =
    | Null
    | Odd of Leibniz
    | Even of Leibniz
```

22. Natürliche Zahlen — Leibniz

Wir zählen

Null
Odd Null
Even Null
Odd (*Odd Null*)
Even (*Odd Null*)
Odd (*Even Null*)
Even (*Even Null*)
Odd (*Odd* (*Odd Null*))
Even (*Odd* (*Odd Null*))
Odd (*Even* (*Odd Null*))
...

22. Nachfolgerfunktion — Leibniz

Um das Bildungsgesetz zu sehen, ist es am einfachsten, wenn wir die Nachfolgerfunktion programmieren.

```
let rec succ (n : Leibniz) : Leibniz =
  match n with
    | Null     → ...
    | Odd  n'  → ... succ n' ...
    | Even n'  → ... succ n' ...
```

- *Rekursionsbasis*: $0 + 1 = 1$.

```
let rec succ (n : Leibniz) : Leibniz =
  match n with
    | Null     → Odd Null
    | Odd  n'  → ... succ n' ...
    | Even n'  → ... succ n' ...
```

22. Nachfolgerfunktion — Leibniz

- *Rekursionsschritt*: $(2 * n' + 1) + 1 = 2 * n' + 2$.

```
let rec succ (n : Leibniz) : Leibniz =
  match n with
    | Null    → Odd Null
    | Odd  n' → Even n'
    | Even n' → ... succ n' ...
```

- *Rekursionsschritt*: $(2 * n' + 2) + 1 = 2 * (n' + 1) + 1$.

```
let rec succ (n : Leibniz) : Leibniz =
  match n with
    | Null    → Odd Null
    | Odd  n' → Even n'
    | Even n' → Odd (succ n')
```

22. Addition — Leibniz

Addition:

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
    | Null      → ...
    | Odd  m'   → ... add (m', n) ...
    | Even m'   → ... add (m', n) ...
```

- *Rekursionsbasis*: $0 + n = n$.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
    | Null      → n
    | Odd  m'   → ... add (m', n) ...
    | Even m'   → ... add (m', n) ...
```

22. Addition — Leibniz

- *Rekursionsschritt*: Können wir die Summe berechnen, ohne n zu kennen? Nein, wir müssen eine geschachtelte Fallunterscheidung über n vornehmen.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m' → match n with
                | Null    → ...
                | Odd  n' → ... add (m', n') ...
                | Even n' → ... add (m', n') ...
  | Even m' → match n with
                | Null → ...
                | Odd  n' → ... add (m', n') ...
                | Even n' → ... add (m', n') ...
```

22. Addition — Leibniz

- Fall $n = \mathit{Null}$: $m + 0 = m$.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m'   → match n with
                | Null     → m
                | Odd  n'  → ... add (m', n') ...
                | Even n'  → ... add (m', n') ...
  | Even m'   → match n with
                | Null     → m
                | Odd  n'  → ... add (m', n') ...
                | Even n'  → ... add (m', n') ...
```

22. Addition — Leibniz

- Fall $m = \mathit{Odd}\ m'$ und $n = \mathit{Odd}\ n'$: $(2m' + 1) + (2n' + 1) = 2(m' + n') + 2$.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m'   → match n with
                | Null      → m
                | Odd  n'   → Even (add (m', n'))
                | Even n'   → ... add (m', n') ...
  | Even m'   → match n with
                | Null      → m
                | Odd  n'   → ... add (m', n') ...
                | Even n'   → ... add (m', n') ...
```

22. Addition — Leibniz

- Fall $m = \mathit{Odd}\ m'$ und $n = \mathit{Even}\ n'$: $(2m' + 1) + (2n' + 2) = 2((m' + n') + 1) + 1$.
- Fall $m = \mathit{Even}\ m'$ und $n = \mathit{Odd}\ n'$: analog.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m'   → match n with
                | Null      → m
                | Odd  n'   → Even (add (m', n'))
                | Even n'   → Odd (succ (add (m', n')))
  | Even m'   → match n with
                | Null      → m
                | Odd  n'   → Odd (succ (add (m', n')))
                | Even n'   → ... add (m', n') ...
```

22. Addition — Leibniz

- Fall $m = \mathit{Even}\ m'$ und $n = \mathit{Even}\ n'$: $(2m' + 2) + (2n' + 2) = 2((m' + n') + 1) + 2$.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m'   → match n with
                | Null      → m
                | Odd  n'   → Even (add (m', n'))
                | Even n'   → Odd (succ (add (m', n')))
  | Even m'   → match n with
                | Null      → m
                | Odd  n'   → Odd (succ (add (m', n')))
                | Even n'   → Even (succ (add (m', n')))
```

22. Leibniz Entwurfsmuster — da capo

Das Leibniz Entwurfsmuster entspricht dem Struktur Entwurfsmuster für den Typ *Leibniz*.

Leibniz Entwurfsmuster:

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

Struktur Entwurfsmuster für den Typ *Leibniz*:

```
let rec f (n : Leibniz) : t =    Struktur Entwurfsmuster:
  match n with
    | Null    → ...              Rekursionsbasis
    | Odd  n' → ... f n' ...     Rekursionsschritt
    | Even n' → ... f n' ...     Rekursionsschritt
```

22. Dualsystem

Der Typ *Leibniz* implementiert übrigens das Binär- oder *Dualsystem*. Der Zusammenhang wird deutlich, wenn wir den Typ etwas umschreiben.

$$
\begin{aligned}
& \mathit{Leibniz} \\
\cong \quad & \{ \text{ Definition von } \mathit{Leibniz} \ \} \\
& 1 + \mathit{Leibniz} + \mathit{Leibniz} \\
\cong \quad & \{ \ 1 \text{ ist das neutrale Element von } `\times' \ \} \\
& 1 + (1 \times \mathit{Leibniz}) + (1 \times \mathit{Leibniz}) \\
\cong \quad & \{ \text{ Distributivgesetz } \} \\
& 1 + (1 + 1) \times \mathit{Leibniz}
\end{aligned}
$$

Damit haben wir eine alternative Definition von *Leibniz*:

$$
\begin{aligned}
&\textbf{\textit{type}}\ \mathit{Bit} \ = \ |\ \mathit{One} \ |\ \mathit{Two} \\
&\textbf{\textit{type}}\ \mathit{Bits} = \ |\ \mathit{Nil} \ |\ \mathit{Cons}\ \textbf{\textit{of}}\ \mathit{Bit} * \mathit{Bits}
\end{aligned}
$$

23. Motivation

- Die Zweige einer Fallunterscheidung binden Bezeichner:

  $\mathit{Female}\ f \to \ldots f.\mathit{name} \ldots$

- Wie bei Wertedefinitionen können wir anstelle eines einzelnen Bezeichners auch ein Muster zulassen.

  $\mathit{Female}\ \{\mathit{name} = n\} \to \ldots n \ldots$

- Wir können auch die gesamte linke Seite als Muster lesen: $\mathit{Female}\ p$ ist ein Muster, auf das ein Wert der Form $\mathit{Female}\ \nu$ passt, sofern ν auf p passt.
- Der Musterabgleich kann auch fehlschlagen: der Wert $\mathit{Male}\ \nu$ passt nicht auf das Muster $\mathit{Female}\ x$.
- Man sagt auch, das Muster $\mathit{Female}\ x$ ist *widerlegbar*.
- Die Fallunterscheidung lässt sich als sukzessives Durchprobieren von Mustern deuten.

23. Abstrakte Syntax

Wir erweitern die Syntax von Fallunterscheidungen.

$e ::= \cdots$	*Ausdrücke:*
$\mid$ ***match*** e ***with*** m	erweiterte Fallunterscheidung

Der Rumpf einer Fallunterscheidung ist eine Folge von sogenannten *Regeln* der Form $p \to e$.

$m ::= p \to e$	Regel
$\mid \; m_1 \; ! \; m_2$	Sequenz von Regeln

23. Abstrakte Syntax

Muster werden um disjunkte Muster und Konstruktoranwendungen erweitert.

$p ::= \cdots$	*Muster:*
$\quad \mid \; p_1 \mid p_2$	disjunktives Muster
$\quad \mid \; C\ p$	Konstruktoranwendung

☞ Statische Semantik, siehe Skript/zur Übung.

23. Dynamische Semantik

Beispiele: Wir betrachten die Fallunterscheidung ***match*** e ***with*** $p \to \ldots$ und nehmen an, dass e zu ν auswertet.

p	ν	Umgebung
Nil	Nil	$\varnothing$
Nil	$\mathit{Cons}\ (1, \mathit{Nil})$	$\lightning$
$\mathit{Cons}\ (a, x)$	Nil	$\lightning$
$\mathit{Cons}\ (a, x)$	$\mathit{Cons}\ (1, \mathit{Nil})$	$\{a \mapsto 1, x \mapsto \mathit{Nil}\}$
$\mathit{Cons}\ (a, \mathit{Cons}\ (b, x))$	Nil	$\lightning$
$\mathit{Cons}\ (a, \mathit{Cons}\ (b, x))$	$\mathit{Cons}\ (1, \mathit{Nil})$	$\lightning$
$\mathit{Cons}\ (a, \mathit{Cons}\ (b, x))$	$\mathit{Cons}\ (1, \mathit{Cons}\ (2, \mathit{Nil}))$	$\{a \mapsto 1, b \mapsto 2, x \mapsto \mathit{Nil}\}$

☞ Der Blitz signalisiert, dass das Muster *nicht* auf den Wert passt.

☞ Dynamische Semantik, siehe Skript/zur Übung.

23. Vertiefung

Die Addition lässt sich mit Hilfe geschachtelter Muster sehr viel natürlicher formulieren.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{add}'\ (m : \mathit{Leibniz}, n : \mathit{Leibniz}) : \mathit{Leibniz} = \\
\quad \textbf{\textit{match}}\ (m, n)\ \textbf{\textit{with}} \\
\qquad \mid (\mathit{Null}, \quad k \quad) \\
\qquad \mid (k, \quad \mathit{Null} \quad) \to k \\
\qquad \mid (\mathit{Odd}\ \ m', \mathit{Odd}\ \ n') \to \mathit{Even}\ (\mathit{add}'\ (m', n')) \\
\qquad \mid (\mathit{Odd}\ \ m', \mathit{Even}\ n') \\
\qquad \mid (\mathit{Even}\ m', \mathit{Odd}\ \ n') \to \mathit{Odd}\ \ (\mathit{succ}\ (\mathit{add}'\ (m', n'))) \\
\qquad \mid (\mathit{Even}\ m', \mathit{Even}\ n') \to \mathit{Even}\ (\mathit{succ}\ (\mathit{add}'\ (m', n')))
\end{array}
$$

☞ Wir unterscheiden vier Fälle: ein Argument ist Null, beide Argumente sind ungerade, ein Argument ist ungerade, beide sind gerade.

24. Motivation

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 im gleichen Kontext zwei verschiedene Listentypen benötigt, müssen wir uns zudem unterschiedliche Namen für den Typ und insbesondere für die Datenkonstruktoren ausdenken.

Mon dieu, isch verschtehe die Aufregung nischt. Für Lischten von Zahlen definiere isch:

```
type natslink = ^nats;
     nats     = record
                  elem : nat;
                  next : natslink
                end;
```

Für Lischten von Personen definiere isch:

```
type personslink = ^persons;
     persons     = record
                     elem : person;
                     next : personslink
                   end;
```

Für Lischten von Adressen definiere isch ...

Ja, ja, schon gut Blaise. Wir befinden uns mittlerweile im 21. Jahrhundert.

24. Motivation

Idee: wir abstrahieren von dem Elementtyp und machen ihn zum Parameter der Typdefinition.

$$\textbf{\textit{type}}\ \mathit{List}\ \langle {}'a \rangle = \ |\ \mathit{Nil}\ |\ \mathit{Cons}\ \textbf{\textit{of}}\ {}'a * \mathit{List}\ \langle {}'a \rangle$$

☞ *Typvariablen* fangen mit einem Apostroph an, um sie von Record- und Variantentypen unterscheiden zu können.

Wenden wir *List* auf einen konkreten Typ an, so erhalten wir einen speziellen Listentyp:

- Listen von natürlichen Zahlen: $\mathit{List}\ \langle \mathit{Nat} \rangle$,
- Listen von Binärziffern: $\mathit{List}\ \langle \mathit{Bit} \rangle$,
- Listen von Personen: $\mathit{List}\ \langle \mathit{Person} \rangle$,
- Listen von Adressen: $\mathit{List}\ \langle \mathit{Address} \rangle$,
- Listen von Listen von natürlichen Zahlen: $\mathit{List}\ \langle \mathit{List}\ \langle \mathit{Nat} \rangle \rangle$,
- usw.

☞ *Typparameter* werden in spitze Klammern gesetzt, um sie von Werteparametern auf den ersten Blick unterscheiden zu können.

24. Motivation

- Listen sind ein typisches Beispiel für Behälter (engl. container): eine Liste enthält Daten.
- Der Typ $\mathit{List}\,\langle t \rangle$ ist ein Containertyp.
- Da *List* einen Typparameter hat, entspricht *List* im Prinzip einer Funktion auf Typen.
- *Zur Erinnerung:* Listen sind eine einfache *Datenstruktur*. Eine Datenstruktur verwaltet Daten und unterstützt Zugriff und Manipulation dieser Daten.
- Listen ordnen Daten linear an.

24. Abstrakte Syntax

Wir erweitern Definitionen um parametrisierte Recordtyp- und Variantentypdefinitionen.

$d ::= \cdots$		*Deklarationen:*
$\mid$	$\textbf{\textit{type}}\ T\ \langle {}'a \rangle = \{\ell_1 : t_1; \ell_2 : t_2\}$	parametrisierter Recordtyp
$\mid$	$\textbf{\textit{type}}\ T\ \langle {}'a \rangle = \mid C_1\ \textbf{\textit{of}}\ t_1 \mid C_2\ \textbf{\textit{of}}\ t_2$	parametrisierter Variantentyp

24. Statische Semantik

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 zum Beispiel die Typregel für die Konstruktoranwendung wie folgt angepasst werden.

$$\frac{\Sigma \vdash e : t_i\{{}'a \mapsto t\}}{\Sigma \vdash C_i\ e : T\ \langle t \rangle}$$

☞ In t_i wird die Typvariable ${}'a$ durch den Typ t ersetzt. Notation der *Typsubstitution*: $t_i\{{}'a \mapsto t\}$. *Analogie:* ${}'a$ ist der formale Typparameter, t ist der aktuelle Typparameter.

Beispiel:

$${}'a * List\ \langle {}'a \rangle \{{}'a \mapsto Nat\} = Nat * List\ \langle Nat \rangle$$

24. Statische Semantik

type $List\ \langle 'a \rangle = \ |\ Nil \ |\ Cons$ ***of*** $'a * List\ \langle 'a \rangle$

Beispiele: t ist ein beliebiger Typ.

Ausdruck	Typ
Nil	$List\ \langle t \rangle$
$Cons\ (4711, Nil)$	$List\ \langle Nat \rangle$
$Cons\ (false, Nil)$	$List\ \langle Bool \rangle$
$Cons\ (Nil, Nil)$	$List\ \langle List\ \langle t \rangle \rangle$
$Cons\ (4711, Cons\ (false, Nil))$	nicht wohlgetypt
$Cons\ (Nil, Cons\ (Nil, Nil))$	$List\ \langle List\ \langle t \rangle \rangle$
$Cons\ (Cons\ (Nil, Nil), Nil)$	$List\ \langle List\ \langle List\ \langle t \rangle \rangle \rangle$

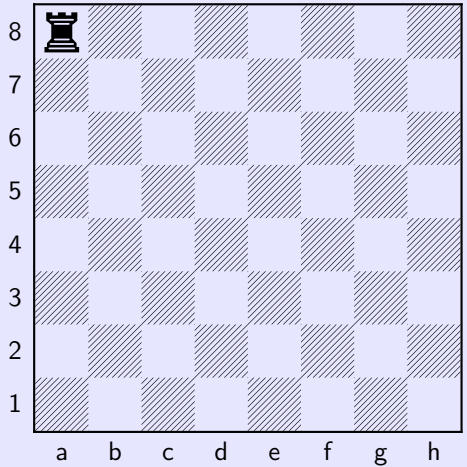
☞ Nil besitzt unendlich viele Typen.

24. Dynamische Semantik

Es ändert sich nichts.

25. Knobelaufgabe #11

- Bewege den Turm von a8 nach h1, so dass jedes Feld *genau einmal* besucht wird.



- Ein Turm darf horizontal und vertikal, aber nicht diagonal ziehen.

25. Motivation

Parametrisierte Typen erhöhen — wie auch Funktionen — die Wiederverwendbarkeit von Programmen („one size fits all“).

Funktionen auf parametrisierten Typen halten mit dieser Entwicklung nicht Schritt. *Beispiel:* Länge einer Liste.

```
let rec length (list : List ⟨Nat⟩) : Nat =
  match list with | Nil → 0 | Cons (_, xs) → 1 + length xs
```

☞ Der Typ des Parameters, $\mathit{List}\,\langle \mathit{Nat} \rangle$, schränkt die Anwendung von *length* auf Listen von *natürlichen Zahlen* ein. Um 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
```

☞ Die Definition ist baugleich zur ersten, nur die Typangabe hat sich geändert. Für das Ausrechnen der Listenlänge spielt der Typ der Elemente aber gar keine Rolle.

25. Motivation

☞ Für jeden Elementtyp eine neue *length* Funktion zu programmieren, ist *mühsam* und *unökonomisch*. Werden im gleichen Kontext Längenfunktionen für zwei verschiedene Elementtypen benötigt, müssen wir uns zudem unterschiedliche Namen für die Funktionen ausdenken.

☞ Das Problem ist ähnlich dem, das die Einführung parametrisierter Typen motivierte. Ähnliches Problem, ähnliche Lösung?

Mon dieu, isch verschtehe die Aufregung nischt. Für die Länge von Lischten von Zahlen definiere isch:

```
function lengthnats (p : natslink) : integer;
begin
   if p = nil then
      lengthnats := 0
   else
      lengthnats := lengthnats (p^.next) + 1
end;
```

Für die Länge von Lischten von Personen definiere isch:

```
function lengthpersons (p : personslink) : integer;
begin
   if p = nil then
      lengthpersons := 0
   else
      lengthpersons := lengthpersons (p^.next) + 1
end;
```

Für die Länge von Lischten von Adressen definiere isch ...

Ich glaub, ich hab ein déjà-vu ...

25. Motivation

Ähnliches Problem, ähnliche Lösung?

Ja! So wie wir Typdefinitionen mit einem Typ parametrisieren, so können wir auch Funktionsdefinitionen mit einem Typ parametrisieren.

$$\textbf{\textit{let rec}}\ \mathit{length}\ \langle'a\rangle(\mathit{list} : \mathit{List}\ \langle'a\rangle) : \mathit{Nat} =$$
$$\quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \mid \mathit{Nil} \to 0 \mid \mathit{Cons}\ (\_, xs) \to 1 + \mathit{length}\ \langle'a\rangle xs$$

☞ Die Funktion *length* hat jetzt zwei Parameter: einen Typparameter ($'a$) und einen Werteparameter (*list*). Der Typparameter wird verwendet, um den Typ des Werteparameters festzulegen.

Wird die Funktion *length* aufgerufen, müssen für die formalen Parameter entsprechend aktuelle Parameter angegeben werden:

- ein Typ und
- eine Liste mit Elementen des angegebenen Typs.

25. Motivation — polymorphe Funktionen

Wenden wir *length* auf einen konkreten Typ an, erhalten wir spezielle Längenfunktionen — Funktionen, die wir bisher mühsam per Hand programmieren mussten.

- *length* ⟨*Nat*⟩ für Listen von natürlichen Zahlen,
- *length* ⟨*Bit*⟩ für Listen von Binärziffern,
- *length* ⟨*Person*⟩ für Listen von Personen,
- *length* ⟨*Address*⟩ für Listen von Adressen,
- *length* ⟨*List* ⟨*Nat*⟩⟩ für Listen von Listen von natürlichen Zahlen,
- usw.

☞ 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.

25. Motivation — polymorphe Funktionen

Labels parametrisierter Recordtypen und Konstruktoren parametrisierter Variantentypen sind im Prinzip polymorphe Funktionen bzw. Werte:

- *Nil* hat den Typ $\mathit{List}\,\langle'a\rangle$;
- *Cons* hat den Typ $'a * \mathit{List}\,\langle'a\rangle \to \mathit{List}\,\langle'a\rangle$.

25. Motivation — Peano Entwurfsmuster, programmiert

Blicken wir zurück, erkennen wir, dass viele Funktionen allgemeiner sind, als die Typangaben es vermuten lassen. *Zum Beispiel:*

$$
\begin{array}{l}
\textbf{\textit{let}}\ \textit{peano-pattern}\ \langle'\textit{soln}\rangle(\textit{zero} : '\textit{soln}, \\
\qquad\qquad\qquad\qquad \textit{succ} : '\textit{soln} \to '\textit{soln}) : \textit{Nat} \to '\textit{soln} = \\
\quad \textbf{\textit{let rec}}\ f\ (n : \textit{Nat}) : '\textit{soln} = \\
\qquad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ \textit{zero} \\
\qquad\qquad\quad \textbf{\textit{else}}\ \ \textit{succ}\ (f\ (n \dot{-} 1)) \\
\quad \textbf{\textit{in}}\ \ f
\end{array}
$$

☞ Jetzt können wir in Mini–F# ausdrücken, dass die Funktion für jeden beliebigen Typ von Lösungen funktioniert (siehe frühere Diskussion).

Super, damit kann ich endlich meine Implementierung der Fakultät ausprobieren.

Mini⟩ *peano-pattern* ⟨*Nat* ∗ *Nat*⟩
((0, 1), ***fun*** (*n*, *s*) → (*n* + 1, *s* ∗ (*n* + 1))) 10
(10, 3628800)

Pass mal auf, Lisa, ich hab mich mittlerweile mit den anonymen Funktionen angefreundet. Jetzt kommt der Hammer: man kommt auch ohne Paare aus!

Mini⟩ *peano-pattern* ⟨*Nat* → *Nat*⟩
(***fun*** *n* → 1, ***fun*** *s* → ***fun*** *n* → *s* (*n* ∸ 1) ∗ *n*) 10 10
3628800

Clever, clever, Harry. Deine Funktion ist sogar noch allgemeiner: wenn Du sie mit *m* und *n* aufrufst (*m* ⩽ *n*), berechnet sie *n* ! ÷ (*n* − *m*) !.

25. Motivation — Sortieren durch Einfügen

Auch die Funktion *sort-by* lässt sich verallgemeinern.

Sortieren durch Einfügen

```
let sort-by ⟨'a⟩(less-equal : 'a * 'a → Bool) : List ⟨'a⟩ → List ⟨'a⟩ =
  let rec insert (elem : 'a, list : List ⟨'a⟩) : List ⟨'a⟩ =
    match list with
      | Nil          → Cons (elem, Nil)
      | Cons (x, xs) → if less-equal (elem, x)
                       then Cons (elem, list)
                       else Cons (x, insert (elem, xs))
  let rec sort (list : List ⟨'a⟩) : List ⟨'a⟩ =
    match list with
      | Nil          → Nil
      | Cons (x, xs) → insert (x, sort xs)
  in sort
```

25. Motivation — Sortieren durch Einfügen

Jetzt können wir beliebige (!) Listen sortieren.

- Listen von natürlichen Zahlen:

$$\mathit{sort\text{-}by}\,\langle \mathit{Nat}\rangle\,(\textbf{\textit{fun}}\ (m, n) \to m \leqslant n)$$
$$\mathit{sort\text{-}by}\,\langle \mathit{Nat}\rangle\,(\textbf{\textit{fun}}\ (m, n) \to m \geqslant n)$$

- Listen von Binärziffern:

$$\begin{aligned}\mathit{sort\text{-}by}\,\langle \mathit{Bit}\rangle\,(\textbf{\textit{fun}}\ (b, c) \to\ &\textbf{\textit{match}}\ (b, c)\ \textbf{\textit{with}}\\ &\mid (\mathit{One}, \_\ \ ) \to \mathit{true}\\ &\mid (\mathit{Two}, \mathit{One}\ ) \to \mathit{false}\\ &\mid (\mathit{Two}, \mathit{Two}) \to \mathit{true}\end{aligned}$$

- Listen von Personen:

$$\mathit{sort\text{-}by}\,\langle \mathit{Person}\rangle\,(\textbf{\textit{fun}}\ (\mathit{ann}, \mathit{bob}) \to \mathit{ann}.\mathit{surname} \leqslant \mathit{bob}.\mathit{surname})$$

- Listen von Listen von natürlichen Zahlen:

$$\mathit{sort\text{-}by}\,\langle \mathit{List}\,\langle \mathit{Nat}\rangle\rangle\,(\textbf{\textit{fun}}\ (\mathit{ns}_1, \mathit{ns}_2) \to \mathit{length}\,\langle \mathit{Nat}\rangle\,\mathit{ns}_1 \leqslant \mathit{length}\,\langle \mathit{Nat}\rangle\,\mathit{ns}_2)$$

25. Pragmatik

Wir sehen davon ab, Syntax und Semantik polymorpher Funktionen formal zu definieren.

Stattdessen geben wir uns pragmatisch und erlauben *Typparameter auszulassen*: sowohl bei der Definition (keine große Sache)

$$\textbf{let rec } \mathit{length}\ (\mathit{list} : \mathit{List}\ \langle' a\rangle) : \mathit{Nat} = \ldots$$
$$\textbf{let } \mathit{sort\text{-}by}\ (\mathit{less\text{-}equal} : {}' a * {}' a \to \mathit{Bool}) : \mathit{List}\ \langle' a\rangle \to \mathit{List}\ \langle' a\rangle = \ldots$$

als auch bei der Anwendung (eine erhebliche Schreiberleichterung).

$$\mathit{sort\text{-}by}\ (\textbf{fun}\ (m, n) \to m \leqslant n)$$
$$\mathit{sort\text{-}by}\ (\textbf{fun}\ (ns_1, ns_2) \to \mathit{length}\ ns_1 \leqslant \mathit{length}\ ns_2)$$

25. Vertiefung

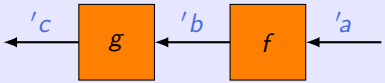
Polymorphen Funktionen ist gemeinsam, dass sie die Elemente eines polymorphen Typs nicht analysieren oder inspizieren (oder gar generieren), sondern nur transportieren (oder womöglich ignorieren).

Mini⟩ ***let*** *id* $x = x$
val *id* : $'a \to 'a$

Mini⟩ ***let*** *constant* $a\ b = a$
val *constant* : $'a \to 'b \to 'a$

Mini⟩ ***let*** *compose* $g\ f\ x = g\ (f\ x)$
val *compose* : $('b \to 'c) \to ('a \to 'b) \to ('a \to 'c)$

☞ *compose* ist ebenfalls eine Funktion höherer Ordnung: sie definiert die Komposition (Hintereinanderschaltung) zweier Funktionen.



25. Vertiefung — Listenzugriff

Aufgabe: Bestimmung des n-ten Elements einer Liste.

```
let rec nth (list : List ⟨'a⟩, n : Nat) : 'a =
```

☞ Was machen wir, wenn die Liste weniger als n Elemente hat?

Idee: die beiden möglichen Resultate des Zugriffs, erfolglos und erfolgreich, mit einem Datentyp darstellen.

```
type Option ⟨'a⟩ = | None | Some of 'a
```

Mit diesem neuen Typ können wir die Signatur von *nth* verfeinern.
Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

```
let rec nth (list : List ⟨'a⟩, n : Nat) : Option ⟨'a⟩ =
  match list with
  | Nil          → ...
  | Cons (x, xs) → ... nth (xs, ...) ...
```

25. Vertiefung — Listenzugriff

- *Rekursionsbasis*: der Index ist zu groß.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{nth}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle, n : \mathit{Nat}) : \mathit{Option}\ \langle'a\rangle = \\
\quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\
\quad \mid \mathit{Nil} \qquad\quad \to \mathit{None} \\
\quad \mid \mathit{Cons}\ (x, xs) \to \ldots \mathit{nth}\ (xs, \ldots) \ldots
\end{array}
$$

- *Rekursionsschritt*: zusätzliche Fallunterscheidung über n.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{nth}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle, n : \mathit{Nat}) : \mathit{Option}\ \langle'a\rangle = \\
\quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\
\quad \mid \mathit{Nil} \qquad\quad \to \mathit{None} \\
\quad \mid \mathit{Cons}\ (x, xs) \to \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ \mathit{Some}\ x \\
\qquad\qquad\qquad\qquad\quad \textbf{\textit{else}}\ \ \mathit{nth}\ (xs, n \dot{-} 1)
\end{array}
$$

☞ Die Funktion *nth* sollte nicht *missbraucht* werden, um über eine Liste zu iterieren!

25. Über den Tellerrand: F#

Listen sind in F# vordefiniert. Allerdings unterscheidet sich die Notation etwas:

- $T\ list$ statt $List\,\langle T\rangle$ (Postfix- statt Präfixnotation),
- $[\,]$ statt Nil, und
- $x :: xs$ statt $Cons\,(x, xs)$ (Infix- statt Präfixnotation).

☞ $[x_1; x_2; x_3]$ kürzt $x_1 :: (x_2 :: (x_3 :: [\,]))$ ab.

Beispiel: Bestimmung des n-ten Elements 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)
```

25. Über den Tellerrand: F#

F# erlaubt, die Typen von Funktionsparametern und -ergebnissen auszulassen — die fehlenden Informationen werden automatisch inferiert (Typinferenz).

Beliebige Grade an Flexibilität:

let rec *length* (*xs* : *'a list*) : *int* = ***match*** *xs* ***with*** ...

let rec *length* (*xs* : *'a list*) = ***match*** *xs* ***with*** ...

let rec *length* (*xs*) = ***match*** *xs* ***with*** ...

let rec *length* = ***function*** ...

☞ ***function*** *m* ist eine beliebte Abkürzung für den Ausdruck ***fun*** $x \rightarrow$ ***match*** x ***with*** m. Beliebt, weil die Einführung eines Parameters vermieden wird (neue Namen zu erfinden ist schwer).

25. Über den Tellerrand: F#

Beispiel: enthält eine Liste ein gegebenes Element?

```
let rec contains key = function
    | []      → false
    | x :: xs → key = x || contains key xs
```

Beispiel: Konkatenation zweier Listen.

```
let rec append list y =
  match list with
    | []      → y
    | x :: xs → x :: append xs y
```

☞ *append x y* ist als Infixoperator vordefiniert: $x \mathbin{@} y$, z.B. $[4;7] \mathbin{@} [1;1] = [4;7;1;1]$.
(Was ist der Typ von *append* aka @? Und von *contains*?)

26. Motivation

Wir haben verschiedene Möglichkeiten kennengelernt, Daten zu aggregieren.

- Tupel und Records:
  - *feste* Anzahl von Daten *verschiedenen* Typs,
  - Zugriff in konstanter Zeit.
- Listen:
  - *beliebige* Anzahl von Daten des *gleichen* Typs,
  - Zugriff in linearer Zeit.
- Fehlt:
  - *beliebige* Anzahl von Daten des *gleichen* Typs,
  - Zugriff in *konstanter* Zeit.

26. Motivation

☞ Diese Lücke schließen Arrays (auch: Felder oder Reihungen).

Konstruktion:

$$[|2;3;5;7;11|]$$

ist ein Array der Größe 5. Mit *a.Length* wird die Größe bzw. Länge eines Arrays ermittelt, z.B. $[|2;3;5;7;11|].\mathit{Length} = 5$.

Zugriff: ist e_2 ein Ausdruck, der zu einem Array auswertet, und e_1 ein arithmetischer Ausdruck, dann kann mit

$$e_2.[e_1]$$

auf die entsprechende Komponente zugegriffen werden.
☞ Nummerierung ab 0, z.B. $[|2;3;5;7;11|].[3] = 7$.

26. Motivation

Arrays können nicht nur durch Aufzählung der Elemente, sondern auch mittels einer Bildungsvorschrift erzeugt werden.

$$[\,|\,\textit{for}\ i\ \textit{in}\ 0\,..\,9 \to i * i\,|\,]$$

ist das Array der ersten zehn Quadratzahlen.

26. Demo

```
Mini⟩ [| |]
[| |]
Mini⟩ [|2; 3; 5; 7; 11|]
[|2; 3; 5; 7; 11|]
Mini⟩ [|for i in 0..9 → 4711|]
[|4711; 4711; 4711; 4711; 4711; 4711; 4711; 4711; 4711; 4711|]
Mini⟩ [|for i in 0..9 → i|]
[|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
Mini⟩ [|for i in 0..9 → i * i|]
[|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
Mini⟩ [|for i in 0..7 → [|for j in 1..i → i * j|]|]
[|[| |];
  [|1|];
  [|2; 4|];
  [|3; 6; 9|];
  [|4; 8; 12; 16|];
  [|5; 10; 15; 20; 25|];
  [|6; 12; 18; 24; 30; 36|];
  [|7; 14; 21; 28; 35; 42; 49|]|]
```

26. Arrays versus Funktionen

Arrays sind Funktionen recht ähnlich.

Typ	$t_1 \to t_2$	$\mathit{Array}\ \langle t_2 \rangle \quad (t_1 = \mathit{Int})$
Konstruktion	—	$[\!\mid e_0; \ldots; e_{n-1} \mid\!]$
	fun $x \to e_3$	$[\!\mid$ ***for*** x ***in*** $0\,..\,n-1 \to e_3 \mid\!]$
Elimination	$e_2\ e_1$	$e_2.[e_1]$
	—	$e.\mathit{Length}$

☞ Der Definitionsbereich eines Arrays ist stets ein Anfangsstück der natürlichen Zahlen. Haben wir das schon einmal gesehen?

26. Arrays versus Sequenzen

Zur Erinnerung: endliche Abbildungen des Typs $\mathbb{N} \to_{\text{fin}} A$ heißen Sequenzen (Folie 60).

- Sequenzen sind Bestandteil der Sprache, mit der wir *über* die Sprache Mini–F# reden.
- Sequenzen sind Bestandteil von Mini–F#.

☞ Fachjargon: Arrays *internalisieren* Sequenzen. Umgekehrt verwenden wir Sequenzen, um die Semantik von Arrays und Operationen auf Arrays präzise zu beschreiben.

26. Abstrakte Syntax

$e ::= \cdots$	*Arrays:*
$\mid [\mid e_0; \ldots; e_{n-1} \mid]$	Konstruktion durch Aufzählung ($n \in \mathbb{Z}$)
$\mid [\mid \textbf{for}\ x\ \textbf{in}\ e_1 \,..\, e_2 \to e_3 \mid]$	Konstruktion durch Bildungsvorschrift
$\mid e_2.[e_1]$	Subskription
$\mid e.\mathit{Length}$	Größe eines Arrays

☞ Eckige Klammern, [| und |], sind das Markenzeichen der Sprachkonstrukte, die Arrays konstruieren.

☞ Das Konstrukt $[\mid \textbf{for}\ x\ \textbf{in}\ e_1 \,..\, e_2 \to e_3 \mid]$ führt den Bezeichner x neu ein; x ist in e_3 sichtbar.

☞ Der Ausdruck e_1 in $e_2.[e_1]$ heißt *Index*.

26. Statische Semantik

Der Typ eines Arrays ist mit dem Typ der Elemente parametrisiert.

$t ::= \cdots$	*Typen:*
$\mid \mathit{Array}\ \langle t \rangle$	Arraytyp

Typregel:

$$\frac{\Sigma \vdash e_i : t \quad \mid \quad i \in \mathbb{N}_n}{\Sigma \vdash [\!|\, e_0; \ldots; e_{n-1} \,|\!] : \mathit{Array}\ \langle t \rangle}$$

☞ 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}.

26. Statische Semantik

Typregel:

$$\frac{\Sigma \vdash e_1 : \mathit{Int} \quad \Sigma \vdash e_2 : \mathit{Int} \quad \Sigma, \{x \mapsto \mathit{Int}\} \vdash e_3 : t}{\Sigma \vdash [|\ \textbf{for}\ x\ \textbf{in}\ e_1 \,..\, e_2 \to e_3\ |] : \mathit{Array}\ \langle t \rangle}$$

$$\frac{\Sigma \vdash e_2 : \mathit{Array}\ \langle t \rangle \qquad \Sigma \vdash e_1 : \mathit{Int}}{\Sigma \vdash e_2.[e_1] : t}$$

$$\frac{\Sigma \vdash e : \mathit{Array}\ \langle t \rangle}{\Sigma \vdash e.\mathit{Length} : \mathit{Int}}$$

☞ ***for*** x ***in*** $e_1 \,..\, e_2$ arbeitet auch mit anderen Zahlentypen, nicht nur mit Int.

26. Statische Semantik — Extremfälle

- $n = 0$: Felder dürfen leer sein.
  - $[|\,|]$,
  - $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ 1\,..\,0 \to x\,|]$,
  - $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ l\,..\,r \to e\,|]$ mit $l > r$,
  - dann gilt: $e.\mathit{Length} = 0$.
  - Das leere Feld $[\![]\!]$ hat den Typ $\mathit{Array}\,\langle t \rangle$ für einen beliebigen Typ t.
- $n = 1$: Felder können genau ein Element enthalten.
  - $[|\,4711\,|]$,
  - $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ 0\,..\,0 \to 4711\,|]$,
  - $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ l\,..\,r \to e\,|]$ mit $l = r$,
  - dann gilt: $e.\mathit{Length} = 1$.
- $n = 2$: ...

26. Dynamische Semantik

Der Wert eines Arrayausdrucks ist eine Sequenz, eine endliche Abbildung $\mathbb{N} \to_{\text{fin}} \text{Val}$; jedem Index wird die entsprechende Komponente zugeordnet.

Beispiele:

- $[|\,|]$ wertet zu ϵ aus;
- $[|4711|]$ wertet zu $\{0 \mapsto 4711\}$ aus;
- $[|2;3;5;7;11|]$ wertet zu $\{0 \mapsto 2, 1 \mapsto 3, 2 \mapsto 5, 3 \mapsto 7, 4 \mapsto 11\}$ aus.

26. Dynamische Semantik

$s \in \mathsf{Val}^*$

$\nu ::= \cdots$	*Werte:*
$\mid\ s$	Array (-wert)

Auswertungsregeln:

$$\frac{\delta \vdash e_i \Downarrow \nu_i \quad \mid \quad i \in \mathbb{N}_n}{\delta \vdash [\!|\, e_0, \ldots, e_{n-1} \,|\!] \Downarrow \{\, i \mapsto \nu_i \mid i \in \mathbb{N}_n \}}$$

$$\frac{\begin{array}{ll} \delta \vdash e_1 \Downarrow l & \\ \delta \vdash e_2 \Downarrow r & \delta, \{x \mapsto i\} \vdash e_3 \Downarrow \nu_i \quad \mid \quad i \in \{l \,..\, r\} \end{array}}{\delta \vdash [\!|\, \textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1 \,..\, e_2 \to e_3 \,|\!] \Downarrow \{\, i - l \mapsto \nu_i \mid i \in \{l \,..\, r\} \}}$$

☞ Es werden zunächst die Arraygrenzen ausgerechnet, dann wird der Bezeichner x nacheinander an die Werte l, ..., r gebunden und bezüglich jeder Bindung wird der Rumpf e_3 ausgerechnet.

26. Dynamische Semantik

Auswertungsregeln:

$$\frac{\delta \vdash e_2 \Downarrow s \qquad \delta \vdash e_1 \Downarrow i}{\delta \vdash e_2.[e_1] \Downarrow s(i)} \quad i < \mathit{len}\ s$$

$$\frac{\delta \vdash e \Downarrow s}{\delta \vdash e.\mathit{Length} \Downarrow \mathit{len}\ s}$$

☞ Ein Array ist eine endliche Abbildung; Subskription ist entsprechend Funktionsapplikation.
☞ Die Subskription $e_2.[e_1]$ ist nur definiert, wenn der Index e_1 im Definitionsbereich der endlichen Abbildung liegt.

26. Vertiefung — Spiegelung

Arraytransformationen lassen sich oftmals mit dem ***for***-Konstrukt (Konstruktion durch Bildungsvorschrift) programmieren.

Beispiel: Spiegelung eines Arrays.

```
let reverse ⟨'a⟩(a : Array ⟨'a⟩) =
  let n = a.Length in
    [| for i in 1 .. n → a.[n − i] |];
```

☞ *reverse* ist eine polymorphe Funktion.

```
Mini⟩  reverse[| for i in 0 .. 9 → i ∗ i |]
[|81; 64; 49; 36; 25; 16; 9; 4; 1; 0|]
```

26. Vertiefung — Konkatenation

Beispiel: Konkatenation zweier Arrays.

$$\textbf{\textit{let}}\ \mathit{append}\ \langle'a\rangle(a : \mathit{Array}\ \langle'a\rangle, b : \mathit{Array}\ \langle'a\rangle) : \mathit{Array}\ \langle'a\rangle =$$
$$[|\ \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,a.\mathit{Length} + b.\mathit{Length} - 1 \rightarrow$$
$$\textbf{\textit{if}}\ i < a.\mathit{Length}\ \textbf{\textit{then}}\ a.[i]$$
$$\textbf{\textit{else}}\ \ b.[i - a.\mathit{Length}]\ |]$$

☞ *append* ist eine polymorphe Funktion.

☞ Vergleiche mit der Konkatenation $s_1 \cdot s_2$ von Sequenzen (Folie 61; Arrays internalisieren Sequenzen).

Mini⟩ $\mathit{append}\ ([|\ \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,9 \rightarrow i\ |], \mathit{reverse}[|\ \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,9 \rightarrow i\ |])$
$[|0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]$

26. Vertiefung — Entwurfsmuster

Der „Definitionsbereich“ des Arrays a ist durch das Intervall $(0, a.Length - 1)$ gegeben. Somit können wir die Programmierung von arrayverarbeitenden Funktionen so ähnlich angehen, wie die von Funktionen auf Suchintervallen.

Entwurfsmuster für Arrays (à la Peano):

```
let f (a : Array ⟨t1⟩) : t2 =
  let rec g (i : Int) : t2 =
    if i = a.Length then ...
                    else ... g (i + 1) ...
  in g 0
```

26. Entwurfsmuster — Summe

Beispiel: Elemente eines Arrays aufaddieren (das Pendant zur Funktion $sum : List\ \langle Nat \rangle \to Nat$).

Das Entwurfsmuster für Arrays gibt vor:

$$
\begin{array}{l}
\textbf{\textit{let}}\ sum\ (a : Array\ \langle Nat \rangle) : Nat = \\
\quad \textbf{\textit{let rec}}\ s\ (i : Int) : Nat = \\
\qquad \textbf{\textit{if}}\ \ i = a.Length\ \textbf{\textit{then}} \ldots \\
\qquad\qquad\qquad\qquad \textbf{\textit{else}}\ \ \ldots s\ (i + 1) \ldots \\
\quad \textbf{\textit{in}}\ s\ 0
\end{array}
$$

26. Entwurfsmuster — Summe

- *Rekursionsbasis*: $sum[||] = 0$!

```
let sum (a : Array ⟨Nat⟩) : Nat =
  let rec s (i : Int) : Nat =
    if i = a.Length then 0
                    else ... s (i + 1) ...
  in s 0
```

- *Rekursionsschritt*: Wir addieren $a.[i]$ zur Summe des Restarrays.

```
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
```

26. Entwurfsmuster — Produkt

Beispiel: Elemente eines Arrays miteinander multiplizieren (das Pendant zur Funktion $product : List\,\langle Nat\rangle \to Nat$).

Das Entwurfsmuster für Arrays gibt vor:

$$\begin{array}{l}\textbf{\textit{let}}\ product\ (a : Array\,\langle Nat\rangle) : Nat = \\ \quad \textbf{\textit{let rec}}\ p\ (i : Int) : Nat = \\ \quad\quad \textbf{\textit{if}}\ \ i = a.Length\ \textbf{\textit{then}} \ldots \\ \quad\quad\quad\quad\quad\quad\quad\quad \textbf{\textit{else}}\ \ \ldots p\ (i+1) \ldots \\ \quad \textbf{\textit{in}}\ p\ 0\end{array}$$

26. Entwurfsmuster — Produkt

- *Rekursionsbasis*: $product[|\,|] = 1$!

```
let product (a : Array ⟨Nat⟩) : Nat =
  let rec p (i : Int) : Nat =
    if i = a.Length then 1
                    else ... p (i + 1) ...
  in p 0
```

- *Rekursionsschritt*: Wir multiplizieren $a.[i]$ mit dem Produkt des Restarrays.

```
let product (a : Array ⟨Nat⟩) : Nat =
  let rec p (i : Int) : Nat =
    if i = a.Length then 1
                    else a.[i] * p (i + 1)
  in p 0
```

26. Alte Funktionen neu

Eine weitere Implementierung der Fakultät und der Potenzfunktion:

$$\textbf{\textit{let}}\ \mathit{factorial}\ (n : \mathit{Nat}) : \mathit{Nat} =$$
$$\quad \mathit{product}[|\ \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 1\,..\,n \to i\ |]$$
$$\textbf{\textit{let}}\ \mathit{power}\ (x : \mathit{Nat}, n : \mathit{Nat}) : \mathit{Nat} =$$
$$\quad \mathit{product}[|\ \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 1\,..\,n \to x\ |]$$

26. Zusammenfassung

Wir haben das Repertoire von Mini–F# beträchtlich erweitert. *Teil III:* Werte, die von Werten abhängen. *Teil IV:* Typen, die von Typen abhängen und Werte, die von Typen abhängen.

---

hängt ab von	Wert	Typ
Wert	Funktion	polymorphe Funktion
Typ	?	parametrisierter Typ

Macht man die Größe von Arrays zum Bestandteil des Typs, dann erhält man ein Beispiel für einen Typ, der von Werten abhängt (sogenannter „dependent type“).

26. Zusammenfassung

Wir haben

- verschiedene Möglichkeiten kennengelernt, Daten zu aggregieren:
  - Tupel und Records,
  - Containertypen wie Listen,
  - Arrays;
- eine Möglichkeit kennengelernt, Daten zu vereinigen:
  - Variantentypen;
- Verschiedene Formen der Abstraktion kennengelernt:
  - parametrisierte Record- und Variantentypen,
  - polymorphe Funktionen;
- gesehen, dass jeder Typ ein Entwurfsmuster mitbringt;
- verschiedene Datenstrukturen kennengelernt:
  - Listen,
  - Arrays;
- und verschiedene Funktionen auf diesen Datenstrukturen programmiert.

26. Paradigmen: funktionale Programmierung

calculare
den Wert bestimmen; rechnen; zählen

functio
Verrichtung; Ausführung

fungi
ausüben

Teil V

Algorithmik

26. Knobelaufgabe #12: Geht's auch schneller?

Wir haben Listen und Arrays; sprechen die auch miteinander?

Du meinst, ob ich eine Liste in ein Array konvertiert kriege und umgekehrt? Das ist fix programmiert:

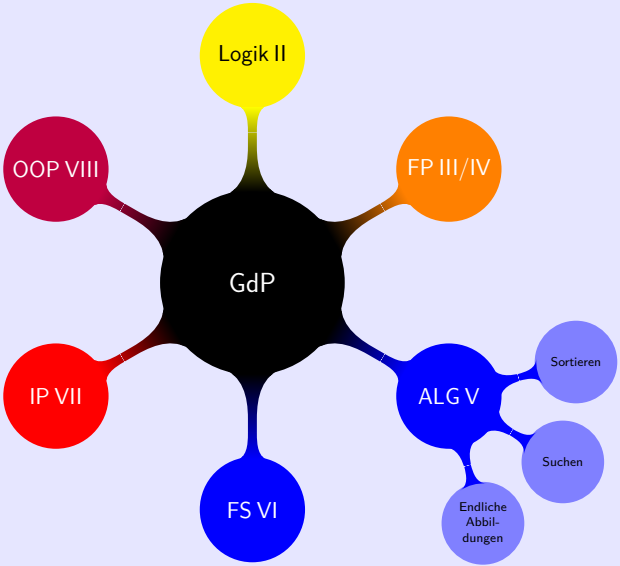
$$\begin{array}{l} \textbf{\textit{let}}\ \textit{to-list}\ \ (xs : \textit{'a array}) : \textit{'a list} = [\ \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,xs.\mathit{Length} \div 1 \to xs.[i]\ ] \\ \textbf{\textit{let}}\ \textit{to-array}\ (xs : \textit{'a list}\ \ ) : \textit{'a array} = [|\ \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,xs.\mathit{Length} \div 1 \to xs.[i]\ |] \end{array}$$

Nicht schlecht — man kann ***for*** auch verwenden, um eine Liste zu konstruieren, ...

Yup. *Mein Tipp:* nicht nur in den Vorlesungsfolien blättern, auch mal in der Online Doku stöbern ;-).

...und $xs.[i]$ um auf das i-te Listenelement zuzugreifen. Das ist aber teuer ...dein *to-array* hat eine quadratische Laufzeit!

26. Gliederung


Logik II
OOP VIII
FP III/IV
GdP
IP VII
ALG V
Sortieren
Suchen
Endliche Abbil-dungen
FS VI

26. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- verschiedene Sortieralgorithmen kennen,
- elementare Suchstrukturen kennen,
- die Laufzeit einfacher Algorithmen abschätzen können,
- einfache Korrektheitsbeweise führen können,
- einfache Terminierungsbeweise führen können,
- das Konzept des „Abstrakten Datentyps“ verstanden haben,
- mit verschiedenen Programmiertechniken vertraut sein.

27. Algorithmisches Lösen von Problemen

- *Rechnen lassen:* mit der Existenz von Rechenmaschinen wird es interessant, Aufgaben in Rechenaufgaben zu verwandeln, die es von Natur aus nicht sind.
- Wie lassen sich Probleme systematisch lösen? Entwurfsmuster!
  - *Peano:* Problem für n wird auf das Problem für $n \dot{-} 1$ zurückgeführt. *Allgemeiner:* Problem der Größe n wird auf Problem der Größe $n \dot{-} 1$ zurückgeführt.
  - *Leibniz:* Problem für n wird auf das Problem für $n \div 2$ zurückgeführt. *Allgemeiner:* Problem der Größe n wird auf Probleme der Größe $n \div 2$ zurückgeführt.
  - *Struktur:* Problem für eine (Daten-) Struktur wird auf Probleme für Teilstrukturen zurückgeführt.
- Den Entwurfsmustern ist gemeinsam, dass sie Lösungen für Probleme aus Lösungen für „kleinere“ Probleme konstruieren.

27. Algorithmisches Lösen von Problemen

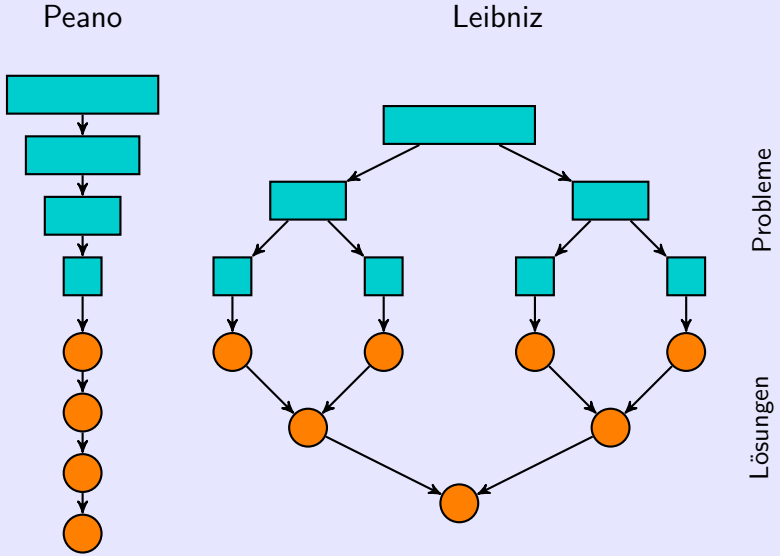
Allgemeiner Ansatz:

Um ein Problem zu lösen, genügt es zu zeigen, dass sich eine Lösung für jede Probleminstanz aus Lösungen kleinerer Probleminstanzen konstruieren lässt.

☞ Es ist nicht nötig, jede Probleminstanz von Grund auf zu lösen.

☞ Stattdessen versuche, eine Probleminstanz auf kleinere Probleminstanzen zu *reduzieren*.

27. Algorithmisches Lösen von Problemen



27. Sortieren

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 taken 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

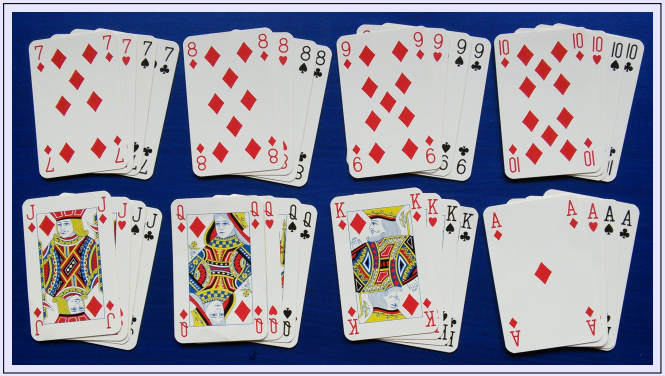
(i) *there are many important applications of sorting, or*
(ii) *many people sort when they shouldn't, or*
(iii) *inefficient sorting algorithms have been in common use.*

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, TAOCP 3*

27. Beispiel: Spielkarten

27. Beispiel: Spielkarten



27. Beispiel: Musikstücke

- Gegeben:

Künstler	Titel	Länge	Genre	Popularität
Yes	Siberian Khatru	8:57	rock	119
Beatles	Helter Skelter	4:27	rock	147
Lorde	Royals	3:10	pop	2677
Genesis	Supper's Ready	22:53	rock	298
Kraftwerk	Autobahn	22:30	pop	346

- nach dem Künstler sortiert:

Künstler	Titel	Länge	Genre	Popularität
Beatles	Helter Skelter	4:27	rock	147
Genesis	Supper's Ready	22:53	rock	298
Kraftwerk	Autobahn	22:30	pop	346
Lorde	Royals	3:10	pop	2677
Yes	Siberian Khatru	8:57	rock	119

27. Beispiel: Musikstücke

- nach dem Genre sortiert (es gibt 12 mögliche Antworten, warum?):

Künstler	Titel	Länge	Genre	Popularität
Lorde	Royals	3:10	pop	2677
Kraftwerk	Autobahn	22:30	pop	346
Yes	Siberian Khatru	8:57	rock	119
Beatles	Helter Skelter	4:27	rock	147
Genesis	Supper's Ready	22:53	rock	298

- nach dem Genre und dann nach der Popularität sortiert:

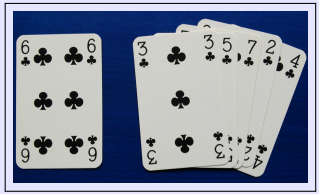
Künstler	Titel	Länge	Genre	Popularität
Lorde	Royals	3:10	pop	2677
Kraftwerk	Autobahn	22:30	pop	346
Genesis	Supper's Ready	22:53	rock	298
Beatles	Helter Skelter	4:27	rock	147
Yes	Siberian Khatru	8:57	rock	119

27. Sortieralgorithmen

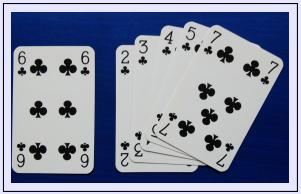
- *Peano Entwurfsmuster:* Problem der Größe n wird auf Problem der Größe $n-1$ zurückgeführt.
- *Sortieren:* die Problemgröße entspricht der Anzahl der zu sortierenden Elemente.
- *Sortieren durch Einfügen* (“bridge player method“):
  - Lege die erste Karte zur Seite,
  - sortiere den restlichen Stapel,
  - *füge* die erste Karte in den sortierten Stapel *ein*.
- Fokussiert auf die Eingabe.
- *Sortieren durch Auswählen*:
  - *Wähle* die kleinste Karte *aus*,
  - sortiere den restlichen Stapel,
  - lege die kleinste Karte auf den sortieren Stapel.
- Fokussiert auf die Ausgabe.

27. Sortieren durch Einfügen

lege erste Karte zur Seite



sortiere Reststapel



füge erste Karte ein

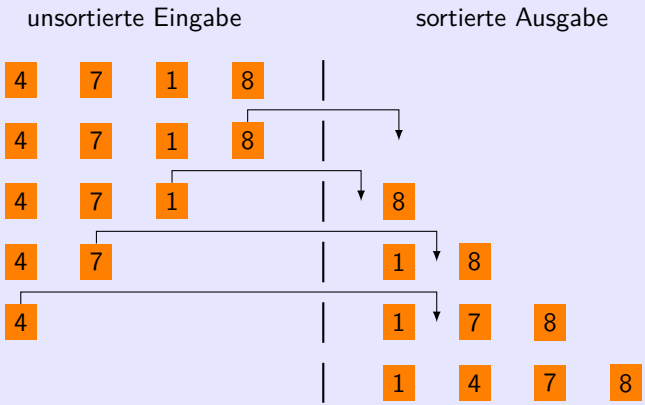
27. Sortieren durch Einfügen

Sortieren durch Einfügen in F#:

```
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
```

☞ Wir abstrahieren von der Ordnungsrelation $\leqslant$.

27. Sortieren durch Einfügen: Korrektheit



Zu zeigen:

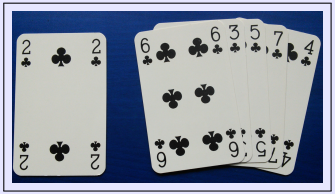
Wenn *xs* sortiert ist, dann ist auch *insert k xs* sortiert.

27. Sortieren durch Einfügen: Laufzeit

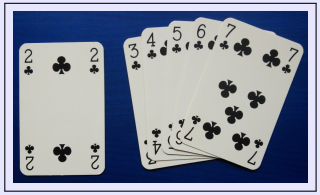
- Sortieren durch Einfügen ist *adaptiv:* „vorsortierte“ Eingaben werden schneller verarbeitet.
- *Der beste Fall:* die Eingabe ist bereits aufsteigend sortiert.
  - Einfügen: lediglich ein Vergleich wird benötigt.
  - Sortieren: benötigt insgesamt $n - 1$ Vergleiche (optimal).
- *Der schlechteste Fall:* die Eingabe ist absteigend sortiert.
  - Einfügen: durchläuft die gesamte Liste und benötigt $i - 1$ Vergleiche.
  - Sortieren: benötigt insgesamt $\sum_{i=0}^{n-1} i = n \cdot (n-1)/2$ Vergleiche.
- Kurz: die Laufzeit ist quadratisch.

27. Sortieren durch Auswählen

lege kleinste
Karte zur Seite



sortiere Reststapel



füge kleinste
Karte hinzu

27. Sortieren durch Auswählen

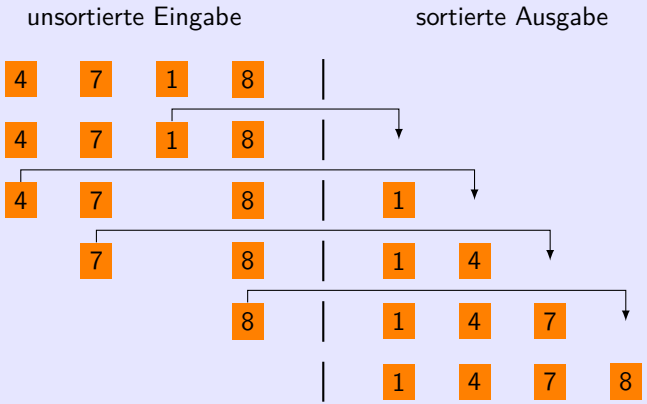
Sortieren durch Auswählen in F#:

```
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
```

Der Aufruf *split-min xs* gibt das kleinste Element von *xs* und die restliche Liste zurück (sofern ein kleinstes Element existiert).

☞ Die Definition von *sort* folgt *nicht* dem Struktur Entwurfsmuster: *ys* ist im allgemeinen nicht die Restliste von *xs*.

27. Sortieren durch Auswählen: Korrektheit



Zu zeigen:

Wenn *xs* nicht leer ist, dann ist *split-min xs* = *Some* (a, ys), wobei *a* das kleinste Element von *xs* ist und *ys* die restliche Liste. Anderenfalls ist *split-min xs* = *None*.

27. Sortieren durch Auswählen: Laufzeit

- Sortieren durch Auswählen ist „vergesslich“ (engl. oblivious): die Laufzeit hängt nicht von den Eingabeelementen ab.
- *Bester und schlechtester Fall:*
  - Auswahl: durchläuft die gesamte Liste und benötigt $i - 1$ Vergleiche (optimal, warum?).
  - Sortieren: benötigt insgesamt $\sum_{i=0}^{n-1} i = n \cdot (n-1)/2$ Vergleiche.
- Kurz: die Laufzeit ist quadratisch.

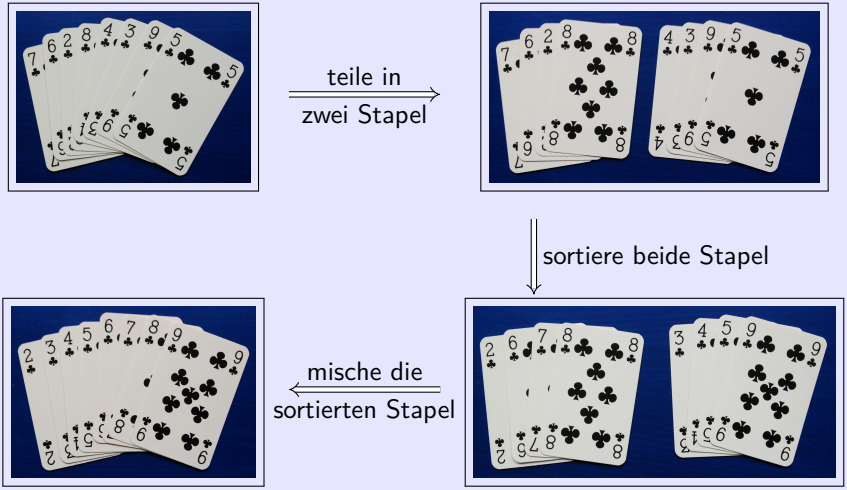
Writing programs needs genius to save the last order or the last millisecond. It is great fun, but it is a young man's game. You start it with great enthusiasm when you first start programming, but after ten years you get a bit bored with it, and then you turn to automatic-programming languages and use 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*

27. Teile und Herrsche

- *Leibniz:* Problem der Größe n wird auf Probleme der Größe $n \div 2$ zurückgeführt.
- *Sortieren durch Mischen:* (engl. sorting by merging)
  - Teile den Eingabestapel in zwei ungefähr gleich große Stapel,
  - sortiere jeden der Teilstapel,
  - „*mische*“ die zwei sortierten Teilstapel.
- (Ein alternativer Ansatz ist *Sortieren durch Austauschen:*
  - Wähle eine Pivotkarte aus und teile den Eingabestapel in kleinere und größere Karten,
  - sortiere jeden der Teilstapel,
  - hänge die beiden sortierten Teilstapel aneinander.
- Auch bekannt unter dem Namen *Quicksort.*)

27. Sortieren durch Mischen



27. Sortieren durch Mischen

Teilaufgabe: eine Liste halbieren.

Das Struktur Entwurfsmuster führt unmittelbar zum Ziel:

$$\begin{aligned}&\textbf{\textit{let rec}}\ \mathit{unzip} = \textbf{\textit{function}}\\ &\quad \mid [] \quad\ \to ([], [])\\ &\quad \mid x :: xs \to \textbf{\textit{let}}\ (xs_1, xs_2) = \mathit{unzip}\ xs\\ &\qquad\qquad\ \ (x :: xs_2, xs_1)\end{aligned}$$

Der Aufruf *unzip xs* teilt die Liste *xs* in die Liste der Elemente an geraden und die Liste der Elemente an ungeraden Positionen.

27. Sortieren durch Mischen

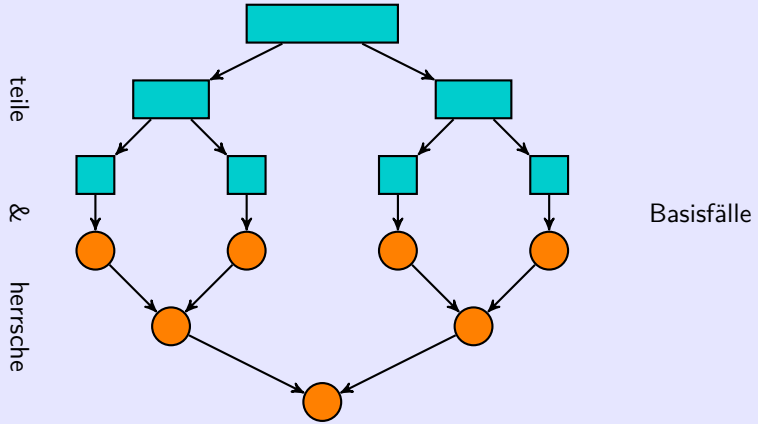
Sortieren durch Mischen in F#:

```
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
               merge (sort xs1, sort xs2)
  in sort
```

☞ *merge* verallgemeinert *insert* ($A \cup B$ versus $\{a\} \cup B$).

27. Teile und Herrsche: Laufzeit

- Leibniz Entwurfsmuster (teile und herrsche, divide et impera):
- *Fall* $n < 2$*:* ad hoc.
- *Fall* $n \geqslant 2$*:* teile in Probleme der Größen $\lfloor n/2 \rfloor$ und $\lceil n/2 \rceil$ auf.



- Wie tief ist der „Rekursionsbaum“? Mit anderen Worten, wie oft lässt sich n halbieren, bis wir 1 erhalten?

27. Teile und Herrsche: Laufzeit

- *Frage:* wie oft lässt sich n halbieren, bis wir 1 erhalten?
- *Antwort:* der binäre Logarithmus von n.

$$\lg n = \log_2 n$$

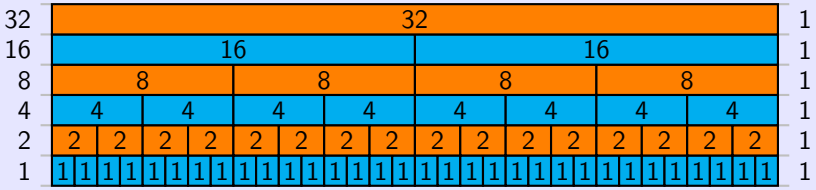
- Wachstum von $\lg n$:

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	$\approx 20,0$

- Zur Erinnerung: $\lg(ab) = \lg a + \lg b$, zum Beispiel ist $\lg 1.000.000 = 2 \cdot \lg 1.000 \approx 20,0$.

27. Sortieren durch Mischen: Laufzeit

- Die Laufzeit von *unzip* ist proportional zur Eingabegröße.
- Die Laufzeit von *merge* ist proportional zur Ausgabegröße.
- Sortieren durch Mischen: die Laufzeit ergibt sich als Produkt der Rekursionstiefe ($\lg n$) und des Aufwands pro Rekursionsebene (n).



- Die Laufzeit von *merge-sort* beträgt entsprechend $n \lg n$.

27. Sortieren durch Mischen: Laufzeit

- Wachstum von $n \lg n$:

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

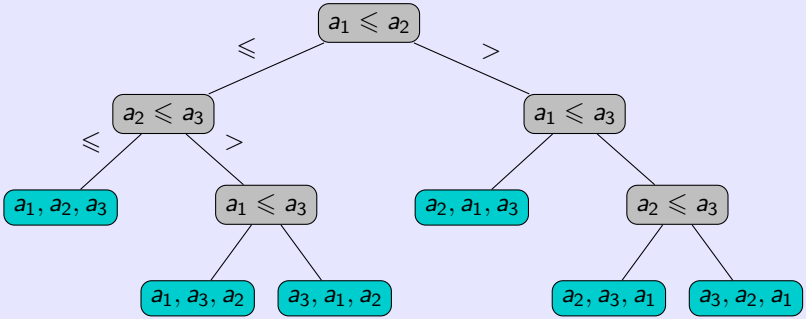
- Für hinreichend große n ist Sortieren durch Mischen wesentlich schneller als Sortieren durch Einfügen oder Auswählen.

27. Sortieren: Problemkomplexität

- Sortieren durch Mischen hat eine Laufzeit von $n \lg n$.
- Geht's noch schneller? Können wir in linearer Zeit sortieren?
- Leider nein: jedes Sortierverfahren, *das auf dem Vergleichen von Elementen basiert*, benötigt im schlechtesten Fall $n \log n$ Vergleiche.
- Damit ist *merge-sort asymptotisch optimal*.
- Die *Komplexität* des Sortierproblems ist $n \log n$.

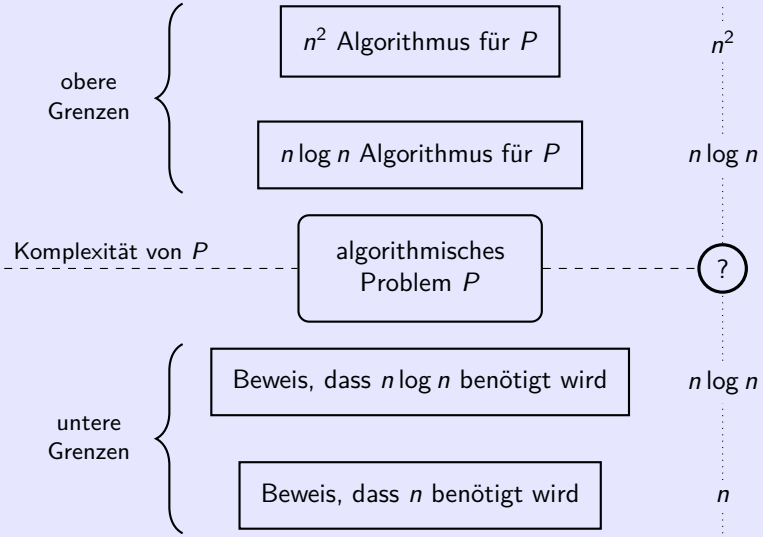
27. Sortieren: Problemkomplexität

- Ein Entscheidungsbaum, um drei Elemente zu sortieren:



- Eine Ausführung des korrespondieren Sortierprogramms entspricht einem Pfad von der Wurzel zu einem Blatt, der sortierten Permutation der Eingabe.
- Ein Entscheidungsbaum, der n Elemente sortiert, muss notwendigerweise $n!$ Blätter besitzen.
- Man kann zeigen, dass die Höhe des Baums ungefähr $n \log n$ ist.

27. Untere und obere Schranken



28. Knobelaufgabe #13

Lisa Lista hat sich folgende Methode ausgedacht, um schrittweise einen Binärbaum zu konstruieren.

```
let rec push (n : 'a, t : Tree ⟨'a⟩) : Tree ⟨'a⟩ =
  match t with
  | Leaf           → Node (Leaf, n, Leaf)
  | Node (l, a, r) → Node (push (n, r), a, l)
```

☞ Um zu verhindern, dass der Baum zu einem Strunk entartet, fügt die Funktion stets rechts ein, vertauscht aber in jedem Schritt den linken und den rechten Teilbaum.

Helfen Sie Lisa bei der Analyse: Welche Form hat der Baum, wenn nacheinander die Zahlen 1 bis n eingefügt werden: $push\ (1, push\ (2, \ldots (push\ (n, Leaf))))$? Wie sieht der Baum für den Fall $n = 2^k - 1$ aus?

28. Suchen

Mini Softwareprojekt: Verwaltung von Personaldaten eines Unternehmens.

Der Personalstamm wird durch eine Liste von Einträgen der Typs *Entry* repräsentiert.

type $Entry = \{ key : Nat; person : Person \}$

☞ Jeder Eintrag besteht aus einer *eindeutigen* Personalnummer und den eigentlichen Personendaten.

$$\begin{aligned} \textbf{\textit{let}}\ team = [&\{ key = 7; \quad person = ralf \quad \}; \\ &\{ key = 815; \ person = melanie \}; \\ &\{ key = 4711; person = julia \quad \}; \\ &\{ key = 4712; person = andres \ \}] \end{aligned}$$

28. Suchen in einer Liste

Wiederkehrende Aufgabe: zu einer gegebenen Personalnummer die zugehörigen Personendaten heraussuchen.

look-up (*key* : *Nat*, *staff* : *List* ⟨*Entry*⟩) : *Person*

☞ Was machen wir, wenn kein passender Eintrag existiert?

Idee: die beiden möglichen Resultate einer Suche, erfolglos und erfolgreich, mit einem Datentyp darstellen. *Zur Erinnerung:*

type *Option* ⟨*'a*⟩ = | *None*
| *Some* ***of*** *'a*

Damit können wir die Signatur von *look-up* verfeinern.

look-up (*key* : *Nat*, *staff* : *List* ⟨*Entry*⟩) : *Option* ⟨*Person*⟩

☞ Schlägt die Suche fehl, wird *None* zurückgegeben, sonst *Some p*, wobei *p* die gesuchte Person ist.

28. Suchen in einer Liste

Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []              → ...
  | entry :: entries → ... look-up (key, entries) ...
```

28. Suchen in einer Liste

- *Rekursionsbasis*: die Suche schlägt fehl.

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []                  → None
  | entry :: entries →  . . . look-up (key, entries) . . .
```

- *Rekursionsschritt*: Ist das Kopfelement die gesuchte Person? Anderenfalls weitersuchen.

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []                  → None
  | entry :: entries →
    if key = entry.key then Some entry.person
                       else look-up (key, entries)
```

28. Suchen in einer geordneten Liste

Die (erfolglose) Suche lässt sich etwas beschleunigen, wenn wir annehmen, dass die Liste nach der Personalnummer geordnet ist.

Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []              → ...
  | entry :: entries → ... look-up (key, entries) ...
```

Vorbedingung: staff ist nach der Personalnummer geordnet. Das Typsystem stellt diese Eigenschaft nicht sicher, darum müssen wir uns kümmern.

28. Suchen in einer geordneten Liste

- *Rekursionsbasis*: die Suche schlägt fehl.

```
let rec look-up (key : Nat, staff : List ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | []                  → None
  | entry :: entries →  . . . look-up (key, entries) . . .
```

- *Rekursionsschritt*: aus dem 2-Wege wird ein 3-Wege Vergleich.

```
let rec look-up (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  look-up (key, entries)
```

☞ Die erfolgreiche Suche wird nicht beschleunigt; im Durchschnitt werden genauso viele rekursive Aufrufe benötigt.

28. Binärbäume

☞ Die Implementierung von *look-up* erinnert an die lineare Suche aus Teil III. Können wir die binäre Suche adaptieren?
Nein, nicht ohne den Geschwindigkeitsvorteil zu verlieren: im Gegensatz zur Halbierung des Suchintervalls ist die Halbierung einer Liste aufwändig (wie aufwändig?).

Wenn wir schnell auf das mittlere Element zugreifen wollen, brauchen wir einen anderen Containertyp.

```
type Tree ⟨'a⟩ =
    | Leaf
    | Node of Tree ⟨'a⟩ * 'a * Tree ⟨'a⟩
```

- *Leaf* tritt an die Stelle von *Nil*;
- *Node* (l, x, r) tritt an die Stelle von *Cons* (x, xs) und repräsentiert eine mindestens einelementige Folge, bestehend aus der „linken“ Teilfolge l, dem Element x und der „rechten“ Teilfolge r.

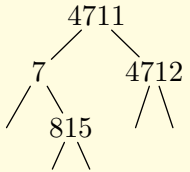
28. Binärbäume

Die Folge $7\ 815\ 4711\ 4712$ kann zum Beispiel durch

$$\begin{aligned}&\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}, 7, \mathit{Node}\ (\mathit{Leaf}, 815, \mathit{Leaf})), 4711,\\ &\qquad \mathit{Node}\ (\mathit{Leaf}, 4712, \mathit{Leaf}))\end{aligned}$$

repräsentiert werden (es gibt noch 13 andere Möglichkeiten).

Die Elemente von *Tree* heißen auch *Binärbäume*; Baum wegen der hierarchischen Struktur; *binär*, da jeder nicht-leere Baum in zwei Teilbäume verzweigt.



☞ Sind die Elemente von links nach rechts geordnet (siehe oben), so spricht man weiterhin von einem *Suchbaum*.

28. Suchen in einem Binärbaum

Mit dem Struktur Entwurfsmuster für *Tree* erhalten wir:

```
let rec look-up (key : Nat, staff : Tree ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
    | Leaf                    → ...
    | Node (left, entry, right) →
      ... look-up (key, left) ... look-up (key, right) ...
```

☞ Im Rekursionsschritt dürfen wir die Teillösungen für den linken *und* den rechten Teilbaum verwenden.

28. Suchen in einem Binärbaum

- *Rekursionsbasis*: die Suche schlägt fehl.

```
let rec look-up (key : Nat, staff : Tree ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | Leaf                       → None
  | Node (left, entry, right) →
    ... look-up (key, left) ... look-up (key, right) ...
```

- *Rekursionsschritt*: der 3-Wege Vergleich bleibt erhalten.

```
let rec look-up (key : Nat, staff : Tree ⟨Entry⟩) : Option ⟨Person⟩ =
  match staff with
  | Leaf                       → None
  | Node (left, entry, right) →
    if    key < entry.key    then look-up (key, left)
    elif  key = entry.key    then Some entry.person
    (* key > entry.key *)    else look-up (key, right)
```

28. Suchen in einem Binärbaum

Welche Laufzeit hat die neue Version von *look-up*?

☞ Das kommt auf die Form des Suchbaums an.

- Ein Binärbaum heißt *ausgeglichen* oder *balanciert*, wenn die Elemente links und rechts jeweils „gleichmäßig“ verteilt sind.
- Ein Binärbaum heißt *degeneriert*, wenn einer der Teilbäume jeweils leer ist (der Baum entspricht einer Liste).

Laufzeit von *look-up*:

- ausgeglichener Baum: logarithmische Laufzeit,
- degenerierter Baum: lineare Laufzeit.

Die Suche ist linear zur *Höhe* des Suchbaums.

28. Schlag die Nachbarn!

- *Aufgabe:*

  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!

- Zum Beispiel:

0	1	2	3	4	5	6	7	8	9
0	4	2	7	6	5	3	9	8	1

- Schachteln #1, #3, and #7 schlagen ihre Nachbarn.
- Schachtel #7 enthält die größte Zahl.

Gibt es denn tatsächlich immer eine Schachtel, die ihre Nachbarn schlägt?

Nö — alle Schachteln enthalten die gleiche Zahl.

0	0	0	0	0	0	0	0	0	0

Vielleicht ist mit „größer“ tatsächlich „größer gleich“ gemeint?

Na ja, selbst dann nicht.

0	1	2	3	4	5	6	7	8	9

Du meinst, die letzte Schachtel zählt nicht, weil sie keinen rechten Nachbarn hat?

Genau ;-).

28. Schlag die Nachbarn: Spielregeln

- Die mittlere Schachtel schlägt ihre Nachbarn, wenn

$\cdots$	l	a	r	$\cdots$

$$l \leqslant a \geqslant r$$

- 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	$-\infty$

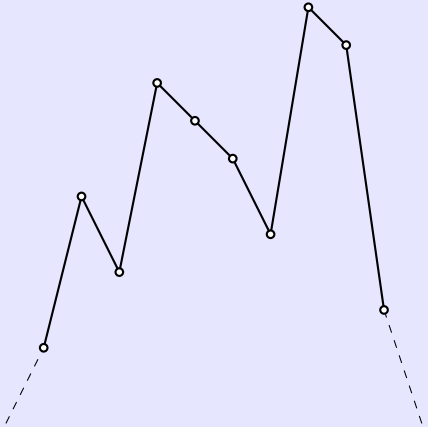
- (*Trick:* so lassen sich Sonderfälle vermeiden. Wenn wir später die Korrektheit beweisen, lassen wir -1 als Hausnummer zu, obwohl -1 keine natürliche Zahl ist.)

28. Schlag die Nachbarn: lokale Maxima

- *Einsicht*: wir suchen *ein* lokales Maximum.

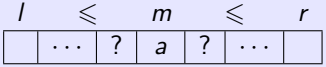
-1	0	1	2	3	4	5	6	7	8	9	10
$-\infty$	0	4	2	7	6	5	3	9	8	1	$-\infty$

- Insgesamt gibt es drei lokale Maxima (und zwei lokale Minima).

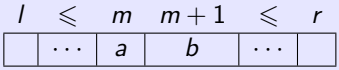


28. Schlag die Nachbarn: Lösungsideen

- *Naive Lösung:* wir öffnen alle Schachteln (globales Maximum).
- Wir suchen aber lediglich ein *lokales* Maximum.
- *Idee:* binäre Suche?

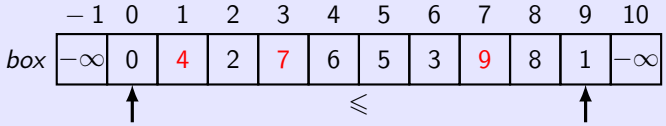


- ...nicht sehr informativ.
- *Idee:* wir öffnen zwei benachbarte Schachteln, zum Beispiel die beiden mittleren Schachteln:

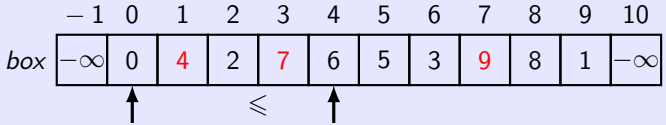


  - wenn $a \leqslant b$, setzen wir die Suche im Intervall $m+1 \ldots r$ fort;
  - wenn $a \geqslant b$, setzen wir die Suche im Intervall $l \ldots m$ fort.
- Finden wir so tasächlich ein lokales Maximum?

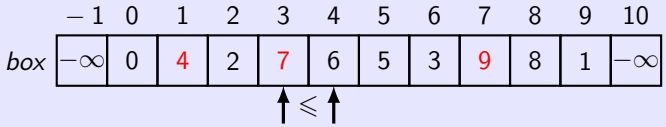
28. Schlag die Nachbarn: Beispiel



- *search* $(0, 9)$: *box* $4 \leqslant$ *box* 5? Nein!



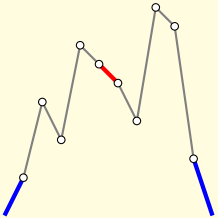
- *search* $(0, 4)$: *box* $2 \leqslant$ *box* 3? Ja!



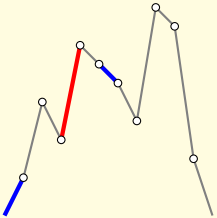
- *search* $(3, 4)$: *box* $3 \leqslant$ *box* 4? Nein!
- *search* $(3, 3)$: *box* $3 = 7$ ist ein lokales Maximum.

28. Schlag die Nachbarn: Beispiel

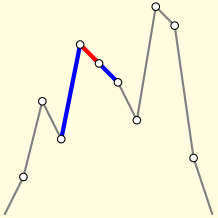
- Bisheriges Wissen in blau; Test in rot.



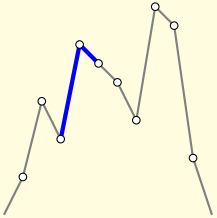
$box\ 4 > box\ 5$



$box\ 2 \leqslant box\ 3$



$box\ 3 > box\ 4$



$box\ 2 \leqslant box\ 3 \geqslant box\ 4$

28. Schlag die Nachbarn: Implementierung

Implementierung in Mini–F#:

```
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)
  search (lower, upper)
```

28. Schlag die Nachbarn: Korrektheit

- Wie können wir die Korrektheit von *beat-your-neighbours* zeigen?
- Die Problembeschreibung suggeriert die *Spezifikation*:

$$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1) \quad \text{wobei} \quad i = search\ (l, u)$$

- Eine Spezifikation beschreibt, *was* eine Funktion leisten soll; im Unterschied zu einer Implementierung, die genau festlegt, *wie* eine Funktion ihr Ergebnis ermittelt.

28. Schlag die Nachbarn: Korrektheit

- *Spezifikation*:

  $$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1) \quad \text{wobei } i = search\ (l, u)$$

- Das Programm erfüllt die Spezifikation nicht!

-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 Aufruf *search* $(5, 5)$ zum Beispiel ergibt 5; *box* $5 = 5$ ist aber kein lokales Maximum.
- Was läuft schief? Drei Möglichkeiten:
  - das Programm ist falsch oder
  - die Spezifikation ist falsch oder
  - beide sind falsch.

28. Schlag die Nachbarn: Korrektheit

Die Korrektheit von *beat-your-neighbours* hängt von einer Annahme ab.

- *Vorbedingung: search* erwartet, dass

$$box\ (l-1) \leqslant box\ l \quad \wedge \quad box\ u \geqslant box\ (u+1)$$

- *Nachbedingung: search* garantiert, dass

$$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1) \quad \text{wobei} \quad i = search\ (l, u)$$

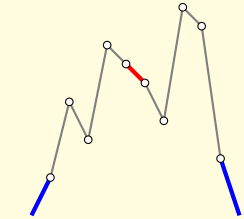
- Die Spezifikation hat die Form einer Implikation:

$$\text{Vorbedingung} \Longrightarrow \text{Nachbedingung}$$

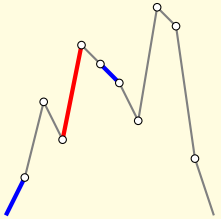
- Die Vorbedingung heißt auch *Invariante*, da sie über rekursive Aufrufe hinweg unverändert bleibt.

28. Schlag die Nachbarn: Korrektheit

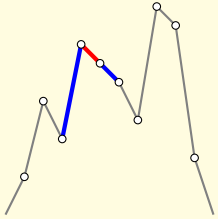
- Invariante in blau; Test in rot.



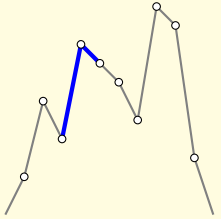
Invariante wird etabliert



Invariante wird erhalten



Invariante wird erhalten



Invariante impliziert das Ergebnis

28. Schlag den Nachbarn: Korrektheit

Invariante von *search* (l, u):
$box\ (l-1) \leqslant box\ l \ \wedge \ box\ u \geqslant box\ (u+1)$

- der initiale Aufruf *search* $(lower, upper)$ etabliert die Invariante:

$box\ (lower-1) = -\infty < box\ lower$
$box\ upper > -\infty = box\ (upper+1)$

- die rekursiven Aufrufe *search* $(m+1, u)$ und *search* (l, m) erhalten die Invariante: die Bedingungen

$box\ m \leqslant box\ (m+1) \ \wedge \ box\ u \geqslant box\ (u+1)$
$box\ (l-1) \leqslant box\ l \ \wedge \ box\ m \geqslant box\ (m+1)$

folgen aus der Invariante und der Abfrage $box\ m \leqslant box\ (m+1)$.
- die Invariante impliziert die gewünschte Nachbedingung:

$box\ (i-1) \leqslant box\ i \geqslant box\ (i+1)$

28. Schlag den Nachbarn: Korrektheit

Hmm, ist damit schon alles bewiesen? Müssen wir nicht auch sicherstellen, dass nur Schachteln aus dem Intervall *lower* . . *upper* geöffnet werden?

Gut beobachtet Lisa!

Und terminiert das Programm auch immer?

Wenden wir uns zunächst einem alten Bekannten zu ...

28. Binäre Suche — da capo

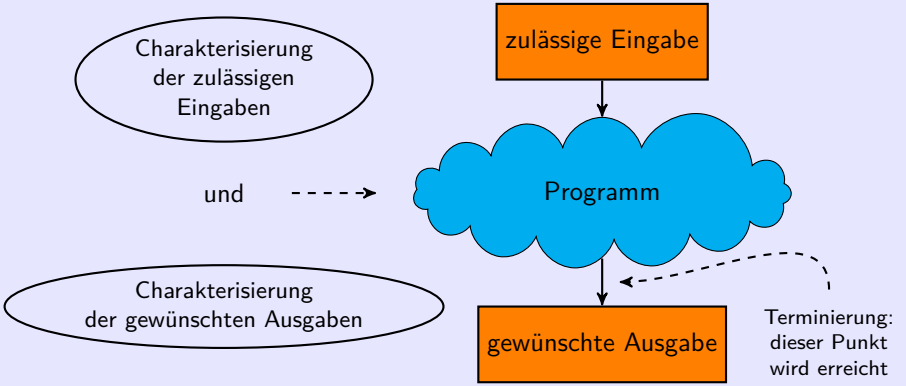
Binäre Suche:

```
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)
  search (lower, upper)
```

Zur Erinnerung: das Orakel gibt zu einem gegebenen n Auskunft, ob die gesuchte Zahl gleich oder kleiner als n ist.

28. Partielle und totale Korrektheit

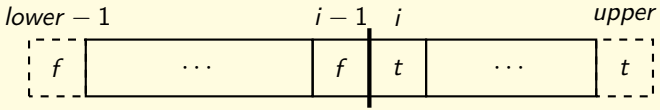
- *Partielle Korrektheit: wenn* das Programm terminiert, *dann* produziert es die gewünschte Ausgabe (für alle zulässigen Eingaben).
- *Totale Korrektheit:* das Programm terminiert *und* es produziert die gewünschte Ausgabe (für alle zulässigen Eingaben).



- totale Korrektheit $\cong$ partielle Korrektheit und Terminierung

28. Partielle Korrektheit der binären Suche

- Wie können wir formalisieren, dass das Orakel nicht mogelt?



- Zunächst gibt *oracle* immer *false* zurück; ab dem gesuchten Index immer *true*. Mit anderen Worten: *oracle* ist eine *monotone* Funktion.
- *Vorbedingungen:*

1. $i \leqslant j \implies \mathit{oracle}\ i \leqslant \mathit{oracle}\ j$
2. $\mathit{lower} \leqslant \mathit{upper}$
3. $\mathit{oracle}\ (\mathit{lower} - 1) < \mathit{oracle}\ \mathit{upper}$

- *Nachbedingung:*

$\mathit{oracle}\ (i-1) < \mathit{oracle}\ i \quad \text{wobei} \quad i = \mathit{binary\text{-}search}\ \mathit{oracle}\ (\mathit{lower}, \mathit{upper})$

28. Zwischenspiel: Ordnung auf Booleschen Werten

Die Booleschen Werte sind wie folgt angeordnet:

$$\mathit{false} < \mathit{true}$$

Wahrheitstafel für die Vergleichsoperationen:

a	b	$a \leqslant b$	$a < b$	$a = b$	$a <> b$
false	*false*	*true*	*false*	*true*	*false*
false	*true*	*true*	*true*	*false*	*true*
true	*false*	*false*	*false*	*false*	*true*
true	*true*	*true*	*false*	*true*	*false*

☞ $\leqslant$ ist die Implikation $\Longrightarrow$ („ex falso quodlibet“). (Gibt es auch andere Namen für die Vergleichsoperationen $<$, $=$ und $<>$?)

28. Partielle Korrektheit der binären Suche

1. Invariante von *search* (l, u):
$\mathit{oracle}\ (l-1) < \mathit{oracle}\ u$

- der initiale Aufruf *search* $(\mathit{lower}, \mathit{upper})$ etabliert die Invariante:

  $\mathit{oracle}\ (\mathit{lower}-1) < \mathit{oracle}\ \mathit{upper}$

- die rekursiven Aufrufe *search* $(m+1, u)$ und *search* (l, m) erhalten die Invariante: die Bedingungen

  $\mathit{oracle}\ m < \mathit{oracle}\ u$
  $\mathit{oracle}\ (l-1) < \mathit{oracle}\ m$

  folgen aus der Invariante und der Abfrage $\mathit{oracle}\ m$.
- die Invariante impliziert das gewünschte Ergebnis:

  $\mathit{oracle}\ (i-1) < \mathit{oracle}\ i$

  folgt aus $l = i = u$. *Aber:* Warum gilt $l = u$?

28. Partielle Korrektheit der binären Suche

Wir müssen zusätzlich garantieren, dass die Funktion *oracle* nur mit Werten aus ihrem Definitionsbereich aufgerufen wird.

2. Invariante von *search* (l, u):
$lower \leqslant l \leqslant u \leqslant upper$

- der initiale Aufruf *search* $(lower, upper)$ etabliert die Invariante.
- die rekursiven Aufrufe *search* $(m+1, u)$ und *search* (l, m) erhalten die Invariante: die Bedingungen folgen aus

$$l < u \implies l \leqslant m < m+1 \leqslant u$$

mit $m = (l+u) \div 2$. *Zur Erinnerung:* $m = \lfloor (l+u)/2 \rfloor$.

- die Invariante stellt sicher, dass *oracle* m definiert ist:

$$lower \leqslant m < upper$$

28. Terminierung der binären Suche

Wir müssen zeigen, dass bei jedem rekursiven Aufruf die Argumente „echt kleiner“ werden.

☞ 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:

$$\begin{aligned} u \dot{-} (m+1) < u \dot{-} l &\iff l < m+1 \\ m \dot{-} l \quad\;\; < u \dot{-} l &\iff m < u \end{aligned}$$

Beide Bedingungen folgen aus der Eigenschaft des „Mittelwerts“ m mit $m = \lfloor (l+u)/2 \rfloor$.

$$l < u \implies l \leqslant m < m+1 \leqslant u$$

28. Anforderungen und Garantien

Metapher: die Benutzer*in und die Bibliotheksfunktion *binary-search* gehen einen Vertrag ein (engl. contract).

Beobachtung:

- die Benutzer*in stellt Anforderungen an das Ergebnis von *binary-search*;
- und an die Argumente von *oracle*;
- die Funktion *binary-search* stellt Anforderungen an ihr Argument;
- und an die Ergebnisse von *oracle*.

Die Gegenseite muss die Anforderungen jeweils erfüllen bzw. garantieren.

☞ *Allgemein:* die Benutzer*in stellt Anforderungen an das Ergebnis; die Gegenseite stellt Anforderungen an die Argumente. Für funktionale Argumente wie *oracle* kehren sich Anforderungen (engl. requirements) und Garantien (engl. guarantees) um.

Komisch: der Korrektheitsbeweis benutzt gar nicht, dass die Funktion *oracle* nicht mogelt.

In der Tat! Wenn ein Beweis eine Annahme nicht verwendet, hat das in der Regel zwei mögliche Ursachen: entweder der Beweis ist schlicht und einfach falsch, oder das Theorem ist tatsächlich allgemeiner als angenommen.

Ich hab noch mal fix nachgerechnet: der Korrektheitsbeweis ist wirklich korrekt ;-).

Okay, wenn das Orakel mogelt, dann gibt es möglicherweise mehrere Stellen i mit $\mathit{oracle}\ (i-1) < \mathit{oracle}\ i$. Die binäre Suche findet dann zumindest irgendeine dieser Stellen.

Unter der Annahme, dass $\mathit{oracle}\ (\mathit{lower}-1) < \mathit{oracle}\ \mathit{upper}$.

Heißt das nicht, dass sich *beat-your-neighbours* auf *binary-search* zurückführen lässt?

Hmm, die Programme sind sich in der Tat ziemlich ähnlich:

```
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)
```

OK, die Tests $l \geqslant u$ und $l = u$ sind gleichwertig, da wir $l \leqslant u$ annehmen. Dann müssen wir noch die Zweige der zweiten Fallunterscheidung vertauschen ...

...indem wir die Bedingung negieren. Funzt:

```
let beat-your-neighbours (box : Nat → Nat) =
  binary-search (fun i → box i > box (i + 1))
```

29. Knobelaufgabe #14

Harry Hacker behauptet eine Funktion definiert zu haben, die einen Binärbaum der Größe n in logarithmischer (!) Zeit konstruiert. Seine Funktion erfüllt die Eigenschaft

$$size\ (create\ n) = n$$

wobei *size* wie folgt definiert ist:

```
let rec size = function
  | Leaf          -> 0
  | Node (l, a, r) -> size l + 1 + size r
```

Genie oder Scharlatan?

29. Mengen und endliche Abbildungen

Zurück zum *Mini Softwareprojekt:* Verwaltung von Personaldaten eines Unternehmens.

Weitere typische Aufgaben:

- Personen zum Personalstamm hinzufügen;
- Personen aus dem Personalstamm entfernen.

☞ Wir realisieren im Prinzip eine *endliche Abbildung* von Personalnummern auf Personendaten.

Ambitionierter: wir abstrahieren von unserer Anwendung und programmieren endliche Abbildungen (wir internalisieren $A \rightarrow_{\mathrm{fin}} B$).

29. Abstrakte Datentypen

- Ein Typ mit einer zugehörigen Menge von Operationen heißt *abstrakter Datentyp* (ADT).
- Zu einem abstrakten Datentyp gehört
  - eine *Schnittstelle* (engl. interface), die die verfügbaren Typen und Operationen beschreibt („was“), und
  - eine oder mehrere *Implementierungen*, die die Typen und Operationen realisieren („wie“).
- *Idee:* Implementierungsdetails werden vor den Klienten verborgen.
- *Vorteil:* Implementierungen lassen sich einfach austauschen — ein Gewinn an Modularität.
- (Good SE practice: programming against an interface.)
- konkrete versus abstrakte Datentypen:
  - ein konkreter Datentyp wird durch seine Elemente definiert;
  - ein abstrakter Datentyp wird durch seine Operationen definiert.

29. Abstrakter Datentyp: endliche Menge

Statt endlicher Abbildungen behandeln wir den konzeptionell etwas einfacheren ADT „endliche Menge“ — endliche Abbildungen im Skript.

Schnittstelle:

type $Set\ \langle 'elem$ ***when*** $'elem : comparison \rangle$

val *empty* : $Set\ \langle 'elem \rangle$
val *add* : $'elem * Set\ \langle 'elem \rangle \to Set\ \langle 'elem \rangle$
val *remove* : $'elem * Set\ \langle 'elem \rangle \to Set\ \langle 'elem \rangle$
val *is-empty* : $Set\ \langle 'elem \rangle \to Bool$
val *contains* : $'elem * Set\ \langle 'elem \rangle \to Bool$
val *from-list* : $List\ \langle 'elem \rangle \to Set\ \langle 'elem \rangle$
val *to-list* : $Set\ \langle 'elem \rangle \to List\ \langle 'elem \rangle$

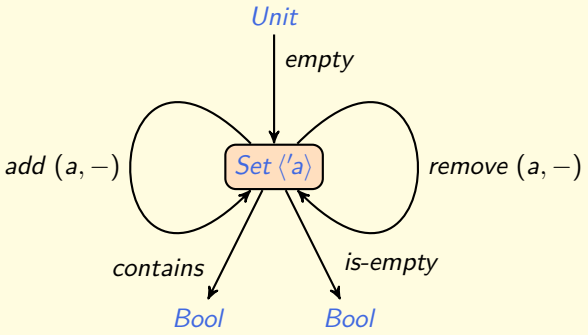
☞ Mengen sind wie Listen Containertypen; der Zusatz ***when*** $'elem : comparison$ schränkt den Elementtyp auf Typen ein, die die Vergleichsoperationen $\leqslant$, $<$ etc unterstützen.

Mathematikbrille: $\emptyset$, $\{a\} \cup A$, $A \setminus \{a\}$, $A = \emptyset$, $a \in A$.

29. Abstrakte Datentypen: Operationen

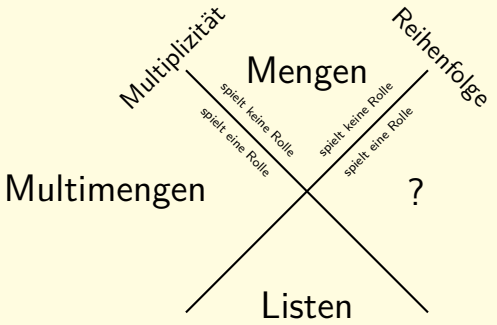
Plausibilitätsprüfung: ein ADT sollte Operationen bereitstellen

- um Elemente des ADTs zu *konstruieren*;
- um Elemente des ADTs in andere Elemente zu *transformieren*;
- um Elemente des ADTs zu *analysieren*.



29. Boom Hierarchie

- Mengen, Multimengen (engl. bags) und Listen sind Containertypen (engl. container types, collection types).
- Eine Menge ist eine ungeordnete Sammlung von Elementen; auch spielt die Multiplizität von Elementen keine Rolle.
- Eine Multimenge ist eine ungeordnete Sammlung von Elementen.
- Eine Liste ist eine geordnete Sammlung von Elementen.



29. Boom Hierarchie

- Die Mengen $\{b; o; m\}$, $\{m; o; b\}$ und $\{b; o; o; m\}$ sind gleich.
- Die Multimengen $\wr b; o; m \int$ und $\wr m; o; b \int$ sind gleich; die Multimengen $\wr b; o; m \int$ und $\wr b; o; o; m \int$ sind verschieden.
- Die Listen $[b; o; m]$, $[m; o; b]$ und $[b; o; o; m]$ sind alle verschieden.
- ACI Eigenschaften:
  - *assoziativ:* $(a + b) + c = a + (b + c)$;
  - *kommutativ:* $a + b = b + a$ (Reihenfolge spielt keine Rolle);
  - *idempotent:* $a + a = a$ (Multiplizität spielt keine Rolle).
- Mengen-, Multimengen- und Listenoperationen unterscheiden sich in den Gesetzen, die sie erfüllen:

	A	C	I
Mengenvereinigung $\cup$	ja	ja	ja
Multimengenvereinigung $\uplus$	ja	ja	nein
Listenkonkatenation @	ja	nein	nein

29. Endliche Mengen: Implementierung — Listen

Implementierung 1: wir repräsentieren endliche Mengen durch ungeordnete Listen, die möglicherweise Duplikate enthalten.

Der Containertyp *Set* wird durch einen Variantentyp mit einer einzigen Varianten implementiert (siehe auch Folie 316).

```
type Set ⟨'elem when 'elem : comparison⟩ =
    | Rep of List ⟨'elem⟩
let empty = Rep []
let add (key, Rep list) = Rep (key :: list)
```

☞ Der Konstruktor *Rep* überführt die Repräsentation einer Menge, eine Liste, in den abstrakten Typ der Menge.

29. Endliche Mengen: Implementierung — Listen

```
let remove (key, Rep list) =
  let rec del = function
   | []      → []
   | x :: xs → if key = x then del xs else x :: del xs
  in Rep (del list)
```

☞ Das zu löschende Element kann mehrfach vorkommen; wir müssen alle Vorkommen entfernen.

29. Endliche Mengen: Implementierung — Listen

```
let is-empty (Rep list) = List.is-empty list

let contains (key, Rep list) =
  let rec find = function
  | []      → false
  | x :: xs → key = x || find xs
  in find list

let from-list list = Rep list

let to-list (Rep list) = list
```

29. Endliche Mengen: Implementierung—Suchlisten

Implementierung 2: wir repräsentieren endliche Mengen durch aufsteigend geordnete Listen, die keine Duplikate enthalten.

Kurz: durch Suchlisten (das ist kein etablierter Begriff).

```
type Set ⟨'elem when 'elem : comparison⟩ =
    | Rep of List ⟨'elem⟩

let empty = Rep []

let add (key, Rep list) =
  let rec ins = function
    | []      → [key]
    | x :: xs → if    key < x   then key :: x :: xs
                elif  key = x   then key :: xs
                (* key > x *)  else  x :: ins xs
  in Rep (ins list)
```

☞ Beim Einfügen wird ein bereits vorhandenes Element „überschrieben“.

29. Endliche Mengen: Implementierung—Suchlisten


```
let remove (key, Rep list) =
  let rec del = function
    | []      → []
    | x :: xs → if    key < x   then x :: xs
                elif  key = x   then xs
                (* key > x *)  else  x :: del xs
  in Rep (del list)
```

☞ Es muss höchstens ein Element entfernt werden, da die Listen keine Duplikate enthalten.

29. Endliche Mengen: Implementierung—Suchlisten

```
let is-empty (Rep list) = List.is-empty list

let contains (key, Rep list) =
  let rec find = function
  | []      → false
  | x :: xs → key = x || key > x && find xs
  in find list

let from-list list = Rep (List.distinct (List.sort list))

let to-list (Rep list) = list
```

29. Endliche Mengen: Implementierung—Suchbäume

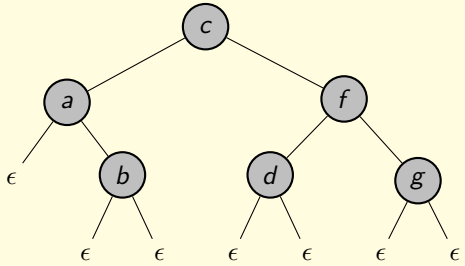
Implementierung 3: wir repräsentieren endliche Mengen durch binäre Suchbäume, die keine Duplikate enthalten.

```
type Tree ⟨'a⟩ =
    | Leaf
    | Node of Tree ⟨'a⟩ * 'a * Tree ⟨'a⟩
```

Schauen wir uns Binärbäume und binäre Suchbäume noch einmal in Ruhe an ...

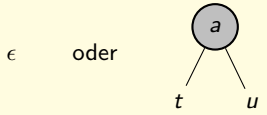
29. Binärbäume

- Der Rekursionsbaum der binären Suche ist ein *Binärbaum*.
- *Idee:* eine *Kontrollstruktur wird zu einer Datenstruktur*, so dass das Leibiz Entwurfsmuster zum Struktur Entwurfsmuster wird.



29. Binärbäume: Definition

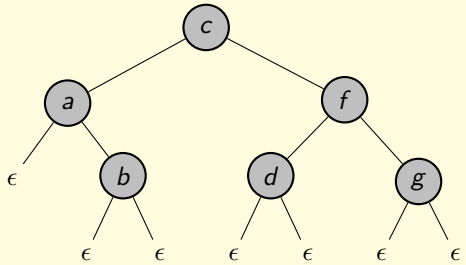
- Ein *binärer Baum* ist entweder
  - leer oder
  - ein Knoten, der aus einem linken Baum, einem Element und einem rechten Baum besteht.



- Leere Bäume heißen auch *Blätter*.
- Der linke und der rechte Baum sind *Teilbäume*.

29. Binärbäume: Begriffe

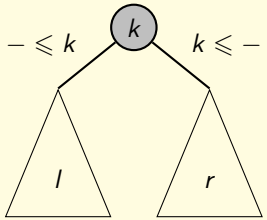
- In der Informatik wachsen Bäume typischerweise von oben nach unten; Bäume werden mit der Wurzel nach oben gezeichnet.



- Der Knoten c ist die *Wurzel*; die leeren Teilbäume sind die *Blätter*.
- Alle Knoten, mit Ausnahme der Wurzel, haben einen *Vorgänger*.
- d's Vorgänger ist f; f's Vorgänger ist c.
- Knoten können *Kinder* haben.
- f hat die Kinder d und g; d hat keine Kinder; a hat ein Kind.
- a und f sind *Geschwister*; d und g sind *Geschwister*.

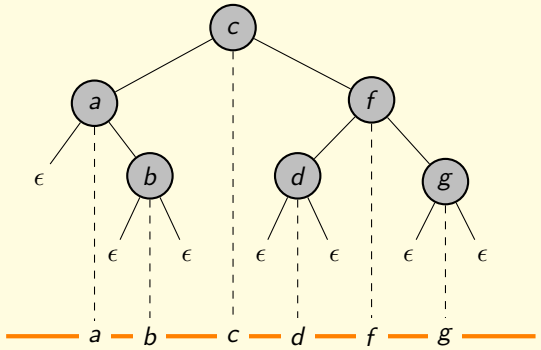
29. Binäre Suchbäume

- Ein *binärer Suchbaum* ist ein binärer Baum, so dass
  - der linke Teilbaum jedes Knotens nur Elemente enthält, die gleich oder kleiner sind als das Element in dem Knoten selbst;
  - der rechte Teilbaum jedes Knotens nur Elemente enthält, die gleich oder größer sind als das Element in dem Knoten selbst.



29. Binäre Suchbäume: Beispiel

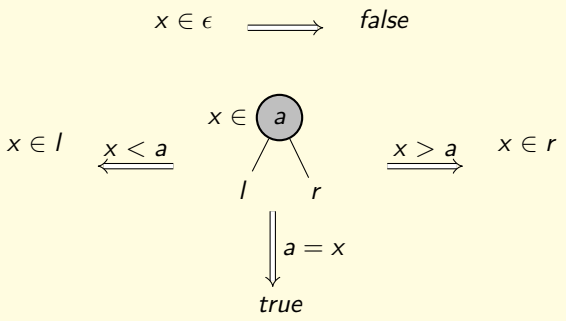
- Unser laufendes Beispiel ist ein binärer Suchbaum:



- Wenn wir die Elemente auf eine horizontale Linie projizieren, dann erhalten wir eine aufsteigend geordnete Sequenz.

29. Binäre Suchbäume: Suche

- Suchen eines Elements in einem binären Suchbaum mit einem 3-Wege Vergleich:



- Das Wurzelelement dient als Wegweiser.

29. Binäre Suchbäume: Suche

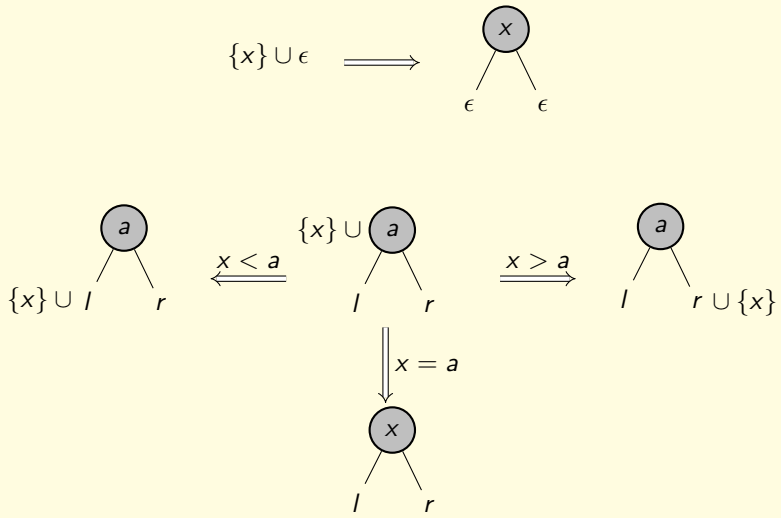
Der 3-Wege Vergleich wird mit einer geschachtelten Fallunterscheidung realisiert:

```
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
```

☞ *contains* ist die Mutter aller Algorithmen auf Suchbäumen.

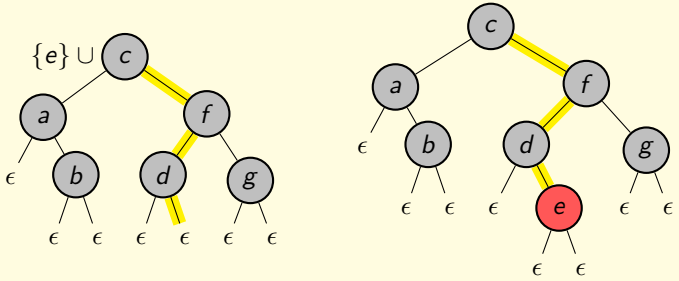
29. Binäre Suchbäume: Einfügen

Das Einfügen verwendet das gleiche Rekursionsmuster wie das Suchen.



29. Binäre Suchbäume: Einfügen — Beispiel

- Wir fügen e ein:



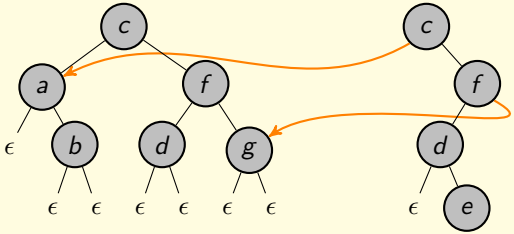
- Das neue Element e ersetzt einen leeren Teilbaum.

29. Binäre Suchbäume: Einfügen

```
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)
```

29. Einfügen — „sharing“

- Die Knoten entlang des Suchpfades werden neu angelegt.



- Die Teilbäume, die nicht traversiert werden, werden „geteilt“:
  - sie sind sowohl Teilbäume im ursprünglichen Baum (Eingabe)
  - als auch im erweiterten Baum (Ausgabe).
- (im Englischen spricht man von „sharing“)

29. Endliche Mengen: Implementierung—Suchbäume

Zur Erinnerung:

```
type Tree ⟨'a⟩ =
    | Leaf
    | Node of Tree ⟨'a⟩ * 'a * Tree ⟨'a⟩
```

Implementierung der Schnittstelle:

```
type Set ⟨'elem when 'elem : comparison⟩ =
    | Rep of Tree ⟨'elem⟩

let empty = Rep Leaf

let add (key, Rep tree) =
  let rec ins = function
    | Leaf           → Node (Leaf, key, Leaf)
    | Node (l, x, r) → if   key < x   then Node (ins l, x, r)
                       elif key = x   then Node (l, key, r)
                       (* key > x *) else  Node (l, x, ins r)
  in Rep (ins tree)
```

29. Endliche Mengen: Implementierung—Suchbäume

```
let is-empty (Rep tree) =
  match tree with
  | Leaf              → true
  | Node (_, _, _)    → false

let contains (key, Rep tree) =
  let rec find = function
  | Leaf              → false
  | Node (l, x, r)    → if    key < x    then find l
                        elif  key = x    then true
                        (* key > x *)   else  find r
  in find tree

let from-list list = Rep (balanced-tree (List.distinct (List.sort list)))

let to-list (Rep tree) = inorder tree
```

☞ Löschen von Elementen zur Übung (siehe auch Skript).

☞ *balanced-tree* und *inorder* gehen wir als nächstes an ...

29. Wechsel der Repräsentation

Wir haben drei Implementierungen des ADTs „endliche Menge“ kennengelernt: Listen, Suchlisten und Suchbäume. Qual der Wahl?

Wunsch: die verschiedenen Repräsentationen einer Menge ineinander zu überführen (zu konvertieren).

Als nächstes beschäftigen wir uns mit der Konstruktion und der Linearisierung von Binärbaumen.

☞ Lässt sich ein Suchbaum in linearer Zeit konstruieren?

29. Konstruktion eines Binärbaums

Aufgabe: Konstruktion eines balancierten Suchbaums aus einer geordneten Liste.

Idee: wir orientieren uns an der Struktur eines Suchbaums:

- *Rekursionsbasis*: Ist die Liste leer, so geben wir den leeren Suchbaum zurück.
- *Rekursionsschritt*: Eine mindestens einelementige Liste teilen wir in drei Teile auf, den linken Teil, das Wurzelelement und den rechten Teil. Um die Ausgeglichenheit des Suchbaums zu gewährleisten, müssen die beiden Teillisten möglichst gleich lang sein.
- *Teilaufgabe:* halbieren einer Liste.

29. Halbierung einer Liste

Spezifikation: wir suchen *eine* Umkehrfunktion der Listenkonkatenation.

$$x \mathbin{@} y = z \quad \text{wobei} \quad (x, y) = \mathit{halve}\ z$$

☞ Die Funktion *unzip* lässt sich somit nicht verwenden.

Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

```
let rec halve (list : List ⟨'a⟩) : List ⟨'a⟩ * List ⟨'a⟩ =
  match list with
  | []      → ...
  | x :: xs → ... halve xs ...
```

29. Halbierung einer Liste

- *Rekursionsbasis*:

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{halve}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle) : \mathit{List}\ \langle'a\rangle * \mathit{List}\ \langle'a\rangle = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\ \quad \mid [] \quad\ \to ([], []) \\ \quad \mid x :: xs \to \ \ldots \mathit{halve}\ xs \ldots\end{array}$$

- *Rekursionsschritt*: Wie können wir aus der halbierten Restliste eine halbierte Liste konstruieren?

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{halve}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle) : \mathit{List}\ \langle'a\rangle * \mathit{List}\ \langle'a\rangle = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\ \quad \mid [] \quad\ \to ([], []) \\ \quad \mid x :: xs \to \textbf{\textit{let}}\ (xs_1, xs_2) = \mathit{halve}\ xs\ \textbf{\textit{in}} \ldots\end{array}$$

Das Kopfelement x muss vor xs_1 gesetzt werden; eventuell muss das letzte Element von xs_1 zu xs_2 verschoben werden.
☞ Machbar, aber langsam!

29. Aufteilung einer Liste

Wir *verallgemeinern* die Aufgabe: eine Liste *list* wird in zwei Teillisten der Längen n und $len - n$ zerteilt, wobei *len* die Länge der Liste *list* ist.

☞ Halbierung ist dann ein Spezialfall mit $n = len \div 2$.

Jetzt kommt das Peano Entwurfsmuster zum Einsatz.

let rec *split* $(n : Nat, list : List\ \langle'a\rangle) : List\ \langle'a\rangle * List\ \langle'a\rangle =$
  if $n = 0$ ***then*** $\ldots$
    else $\ldots split\ (n \div 1, \ldots) \ldots$

29. Aufteilung einer Liste

- *Rekursionsbasis*: die erste Teilliste ist leer.

$$\begin{array}{l}\textbf{let rec } split\ (n : Nat, list : List\ \langle'a\rangle) : List\ \langle'a\rangle * List\ \langle'a\rangle = \\ \quad \textbf{if } n = 0 \textbf{ then } ([], list) \\ \qquad\qquad \textbf{else } \ldots split\ (n \dot{-} 1, \ldots) \ldots\end{array}$$

- *Rekursionsschritt*: Wir machen zusätzlich eine Fallunterscheidung über das Listenargument.

$$\begin{array}{l}\textbf{let rec } split\ (n : Nat, list : List\ \langle'a\rangle) : List\ \langle'a\rangle * List\ \langle'a\rangle = \\ \quad \textbf{if } n = 0 \\ \quad \textbf{then } ([], list) \\ \quad \textbf{else } \textbf{match } list \textbf{ with} \\ \qquad\quad | \ [] \qquad\ \to ([], []) \\ \qquad\quad | \ x :: xs \to \textbf{let } (xs_1, xs_2) = split\ (n \dot{-} 1, xs) \\ \qquad\qquad\qquad\quad\ \ (x :: xs_1, xs_2)\end{array}$$

29. Konstruktion eines Binärbaums

Zurück zur ursprünglichen Aufgabe: der Konstruktion eines balancierten Suchbaums aus einer geordneten Liste.

Wir verwenden das verallgemeinerte Leibniz Entwurfsmuster — wir „kämpfen“ gegen die Struktur von Listen an.

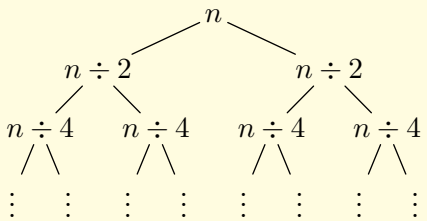
```
let rec balanced-tree (list : List ⟨'a⟩) : Tree ⟨'a⟩ =
  let n = length list
  if n = 0 then Leaf
           else let (xs₁, x :: xs₂) = split (n ÷ 2, list)
                Node (balanced-tree xs₁, x, balanced-tree xs₂)
```

☞ Der Abgleich mit dem Muster $(xs_1, x :: xs_2)$ kann nicht scheitern. (Warum?)

29. Konstruktion eines Binärbaums

☞ Welche Laufzeit hat *balanced-tree*?

Rekursive Aufrufstruktur von *balanced-tree* in Abhängigkeit von der Listenlänge:



Abschätzung:

- Höhe des Rekursionsbaumes: $\lg n$.
- Laufzeit pro Rekursionsebene: linear.
- Gesamtlaufzeit: $n \lg n$.

☞ Lässt sich die Laufzeit verbessern?

29. Linearisierung eines Binärbaums

Aufgabe: Überführung eines Binärbaums in eine Liste — ein Suchbaum soll dabei auf eine geordnete Liste abgebildet werden.

Das Struktur Entwurfsmuster für *Tree* führt fast direkt zum Ziel.

```
let rec inorder (tree : Tree ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → ...
  | Node (left, x, right) → ... inorder left ... inorder right ...
```

☞ *Zur Erinnerung:* Im Rekursionsschritt dürfen wir die Teillösungen für den linken *und* den rechten Teilbaum verwenden.

29. Linearisierung eines Binärbaums

- *Rekursionsbasis*: der leere Baum wird zur leeren Liste.

```
let rec inorder (tree : Tree ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → []
  | Node (left, x, right) → ... inorder left ... inorder right ...
```

- *Rekursionsschritt*: wir müssen zwei Listen aneinanderhängen.

```
let rec inorder (tree : Tree ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → []
  | Node (left, x, right) → inorder left @ x :: inorder right
```

☞ Die relative Reihenfolge der Elemente bleibt erhalten. Wegen der Position des Wurzelelements heißt die Funktion *inorder*.

29. Linearisierung eines Binärbaums: Laufzeit

- Wie schnell ist die Funktion *inorder*?
- Das hängt wie so oft von der Form des Binärbaums ab.
- Zunächst: wie schnell ist die Listenkonkatenation @?
- Die Funktion @ rekurriert über das erste Argument, die Laufzeit ist also proportional zur Länge der ersten Liste.
- Zurück zu *inorder*:
  - Wenn der linke Teilbaum immer leer ist: lineare Laufzeit.
  - Ist der rechte Teilbaum immer leer, dann werden nacheinander Listen der Längen 1, 2, ..., $n-2$, $n-1$ durchlaufen. Insgesamt: $1+2+\cdots+n-2+n-1=(n-1)\cdot n/2$ Schritte, also eine *quadratische* Laufzeit.

29. Linearisierung eines Binärbaums: Laufzeit

Die Laufzeit von *inorder* ist vielleicht unerwartet, auf jeden Fall ist sie unbefriedigend. Die folgende Tabelle zeigt warum.

$\lg n$	n	$n \lg n$	n^2
≈ 7	100	≈ 700	10.000
≈ 10	1.000	≈ 10.000	1.000.000
≈ 14	10.000	≈ 140.000	100.000.000
≈ 17	100.000	$\approx 1.700.000$	10.000.000.000
≈ 20	1.000.000	$\approx 20.000.000$	1.000.000.000.000

☞ Um zum Beispiel einen Baum mit zehntausend Elementen zu linearisieren — Bäume dieser Größenordnung sind nicht ungewöhnlich —, werden hundertmillionen Schritte benötigt.

29. Linearisierung eines Binärbaums

Wie können wir *inorder* verbessern?

Idee: wir *verallgemeinern* die Aufgabenstellung und programmieren eine Funktion, die gleichzeitig linearisiert *und* konkateniert.

Spezifikation:

$$\mathit{inorder\text{-}append}\ (\mathit{tree}, \mathit{list}) = \mathit{inorder}\ \mathit{tree}\ @\ \mathit{list}$$

29. Linearisierung eines Binärbaums

Mit dem Struktur Entwurfsmuster für *Tree* erhalten wir:

```
let rec inorder-append (tree : Tree ⟨'a⟩, list : List ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → ...
  | Node (left, x, right) →
    ... inorder-append (left, ...) ... inorder-append (right, ...) ...
```

29. Linearisierung eines Binärbaums

- *Rekursionsbasis*: die Liste wird zurückgegeben.

```
let rec inorder-append (tree : Tree ⟨'a⟩, list : List ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → list
  | Node (left, x, right) →
    ... inorder-append (left, ...) ... inorder-append (right, ...) ...
```

- *Rekursionsschritt*: wir müssen die rekursiven Aufrufe nur ineinander schachteln.

```
let rec inorder-append (tree : Tree ⟨'a⟩, list : List ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → list
  | Node (left, x, right) →
    inorder-append (left, x :: inorder-append (right, list))
```

29. Programmiertechnik: Rekursionsparadoxon

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.

(Beim Beweisen verwendet man ähnliche Techniken: Verstärkung der Induktion; auch bekannt unter dem Namen „Inventor's paradox".)

29. Übersicht — Laufzeit

	Liste	Suchliste	Suchbaum
empty	konstant	konstant	konstant
add	konstant	linear	linear zur Höhe
remove	linear	linear	linear zur Höhe
is-empty	konstant	konstant	konstant
contains	linear	linear	linear zur Höhe
from-list	konstant	linear-logarithmisch	linear-logarithmisch
to-list	konstant	konstant	linear

☞ Die Höhe eines Binärbaums ist im schlechtesten Fall linear zur Größe!

29. Zusammenfassung

Wir haben

- verschiedene Sortieralgorithmen kennengelernt (unter anderem Sortieren durch Mischen),
- mit Rekursionsbäumen die Laufzeit von Programmen abgeschätzt,
- elementare Suchstrukturen kennengelernt (unter anderem binäre Suchbäume),
- Vorbedingungen, Nachbedingungen und Invarianten eingeführt,
- zwischen partieller und totaler Korrektheit unterschieden,
- die Korrektheit der binären Suche bewiesen,
- gesehen, dass schwierigere Probleme manchmal einfacher zu lösen sind: Rekursionsparadoxon.

Teil VI

Grammatiken

29. Knobelaufgabe #15

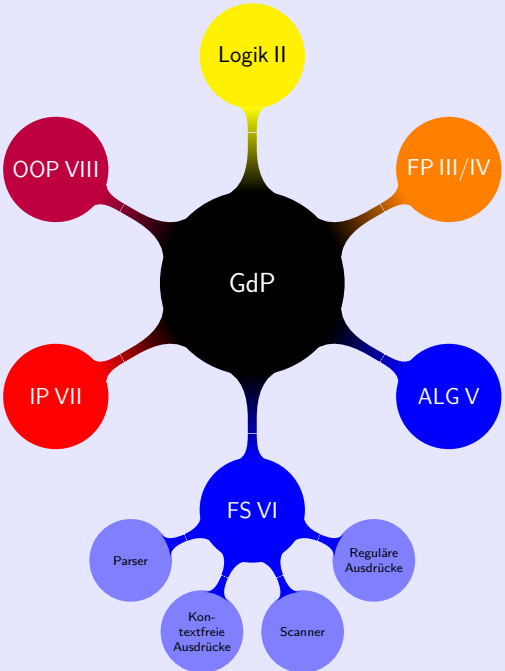
Ist es möglich, ein *nicht-rekursives* Programm zu schreiben, das *nicht terminiert*?

Nicht-rekursiv bedeutet, dass in dem Programm keine rekursiven Funktionsdefinitionen

$$\textbf{\textit{let rec}}\ f\ (x_1 : t_1) : t_2 = \ldots f \ldots$$

verwendet werden dürfen.

29. Gliederung


Logik II
OOP VIII
FP III/IV
GdP
IP VII
ALG V
FS VI
Parser
Reguläre Ausdrücke
Kon-
textfreie
Ausdrücke
Scanner

29. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- den Unterschied zwischen konkreter und abstrakter Syntax verstanden haben,
- Syntax und Semantik regulärer Ausdrücke kennen,
- aus einem regulären Ausdruck einen Akzeptor konstruieren können,
- den Unterschied zwischen Interpreter und Übersetzer verstanden haben,
- Syntax und Semantik kontextfreier Ausdrücke kennen,
- (aus einem kontextfreien Ausdruck einen Parser konstruieren können,)
- Möglichkeiten und Grenzen der Sprachfamilien kennen.

29. Überblick

Dieses Kapitel erzählt eine der großen Erfolgsgeschichten der Informatik: die automatische Umsetzung einer deskriptiven Beschreibung („was“) in ein ausführbares Programm („wie“).

Wir führen Formalismen ein, mit denen man die konkrete Syntax einer Programmiersprache beschreiben kann:

- die lexikalische Syntax und
- die kontextfreie Syntax.

29. Wiederholung: Lexikalische Syntax

Betrachten wir ein einfaches Beispielprogramm:

```
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.

29. Wiederholung: Lexeme

Als menschlicher Leser sind wir gewohnt — bzw. durch jahrelanges Training geschult — mehrere Zeichen zu einer Einheit zusammenzufassen.

4711	∗	(	a11	+	815	)

☞ Nicht alle Zeichen sind für den Rechner gedacht: `(* speed *)` ist ein Kommentar, der sich an den menschlichen Leser richtet.

In der *lexikalischen Syntax* einer Programmiersprache wird festgelegt, wie Zeichen zu größeren Einheiten, sogenannten *Lexemen* (engl. tokens), zusammengefasst werden.

29. Wiederholung: Kontextfreie Syntax

Nicht alle Folgen von Lexemen stellen ein gültiges Programm dar:

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

umfasst die gleichen Lexeme, ist aber *kein* Mini–F# Programm.

In der *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.

30. Lexeme in Mini–F#

- *Numerale:* ein Numeral in Mini–F# besteht aus einer nicht-leeren Folge von Dezimalziffern.
- *Bezeichner:* ein Bezeichner in Mini–F# beginnt mit einem Buchstaben, gefolgt von weiteren Buchstaben, Ziffern und Sonderzeichen, wie einem Unterstrich oder einem Apostroph.

☞ Wie können wir den Aufbau von Numeralen und Bezeichnern präzise beschreiben?

☞ Wir müssen uns eine Sprache ausdenken — eine Sprache, um Sprachen zu beschreiben.

30. Begriffe

- Ein *Alphabet* A ist eine Menge von Zeichen.
  Beispiele:
  - $\{\mathtt{a}, \mathtt{b}\}$,
  - $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), *, +\}$,
  - ASCII-Alphabet,
  - Menge aller Mini–F#-Lexeme.
- Eine *Sprache* ist eine Teilmenge von A^*, der Menge der Sequenzen über A.
- Statt von Sequenzen spricht man auch von *Wörtern*.
- Die Menge aller Sprachen ist $\mathbb{P}(A^*)$.

30. Motivation

Eine Sprache ist eine Menge von Wörtern; auf Wörtern haben wir bereits einige Operationen eingeführt.

- das *leere Wort*: ϵ,
- die *Konkatenation* von Wörtern: $w_1\ w_2$,
- die *n-fache Wiederholung*: w^n.

☞ Verallgemeinern wir w^n zu einer beliebigen Wiederholung w^*, so können wir Numerale beschreiben:

$$\mathit{digit}\ \mathit{digit}^*$$

☞ *digit* steht für die Sprache der Ziffern. *Lies:* ein Numeral ist eine Ziffer gefolgt von einer beliebigen Folge von Ziffern.

30. Motivation

Wie können wir *digit* beschreiben? Eine Ziffer ist entweder 0, oder 1, oder ... Erfinden wir eine Notation für die Alternative, '|', 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* | *digit* | _ | `)*

wobei *letter* durch a | ··· | z | A | ··· | Z gegeben ist.

30. Motivation

Jetzt können wir unser Routineprogramm abspulen!

Wir definieren die

- die abstrakte Syntax der Sprachbeschreibungssprache und
- die Semantik der Sprachbeschreibungssprache.

Mir schwirrt der Kopf, eine Sprachbeschreibungssprache! Drehen wir uns da nicht im Kreis? Ich meine, wir wollen doch die konkrete Syntax von Mini–F# festlegen und jetzt müssen wir zuerst die Syntax der Sprachbeschreibungssprache festlegen.

Aber, welche Alternativen gibt es? Willst Du die konkrete Syntax umgangssprachlich festlegen?

Ist mir schon bewusst, dass das keine gute Idee ist.

Wenn Du einen Formalismus verwendest, muss doch klar sein, was erlaubt ist und was gemeint ist.

Die Gründe liegen aber noch tiefer: Im Endeffekt wollen wir die Feststellung, ob ein Stück Text ein gültiges Mini–F# Programm ist, dem Rechner übertragen. Dazu muss eben Syntax und Semantik präzise festgelegt werden.

Mir schwant Böses: ein Mini–F# Programm, das überprüft, ob Mini–F# Programme korrekt sind ...

30. Abstrakte Syntax

$a \in A$		*Alphabet*
$r \in \mathrm{Reg} ::=$		*reguläre Ausdrücke:*
	ϵ	das leere Wort
	a	einzelnes Zeichen ≙ Terminalsymbol
	$r_1\ r_2$	Konkatenation ≙ Sequenz
	$\emptyset$	die leere Sprache
	$r_1 \mid r_2$	Alternative
	r^*	Wiederholung

☞ Die Symbole ϵ und $\emptyset$ sind *überladen*: je nach Kontext bedeuten sie etwas Verschiedenes — ϵ steht auch für das leere Wort und $\emptyset$ für die leere Menge.

30. Konkrete Syntax

Zwei konkrete Beispiele für konkrete Syntaxen:

	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	$a_1 \mid \cdots \mid a_n$	`.`	`.`
optionales Vorkommen	$\epsilon \mid r$	r`\?`	r`?`
positive Wiederholung	$r\ r^*$	r`\+`	r`+`

☞ `grep` und `egrep` durchsuchen Sequenzen (typischerweise Textdateien) nach Teilsequenzen, die auf ein bestimmtes Muster passen.
☞ \ ist ein Fluchtsymbol.

30. Semantik

☞ Eine Aufteilung in statische und dynamische Semantik ist nicht notwendig: reguläre Ausdrücke sind beliebig kombinierbar.

Wir definieren zwei verschiedene Semantiken:

- *Denotationelle Semantik:* „Welche Sprache bezeichnet der reguläre Ausdruck?“.
- *Reduktionssemantik:* „Wie kann ich ein einzelnes Wort aus dem regulären Ausdruck ableiten?“.

30. Denotationelle Semantik

- Die Bedeutung von ϵ ist $\{\epsilon\}$.
- Die Bedeutung von a ist $\{a\}$, wobei a das Wort $\{0 \mapsto a\}$ abkürzt.
- *Zur Erinnerung:* Die Bedeutung eines regulären Ausdrucks ist eine Sprache, kein Wort.
- Wir haben die Konkatenation von Wörtern definiert; jetzt müssen wir die Konkatenation auf Mengen von Wörtern fortsetzen.

$$L_1 \cdot L_2 = \{ w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2 \}$$

☞ Damit ist auch '$\cdot$' *überladen*.

- Beliebige Wiederholung einer Sprache:

$$L^* = \bigcup \{ L^n \mid n \in \mathbb{N} \}$$

mit $L^0 = \{\epsilon\}$ und $L^{n+1} = L \cdot L^n$.

30. Denotationelle Semantik

Die Semantik wird durch eine Bedeutungsfunktion angegeben:

$$
\begin{array}{ll}
[\![a]\!] & = \{\, a \,\} \\
[\![\epsilon]\!] & = \{\, \epsilon \,\} \\
[\![r_1\ r_2]\!] & = [\![r_1]\!] \cdot [\![r_2]\!] \\
[\![\emptyset]\!] & = \emptyset \\
[\![r_1 \mid r_2]\!] & = [\![r_1]\!] \cup [\![r_2]\!] \\
[\![r^*]\!] & = [\![r]\!]^*
\end{array}
$$

☞ Die doppelten Klammern sind die sogenannten *Oxford* oder *Strachey Klammern* — sie umschließen die Syntax.

30. Beispiele

Zugrundeliegendes Alphabet $A = \{\mathtt{a}, \mathtt{b}\}$.

$$
\begin{aligned}
& [\![\mathtt{a\ b} \mid \mathtt{b\ a}]\!] \\
= \ & [\![\mathtt{a\ b}]\!] \cup [\![\mathtt{b\ a}]\!] \\
= \ & ([\![\mathtt{a}]\!] \cdot [\![\mathtt{b}]\!]) \cup ([\![\mathtt{b}]\!] \cdot [\![\mathtt{a}]\!]) \\
= \ & (\{\mathtt{a}\} \cdot \{\mathtt{b}\}) \cup (\{\mathtt{b}\} \cdot \{\mathtt{a}\}) \\
= \ & \{\mathtt{ab}\} \cup \{\mathtt{ba}\} \\
= \ & \{\mathtt{ab}, \mathtt{ba}\}
\end{aligned}
$$

☞ Der reguläre Ausdruck $[\![\mathtt{a\ b} \mid \mathtt{b\ a}]\!]$ bezeichnet die Sprache $\{\mathtt{ab}, \mathtt{ba}\}$.

30. Beispiele

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 aller Wörter der Länge 2.

30. Beispiele

$$
\begin{aligned}
& [\![((\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}))^*]\!] \\
= \quad & [\![(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})]\!]^* \\
= \quad & \{\mathtt{ab}, \mathtt{aa}, \mathtt{bb}, \mathtt{ba}\}^*
\end{aligned}
$$

☞ Die bezeichnete Sprache umfasst alle Wörter gerader Länge.

30. Reduktionssemantik

Zur Erinnerung:

- *Denotationelle Semantik:* „Welche Sprache bezeichnet der reguläre Ausdruck?“.

  ☞ Globale Sicht.
- *Reduktionssemantik:* „Wie kann ich ein einzelnes Wort aus dem regulären Ausdruck ableiten?“.

  ☞ Lokale Sicht.

30. Reduktionssemantik

☞ Um die Semantik von Mini–F# festzulegen, haben wir Werte eingeführt, das sind besonders einfache Ausdrücke.

☞ In Analogie dazu: für die Reduktionssemantik sind Wörter besonders einfache reguläre Ausdrücke.

$w ::= \epsilon$		das leere Wort
	a	einzelnes Zeichen ∖ Terminalsymbol
	$w_1\ w_2$	Konkatenation ∖ Sequenz

☞ Ein Wort zeichnet sich dadurch aus, dass seine Denotation eine einelementige Menge ist.

30. Reduktionssemantik

Die schrittweise Reduktion $r \longrightarrow r'$ (lies: r kann in einem Schritt zu r' reduziert werden) wird durch ein Beweissystem formalisiert.

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 letzten vier Regeln formalisieren Wahlmöglichkeiten.

30. Beispiele

Für den regulären Ausdruck `a b | b a` existieren zwei mögliche Reduktionsfolgen.

`a b | b a` $\longrightarrow$ `a b`

`a b | b a` $\longrightarrow$ `b a`

☞ Die Sprache, die durch einen regulären Ausdruck beschrieben wird, ist die Menge aller *Worte*, die aus dem Ausdruck ableitbar sind. Für unser Beispiel: $\{\texttt{ab}, \texttt{ba}\}$.

30. Reduktionssemantik

Um den Ausdruck (a | b) (b | a) zu vereinfachen, benötigen wir weitere Regeln. Die obigen Beweisregeln erlauben nur den *gesamten* Ausdruck umzuformen; wir brauchen also Regeln, die es uns ermöglichen, einen Ausdruck „mittendrin“ zu manipulieren.

Kongruenz- oder *Kontextregeln*:

$$\frac{r_1 \longrightarrow r_1'}{r_1\ r_2 \longrightarrow r_1'\ r_2} \qquad \frac{r_2 \longrightarrow r_2'}{r_1\ r_2 \longrightarrow r_1\ r_2'}$$

30. Beispiele

Für $(a \mid b)\,(b \mid a)$ gibt es insgesamt vier mögliche Reduktionsfolgen.

$$(a \mid b)\,(b \mid a) \longrightarrow a\,(b \mid a) \longrightarrow a\,b$$
$$(a \mid b)\,(b \mid a) \longrightarrow a\,(b \mid a) \longrightarrow a\,a$$
$$(a \mid b)\,(b \mid a) \longrightarrow b\,(b \mid a) \longrightarrow b\,b$$
$$(a \mid b)\,(b \mid a) \longrightarrow b\,(b \mid a) \longrightarrow b\,a$$

☞ Die bezeichnete Sprache ist somit $\{ab, aa, bb, ba\}$.

30. Beispiele

Wesentlich mehr Möglichkeiten gibt es für $((a \mid b)\ (b \mid a))^*$, nämlich unendlich viele.

$$
\begin{aligned}
((a \mid b)\ (b \mid a))^* &\longrightarrow \epsilon \\
((a \mid b)\ (b \mid a))^* &\longrightarrow (a \mid b)\ (b \mid a)\ ((a \mid b)\ (b \mid a))^* \\
&\longrightarrow a\ (b \mid a)\ ((a \mid b)\ (b \mid a))^* \\
&\longrightarrow a\ b\ ((a \mid b)\ (b \mid a))^* \\
&\longrightarrow a\ b\ \epsilon \\
&\longrightarrow a\ b \\
&\ldots
\end{aligned}
$$

30. Beispiele

Zur Erinnerung: Die Sprache, die durch einen regulären Ausdruck beschrieben wird, ist die Menge aller *Worte*, die aus dem Ausdruck ableitbar sind.

$$\emptyset^* \longrightarrow \emptyset\ \emptyset^* \longrightarrow \emptyset\ \emptyset\ \emptyset^* \longrightarrow \cdots$$

☞ Da es keine Regel gibt, $\emptyset$ zu reduzieren, lässt sich mit dieser Reduktionsfolge kein Wort ableiten.

Die einzige mögliche Reduktionsfolge ist

$$\emptyset^* \longrightarrow \epsilon$$

☞ Somit ist die Semantik von $\emptyset^*$ die Sprache $\{\epsilon\}$. (Auch mit der denotationellen Semantik erhalten wir $[\![\emptyset^*]\!] = \{\epsilon\}$.)

30. Vertiefung

Eine Sprache kann in der Regel auf verschiedene Art und Weisen beschrieben werden.

Beispiel: Sprache aller Wörter mit einer geraden Anzahl von as und beliebig vielen bs.

$$(\mathtt{a}\ \mathtt{b}^*\ \mathtt{a} \mid \mathtt{b})^*$$

Alternative Formulierung:

$$(\mathtt{b} \mid \mathtt{a}\ \mathtt{b}^*\ \mathtt{a})^*$$

Alternative Formulierung ohne '|':

$$\mathtt{b}^*\ (\mathtt{a}\ \mathtt{b}^*\ \mathtt{a}\ \mathtt{b}^*)^*$$

☞ Allgemein gilt: $(r_1 \mid r_2)^* = r_1^*\ (r_2\ r_1^*)^*$.

30. Gleichheit regulärer Ausdrücke

☞ Wie zeigt man, dass zwei reguläre Ausdrücke r_1 und r_2 gleichwertig sind?

Mit Hilfe der Semantik:

- *Denotationelle Semantik:* $[\![r_1]\!] = [\![r_2]\!]$;
- also: Rückgriff auf die (naive) Mengenlehre;
- *Reduktionssemantik:* zu jeder Reduktionsfolge $r_1 \longrightarrow \cdots \longrightarrow w$ gibt es eine korrespondierende Reduktionsfolge $r_2 \longrightarrow \cdots \longrightarrow w$;
- also: Rückgriff auf Beweissysteme (Logik).

30. Vertiefung

Die Sprache aller Wörter, die eine gerade Anzahl von as (bs beliebig) *oder* eine ungerade Anzahl von bs (as beliebig) haben, lässt sich unter Rückgriff auf das vorherige Beispiel (Folie 573) leicht definieren.

$$(\mathtt{a}\ \mathtt{b}^*\ \mathtt{a} \mid \mathtt{b})^* \quad \mid \quad \mathtt{a}^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b}\ \mathtt{a}^*\ \mathtt{b})^*$$

☞ Aber wie beschreibt man die Sprache aller Wörter, die eine gerade Anzahl von as (bs beliebig) *und* eine ungerade Anzahl von bs (as beliebig) haben?

30. Vertiefung

Stände ein Pendant zum Mengendurchschnitt bereit, sagen wir &, könnten wir formulieren:

$$(a\ b^*\ a \mid b)^* \ \&\ a^*\ b\ (a \mid b\ a^*\ b)^*$$

☞ Es spricht nichts dagegen, den Durchschnitt zu der Sprache der regulären Ausdrücke hinzuzunehmen.
☞ Interessanterweise ist die Erweiterung zwar bequem, aber nicht notwendig: sie erhöht nicht die Ausdruckskraft.

Versuchen wir also die Sprache mit den bisherigen Bordmitteln zu definieren ...

30. Vertiefung

Nützliche Abkürzungen: $g = (\mathtt{aa})^*$ und $u = \mathtt{a}\,(\mathtt{aa})^*$.

1. Schritt: genau ein b und eine gerade Anzahl von as.

$$g\ \mathtt{b}\ g \mid u\ \mathtt{b}\ u$$

☞ Entweder kommen vor und hinter dem b eine gerade Anzahl von as vor oder an beiden Positionen eine ungerade Anzahl.

2. Schritt: wir erhöhen die Anzahl der bs 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 as über drei Positionen verteilt; entweder befinden sich an allen Stellen eine gerade Anzahl von as oder an exakt zwei Positionen eine ungerade Anzahl.

30. Vertiefung

3. Schritt: eine ungerade Zahl hat die Form $1 + 2n$, entsprechend definieren wir

$$(g\ \mathsf{b}\ g \mid u\ \mathsf{b}\ u)\ \ (g\ \mathsf{b}\ g\ \mathsf{b}\ g \mid g\ \mathsf{b}\ u\ \mathsf{b}\ u \mid u\ \mathsf{b}\ g\ \mathsf{b}\ u \mid u\ \mathsf{b}\ u\ \mathsf{b}\ g)^*$$

☞ Man sieht, ohne den Durchschnitt müssen wir bei der Formulierung von Sprachen sehr viel mehr Grips investieren.

30. Grenzen regulärer Ausdrücke

- Können wir mit regulären Ausdrücken alle Sprachen beschreiben?
- Nein, es gibt sehr viel mehr Sprachen als Ausdrücke, mit denen Sprachen beschrieben werden können (überabzählbar versus abzählbar).
- Können wir mit regulären Ausdrücken die Syntax von Programmiersprachen festlegen?
- Nein, einfache Hygienevorschriften lassen sich nicht fassen: etwa, dass es in Mini–F# Ausdrücken zu jeder „Klammer auf“ eine korrespondierende „Klammer zu“ geben muss. Die Sprache

$$\{\mathtt{a}^n\ \mathtt{b}^n \mid n \in \mathbb{N}\}$$

ist nicht regulär — lies $\mathtt{a}$ als (und $\mathtt{b}$ als) .

30. Lösung Knobelaufgabe #4

Endlich!!

Also: das sichselbstausgebende Programm muss mindestens einen String enthalten; ein String alleine reicht aber nicht, es muss noch etwas dazukommen. *Idee:* der String enthält den Programmcode des anderen Teils; damit das Programm sich selbst ausgibt, muss der String verdoppelt werden. Etwa so:

$$s \mathbin{\hat{}} \mathit{show}\ s$$

Warum einmal s und einmal $\mathit{show}\ s$?

s steht für den Programmcode und $\mathit{show}\ s$ für den Text des Programmcodes.

$$(\textbf{\textit{fun}}\ s \to s \mathbin{\hat{}} \mathit{show}\ s)\ \texttt{"(fun s -> s ^ show s)"}$$

30. Lösung Knobelaufgabe #4 — Demo

```
Mini⟩ "Hello, world!"
"Hello, world!"
Mini⟩ show "Hello, world!"
"\"Hello, world!\""
Mini⟩ length "Hello, world!"
13
Mini⟩ length (show "Hello, world!")
15
Mini⟩ (fun s → s ^ show s) "(fun s -> s ^ show s)"
"(fun s -> s ^ show s)\"(fun s -> s ^ show s)\""
Mini⟩ putline it
(fun s → s ^ show s) "(fun s -> s ^ show s)"
```

31. Knobelaufgabe #16

Einem Wort über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$ kann man ansehen, ob es eine gerade Anzahl von as enthält (bs beliebig).

Einer *endlichen* Sprache kann man ansehen, ob *alle* in der Sprache enthaltenen Wörter eine gerade Anzahl von as enthalten.

Kann man einem regulären Ausdruck ansehen, ob alle in der von dem regulären Ausdruck bezeichneten Sprache enthaltenen Wörter eine gerade Anzahl von as enthalten?

31. Akzeptoren

- Ein regulärer Ausdruck r beschreibt eine Sprache.
- Wie können wir feststellen, ob ein gegebenes Wort w in der Sprache enthalten ist: $w \in [\![r]\!]$?
- Können wir ein Mini–F# Programm schreiben, das uns diese Aufgabe abnimmt?
- Ja! Ein solches Programm nennt man *Akzeptor*.
- Im Folgenden führen wir die Konstruktion exemplarisch für den regulären Ausdruck

$$r_{00} = (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b})$$

durch. Der Ausdruck beschreibt die Sprache aller Wörter, die an der vorletzten Position ein $\mathtt{b}$ haben.

31. Akzeptoren — Schnittstelle

Wir gehen davon aus, dass das Alphabet durch eine Variantentypdefinition gegeben ist.

```
type Alphabet = | A | B

let ord-Alphabet (a : Alphabet) : Int =
  match a with A → 0 | B → 1
```

☞ Zu einem Alphabet gehört eine injektive Funktion, die sogenannte Codierungsfunktion, die jedem Zeichen eine ganze Zahl zuordnet, den sogenannten Zeichencode.

Ein Akzeptor ist dann eine Funktion des Typs

$$\textit{accept-}r_{00} : \textit{List}\ \langle \textit{Alphabet} \rangle \to \textit{Bool}$$

☞ *accept-*r_{00} testet, ob die Eingabe w ein Element der durch r_{00} bezeichneten Sprache ist: $w \in [\![r_{00}]\!]$.

31. Akzeptoren — Implementierung

Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept\text{-}r_{00}}\ (\mathit{input} : \mathit{List}\ \langle \mathit{Alphabet} \rangle) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ \mathit{input}\ \textbf{\textit{with}} \\
\quad\quad \mid [\,] \qquad\quad \to \ldots \\
\quad\quad \mid A :: \mathit{rest} \to \ldots\ \mathit{accept\text{-}r_{00}}\ \mathit{rest} \ldots \\
\quad\quad \mid B :: \mathit{rest} \to \ldots\ \mathit{accept\text{-}r_{00}}\ \mathit{rest} \ldots
\end{array}
$$

☞ Der rekursive Aufruf von $\mathit{accept\text{-}r_{00}}$ hilft uns an dieser Stelle *nicht* weiter: aus $w \in [\![r_{00}]\!]$ können wir im Allgemeinen keine Rückschlüsse auf $a\ w \in [\![r_{00}]\!]$ ziehen.

31. Implementierung — Rekursionsbasis

Im Fall $input = []$ müssen wir überlegen, ob das leere Wort in der Sprache enthalten ist: $\epsilon \in [\![r]\!]$. Können wir dem regulären Ausdruck das ansehen?

- $[\![a]\!]$ enthält ϵ nicht;
- $[\![\epsilon]\!]$ enthält ϵ;
- $[\![r_1\ r_2]\!]$ enthält ϵ, wenn $[\![r_1]\!]$ *und* $[\![r_2]\!]$ das leere Wort enthalten;
- $[\![\emptyset]\!]$ enthält ϵ nicht;
- $[\![r_1 \mid r_2]\!]$ enthält ϵ, wenn $[\![r_1]\!]$ *oder* $[\![r_2]\!]$ das leere Wort enthalten;
- $[\![r^*]\!]$ enthält ϵ.

31. Enthält eine Sprache das leere Wort?

Formalisierung:

$$
\begin{array}{ll}
\mathit{nullable}(a) & = \mathit{false} \\
\mathit{nullable}(\epsilon) & = \mathit{true} \\
\mathit{nullable}(r_1\ r_2) & = \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2) \\
\mathit{nullable}(\emptyset) & = \mathit{false} \\
\mathit{nullable}(r_1 \mid r_2) & = \mathit{nullable}(r_1) \vee \mathit{nullable}(r_2) \\
\mathit{nullable}(r^*) & = \mathit{true}
\end{array}
$$

☞ *nullable* ist eine mathematische Funktion, kein Mini–F# Programm.

☞ Wie auch $[\![-]\!]$ orientiert sich *nullable* an der Struktur regulärer Ausdrücke.

☞ Eine semantische Eigenschaft, $\epsilon \in [\![r]\!]$, wird anhand der Syntax geklärt, $\mathit{nullable}(r)$.

31. Enthält eine Sprache das leere Wort?

Für r_{00} erhalten wir:

$$\begin{aligned} & \mathit{nullable}((\mathtt{a} \mid \mathtt{b})^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b})) \\ = \quad & \mathit{nullable}((\mathtt{a} \mid \mathtt{b})^*) \wedge \mathit{nullable}(\mathtt{b}\ (\mathtt{a} \mid \mathtt{b})) \\ = \quad & \mathit{nullable}(\mathtt{b}\ (\mathtt{a} \mid \mathtt{b})) \\ = \quad & \mathit{nullable}(\mathtt{b}) \wedge \mathit{nullable}(\mathtt{a} \mid \mathtt{b}) \\ = \quad & \mathit{false} \end{aligned}$$

☞ ϵ ist *nicht* in $[\![r_{00}]\!]$ enthalten.

31. Implementierung — Rekursionsschritt

Somit können wir den ersten Zweig von $\mathit{accept\text{-}r}_{00}$ mit Leben füllen:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept\text{-}r}_{00}\ (\mathit{input} : \mathit{List}\ \langle \mathit{Alphabet} \rangle) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ \mathit{input}\ \textbf{\textit{with}} \\
\quad\ \mid [\,] \qquad\ \to \mathit{false} \\
\quad\ \mid A :: \mathit{rest} \to \ldots \mathit{accept\text{-}r}_{00}\ \mathit{rest} \ldots \\
\quad\ \mid B :: \mathit{rest} \to \ldots \mathit{accept\text{-}r}_{00}\ \mathit{rest} \ldots
\end{array}
$$

☞ Im Fall $\mathit{input} = A :: \mathit{rest}$ 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“.

31. Rechtsfaktoren

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.

☞ Wenn L eine reguläre Sprache ist, ist dann $L \,/\, w$ ebenfalls regulär? Zunächst einmal gilt

$$L \,/\, \epsilon \quad = L$$
$$L \,/\, a\, w = (L \,/\, a) \,/\, w$$

☞ Somit können wir uns auf die Definition von $L \,/\, a$ konzentrieren.

31. Rechtsfaktoren

Ziel: Definition von '/' auf regulären Ausdrücken, so dass

$$[\![r \,/\, x]\!] = [\![r]\!] \,/\, x$$

gilt. Eine semantische Operation auf Sprachen, $L \,/\, x$, wird auf reguläre Ausdrücke (Syntax), $r \,/\, x$, übertragen.

Schwierigster Fall: Konkatenation $r_1\ r_2 \,/\, x$.

- Entweder wir entfernen x aus r_1
- oder aus r_2; das geht aber nur, wenn $\epsilon \in [\![r_1]\!]$. Wir definieren:

$$\Delta(r) \quad = \quad \begin{cases} \epsilon & \text{falls } \mathit{nullable}(r) \\ \emptyset & \text{sonst} \end{cases}$$

31. Rechtsfaktoren

Damit erhalten wir

$$
\begin{array}{lcl}
a \,/\, x & = & \begin{cases} \epsilon & \text{falls } a = x \\ \emptyset & \text{sonst} \end{cases} \\
\epsilon \,/\, x & = & \emptyset \\
r_1\ r_2 \,/\, x & = & (r_1 \,/\, x)\ r_2 \mid \Delta(r_1)\ (r_2 \,/\, x) \\
\emptyset \,/\, x & = & \emptyset \\
r_1 \mid r_2 \,/\, x & = & (r_1 \,/\, x) \mid (r_2 \,/\, x) \\
r^* \,/\, x & = & (r \,/\, x)\ r^*
\end{array}
$$

☞ 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).

☞ Wir sehen: es war eine gute Idee, $\emptyset$ mit zur Sprache hinzuzunehmen.

31. Rechtsfaktoren

Wie sehen die Rechtsfaktoren für unser Beispiel aus?

$$
\begin{aligned}
& r_{00} \,/\, \mathtt{a} \\
= \;& (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{a} \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{a}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\Delta((\mathtt{a} \mid \mathtt{b})^*)) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{a}) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{a}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{a}) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{a}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid ((\mathtt{b} \,/\, \mathtt{a}) \; (\mathtt{a} \mid \mathtt{b})) \mid ((\Delta \mathtt{b}) \; ((\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{a})) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{a}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \\
= \;& (\mathtt{a} \mid \mathtt{b} \,/\, \mathtt{a}) \; (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \\
= \;& ((\mathtt{a} \,/\, \mathtt{a}) \mid (\mathtt{b} \,/\, \mathtt{a})) \; (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \\
= \;& (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \\
= \;& r_{00}
\end{aligned}
$$

☞ $r_{00} \,/\, \mathtt{a} = r_{00}$!

31. Rechtsfaktoren

Fortsetzung:

$$
\begin{aligned}
& r_{00} \,/\, \mathtt{b} \\
= \;& (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{b} \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{b}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\Delta((\mathtt{a} \mid \mathtt{b})^*)) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{b}) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{b}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{b}) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{b}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid ((\mathtt{b} \,/\, \mathtt{b}) \; (\mathtt{a} \mid \mathtt{b})) \mid ((\Delta \mathtt{b}) \; ((\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{b})) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{b}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\mathtt{a} \mid \mathtt{b}) \\
= \;& (\mathtt{a} \mid \mathtt{b} \,/\, \mathtt{b}) \; (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \\
= \;& ((\mathtt{a} \,/\, \mathtt{b}) \mid (\mathtt{b} \,/\, \mathtt{b})) \; (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \\
= \;& (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \\
= \;& r_{00} \mid (\mathtt{a} \mid \mathtt{b})
\end{aligned}
$$

☞ Wir taufen den resultierenden Ausdruck r_{10}.

31. Implementierung — Rekursionsschritt

Damit können wir endlich die Definition von $\mathit{accept}\text{-}r_{00}$ vervollständigen.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept}\text{-}r_{00}\ (\mathit{input} : \mathit{List}\ \langle \mathit{Alphabet} \rangle) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ \mathit{input}\ \textbf{\textit{with}} \\
\quad\ \mid [\,] \qquad\ \to \mathit{false} \\
\quad\ \mid A :: \mathit{rest} \to \mathit{accept}\text{-}r_{00}\ \mathit{rest} \\
\quad\ \mid B :: \mathit{rest} \to \mathit{accept}\text{-}r_{10}\ \mathit{rest}
\end{array}
$$

☞ Ist das erste Zeichen ein A, dann erfolgt ein rekursiver Aufruf;
ist das Zeichen ein B, dann wird die weitere Arbeit an $\mathit{accept}\text{-}r_{10}$ delegiert.

31. Implementierung

Es verbleibt einen Akzeptor für r_{10} zu schreiben. Diese Aufgabe gehen wir nach dem gleichen Schema wie für r_{00} an.

$r_{10} = r_{00} \mid (\mathtt{a} \mid \mathtt{b})$
$\mathit{nullable}(r_{10}) = \mathit{false}$
$r_{10} \,/\, \mathtt{a} = r_{00} \mid \epsilon \qquad = r_{01}$
$r_{10} \,/\, \mathtt{b} = r_{00} \mid (\mathtt{a} \mid \mathtt{b}) \mid \epsilon = r_{11}$

Wir erhalten zwei neue reguläre Ausdrücke und rechnen weiter ...

31. Implementierung

$r_{01} = r_{00} \mid \epsilon$

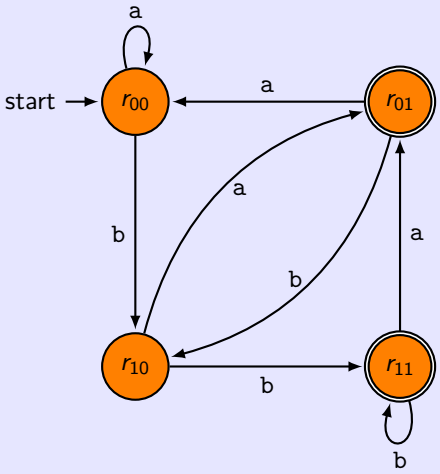
$nullable(r_{01}) = true$
$r_{01} \,/\, \mathtt{a} = r_{00}$
$r_{01} \,/\, \mathtt{b} = r_{10}$

$r_{11} = r_{00} \mid (\mathtt{a} \mid \mathtt{b}) \mid \epsilon$

$nullable(r_{11}) = true$
$r_{11} \,/\, \mathtt{a} = r_{01}$
$r_{11} \,/\, \mathtt{b} = r_{11}$

☞ Die Aufrufstruktur ist 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}.

31. Implementierung — Aufrufgraph



☞ Die vier Akzeptoren sind *verschränkt rekursiv*: die Definitionen müssen mit dem Schlüsselwort ***and*** verbunden werden.

31. Implementierung

```
let rec accept-r00 (input : List ⟨Alphabet⟩) : Bool =
  match input with | []        → false
                   | A :: rest → accept-r00 rest
                   | B :: rest → accept-r10 rest
and accept-r01 (input : List ⟨Alphabet⟩) : Bool =
  match input with | []        → true
                   | A :: rest → accept-r00 rest
                   | B :: rest → accept-r10 rest
and accept-r10 (input : List ⟨Alphabet⟩) : Bool =
  match input with | []        → false
                   | A :: rest → accept-r01 rest
                   | B :: rest → accept-r11 rest
and accept-r11 (input : List ⟨Alphabet⟩) : Bool =
  match input with | []        → true
                   | A :: rest → accept-r01 rest
                   | B :: rest → accept-r11 rest
```

r_{00}, r_{10}, r_{01}, r_{11}? Ich blick da nicht mehr durch. Es geht doch darum, zu erkennen, ob das vorletzte Zeichen ein b ist. Dann merke ich mir einfach immer die beiden letzten Zeichen und überprüfe am Ende, ob das vorletzte Zeichen gleich b ist.

Genau das machen r_{00}, r_{10}, r_{01}, r_{11}!

Kapier ich nicht.

Schau Dir die regulären Ausdrücke genau an:

$$
\begin{array}{lcl}
r_{00} & = & (\mathtt{a} \mid \mathtt{b})^* \, \mathtt{b} \, (\mathtt{a} \mid \mathtt{b}) \mid \emptyset \mid \emptyset \\
r_{10} & = & (\mathtt{a} \mid \mathtt{b})^* \, \mathtt{b} \, (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \mid \emptyset \\
r_{01} & = & (\mathtt{a} \mid \mathtt{b})^* \, \mathtt{b} \, (\mathtt{a} \mid \mathtt{b}) \mid \emptyset \mid \epsilon \\
r_{11} & = & (\mathtt{a} \mid \mathtt{b})^* \, \mathtt{b} \, (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \mid \epsilon
\end{array}
$$

(a ∣ b) besagt „das letzte Zeichen war ein b“ (danach muss *ein* beliebiges Zeichen kommen); ϵ besagt „das vorletzte Zeichen war ein b“ (danach muss das Eingabeende kommen).

Die Nummerierung r_{ij} spiegelt das übrigens wieder: $i = 1$ gdw. das letzte Zeichen ein b war; $j = 1$ gdw. das vorletzte Zeichen ein b war. *Zum Knobeln:* Wieviele Rechtsfaktoren hat $(\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b})^n$?

So langsam dämmert's. Aber meine Lösung finde ich trotzdem einfacher.

Für dieses spezielle Beispiel magst Du Recht haben. Der Punkt ist aber, dass die Konstruktion der Akzeptoren *mechanisch* — ohne Nachdenken — funktioniert. Sie klappt für beliebige reguläre Ausdrücke, eben nicht nur für dieses Beispiel. Die kompliziertere Herangehensweise zahlt sich durch die größere Allgemeinheit aus!

Ok, ok, ich gebe mich geschlagen. Aber einfacher ist sie trotzdem ...

31. Rechtsfaktoren — Eigenschaften

- Der reguläre Ausdruck $(\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b})$ hat vier verschiedene Rechtsfaktoren.
- Der Ausdruck $(\mathtt{a} \; \mathtt{b}^* \; \mathtt{a} \mid \mathtt{b})^*$ hat zwei (welche?).
- Allgemein: jede reguläre Sprache hat nur *endlich* viele Rechtsfaktoren.
- Diese Eigenschaft können wir ausnutzen, um zu zeigen, dass eine Sprache *nicht* durch einen regulären Ausdruck bezeichnet werden kann.
- Die Sprache $L = \{\mathtt{a}^n \; \mathtt{b}^n \mid n \in \mathbb{N}\}$ ist ein Beispiel für eine solche Sprache. Die Rechtsfaktoren für Wörter der Form $\mathtt{a}^k$ sind gegeben durch:

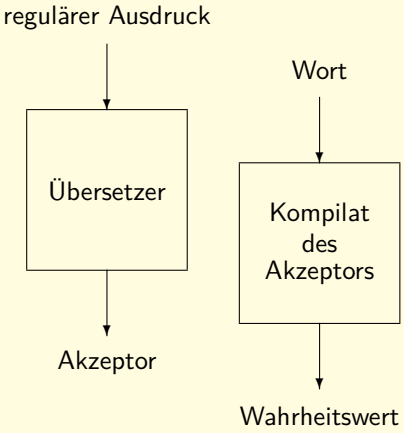
$$L \,/\, \mathtt{a}^k = \{\mathtt{a}^n \; \mathtt{b}^{n+k} \mid n \in \mathbb{N}\}$$

Alle diese Sprachen sind verschieden, also ist L keine reguläre Sprache.

31. Übersetzer und Interpreter

- Da die Anzahl aller Rechtsfaktoren endlich ist, kann man das obige Verfahren verwenden, um aus einem regulären Ausdruck einen Akzeptor zu generieren.
- *Mehr noch:* wir können das Verfahren auch automatisieren: 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 Programm, das ein anderes Programm erzeugt, nennt man *Übersetzer* (engl. compiler).
- In diesem Fall würde man auch von einem *Scanner-Generator* sprechen.
- (Ein bekannter Scanner-Generator ist `lex`).

31. Übersetzer und Interpreter



31. Übersetzer und Interpreter

- Der Übersetzer und der generierte Akzeptor sind zwei getrennte Programme.
- Der Übersetzer erzeugt ein Programm, das in einem zweiten Schritt ausgeführt wird.
- Die beiden Programme können alternativ auch enger miteinander verzahnt werden.
- Die Struktur der einzelnen Akzeptoren ist identisch — das ist nicht weiter verwunderlich, wir haben sie ja nach dem gleichen Schema konstruiert.
- Lassen sich die vier Funktionen zu einer zusammenfassen?
- *Idee:* Wir nummerieren die Funktionen durch und machen die Hausnummer zu einem zusätzlichen Parameter.

$$accept\ (k : Nat, input : List\ \langle Alphabet \rangle) : Bool$$

- Diese allgemeine Funktion müssen wir 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 wurde.

31. Übersetzer und Interpreter

Aus einer Kontrollstruktur wird eine Datenstruktur!

$$\begin{aligned}&\textbf{\textit{type}}\ \mathit{Control} = \{\,\mathit{nullable} : \mathit{Array}\,\langle \mathit{Bool}\rangle;\\ &\qquad\qquad\quad \mathit{next}\quad\ : \mathit{Array}\,\langle \mathit{Array}\,\langle \mathit{Nat}\rangle\rangle\,\}\end{aligned}$$

Für unser laufendes Beispiel erhalten wir die folgenden Daten:

$$\begin{aligned}&\textbf{\textit{let}}\ \mathit{control}\text{-}r_{00} = \{\,\mathit{nullable} = [|\ \mathit{false}; \mathit{true}; \mathit{false}; \mathit{true}\ |];\\ &\qquad\qquad\quad \mathit{next}\quad\ = [|\ [|\ 0; 0; 1; 1\ |];\qquad (*\ A\ *)\\ &\qquad\qquad\qquad\qquad\ \ [|\ 2; 2; 3; 3\ |]\ |]\}\qquad (*\ B\ *)\end{aligned}$$

☞ Die Funktionen sind fortlaufend von oben nach unten beginnend mit 0 durchnummeriert.

31. Übersetzer und Interpreter

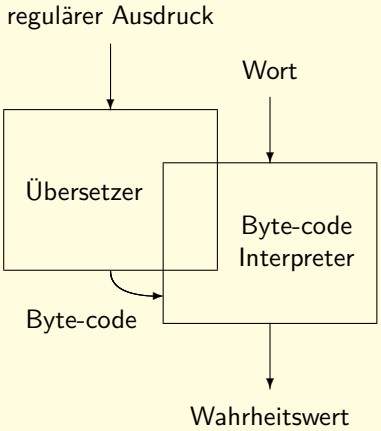
Ein generischer Akzeptor:

```
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)
let accept-r00 = generic-accept control-r00
```

31. Übersetzer und Interpreter

- 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*.
- Der allgemeine Akzeptor *generic-accept interpretiert* 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 regulären Ausdrücke durch kleine Zahlen repräsentiert werden und die Funktionen *nullable* und $r \,/\, x$ mit Hilfe von *done* 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.

31. Übersetzer und Interpreter



31. Übersetzer und Interpreter

- 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.
- Aus dem Byte-code Interpreter wird ein „echter“ Interpreter.
- Zu diesem Zweck müssen wir
  - die regulären Ausdrücke durch einen Datentyp modellieren und
  - die Funktionen *nullable* und $r \,/\, x$ in Mini–F# Programme überführen.

31. Interpreter

Wir transliterieren die abstrakte Syntax regulärer Ausdrücke in einen Variantentyp.

```
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 direkt in Mini–F# definiert werden.

```
let any = Alt (Sym A, Sym B)
let r00 = Cat (Rep any, Cat (Sym B, any))
```

31. Interpreter

Die Funktion *nullable* lässt sich direkt übertragen: die sechs Gleichungen werden zu den sechs Zweigen eines ***match***-Ausdrucks.

```
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
```

31. Interpreter

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)
```

31. Interpreter — Demo


```
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)))
```

☞ *divide* (r_{00}, A) ist semantisch äquivalent zu r_{00}, aber nicht syntaktisch gleich. Das liegt daran, dass bei der Konstruktion des Rechtsfaktors keine algebraischen Vereinfachungen wie etwa $r\ \epsilon = r$ vorgenommen werden.

31. Interpreter

Der generische Akzeptor:

```
let rec generic-accept (reg : Reg, input : List ⟨Alphabet⟩) : Bool =
  match input with
  | []        → nullable reg
  | a :: rest → generic-accept (divide (reg, a), rest)
```

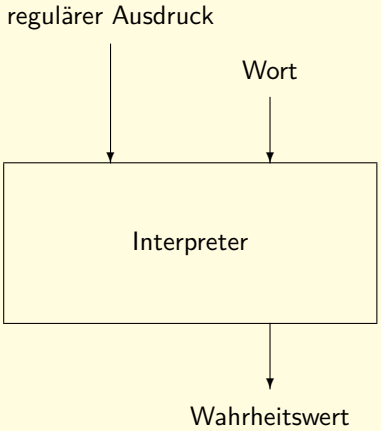
☞ An die Stelle der Hausnummern sind die regulären Ausdrücke selbst getreten.

```
let accept-r00 = fun input → generic-accept (r00, input)
```

31. Interpreter — Demo

```
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))
val even-no-of-as : Reg
Mini⟩ generic-accept (even-no-of-as, A :: B :: A :: [])
true
Mini⟩ generic-accept (even-no-of-as, A :: B :: B :: [])
false
```

31. Übersetzer und Interpreter



31. Übersetzer und Interpreter

Aus einem regulären Ausdruck lässt sich automatisch ein passender Akzeptor konstruieren. Je nach Verzahnungsgrad kann man mindestens drei Ansätze unterscheiden:

- reiner Übersetzer,
- Mischform aus Übersetzer und Interpreter,
- 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.

31. Lösung Knobelaufgabe #15

Ist es möglich, ein *nicht-rekursives* Programm zu schreiben, das *nicht terminiert*?

Ja! Wir zeigen, wie man die Fakultät ohne Rekursion programmieren kann. *Idee:* wir machen den rekursiven Aufruf zum Parameter und rufen die resultierende Funktion mit sich selbst auf (Selbstapplikation).

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

31. Lösung Knobelaufgabe #15

Was ist der Typ des formalen Parameters *self*?

```
let fac (self, n : Nat) : Nat =
  if n = 0 then 1
           else self (self, n − 1) ∗ n
```

☞ Sei $self : t$. Da *self* auf sich selbst angewendet wird, muss gelten: $t = t * Nat \to Nat$.

31. Lösung Knobelaufgabe #15

☞ Wir können keinen Typ t definieren, der *gleich* dem Typ $t * Nat \to Nat$ ist.

Aber wir können einen Typ definieren, der *isomorph* zu dem Typ $t * Nat \to Nat$ ist.

$$
\begin{array}{l}
\textbf{\textit{type}}\ Factorial = \{\, apply : Factorial * Nat \to Nat \,\} \\
\textbf{\textit{let}}\ fac\ (self : Factorial, n : Nat) : Nat = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\
\qquad\qquad \textbf{\textit{else}}\ \ self.apply\ (self, n - 1) * n \\
\textbf{\textit{let}}\ factorial\ (n : Nat) : Nat = \\
\quad fac\ (\{\, apply = fac \,\}, n)
\end{array}
$$

☞ $\textbf{\textit{fun}}\ f \to f.apply$ und $\textbf{\textit{fun}}\ g \to \{\, apply = g \,\}$ sind die Isomorphismen zwischen $Factorial$ und $Factorial * Nat \to Nat$.

32. Knobelaufgabe #17

Welche Sprache bezeichnet der folgende reguläre Ausdruck?

$$\mathit{riddle} = (\mathtt{a}\ \mathtt{a} \mid \mathtt{b}\ \mathtt{b})^* \ ((\mathtt{a}\ \mathtt{b} \mid \mathtt{b}\ \mathtt{a})\ (\mathtt{a}\ \mathtt{a} \mid \mathtt{b}\ \mathtt{b})^*\ (\mathtt{a}\ \mathtt{b} \mid \mathtt{b}\ \mathtt{a})\ (\mathtt{a}\ \mathtt{a} \mid \mathtt{b}\ \mathtt{b})^*)^*$$

Und welche Sprachen bezeichnen $\mathit{riddle} \,/\, \mathtt{a}$ und $\mathit{riddle} \,/\, \mathtt{b}$?

32. Motivation

- Reguläre Ausdrücke sind wenig ausdrucksstark — die jeweiligen Akzeptoren haben nur ein *endliches* Gedächtnis.
- Hygienevorschriften wie „zu jeder offenen Klammer muss es eine schließende Klammer geben“ lassen sich nicht formulieren.

$$\{\mathtt{a}^n\,\mathtt{b}^n \mid n \in \mathbb{N}\}$$

- Die Sprache der wohlgeformten Klammerausdrücke lässt sich aber schrittweise (im Fachjargon: induktiv) definieren:
  - ϵ ist wohlgeformt;
  - wenn w wohlgeformt ist, dann auch $\mathtt{a}\ w\ \mathtt{b}$.
- Wir benötigen *Rekursion*!
- Wir erweitern reguläre Ausdrücke um rekursiv definierte Sprachen:

$$\textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$$

Lies: wenn x ein Element der Sprache ist, dann auch ϵ und $\mathtt{a}\ x\ \mathtt{b}$.

32. Motivation

- Die Erweiterung bekommt einen neuen Namen: wir sprechen von *kontextfreien Ausdrücken* oder *kontextfreien Grammatiken*.
- Es gibt auch *kontextsensitive Sprachen*, mehr zu diesem Thema später aus der Abteilung der Theoretischen Informatik.
- Zum Begriff „kontextfrei“:

  rec *expr* → 0 | `false` | *expr* + *expr*

  Die Grammatik erlaubt einen Booleschen Ausdruck, etwa `false`, in einem Kontext zu verwenden, in dem ein arithmetischer Ausdruck erwartet wird: `false` + 0.
- Die Einschränkung auf wohlgetypte Ausdrücke können wir *nicht* mit der kontextfreien Syntax ausdrücken.
- Um diese Dinge kümmert sich bei uns die statische Semantik.
- Durch diese klare Trennung von Zuständigkeiten wird die Sprachdefinition von Mini–F# ungemein erleichtert.

32. Abstrakte Syntax

$x \in \mathsf{Id}$

$c \in \mathsf{CF} ::=$		*kontextfreie Ausdrücke:*
	a	Terminalsymbol
$\mid$	x	Bezeichner$\setminus$Nichtterminalsymbol
$\mid$	ϵ	das leere Wort
$\mid$	$c_1\ c_2$	Konkatenation$\setminus$Sequenz
$\mid$	$\emptyset$	die leere Sprache
$\mid$	$c_1 \mid c_2$	Alternative
$\mid$	$\mathbf{rec}\ x \to c$	Rekursion

☞ *Zusätzliche Konstrukte:* Bezeichner und Rekursion.

☞ Die Wiederholung ist kein primitives Konzept mehr; sie kann mit Hilfe der Rekursion ausgedrückt werden: c^* durch $\mathbf{rec}\ x \to \epsilon \mid c\ x$ oder $\mathbf{rec}\ x \to \epsilon \mid x\ c$.

32. Reduktionssemantik

Teaser: Was ist die Bedeutung von

- ***rec*** $x \rightarrow x$,
- ***rec*** $x \rightarrow \mathtt{a}\ x\ \mathtt{b}$,
- ***rec*** $x \rightarrow x^*$,
- ***rec*** $x \rightarrow x^*\ \mathtt{a}$,
- ***rec*** $x \rightarrow$ ***rec*** $y \rightarrow \epsilon \mid \mathtt{a} \mid x\ y$?

☞ Semantik ist insbesondere dazu da, die Bedeutung von Randfällen oder Extremfällen zu klären.

32. Reduktionssemantik — Rechenregel

Rechenregel:

$$\overline{(\textbf{\textit{rec}}\ x \to c) \longrightarrow c\{x \mapsto \textbf{\textit{rec}}\ x \to c\}}$$

☞ Ein rekursiver Ausdruck wird einmal „aufgefaltet“: der Bezeichner x wird im Rumpf c durch den gesamten Ausdruck ***rec*** $x \to c$ ersetzt.

32. Reduktionssemantik — Beispiel

Mit Hilfe der Rechenregel können wir zum Beispiel aabb aus dem Ausdruck $\mathit{rec}\ x \to \epsilon \mid \mathrm{a}\ x\ \mathrm{b}$ ableiten.

$$
\begin{aligned}
& \mathit{rec}\ x \to \epsilon \mid \mathrm{a}\ x\ \mathrm{b} \\
\longrightarrow\ & \epsilon \mid \mathrm{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathrm{a}\ x\ \mathrm{b})\ \mathrm{b} \\
\longrightarrow\ & \mathrm{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathrm{a}\ x\ \mathrm{b})\ \mathrm{b} \\
\longrightarrow\ & \mathrm{a}\ (\epsilon \mid \mathrm{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathrm{a}\ x\ \mathrm{b})\ \mathrm{b})\ \mathrm{b} \\
\longrightarrow\ & \mathrm{a}\ (\mathrm{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathrm{a}\ x\ \mathrm{b})\ \mathrm{b})\ \mathrm{b} \\
\longrightarrow\ & \mathrm{a}\ (\mathrm{a}\ (\epsilon \mid \mathrm{a}\ (\mathit{rec}\ x \to \epsilon \mid \mathrm{a}\ x\ \mathrm{b})\ \mathrm{b})\ \mathrm{b})\ \mathrm{b} \\
\longrightarrow\ & \mathrm{a}\ (\mathrm{a}\ \epsilon\ \mathrm{b})\ \mathrm{b} \\
\longrightarrow\ & \mathrm{a}\ \mathrm{a}\ \mathrm{b}\ \mathrm{b}
\end{aligned}
$$

32. Reduktionssemantik — Substitution

Die Ersetzung von Bezeichnern durch Ausdrücke nennt man *Substitution*.

$$c\sigma$$

☞ In c werden die Bezeichner aus $dom\ \sigma$ mit $\sigma \in \mathsf{Id} \to_{\mathrm{fin}} \mathrm{CF}$ durch die zugeordneten Ausdrücke ersetzt.

Beispiel:

$$\begin{aligned}&(g\ \mathtt{b}\ g \mid u\ \mathtt{b}\ u)\{g \mapsto (\mathtt{aa})^*, u \mapsto \mathtt{a}\ (\mathtt{aa})^*\}\\ &\quad = (\mathtt{aa})^*\ \mathtt{b}\ (\mathtt{aa})^* \mid \mathtt{a}\ (\mathtt{aa})^*\ \mathtt{b}\ \mathtt{a}\ (\mathtt{aa})^*\end{aligned}$$

32. Reduktionssemantik — Substitution

Substitution von Bezeichnern in Ausdrücken:

$$
\begin{array}{lcl}
a\sigma & = & a \\
x\sigma & = & \begin{cases} \sigma(x), & \textbf{if } x \in \mathit{dom}\,(\sigma) \\ x, & \textbf{otherwise} \end{cases} \\
\epsilon\sigma & = & \epsilon \\
(c_1\ c_2)\sigma & = & (c_1\sigma)\,(c_2\sigma) \\
\emptyset\sigma & = & \emptyset \\
(c_1 \mid c_2)\sigma & = & (c_1\sigma) \mid (c_2\sigma) \\
(\textbf{\textit{rec}}\ x \to c)\sigma & = & \textbf{\textit{rec}}\ x \to c(\sigma \setminus \{x\})
\end{array}
$$

☞ $\sigma \setminus \{x\}$ ist die *Einschränkung* von σ auf $\mathit{dom}\ \sigma \setminus \{x\}$. Wir müssen sicherstellen, dass gebundene Variablen, z.B x in ***rec*** $x \to c$, *nicht* ersetzt werden.

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck ***rec*** $x \to x$ aus?

$$\begin{aligned} & \textbf{\textit{rec}}\ x \to x \\ \longrightarrow\quad & \textbf{\textit{rec}}\ x \to x \\ & \vdots \end{aligned}$$

☞ Wir machen keinen Fortschritt!

☞ Wir können kein *Wort* ableiten; ***rec*** $x \to x$ bezeichnet die leere Sprache.

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck $\textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b}$ aus?

$$
\begin{aligned}
& \textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ \mathtt{a}\ (\textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b}\ \mathtt{b} \\
& \vdots
\end{aligned}
$$

☞ Der Ausdruck wird immer größer!

☞ Wir können wiederum kein Wort ableiten; auch $\textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b}$ bezeichnet die leere Sprache.

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck ***rec*** $x \to x^*$ aus?

$$\begin{aligned} & \textbf{\textit{rec}}\ x \to x^* \\ \longrightarrow\quad & (\textbf{\textit{rec}}\ x \to x^*)^* \\ \longrightarrow\quad & \epsilon \end{aligned}$$

☞ Ein anderes Wort lässt sich nicht ableiten, also steht ***rec*** $x \to x^*$ für die Sprache $\{\epsilon\}$.

32. Denotationelle Semantik

Bei der Motivation von ***rec*** $x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ haben wir überlegt, wie sich $\mathtt{a}^n\ \mathtt{b}^n$ *schrittweise* bilden lässt: aus ϵ wird ab, daraus wird 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.

$$F(L) = \{\epsilon\} \cup \{\mathtt{a}\} \cdot L \cdot \{\mathtt{b}\}$$

☞ Die mathematische Funktion F hat den Typ $\mathbb{P}(A^*) \to \mathbb{P}(A^*)$.

☞ F beschreibt einen Schritt des Bildungsprozesses.

32. Denotationelle Semantik

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 *nähern* uns der Semantik, da wir in endlich vielen Schritten niemals die unendliche Menge $\{\mathtt{a}^n\ \mathtt{b}^n \mid n \in \mathbb{N}\}$ konstruieren können.

32. Denotationelle Semantik — Fixpunkte

Die Bedeutung eines rekursiven Ausdrucks ergibt sich als Vereinigung aller endlichen Approximationen.

$$\bigcup\{F^n(\emptyset) \mid n \in \mathbb{N}\}$$

☞ $F^n(\emptyset)$ ist die n-fache Anwendung der Funktion F auf $\emptyset$: $F^0(X) = X$ und $F^{n+1}(X) = F(F^n(X))$.

Eigenschaften von $S = \bigcup\{F^n(\emptyset) \mid n \in \mathbb{N}\}$:

- Die Sprache S ist ein *Fixpunkt* von F: es gilt $F(S) = S$.
- Die Sprache ist die *kleinste* Menge mit dieser Eigenschaft; S ist der *kleinste Fixpunkt*.
- Kurz: Alles notwendige ist drin, mehr aber nicht.

32. Denotationelle Semantik — Bezeichner

☞ Bevor wir die semantischen Gleichungen formulieren, müssen wir uns noch um die Bedeutung von Bezeichnern kümmern.

Deren Bedeutung halten wir in einer *Umgebung* fest.

$$\varrho \in \mathsf{Id} \to_{\text{fin}} \mathbb{P}(A^*)$$

☞ Den Begriff Umgebung haben wir schon bei der Auswertung von Mini–F# Ausdrücken verwendet.

- Dort: Abbildung von Bezeichnern auf Werte.
- Hier: Abbildung von Bezeichnern auf Sprachen.

32. Denotationelle Semantik — semantische Gleichungen

Die Semantikfunktion bildet einen kontextfreien Ausdruck und eine Umgebung auf eine Sprache ab.

$$
\begin{array}{ll}
[\![a]\!]\varrho & = \{a\} \\
[\![x]\!]\varrho & = \varrho(x) \\
[\![\epsilon]\!]\varrho & = \{\epsilon\} \\
[\![c_1\ c_2]\!]\varrho & = [\![c_1]\!]\varrho \cdot [\![c_2]\!]\varrho \\
[\![\emptyset]\!]\varrho & = \emptyset \\
[\![c_1 \shortmid c_2]\!]\varrho & = [\![c_1]\!]\varrho \cup [\![c_2]\!]\varrho \\
[\![\textbf{\textit{rec}}\ x \to c]\!]\varrho & = \bigcup\{F^n(\emptyset) \mid n \in \mathbb{N}\} \ \text{ mit } \ F(X) = [\![c]\!](\varrho, \{x \mapsto X\})
\end{array}
$$

☞ ***rec*** $x \to c$ wird auf den kleinsten Fixpunkt der zugehörigen Funktion $F : \mathbb{P}(A^*) \to \mathbb{P}(A^*)$ abgebildet.

☞ Die Umgebung wird verwendet, um die Bedeutung von x zu klären; beim ***rec***-Konstrukt wird die Umgebung erweitert: ',' ist der bekannte und beliebte Kommaoperator.

32. Denotationelle Semantik — Teaser

Die dem Ausdruck ***rec*** $x \to x$ zugeordnete Funktion ist

$$F(L) = L$$

Fixpunktiteration:

$\emptyset$
$F(\emptyset) = \emptyset$

☞ Die Bedeutung von ***rec*** $x \to x$ ist die leere Sprache.

☞ Die Funktion F hat unendlich viele Fixpunkte — jede Sprache ist Fixpunkt dieser Funktion. Der kleinste Fixpunkt ist die leere Sprache.

32. Denotationelle Semantik — Teaser

Die dem Ausdruck ***rec*** $x \to \mathtt{a}\ x\ \mathtt{b}$ zugeordnete Funktion ist

$$G(L) = \{\mathtt{a}\} \cdot L \cdot \{\mathtt{b}\}$$

Fixpunktiteration:

$$\emptyset$$
$$G(\emptyset) = \{\mathtt{a}\} \cdot \emptyset \cdot \{\mathtt{b}\} = \emptyset$$

☞ Die Bedeutung von ***rec*** $x \to \mathtt{a}\ x\ \mathtt{b}$ ist ebenfalls die leere Sprache.

☞ *Zum Knobeln:* Wieviele Fixpunkte hat G?

32. Denotationelle Semantik — Teaser

Die dem Ausdruck ***rec*** $x \to x^*$ zugeordnete Funktion ist

$$H(L) = L^*$$

Fixpunktiteration:

$$\begin{aligned} &\emptyset \\ &H(\emptyset) &&= \emptyset^* &&= \{\epsilon\} \\ &H(H(\emptyset)) &&= \{\epsilon\}^* &&= \{\epsilon\} \end{aligned}$$

☞ Die Bedeutung von ***rec*** $x \to x^*$ ist $\{\epsilon\}$.

32. Vertiefung

Im Folgenden spezifizieren wir schrittweise die kontextfreie Syntax von Mini–F#, eingeschränkt auf die in Teil III eingeführten Konstrukte.

Das zugrundeliegende Alphabet ist die Menge aller Mini–F# Lexeme. Wir verwenden

- *num* als Bezeichner für die Sprache aller Numerale,
- *id* steht für kleine Bezeichner und
- *Id* entsprechend für große Bezeichner.

32. Vertiefung

Versuchen wir uns an der kontextfreien Syntax einfacher arithmetischer Ausdrücke.

rec $expr \to num \mid expr + expr \mid expr * expr$

☞ 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.

32. Vertiefung

Aus dem kontextfreien Ausdruck lässt sich das Wort $4711 + 815 * 2765$ ableiten — wir kürzen den Ausdruck mit E ab und führen der Übersichtlichkeit halber nicht alle Reduktionsschritte auf:

$$
\begin{aligned}
E &\longrightarrow E + E \\
&\longrightarrow num + E \\
&\longrightarrow 4711 + E \\
&\longrightarrow 4711 + E * E \\
&\longrightarrow 4711 + num * E \\
&\longrightarrow 4711 + 815 * E \\
&\longrightarrow 4711 + 815 * num \\
&\longrightarrow 4711 + 815 * 2765
\end{aligned}
$$

32. Vertiefung

Es gibt aber noch eine zweite mögliche Reduktionsfolge:

$$
\begin{aligned}
E &\longrightarrow E * E \\
&\longrightarrow E + E * E \\
&\longrightarrow \mathit{num} + E * E \\
&\longrightarrow 4711 + E * E \\
&\longrightarrow 4711 + \mathit{num} * E \\
&\longrightarrow 4711 + 815 * E \\
&\longrightarrow 4711 + 815 * \mathit{num} \\
&\longrightarrow 4711 + 815 * 2765
\end{aligned}
$$

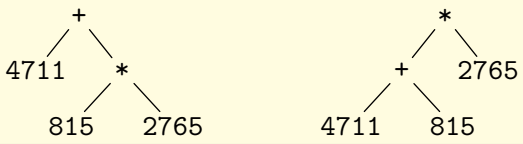
☞ *Problem*: der obige kontextfreie Ausdruck ist *mehrdeutig*; ein Wort kann auf verschiedene Weisen abgeleitet werden.

32. Vertiefung — mehrdeutige Grammatiken

☞ Warum ist das ein Problem?

Die kontextfreie Syntax dient einzig und allein dem Zweck, aus der linearen Folge von Lexemen die hierarchische Struktur eines Programms zu rekonstruieren.

Die beiden unterschiedlichen Reduktionsfolgen legen aber einen unterschiedlichen hierarchischen Aufbau nahe:



Die Semantik ordnet den beiden Syntaxbäumen eine unterschiedliche Bedeutung zu.

☞ Der obige kontextfreie Ausdruck ist ungeeignet zur Beschreibung arithmetischer Ausdrücke, da er nicht jedem Ausdruck einen eindeutigen abstrakten Syntaxbaum zuordnet.

32. Operatoren — Präfix- und Postfixnotation

Was ist zu tun?

Wir können die Syntax von Mini–F# Ausdrücken überdenken und arithmetische Operatoren nicht zwischen die Operatoren schreiben, sondern davor oder dahinter.

- *Präfixnotation* (wie in *Scheme*):

  rec *expr* → *num* | + *expr expr* | ∗ *expr expr*

- *Postfixnotation* (wie in *PostScript*):

  rec *expr* → *num* | *expr expr* + | *expr expr* ∗

☞ Beide Syntaxen sind *eindeutig*. (Warum?)

32. Operatoren — Infixnotation

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. *Infixnotation*:

```
4711 + 815 + 2765
```

- *Präfixnotation*: + 4711 + 815 2765 und + + 4711 815 2765.
- *Postfixnotation*: 4711 815 + 2765 + und 4711 815 2765 + +.

☞ Semantisch sind beide Varianten gleich, da die Addition *assoziativ* ist. Präfix- und Postfixnotation machen einen Unterschied, wo es keinen gibt. Allein die Infix-Notation stellt uns nicht vor die Wahl.

32. Operatoren — Assoziativität

Assoziative Operatoren und Funktionen:

- Disjunktion: `||`,
- Konjunktion: `&&`,
- Addition: $+$,
- Multiplikation: $*$,
- Konkatenation von Strings: `^`,
- Minimum: *min*,
- Maximum: *max*,
- Konkatenation von Listen: `@`,
- der Kommaoperator: ‘,’.

☞ Nicht alle assoziativen Funktionen notieren wir tatsächlich infix, bei allen würde es sich aber anbieten.

32. Operatoren — Assoziativität

☞ Wo Licht ist, ist auch Schatten.

Es hat sich eingebürgert, auch *nicht assoziative* Funktionen infix zu notieren:

- Subtraktion:

```
4711 - 815 - 2765
```

  ☞ `- 4711 - 815 2765` und `- - 4711 815 2765` sind unterschiedlich. Die Infixschreibweise klärt nicht, welche Variante gemeint ist.

- Potenzfunktion (*power* (x, n) wird zu x `**` n):

```
4711 ** 815 ** 2765
```

  ☞ Das gleiche Problem.

32. Operatoren — Links- und Rechtsassoziierung

Es bedarf einer Festlegung:

- Die Subtraktion wird allgemein als *linksassoziierend* festgelegt:

  $a - b - c$ steht für $- - a\ b\ c$

- Die Potenzfunktion wird allgemein als *rechtsassoziierend* festgelegt:

  $a \mathbin{**} b \mathbin{**} c$ steht für $\mathbin{**} a \mathbin{**} b\ c$

32. Operatoren — Klammern

Was machen wir, wenn wir die andere Variante benötigen, die wir infix nicht ausdrücken können?

In diesem Fall behilft man sich mit *Klammern*, die die Gruppierung explizit machen:

- Subtraktion:

```
4711 - (815 - 2765)
```

- Potenzfunktion:

```
(4711 ** 815) ** 2765
```

☞ Klammern sind ein Hilfsmittel der konkreten Syntax.

32. Operatoren — Klammern

Mit Hilfe von Klammern lassen sich die Begriffe links- und rechtsassoziierend noch einmal verdeutlichen:

- Ist der Operator '$\oplus$' linksassoziierend, dann steht

  $a \oplus b \oplus c$ für $(a \oplus b) \oplus c$

- ist der Operator rechtsassoziierend, dann steht

  $a \oplus b \oplus c$ für $a \oplus (b \oplus c)$

32. Operatoren — Assoziativität

Zwischenfazit: die Infixschreibweise ist *semantisch motiviert*:

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

Bei nicht assoziativen Funktionen bedarf es einer *syntaktischen Festlegung*:

- linksassoziierend: $(a \oplus b) \oplus c$,
- rechtsassoziierend: $a \oplus (b \oplus c)$.

☞ Eine solche Festlegung kann auch für assoziative Funktionen sinnvoll sein: $xs_1 \mathbin{@} (xs_2 \mathbin{@} xs_3)$ und $(xs_1 \mathbin{@} xs_2) \mathbin{@} xs_3$ sind zwar gleichwertig, aber der erste Ausdruck ist schneller ausgerechnet. (Warum?)

32. Operatoren — Bindungsstärke

Bietet eine Sprache mehrere Operatoren an, dann muss man klären, was passiert, wenn zwei Operatoren aufeinandertreffen.

```
4711 + 815 * 2765
```

Ist damit

- (4711 + 815) * 2765 oder
- 4711 + (815 * 2765) gemeint?

☞ Gängige Konvention — Punkt- vor Strichrechnung — gibt der zweiten Alternative den Vorzug.

32. Operatoren — Bindungsstärke

Hat man zwei beliebige Operatoren vor sich, $\oplus$ 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.

- Hat $\oplus$ die höhere Bindungsstärke, dann ist gemeint

$$(a \oplus b) \otimes c$$

- Hat $\otimes$ die höhere Bindungsstärke, dann entsprechend

$$a \oplus (b \otimes c)$$

☞ Die Bindungsstärke wird oft mit Hilfe natürlicher Zahlen spezifiziert, etwa '$+$' hat die Bindungsstärke 0 und '$*$' hat die Bindungsstärke 1.

32. Operatoren — eindeutige Grammatiken

Zurück zu unserer Aufgabe, der Aufstellung einer Syntax für einfache arithmetische Ausdrücke.

Wir können die Grammatik eindeutig machen, indem wir Assoziierung und Bindungsstärke der Operatoren in die Beschreibung „hineinprogrammieren“.

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*: die Bezeichner $expr_0$, $expr_1$ und $expr_2$ sind in allen rechten Seiten sichtbar.

32. Operatoren — eindeutige Grammatiken

$$
\begin{array}{lll}
\textbf{\textit{rec}} & expr_0 & \to expr_1 \mid expr_1 + expr_0 \\
\textbf{\textit{and}} & expr_1 & \to expr_2 \mid expr_2 * expr_1 \\
\textbf{\textit{and}} & expr_2 & \to num \mid (expr_0)
\end{array}
$$

Idee: E_i umfasst nur Ausdrücke, deren oberster Operator eine Bindungsstärke von i oder mehr hat.

- E_0 umfasst alle Ausdrücke,
- E_1 umfasst nur Produkte und
- E_2 umfasst nur atomare oder geklammerte Ausdrücke.

☞ Die Grammatik legt + und ∗ als rechtsassoziierend fest. (Warum?)

32. Operatoren — eindeutige Grammatiken

Die Grammatik ist *eindeutig*: jeder arithmetische Ausdruck lässt sich auf genau eine Art und Weise ableiten.

Zum Beispiel:

$$
\begin{aligned}
E_0 &\longrightarrow E_1 + E_0 \\
&\longrightarrow E_2 + E_0 \\
&\longrightarrow num + E_0 \\
&\longrightarrow 4711 + E_0 \\
&\longrightarrow 4711 + E_1 \\
&\longrightarrow 4711 + E_2 * E_1 \\
&\longrightarrow 4711 + num * E_1 \\
&\longrightarrow 4711 + 815 * E_1 \\
&\longrightarrow 4711 + 815 * E_2 \\
&\longrightarrow 4711 + 815 * num \\
&\longrightarrow 4711 + 815 * 2765
\end{aligned}
$$

32. Operatoren — eindeutige Grammatiken

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 $(0 \ldots)$ 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.

32. Syntax von Mini–F#

Die Erweiterung von kontextfreien Ausdrücken um verschränkte Rekursion ist auch notwendig für die Beschreibung von ***in***-Ausdrücken. Diese involvieren eine zweite syntaktische Kategorie: Definitionen.

rec *expr* $\rightarrow$ *id* | *num* | *expr* + *expr* | *expr* * *expr* | *decl* `in` *expr*
and *decl* $\rightarrow$ `let` *id* = *expr*

Ausdrücke und Definitionen sind verschränkt rekursiv: Ausdrücke beinhalten Definitionen und umgekehrt.

32. Syntax von Mini–F# — Mehrdeutigkeiten

Auch diese Grammatik ist mehrdeutig: der ***in***-Ausdruck

let $n = 4711$ ***in*** $n + n$

hat zwei mögliche Interpretationen:

- (***let*** $n = 4711$ ***in*** n) $+ n$
- ***let*** $n = 4711$ ***in*** $(n + n)$

☞ Der Unterschied ist groß, meint doch das zweite Vorkommen von n in beiden Ausdrücken etwas anderes!

32. Syntax von Mini–F# — *so weit nach rechts wie möglich*

Metaregel: wir vereinbaren, dass sich ein ***in***-Ausdruck *so weit nach rechts wie möglich erstreckt*.

let $n = 4711$ ***in*** $n + n$ steht für ***let*** $n = 4711$ ***in*** $(n + n)$

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 kann man das Ende des Sichtbarkeitsbereiches explizit markieren:
let $n = 4711$ ***in*** $n + n$ ***end***.

32. Syntax von Mini–F# — *so weit nach rechts wie möglich*

Die Vereinbarung „*so weit nach rechts wie möglich*“ wird in Mini–F# für zwei andere Konstrukte verwendet:

- Alternativen: ***if*** e_1 ***then*** e_2 ***else*** e_3.
- Funktionsausdrücke (anonyme Funktionen): ***fun*** $x \to e$.

☞ Der Ausdruck ***fun*** $x \to x + x$ ist mehrdeutig: (***fun*** $x \to x$) $+ x$ und ***fun*** $x \to (x + x)$ stehen als Interpretationen zur Wahl.

Der erste Ausdruck ist nicht typkorrekt, deswegen wird der zweiten Variante der Vorzug gegeben.

☞ Da die Metaregel „so weit nach rechts wie möglich“ *in der Regel* die sinnvolle Variante auswählt, wird das Ende von Alternativen und Funktionsausdrücken wie bei ***in***-Ausdrücken nicht explizit markiert.

32. Syntax von Mini–F# — Notation

Für die Syntaxbeschreibung von Mini–F# verwenden wir mehrere abkürzende Notationen:

- c^+ steht für eine mindestens einmalige Wiederholung von c: ***rec*** $x \to c \mid c\ x$;
- c_s^+ für eine mindestens einmalige Wiederholung, bei der die c Elemente durch s Elemente getrennt werden: ***rec*** $x \to c \mid c\ s\ x$;
- c_s^* für eine beliebige Wiederholung, bei der die c Elemente durch s Elemente getrennt werden: $\epsilon \mid c_s^+$.

32. Syntax von Mini–F# — Grammatik

// Ausdrücke			
rec *expr*	→	*aexpr*	atomarer Ausdruck
	|	`if` *expr* `then` *expr* `else` *expr*	Alternative
	|	`let` *decl** `in` *expr*	lokale Deklaration
	|	*expr* `||` *expr*	Disjunktion
	|	...	
	|	*expr* `+` *expr*	Addition
	|	...	
	|	`fun` *apat*$^+$ `->` *expr*	Funktionsabstraktion
	|	*aexpr aexpr*$^+$	Funktionsapplikation
	|	*expr* `:` *type*	Typangabe
and *aexpr*	→	*num*	Numeral
	|	*id*	Bezeichner
	|	`(` *expr* `)`	Gruppierung

☞ Atomare Ausdrücke müssen niemals geklammert werden.

32. Syntax von Mini–F# — Grammatik

// Deklarationen

and *decl*	$\rightarrow$	*pat* = *expr*	Wertedefinition
	|	*funcdecl*	Funktionsdefinition
	|	`rec` $\mathit{fundecl}^{+}_{\texttt{and}}$	rekursive Funktionsdefinitionen
and *fundecl*	$\rightarrow$	*id* apat^{+} : *type* = *expr*	Funktionsdefinition

// Muster

and *pat*	$\rightarrow$	*apat*	atomares Muster
	|	*pat* : *type*	Typangabe
and *apat*	$\rightarrow$	*id*	Bezeichner
	|	($\mathit{pat}^{*}_{,}$)	Tupelmuster oder Gruppierung

// Typausdrücke

and *type*	$\rightarrow$	*atype*	atomarer Typ
	|	*type* * *type*	Tupeltyp
	|	*type* -> *type*	Funktionstyp
and *atype*	$\rightarrow$	*Id*	Typbezeichner
	|	(*type*)	Gruppierung

32. Syntax von Mini–F# — Operatoren

Ausdrücke	
`;`	rechtsassoziierend
`:=`	rechtsassoziierend
`,`	nicht assoziierend
`||`	linksassoziierend
`&&`	linksassoziierend
`< =< = <> >= >`	linksassoziierend
`^`	rechtsassoziierend
`:: @`	rechtsassoziierend
`+ -`	linksassoziierend (infix)
`* / %`	linksassoziierend
`**`	rechtsassoziierend
Funktionsapplikation	linksassoziierend
`+ - !`	linksassoziierend (prefix)
`.`	linksassoziierend
Muster	
`|`	rechtsassoziierend
`&`	rechtsassoziierend
Typen	
`->`	rechtsassoziierend

33. Akzeptoren

- *Zur Erinnerung:* für reguläre Ausdrücke haben wir Akzeptoren generiert, indem wir systematisch alle Rechtsfaktoren berechnet haben.
- Das Verfahren lässt sich nicht auf kontextfreie Ausdrücke übertragen, da diese im Allgemeinen unendlich viele Rechtsfaktoren besitzen.
- *Beispiel:* die „Klammersprache“ $E = \textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ hat die Rechtsfaktoren

$$E \,/\, \mathtt{a} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b}) \,/\, \mathtt{a} = E\ \mathtt{b}$$
$$E \,/\, \mathtt{b} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b}) \,/\, \mathtt{b} = \emptyset$$

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. Und so weiter ...

33. Akzeptoren — Idee

- *Beobachtung:* E tritt im Rechtsfaktor wieder auf, allerdings gefolgt von unterschiedlichen Ausdrücken ($\mathtt{b}$, $\mathtt{b}^2$, ...).

$$E \,/\, \mathtt{a} \quad = (\epsilon \mid \mathtt{a}\; E\; \mathtt{b}) \,/\, \mathtt{a} \quad = E\; \mathtt{b}$$
$$E\; \mathtt{b} \,/\, \mathtt{a} = (\epsilon \mid \mathtt{a}\; E\; \mathtt{b})\; \mathtt{b} \,/\, \mathtt{a} = E\; \mathtt{b}^2$$

- *Idee:* der „Akzeptor“ für E kann durch eine rekursive Funktion implementiert werden, wenn wir ihn mit dem Akzeptor für den Folgeausdruck parametrisieren.

```
type Follow = List ⟨Alphabet⟩ → Bool
let accept-E (follow : Follow) : Follow
```

☞ Der Parameter *follow* legt fest, was *nach E* erwartet wird.
☞ Die Typdefinition führt ein sogenanntes *Typsynonym* ein, eine Abkürzung für den Typ auf der rechten Seite.

33. Akzeptoren — Klammersprache

Mit diesem Ansatz sieht der Akzeptor für den kontextfreien Ausdruck `b` — ein einzelnes Terminalsymbol — wie folgt aus.

```
let accept-b (follow : Follow) : Follow = fun input →
  match input with
  | B :: rest → follow rest
  | _         → false
```

☞ Fängt die Eingabe mit einem `b` an, wird die Überprüfung der restlichen Eingabe an *follow* delegiert.

☞ Anderenfalls ist die Eingabe nicht in der Sprache enthalten — für *keinen* Folgeausdruck.

33. Akzeptoren — Klammersprache

Der Akzeptor für die Klammersprache:

```
let rec accept-E (follow : Follow) : Follow = fun input →
  match input with
  | A :: rest → accept-E (accept-b follow) rest
  | _         → follow input
```

☞ Für jedes gelesene `a` wird der Folgeakzeptor um *accept-b* erweitert; nach dem ersten `b` wird der akkumulierte Zopf von *accept-b*s abgearbeitet.

☞ Wir können zählen, ohne die natürlichen Zahlen bemühen zu müssen `;-)`.

33. Akzeptoren — Klammersprache

Den gewünschten Akzeptor für E erhalten wir, indem wir *accept-E* mit dem Akzeptor für ϵ aufrufen: *accept-E end-of-input* wobei *end-of-input* wie folgt definiert ist.

```
let end-of-input = fun input →
  match input with
  | []      → true
  | _ :: _ → false
```

33. Akzeptoren

Zum Vergleich:

- Akzeptor für einen regulären Ausdruck:

  $$\mathit{accept\text{-}r}_{00} : \mathit{List}\ \langle \mathit{Alphabet} \rangle \to \mathit{Bool}$$

  Ein Akzeptor für einen „isolierten“ regulären Ausdruck.

- „Akzeptor“ für einen kontextfreien Ausdruck:

  $$\textbf{\textit{type}}\ \mathit{Follow} = \mathit{List}\ \langle \mathit{Alphabet} \rangle \to \mathit{Bool}$$
  $$\mathit{accept\text{-}E} : \mathit{Follow} \to \mathit{Follow}$$

  Ein Akzeptor für einen kontextfreien Ausdruck gefolgt von einem beliebigen kontextfreien Ausdruck.

☞ Ein weiteres Beispiel für die Programmiertechnik der *Verallgemeinerung*.

33. Akzeptoren mit System

Mit Hilfe dieses Ansatzes lässt sich zu jedem kontextfreien Ausdruck systematisch ein korrespondierender Mini–F# Ausdruck, ein Akzeptor des Typs *Follow* → *Follow*, konstruieren.

Terminalsymbol a:

```
fun follow →
  fun input →
    match input with
      | A :: rest → follow rest
      | _         → false
```

☞ *A* ist das zu *a* korrespondierende Element des Alphabets.

33. Akzeptoren mit System

Bezeichner x:

accept-x

Das *leere Wort* ϵ:

fun *follow* $\rightarrow$
  fun *input* $\rightarrow$
    follow input

☞ Wir nutzen $\epsilon\ c = c$ aus.
Kürzer: ***fun*** *follow* $\rightarrow$ *follow*. Wie heißt diese Funktion?

33. Akzeptoren mit System

Die *Sequenz* $c_1\ c_2$:
ist $accept_i$ die Implementierung von c_i, dann wird die Sequenz $c_1\ c_2$ wie folgt implementiert.

```
fun follow →
  fun input →
    accept1 (accept2 follow) input
```

☞ Wir nutzen $(c_1\ c_2)\ c = c_1\ (c_2\ c)$ aus.
Kürzer: ***fun*** *follow* $\rightarrow$ $accept_1\ (accept_2\ follow)$. Wie heißt diese Funktion?

33. Akzeptoren mit System

Die *leere Sprache* $\emptyset$:

```
fun follow →
  fun input →
    false
```

☞ Wir nutzen $\emptyset\ c = \emptyset$ aus.

Die *Alternative* $c_1 \mid c_2$:
ist $accept_i$ die Implementierung von c_i, dann wird die Alternative $c_1 \mid c_2$ wie folgt implementiert.

```
fun follow →
  fun input →
    accept1 follow input || accept2 follow input
```

☞ Wir nutzen das Distributivgesetz $(c_1 \mid c_2)\ c = c_1\ c \mid c_2\ c$ aus.

33. Akzeptoren mit System

Der *rekursive* Ausdruck ***rec*** $x \rightarrow c$:
ist *accept* die Implementierung von c, dann wird der rekursive Ausdruck ***rec*** $x \rightarrow c$ wie folgt implementiert.

```
let rec accept-x (follow) =
  fun input →
    accept follow input
in accept-x
```

☞ Tritt x in c auf, so kommt entsprechend *accept-x* in *accept* vor.
Kürzer: ***let rec*** *accept-x* $(follow) =$ *accept follow* ***in*** *accept-x*.

33. Akzeptoren mit System

Der *verschränkt rekursive* Ausdruck ***rec*** $x_1 \to c_1$ ***and*** $\cdots$ ***and*** $x_n \to c_n$ wird entsprechend auf einen verschränkt rekursiven Mini–F# Ausdruck abgebildet.

let rec *accept-*x_1 $(follow) = accept_1\ follow$
and $\ldots$
and *accept-*x_n $(follow) = accept_n\ follow$
in *accept-*x_1

33. Akzeptoren mit System — Beispiel

Setzen wir die Bausteine entsprechend zusammen, ergibt sich für die Klammersprache

$$\textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$$

der folgende kompakte Mini–F# Ausdruck.

```
let rec accept-x (follow : Follow) : Follow =
    fun input ->
        follow input || accept-a (accept-x (accept-b follow)) input
in accept-x
```

☞ Die Struktur des Mini–F# Ausdrucks spiegelt die Struktur des kontextfreien Ausdrucks wider.

33. Akzeptoren mit noch mehr System

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 so eine Bibliothek für die Konstruktion von Akzeptoren erstellen.

```
type Follow = List ⟨Alphabet⟩ → Bool

type Acceptor = Follow → Follow

let symbol (a : Alphabet) : Acceptor =
  fun (follow : Follow) →
    fun input →
      match input with
      | []       → false
      | b :: rest → a = b && follow rest
```

33. Akzeptoren mit noch mehr System

```
let eps : Acceptor =
  fun (follow : Follow) → follow

let seq (accept1 : Acceptor, accept2 : Acceptor) : Acceptor =
  fun (follow : Follow) → accept1 (accept2 follow)

let empty : Acceptor =
  fun (follow : Follow) →
    fun input → false

let alt (accept1 : Acceptor, accept2 : Acceptor) : Acceptor =
  fun (follow : Follow) →
    fun input → accept1 follow input || accept2 follow input
```

33. Akzeptoren mit noch mehr System — Beispiel

Mit Hilfe dieser Bibliothek kann der Akzeptor für die Klammersprache

$$\textbf{\textit{rec}}\ x \rightarrow \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$$

noch etwas kompakter definiert werden.

```
let rec accept-x (follow : Follow) : Follow =
    alt (eps, seq (seq (symbol A, accept-x), symbol B)) follow
in accept-x
```

☞ Die Struktur des Mini–F# Ausdrucks spiegelt exakt die Struktur des kontextfreien Ausdrucks wider.

33. Beispiel — einfache arithmetische Ausdrücke

Probieren wir die Technik an einem weiteren Beispiel aus, den einfachen arithmetischen Ausdrücken.

rec $expr_0 \rightarrow expr_1 \mid expr_1 + expr_0$
and $expr_1 \rightarrow expr_2 \mid expr_2 * expr_1$
and $expr_2 \rightarrow num \mid (expr_0)$

Das diesem Ausdruck zugrundeliegende Alphabet ist *Token*:

type *Token* = | *Num* ***of*** *Nat* | *LParen* | *RParen* | *Asterisk* | *Plus*

☞ Die offene Klammer (wird durch den Konstruktor *LParen* repräsentiert; das Numeral 4711 durch den Wert *Num* 4711.

33. Beispiel — einfache arithmetische Ausdrücke

Die Umsetzung von

$$
\begin{array}{ll}
\textbf{\textit{rec}} & \mathit{expr}_0 \to \mathit{expr}_1 \mid \mathit{expr}_1 \texttt{ + } \mathit{expr}_0 \\
\textbf{\textit{and}} & \mathit{expr}_1 \to \mathit{expr}_2 \mid \mathit{expr}_2 \texttt{ * } \mathit{expr}_1 \\
\textbf{\textit{and}} & \mathit{expr}_2 \to \mathit{num} \mid \texttt{(} \mathit{expr}_0 \texttt{)}
\end{array}
$$

geht mechanisch vonstatten:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept\text{-}expr}_0\ (\mathit{follow} : \mathit{Follow}) : \mathit{Follow} = \\
\quad \mathit{alt}\ (\mathit{accept\text{-}expr}_1, \\
\qquad \mathit{seq}\ (\mathit{seq}\ (\mathit{accept\text{-}expr}_1, \mathit{symbol}\ \mathit{Plus}), \mathit{accept\text{-}expr}_0)) \qquad \mathit{follow} \\
\textbf{\textit{and}}\ \ \mathit{accept\text{-}expr}_1\ (\mathit{follow} : \mathit{Follow}) : \mathit{Follow} = \\
\quad \mathit{alt}\ (\mathit{accept\text{-}expr}_2, \\
\qquad \mathit{seq}\ (\mathit{seq}\ (\mathit{accept\text{-}expr}_2, \mathit{symbol}\ \mathit{Asterisk}), \mathit{accept\text{-}expr}_1)) \quad \mathit{follow} \\
\textbf{\textit{and}}\ \ \mathit{accept\text{-}expr}_2\ (\mathit{follow} : \mathit{Follow}) : \mathit{Follow} = \\
\quad \mathit{alt}\ (\mathit{accept\_num}, \\
\qquad \mathit{seq}\ (\mathit{seq}\ (\mathit{symbol}\ \mathit{LParen}, \mathit{accept\text{-}expr}_0), \mathit{symbol}\ \mathit{RParen}))\ \mathit{follow}
\end{array}
$$

☞ Die Struktur des Mini–F# Ausdrucks (***let*** ... ***in*** $\mathit{accept\text{-}expr}_0$) spiegelt exakt die Struktur des kontextfreien Ausdrucks wider.

33. Beispiel — einfache arithmetische Ausdrücke

Zum Vergleich: der gleiche Akzeptor ohne Verwendung der Bausteine.

```
let rec accept-expr0 (follow : Follow) : Follow =
    fun input →
        accept-expr1 follow input
      || accept-expr1 (symbol Plus (accept-expr0 follow)) input
  and accept-expr1 (follow : Follow) : Follow =
    fun input →
        accept-expr2 follow input
      || accept-expr2 (symbol Asterisk (accept-expr1 follow)) input
  and accept-expr2 (follow : Follow) : Follow =
    fun input →
        accept_num follow input
      || symbol LParen (accept-expr0 (symbol RParen follow)) input
```

33. Linksrekursion

Jeder kontextfreie Ausdruck lässt sich systematisch in einen Mini–F# Ausdruck überführen. Ist damit der Fall abgeschlossen?
Nein! Die Umsetzung rekursiver Sprachen ist *nicht perfekt*: der kontextfreie Ausdruck ***rec*** $x \rightarrow x$ wird auf den Mini–F# Ausdruck

```
let rec accept-x (follow) →
  fun input →
    accept-x follow input
in accept-x
```

abgebildet, eine *nichtterminierende* Funktion.
Die Bedeutung von ***rec*** $x \rightarrow x$ ist aber die leere Sprache. Deren korrekte Implementierung lautet

```
fun follow →
  fun input →
    false
```

eine stets *terminierende* Funktion.

33. Linksrekursion

☞ Das Problem der Nichtterminierung tritt immer dann auf, wenn der kontextfreie Ausdruck *linksrekursiv* ist. Dann erfolgt der rekursive Aufruf, ohne dass die Liste von Tokens verkleinert wurde.

Beispiel: Klammergebirge.

rec $x \rightarrow \epsilon \mid x\ \mathtt{a}\ x\ \mathtt{b}$

Der zugehörige Akzeptor terminiert für die Eingabe `abab` nicht.

Die äquivalente, rechtsrekursive Formulierung bereitet keine Probleme.

rec $x \rightarrow \epsilon \mid \mathtt{a}\ x\ \mathtt{b}\ x$

Beim rekursiven Aufruf des Akzeptors ist sichergestellt, dass die Eingabe verkleinert wurde: der zu a korrespondierende Akzeptor hat vorher einen Buchstaben konsumiert.

33. Linksrekursion

Fazit:

- Bei handgeschriebenen Akzeptoren muss man Sorge tragen, dass die rekursiven Aufrufe auf kleineren Eingaben arbeiten.
- Linksrekursion ist *um jeden Preis* zu vermeiden.
- *Zum Vergleich:* Bei *regulären Ausdrücken* wird durch das Konstruktionsverfahren sichergestellt, dass die Eingabe stets kleiner wird.

33. Parser

- Ein Akzeptor beantwortet 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
  - der Wert des Ausdrucks oder
  - der abstrakte Syntaxbaum des Ausdrucks sein.
  - Der letztere Ansatz ist allgemeiner.

33. Abstrakte Syntax arithmetischer Ausdrücke

Die abstrakte Syntax arithmetischer Ausdrücke kann mit einer rekursiven Variantentypdefinition eingefangen werden.

```
type Expr =
    | Const of Nat
    | Add   of Expr * Expr
    | Mul   of Expr * Expr
```

☞ Rekursive Variantentypen sind das Mini–F# Pendant zu Baumsprachen, so dass wir die Baumsprache für arithmetische Ausdrücke im Wesentlichen übernehmen können.

33. Ein Auswerter für arithmetische Ausdrücke

Aufgabe: evaluate soll einen arithmetischen Ausdruck auswerten.

Mit dem Struktur Entwurfsmuster für *Expr* erhalten wir:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{evaluate}\ (\mathit{expr} : \mathit{Expr}) : \mathit{Nat} = \\
\quad \textbf{\textit{match}}\ \mathit{expr}\ \textbf{\textit{with}} \\
\quad\quad \mid \mathit{Const}\ \mathit{nat} \rightarrow \ldots \\
\quad\quad \mid \mathit{Add}\ (\mathit{expr}_1, \mathit{expr}_2) \rightarrow \ldots\ \mathit{evaluate}\ \mathit{expr}_1 \ldots\ \mathit{evaluate}\ \mathit{expr}_2 \ldots \\
\quad\quad \mid \mathit{Mul}\ (\mathit{expr}_1, \mathit{expr}_2) \rightarrow \ldots\ \mathit{evaluate}\ \mathit{expr}_1 \ldots\ \mathit{evaluate}\ \mathit{expr}_2 \ldots
\end{array}
$$

33. Ein Auswerter für arithmetische Ausdrücke

- *Rekursionsbasis*: Fall $expr = Const\ nat$.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ evaluate\ (expr : Expr) : Nat = \\
\quad \textbf{\textit{match}}\ expr\ \textbf{\textit{with}} \\
\quad\quad \mid Const\ nat \rightarrow nat \\
\quad\quad \mid Add\ (expr_1, expr_2) \rightarrow \ldots evaluate\ expr_1 \ldots evaluate\ expr_2 \ldots \\
\quad\quad \mid Mul\ (expr_1, expr_2) \rightarrow \ldots evaluate\ expr_1 \ldots evaluate\ expr_2 \ldots
\end{array}
$$

- *Rekursionsschritt*: Fall $expr = Add\ (expr_1, expr_2)$.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ evaluate\ (expr : Expr) : Nat = \\
\quad \textbf{\textit{match}}\ expr\ \textbf{\textit{with}} \\
\quad\quad \mid Const\ nat \rightarrow nat \\
\quad\quad \mid Add\ (expr_1, expr_2) \rightarrow evaluate\ expr_1 + evaluate\ expr_2 \\
\quad\quad \mid Mul\ (expr_1, expr_2) \rightarrow \ldots evaluate\ expr_1 \ldots evaluate\ expr_2 \ldots
\end{array}
$$

33. Ein Auswerter für arithmetische Ausdrücke

- *Rekursionsschritt*: Fall $expr = Mul\ (expr_1, expr_2)$.

$$
\begin{array}{l}
\textbf{let rec}\ evaluate\ (expr : Expr) : Nat = \\
\quad \textbf{match}\ expr\ \textbf{with} \\
\quad\quad \mid Const\ nat \rightarrow nat \\
\quad\quad \mid Add\ (expr_1, expr_2) \rightarrow evaluate\ expr_1 + evaluate\ expr_2 \\
\quad\quad \mid Mul\ (expr_1, expr_2) \rightarrow evaluate\ expr_1 * evaluate\ expr_2
\end{array}
$$

☞ Der Auswerter ist ein waschechter Interpreter.

Der Auswerter internalisiert die Auswertungsregeln $e \Downarrow \nu$.

33. Parser — semantische Werte

Ein *Folgeakzeptor* liefert einen Booleschen Wert als Ergebnis:

$$\textbf{\textit{type}}\ \mathit{Follow} = \mathit{List}\ \langle \mathit{Token} \rangle \to \mathit{Bool}$$

Ein *Folgeparser* muss im Erfolgsfall einen semantischen Wert zurückgeben. Aus dem Typ *Bool* wird der Typ *Option*:

$$\textbf{\textit{type}}\ \mathit{Follow}\ \langle' v\rangle = \mathit{List}\ \langle \mathit{Token} \rangle \to \mathit{Option}\ \langle' v\rangle$$

☞ Der Typparameter $'v$ spezifiziert den Typ des resultierendes Wertes, für unser laufendes Beispiel ist der Typ zum Beispiel *Nat* (Semantik) oder *Expr* (abstrakter Syntaxbaum).

33. Parser — semantische Werte

Welche Änderungen ergeben sich für die Programme?

(„Kennen Sie den Weg zum Audimax?“ Statt schroff mit „Ja!“ oder „Nein!“ zu antworten, sagt man im positiven Fall „Ja! Betreten Sie das Gebäude durch den Haupteingang, dann halten Sie sich links …“)

Allgemein: wenn bei einer Programmtransformation aus dem Typ *Bool* der Typ *Option* $\langle t \rangle$ wird, dann ändert sich

- *false* zu *None*,
- *true* zu *Some* e mit $e : t$, und
- e_1 || e_2 bzw. ***if*** e_1 ***then*** *true* ***else*** e_2 wird zu

match e_1 ***with***
| *Some* $a \to$ *Some* a
| *None* $\to e_2$

33. Parser — semantische Werte

Damit erhalten wir für *end-of-input*:

```
let end-of-input : Follow ⟨Expr⟩ =
  fun input →
    match input with
    | []     → Some ?
    | _ :: _ → None
```

☞ Woher nehmen wir das Argument für *Some*?
Wir müssen *end-of-input* den Wert mit auf den Weg geben.

```
let end-of-input (e : Expr) : Follow ⟨Expr⟩ =
  fun input →
    match input with
    | []     → Some e
    | _ :: _ → None
```

☞ Aus einer Funktion des Typs *Follow* ist eine Funktion des Typs $Expr \to Follow\ \langle Expr \rangle$ geworden.

33. Parser — semantische Werte

Idee: Jeder Parser übergibt seinen semantischen Wert an den Folgeparser. Die Funktion *end-of-input* ist der initiale Folgeparser.

Ein *Akzeptor* hat den Typ

$$\textbf{\textit{type}}\ \mathit{Acceptor} = \mathit{Follow} \to \mathit{Follow}$$

Ein *Parser* hat den Typ

$$\textbf{\textit{type}}\ \mathit{Parser}\ \langle'a\rangle = (\text{'}a \to \mathit{Follow}\ \langle \mathit{Expr}\rangle) \to \mathit{Follow}\ \langle \mathit{Expr}\rangle$$

☞ *Expr* ist der Typ des semantischen Wertes, den der Folgeparser als *endgültiges* Ergebnis zurückgibt.

33. Parser — semantische Werte

Welche Änderungen ergeben sich für die Programme?

- Aus

  accept-expr$_0$

  wird zum Beispiel

  parse-expr$_0$ (***fun*** $e \rightarrow \ldots$)

  ☞ *parse-expr*$_0$ selbst übergibt der bereitgestellten Funktion den semantischen Wert.
- Aus

  follow

  wird zum Beispiel

  follow (*Add* (e_1, e_2))

  ☞ Dem Folgeparser wird der semantische Wert übergeben.

33. Parser — einfache arithmetische Ausdrücke

Änderungen an dem Parser für einfache arithmetische Ausdrücke:

```
let rec parse-expr0 (follow : Expr → Follow ⟨Expr⟩) : Follow ⟨Expr⟩ =
  fun input →
    match parse-expr1 (fun e → follow e) input with
    | None →
      parse-expr1       (fun e1 →
        symbol Plus     (
          parse-expr0 (fun e2 →
            follow (Add (e1, e2))))) input
    | Some e → Some e
```

☞ *symbol* muss ebenfalls angepasst werden — zur Übung.

33. Parser — einfache arithmetische Ausdrücke

Fortsetzung:

```
and parse-expr1 (follow : Expr → Follow ⟨Expr⟩) : Follow ⟨Expr⟩ =
  fun input →
    match parse-expr2 (fun e → follow e) input with
    | None →
      parse-expr2          (fun e1 →
        symbol Asterisk (
          parse-expr1      (fun e2 →
            follow (Mul (e1, e2))))) input
    | Some e → Some e
```

33. Parser — einfache arithmetische Ausdrücke

Fortsetzung:

```
and parse-expr2 (follow : Expr → Follow ⟨Expr⟩) : Follow ⟨Expr⟩ =
  fun input →
    match parse-num (fun n → follow (Const n)) input with
    | None →
      symbol LParen        (
        parse-expr0        (fun e →
          symbol RParen (
            follow e))) input
    | Some e → Some e
```

☞ Insgesamt ergibt sich das Bild einer Kollekte: es wird ein Korb herumgereicht; jeder fügt ein paar Münzen hinzu und reicht den Korb an den Nachbarn (den Folgeparser) weiter. Am Ende der Kette wird der Korb geleert (*end-of-input*).

33. Parser — einfache arithmetische Ausdrücke

Jetzt wird es Zeit, den Parser in Aktion zu erleben.

Zu diesem Zweck kombinieren wir den Scanner *lex* mit dem Parser $parse\text{-}expr_0$.

```
let abstract-syntax-tree (input : String) : Option ⟨Expr⟩ =
  parse-expr₀ end-of-input (lex (explode input))
function abstract-syntax-tree (input : String) : Option ⟨Expr⟩ =
  parse-expr₀ end-of-input (lex (explode input))
```

☞ *explode* überführt einen String in eine Liste von Zeichen; *lex* überführt die Liste von Zeichen in eine Liste von Tokens; $parse\text{-}expr_0$ *end-of-input* überführt die Liste von Tokens in einen abstrakten Syntaxbaum.

33. Parser — Demo

```
Mini⟩ abstract-syntax-tree "4711"
Some (Const 4711)
Mini⟩ abstract-syntax-tree "4711+815*2765"
Some (Add (Const 4711, Mul (Const 815, Const 2765)))
Mini⟩ abstract-syntax-tree "1+2+3+4"
Some (Add (Const 1, Add (Const 2, Add (Const 3, Const 4))))
Mini⟩ abstract-syntax-tree "(1+2)+(3+4)"
Some (Add (Add (Const 1, Const 2), Add (Const 3, Const 4)))
Mini⟩ match it with Some expr → evaluate expr
10
Mini⟩ abstract-syntax-tree "(1+2+3)*(4+5+6)"
Some (Mul (Add (Const 1, Add (Const 2, Const 3)),
           Add (Const 4, Add (Const 5, Const 6))))
Mini⟩ match it with Some expr → evaluate expr
90
```

33. Zusammenfassung

Wir haben

- Syntax und Semantik regulärer Ausdrücke definiert,
- gesehen, wie man aus einem regulären Ausdruck systematisch mit Hilfe der Rechtsfaktoren einen Akzeptor herleitet,
- die Begriffe Interpreter und Übersetzer eingeführt,
- Möglichkeiten und Grenzen regulärer Ausdrücke kennengelernt,
- reguläre Ausdrücke um Rekursion erweitert: zu kontextfreien Ausdrücken,
- gesehen, wie man aus einem kontextfreien Ausdruck systematisch einen Akzeptor bzw. einen Parser herleitet,
- die Gefahr der Nichtterminierung bei linksrekursiven Grammatiken besprochen.

Teil VII

Effekte

33. Knobelaufgabe #18

Ist es möglich, ein *nicht-rekursives* Programm zu schreiben, das *nicht terminiert*?

Nicht-rekursiv bedeutet, dass in dem Programm weder rekursive Funktionsdefinitionen

$$\textbf{let rec } f\ (x_1 : t_1) : t_2 = \ldots f \ldots$$

noch rekursive Variantentypen

$$\textbf{type } T = \mid \ldots \mid \ldots T \ldots \mid \ldots$$

verwendet werden dürfen.

33. Gliederung

33. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- die Semantik externer Effekte erklären können,
- das Konzept des Speichers verstanden haben,
- Kontrollstrukturen kennen und verwenden können,
- Ausnahmen und deren Semantik erklären können,
- effektfreie und effektvolle Ausdrücke unterscheiden können.

33. Überblick

Dieses Kapitel ist dem Studium von Effekten gewidmet.

- *Bisher:* Eine Mini–F# Funktion ist eine Funktion im mathematischen Sinne: das Funktionsargument bestimmt das Funktionsergebnis.
- Mathematische Funktionen sind im gewissen Sinne autistisch.
- Im folgenden erweitern wir die Idee des Rechnens: ein Ausdruck kann neben dem Wert zusätzlich einen *Effekt* haben.
- *Externe Effekte:*
  - Ausgabe auf dem Bildschirm,
  - die Anforderung einer Eingabe,
  - das Einlesen von Sensordaten,
  - die Steuerung eines Motors usw.
- *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.

33. Eine Warnung vorneweg

☞ Effekte verändern die Natur des Rechnens!

- Ist eine Funktion effektvoll, dann handelt es sich nicht mehr um eine Funktion im mathematischen Sinne.
- Eine effektvolle Funktion kann bei gleichen Argumenten unterschiedliche Resultate liefern!
- *Gefahr:*
  Setzt man die neuen Sprachkonstrukte nicht mit Bedacht ein, dann
  - leidet die Lesbarkeit von Programmen;
  - leidet die Wartbarkeit von Programmen.

  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.

34. Motivation

Aufgabe: In Kapitel 3 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 Benutzer*in soll die Rolle von Spieler A übernehmen.

Natürlich wird Spieler A weiterhin durch eine Mini–F# Funktion realisiert,

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.

34. Demo

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.

34. Ausgabe

Ausgaben auf dem Bildschirm werden mit Hilfe der Funktion *putstring* getätigt. Der Aufruf

```
putstring "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; das Verhalten entspricht dem der Funktion ***fun*** $(s : \mathit{String}) \to ()$.

☞ Die Funktion *putstring* wird allein wegen ihres Effektes aufgerufen.

34. Eingabe

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. 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.

34. „Query the user“

Wir können *putstring* und *getline* kombinieren, um eine Funktion zu programmieren, die die Benutzer*in zu einer Eingabe auffordert.

```
let query (prompt : String) : String =
  let () = putstring prompt in getline ()
```

☞ Die lokale Bindung **let** $() = e_1$ **in** e_2 dient dazu, die Auswertung von zwei Ausdrücken und damit das Auftreten von Effekten zu *sequentialisieren*:

- Zunächst wird e_1 ausgerechnet,
- dann wird das Ergebnis mit dem Muster ‘()’ abgeglichen,
- anschließend wird e_2 ausgerechnet.

34. Syntaktischer Zucker

☞ „Bindungen ohne Bindungen“ können mit dem bekannten und beliebten Semikolonoperator abgekürzt werden.

let *query* (*prompt* : *String*) : *String* =
  putstring prompt; *getline* ()

Weiterhin erlauben wir die Definition ***let*** () = *e* mit ***do*** *e* abzukürzen. (Wird in Teil VIII häufiger verwendet.)

34. Motivation

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.

Zum Knobeln: Kann die menschliche Gegenspielerin *mogeln*?

34. Reihenfolge

☞ Effekte verändern die Natur des Rechnens: die Reihenfolge und die Multiplizität von Rechnungen spielen nunmehr eine Rolle.

Bisher: Die Komponenten des Paarausdrucks

$$(\mathit{factorial}\ 9, \mathit{factorial}\ 10)$$

konnten in beliebiger Reihenfolge ausgerechnet werden.

Jetzt: Wenn *factorial* zusätzlich einen Effekt hat, dann spielt die Reihenfolge der Teilrechnungen sehr wohl eine Rolle.

34. Multiplizität

Ebenso ist relevant, wie oft eine Rechnung durchgeführt wird.

Bisher: Der Ausdruck

$$\textbf{\textit{let}}\ f = \mathit{factorial}\ 9\ \textbf{\textit{in}}\ (f, f * 10)$$

ist äquivalent zu dem Ausdruck

$$(\mathit{factorial}\ 9, \mathit{factorial}\ 10)$$

Jetzt: Die Ausdrücke haben zwar den gleichen Wert aber wahrscheinlich einen anderen Effekt.

34. Demo

```
let rec factorial (n : Nat) : Nat =
  putstring (show n ^ "\n");
  if n = 0 then 1 else factorial (n - 1) * n
```

```
Mini⟩ (factorial 0, factorial 1)
0
1
0
(1, 1)
Mini⟩ (factorial 1, factorial 0)
1
0
0
(1, 1)
Mini⟩ let f = factorial 0 in (f, f * 1)
0
(1, 1)
```

34. Motivation

Beispiel: Eingabe von Personendaten.

```
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"]}
```

☞ Die Funktion *contains* : $String \to List\,\langle String\rangle \to Bool$ überprüft, ob das erste Argument in der angegebenen Liste enthalten ist.

34. Demo

```
Mini⟩ input-person ()
gender : male
name   : Ralf
bald?  : yes
Male {name = "Ralf"; bald = true}
```

☞ Warum haben wir *input-person* als Funktion des Typs $Unit \to Person$ definiert und nicht einfach als $Person$?

34. Abstrakte Syntax

Die Ein- und Ausgabeoperationen *putstring* und *getline* lassen sich auf Funktionen zurückführen, die ein einzelnes Zeichen ausgeben bzw. einlesen (zur Übung).

$e ::= \cdots$	*Ausdrücke:*
$\mid$ *getchar* e	Einlesen eines Zeichens
$\mid$ *putchar* e	Ausgabe eines Zeichens

34. Statische Semantik

Die Ein- und Ausgabeoperationen verarbeiten einzelne Zeichen.

$$\frac{\Sigma \vdash e : \mathit{Unit}}{\Sigma \vdash \mathit{getchar}\ e : \mathit{Char}} \qquad \frac{\Sigma \vdash e : \mathit{Char}}{\Sigma \vdash \mathit{putchar}\ e : \mathit{Unit}}$$

34. Dynamische Semantik

- Die Auswertung verändert sich mit dem Einzug von Effekten.
- Wie müssen wir die Auswertungsregeln modifizieren, um Interaktionen mit der Umwelt modellieren zu können?
- Beweisregeln sind genauso wenig interaktiv wie mathematische Funktionen!
- Ein Ausdruck hat neben einem *Wert* zusätzlich einen *Effekt*. Die dreistellige Relation

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

taugt nicht mehr für dieses Szenario.

- *Idee:* Wir erweitern die Auswertungsrelation zu einer *vierstelligen* Relation

$$\delta \vdash e \Downarrow_t \nu$$

die eine Umgebung mit einem Ausdruck, einem *externen Effekt* t und einem Wert in Beziehung setzt.

- Was ist ein externer Effekt?

34. Ereignisse

☞ Wir modellieren einen externen Effekt als *Sequenz* von Ereignissen, wobei ein einzelnes Ereignis die Ein- oder Ausgabe eines Zeichens ist.

$c \in \text{Unicode}$

$t \in \text{Event} ::=$		Ereignis
	$in(c)$	Eingabe von c
	$out(c)$	Ausgabe von c

34. Auswertungsregeln

Auswertungsregeln:

$$\frac{\delta \vdash e \Downarrow_t ()}{\delta \vdash \mathit{getchar}\ e \Downarrow_{t \cdot in(c)} c}$$

$$\frac{\delta \vdash e \Downarrow_t c}{\delta \vdash \mathit{putchar}\ e \Downarrow_{t \cdot out(c)} ()}$$

34. Beispiel

$$\dfrac{\dfrac{\varnothing \vdash () \Downarrow_{\epsilon} ()}{\varnothing \vdash \mathit{getchar}\ () \Downarrow_{\mathit{in}(\mathtt{h})}\ \mathtt{'h'}}}{\varnothing \vdash \mathit{putchar}\ (\mathit{getchar}\ ()) \Downarrow_{\mathit{in}(\mathtt{h}) \cdot \mathit{out}(\mathtt{h})} ()}$$

☞ $\mathit{in}(\mathtt{h}) \cdot \mathit{out}(\mathtt{h})$ ist nicht die einzige mögliche Ereignisfolge: auch $\mathit{in}(\mathtt{a}) \cdot \mathit{out}(\mathtt{a})$ oder $\mathit{in}(\mathtt{l}) \cdot \mathit{out}(\mathtt{l})$ usw. sind denkbar, nicht aber $\mathit{in}(\mathtt{h}) \cdot \mathit{out}(\mathtt{a})$ oder $\mathit{in}(\mathtt{a}) \cdot \mathit{out}(\mathtt{h})$.

☞ Die Auswertungsrelation ist eine „echte“ Relation und trägt damit der Tatsache Rechnung, dass viele unterschiedliche Interaktionen mit der Benutzer*in möglich sind.

34. Auswertungsregeln

☞ Da wir die Auswertungsrelation um ein Argument erweitert haben, müssen wir *alle* bisher aufgeführten Auswertungsregeln anpassen!

Jeder Teilausdruck kann einen Effekt haben:

$$(\mathit{putstring}\ \texttt{"Hello, "}; 4700) + (\mathit{putstring}\ \texttt{"world!"}; 11)$$

Die Auswertungsregel wird wie folgt abgeändert.

$$\frac{\delta \vdash e_1 \Downarrow_{t_1} n_1 \qquad \delta \vdash e_2 \Downarrow_{t_2} n_2}{\delta \vdash e_1 + e_2 \Downarrow_{t_1 \cdot t_2} n_1 + n_2}$$

☞ Beide Teilausdrücke haben einen Effekt, t_1 bzw. t_2; der kumulierte Effekt der Summe ist $t_1 \cdot t_2$. Somit treten die Effekte des ersten Summanden vor den Effekten des zweiten Summanden auf.

34. Auswertungsregeln

☞ Allgemein werden Ausdrücke von *links nach rechts* abgearbeitet und Effekte werden in dieser Reihenfolge sichtbar. Die Auswertungsregel

$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \qquad \cdots \qquad \delta_n \vdash e_n \Downarrow \nu_n}{\delta \vdash e \Downarrow \nu}$$

wird wie folgt erweitert:

$$\frac{\delta_1 \vdash e_1 \Downarrow_{t_1} \nu_1 \qquad \cdots \qquad \delta_n \vdash e_n \Downarrow_{t_n} \nu_n}{\delta \vdash e \Downarrow_{t_1 \cdot \ldots \cdot t_n} \nu}$$

☞ Die Reihenfolge der Effekte wird durch die Konkatenation der Ereignissequenzen festgelegt.

34. Auswertungsregeln: Beispiel

Beispiel: Die Regel für ***in***-Ausdrücke

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

wird zu

$$\frac{\delta \vdash d \Downarrow_{t'} \delta' \qquad \delta, \delta' \vdash e \Downarrow_{t} \nu}{\delta \vdash d \textbf{ in } e \Downarrow_{t' \cdot t} \nu}$$

34. Auswertungsregeln: Diskussion

- Die Einführung von effektvollen Ausdrücken hat einen dramatischen Effekt auf die Semantik von Mini–F#.
- Die Auswertungsregeln legen nunmehr pedantisch fest, in welcher Reihenfolge ein Programm abgearbeitet wird.
- Das ist in gewisser Weise ein Rückschritt:

  $e_1 + e_2$

  konnte bis dato gleichzeitig oder im Fachjargon *parallel* ausgerechnet werden.
- Wenn wir weiterhin eine parallele Auswertung wegen des möglichen Geschwindigkeitsvorteils anstreben, dann müssen wir sicherstellen, dass e_1 keine Effekte hat: $\delta \vdash e_1 \Downarrow_\epsilon \nu_1$.
- Diese Eigenschaft ist wie viele andere nicht formal entscheidbar!
- Die Parallelisierung von Programmen ist eine der großen Herausforderungen der Informatik.
- „The free lunch is over.“

34. Modularität

☞ Interaktive Programme sind 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.

Die Funktion *query* prüft die Eingabe nicht auf Plausibilität.

```
Mini⟩ query "age: "
age : Hello, world !
"Hello, world!"
```

34. Eingabe mit Validierung

Idee: query mit einem *Validator* parametrisieren.

```
let checked-query (prompt : String,
                   check : String → Result ⟨'value⟩) : 'value
```

☞ Ein Validator bildet einen String auf ein Element des folgenden Datentyps ab.

```
type Result ⟨'value⟩ =
  | Okay of 'value
  | Error of String
```

☞ 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.

34. Eingabe mit Validierung

☞ Eine validierende Version von *query*:

```
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.

34. Validierung

☞ Validatoren sind einfache, effektfreie Funktionen:

```
let is-nat (s : String) : Result ⟨Nat⟩ =
  if s <> "" && String.forall Char.IsDigit s then
    Okay (Nat.Parse s)
  else
    Error "natural number expected"
```

☞ *forall* : $(Char \to Bool) \to String \to Bool$ überprüft, ob alle Zeichen eines Strings das angegebene Prädikat erfüllen.

☞ *is-nat* ist im Prinzip ein einfacher Parser!

34. Demo

```
Mini⟩ checked-query ("age", is-nat)
age : Hello, world !
∗ ∗ ∗ natural number expected
age : 4711
4711
```

Wir sollten zusätzlich verlangen, dass die Altersangabe kleiner als 123 ist.

```
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
```

34. Validierung

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.

```
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
```

☞ 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* : *String* → *Result* ⟨*Nat*⟩
- *is-less* 123 : *Nat* → *Result* ⟨*Nat*⟩
- *both* (*is-nat*, *is-less* 123) : *String* → *Result* ⟨*Nat*⟩

34. Validierung

Die Funktion *is-less* kleidet 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)
```

34. Anwendungen

☞ Mit Hilfe von *checked-query* können wir z. B. 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.

34. Anwendungen

☞ 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 (zur Übung).

34. Demo

```
Mini⟩ input-person ()
gender : sehr maskulin
*** choices : f, m, female, male
gender : m
name   : Ralf
bald   : ja
*** choices : y, n, yes, no
bald   : y
Male {name = "Ralf"; bald = true}
```

34. Trace

☞ Ausgaben können auch beim Testen von Programmen nützliche Dienste leisten.

```
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.

34. Demo

```
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
```

34. Trace

☞ Die *Fibonacci-Funktion* zeigt 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?“.

34. Demo

```
Mini⟩ Call fibonacci 4
call   4
call   3
call   2
call   1
return 1
call   0
return 0
return 1
call   1
return 1
return 2
call   2
call   1
return 1
call   0
return 0
return 1
return 3
3
```

34. Trace

```
let Return (x : 'a) : 'a =
  putline ("return " ^ show x); x
```

☞ Wertmäßig ist *Return* die Identität: das Argument wird als Ergebnis zurückgegeben.

```
let Call (f : 'a → 'b) : 'a → 'b =
  fun x → putline ("call    " ^ show x); f x
```

☞ Wertmäßig ist *Call* die Identität von Funktionen.

35. Knobelaufgabe #19

Die *unendliche* Folge

$$0\,1\,0\,2\,0\,1\,0\,3\,0\,1\,0\,2\,0\,1\,0\,4\,0\,1\,0\,2\,0\,1\,0\,3\,0\,1\,0\,2\,0\,1\,0\,5\ldots$$

ist invariant unter der folgenden Transformation.

- Jedes Element wird um 1 erhöht.
- An den Anfang und zwischen je zwei Elemente wird eine 0 gesetzt.

Programmieren Sie eine Funktion, die alle Elemente der Folge nacheinander ausgibt.

35. Motivation

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–)*

35. Motivation

Wunsch: Die Protokollierung der Auswertung via *Return* und *Call* soll sich an- und ausschalten lassen.

☞ Ein *globaler Schalter* lässt sich mit den bisherigen Mitteln nicht realisieren. Funktionen lassen sich bis dato nur über ihre Funktionsargumente steuern — oder über Eingaben der Benutzer*in.

☞ Erweitert man *Return* und *Call* um ein Boolesches Argument, dann hat man viele *lokale Schalter*, aber keinen globalen.

35. Motivation

Idee: ein Gedächtnis in Form einer *Speicherzelle* einführen, die abgefragt und manipuliert werden kann.

☞ Ein Schalter kann durch eine Speicherzelle realisiert werden, die einen Booleschen Wert enthält.

let *trace* = ***ref*** *false*

☞ Der Ausdruck ***ref*** *false* legt eine neue Speicherzelle an, einen Behälter für einen Wert. Der initiale *Inhalt* der allokierten Speicherzelle ist *false*. Als Ergebnis des Aufrufs gibt ***ref*** die *Adresse* der allokierten Speicherzelle zurück.

35. Motivation

☞ Mit Hilfe der Adresse können wir den Inhalt der Speicherzelle abfragen.

```
!trace
```

☞ Wertet *e* zu einer Adresse aus, so ermittelt !*e* den Inhalt der adressierten Speicherzelle (lies: „bang e“).

Damit können wir eine Version von *putline* definieren, die sich von außen über den Schalter steuern lässt.

```
let traceline (s : String) =
  if !trace then putline s
```

☞ Nur wenn der Inhalt der Speicherzelle *true* ist, erfolgt die Ausgabe. (Wo ist das **else**?)

35. Motivation

☞ Mit Hilfe der *Zuweisung* $e_1 := e_2$ können wir den Inhalt der von e_1 adressierten Speicherzelle auf den Wert von e_2 setzen.

trace := *true*

Der Ausdruck schaltet die Protokollierung an.

trace := *false*

Der Ausdruck schaltet die Protokollierung aus.

☞ Der *Wert* der Zuweisung ist jeweils '()', unser liebgewonnener Dummywert.

In der Definition von *Return* und *Call* ersetzen wir *putline* durch *traceline*.

35. Demo

```
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
```

☞ ' !' ist keine Funktion im mathematischen Sinne: der gleiche Aufruf, *!trace*, führt zu zwei unterschiedlichen Ergebnissen.

35. Exkursion: Module

☞ Namen zu erfinden ist schwer!

F# erlaubt es, Definitionen in sogenannten Modulen zusammenzufassen, die jeweils einen eigenen „Namensraum“ bilden.

```
module Values.Modules

module Ann =
  let name = "Ann"
module Bob =
  let name = "Bob"

let hello = "Hello " ^ Ann.name ^ " and " ^ Bob.name ^ "!"
```

Das Programm definiert das Modul *Values.Modules*, das zwei lokale Module, *Ann* und *Bob*, enthält. Der Bezeichner *name* wird zweimal definiert, lebt aber in zwei unterschiedlichen Namensräumen.

Auf die definierten Werte kann mittels qualifizierter Namen zugegriffen werden: *Ann.name* und *Bob.name*. (Von „außen“ entsprechend: *Values.Modules.Ann.name*.)

35. Exkursion: Module

Die Verwendung qualifizierter Namen ist oft mühselig.

```
module Values.Modules

module Ann =
  let name = "Ann"
module Bob =
  let name = "Bob"

open Ann

let hello = "Hello " ^ name ^ " and " ^ Bob.name ^ "!"
```

Mit **open** werden Namensräume geöffnet.

35. Bankkonto

☞ Speicherzellen können beliebige Werte enthalten:

- Boolesche Werte,
- natürliche Zahlen,
- Funktionen,
- Adressen anderer Speicherzellen usw.

☞ Eine Speicherzelle, die eine *natürliche* Zahl enthält, kann zum Beispiel verwendet werden, um ein *Bankkonto* zu modellieren: der Inhalt repräsentiert den Kontostand.

35. Bankkonto

```
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
```

☞ '$\dot{-}$' bezeichnet die Subtraktion auf den natürlichen Zahlen („monus“). Die Differenz zwischen dem alten und dem neuen Kontostand wird zurückgegeben.

35. Kapselung

- Die Speicherzelle *funds* ist lokal zu *deposit*, *withdraw* und *balance*.
- Der Zusatz ***private*** stellt sicher, dass der Bezeichner *funds* nur innerhalb des Moduls sichtbar ist.
- Man sagt auch, der Zustand ist *gekapselt*; von außen ist nicht sichtbar, dass die Funktionen eine Speicherzelle verwenden.
- Der Kontostand kann nur mit Hilfe der Funktion *balance* eingesehen werden.
- Der Kontostand kann nicht direkt über eine Zuweisung verändert werden, sondern nur indirekt mit *deposit* und *withdraw*.
- Heimliche Kontomanipulationen sind in der Bankenwelt sehr ungern gesehenen.

35. Demo

```
Mini⟩ Account.deposit 4711
()
Mini⟩ Account.withdraw 815
815
Mini⟩ Account.withdraw 2765
2765
Mini⟩ Account.withdraw 2765
1131
Mini⟩ Account.withdraw 2765
0
```

☞ *withdraw* ist keine mathematische Funktion: drei Aufrufe der Form *withdraw* 2765, drei unterschiedliche Funktionsergebnisse.

Endlich mal was, das einigermaßen bekannt aussieht. Statt $funds := !funds + amount$ würde ich ja schreiben

```
funds = funds + amount
```

Und ich habe schon gedacht, du hättest C++ vergessen ;-).

Mon dieu, dosch nischt '='. Dasch ischt doch kein Vergleich, sondern eine Zuweischung.

```
funds := funds + amount
```

Na ja, es *ist* nicht gleich, aber es wird doch gleich *gemacht*.

Aber die Ausdrücke links und rechts haben doch noch nicht einmal den gleichen Typ!

Wieso das denn nicht? Links *Nat* (na ja, eigentlich `unsigned int`) und rechts *Nat*.

Dann darf ich auch schreiben:

```
4711 = 815
```

Links *Nat*, rechts *Nat*.

Natürlich nicht! Links muss eine Variable stehen!

Genau. Links muss ein Ausdruck stehen, der zu einer *Adresse* auswertet. Der neue *Inhalt* der adressierten Speicherzelle steht dann auf der rechten Seite.

Aber dann ergibt doch `funds + amount` keinen Sinn: `funds` ist eine Adresse und `amount` eine Zahl.

Genau. Deswegen schreiben wir in Mini–F# ja auch $funds := !funds + amount$.

35. Abstrakte Syntax

Wir erweitern Mini–F# um Speichermanipulationen.

$e ::= \cdots$	*Ausdrücke:*
$\mid$ ***ref*** e	Allokation
$\mid$ $!e$	Dereferenzierung
$\mid$ $e_1 := e_2$	Zuweisung

☞ ***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.

35. Statische Semantik

Adressen erhalten einen Referenztyp; dieser ist mit dem Typ des Inhaltes parametrisiert.

$t ::= \cdots$	*Typen:*
$\quad \mid \mathit{Ref}\,\langle t \rangle$	Referenztyp

Typregeln:

$$\frac{\Sigma \vdash e : t}{\Sigma \vdash \textbf{\textit{ref}}\ e : \mathit{Ref}\,\langle t \rangle}$$

$$\frac{\Sigma \vdash e : \mathit{Ref}\,\langle t \rangle}{\Sigma \vdash\ !e : t}$$

$$\frac{\Sigma \vdash e_1 : \mathit{Ref}\,\langle t \rangle \qquad \Sigma \vdash e_2 : t}{\Sigma \vdash e_1 := e_2 : \mathit{Unit}}$$

35. Statische Semantik

Zur Erinnerung: Ausdrücke können beliebig miteinander kombiniert werden.

$$\textbf{\textit{let}}\ p = (\textbf{\textit{ref}}\ \mathit{false}, \textbf{\textit{ref}}\ 0)$$

☞ p hat den Typ $\mathit{Ref}\langle \mathit{Bool}\rangle * \mathit{Ref}\langle \mathit{Nat}\rangle$.

Alle folgenden Ausdrücke sind zulässig:

$!(\mathit{fst}\ p)$

$\mathit{snd}\ p := 4711$

$\mathit{fst}\ (\mathit{swap}\ p) := !(\mathit{snd}\ p) + 1$

35. Dynamische Semantik

Die drei neuen Konstrukte manipulieren einen sogenannten (Haupt-) Speicher.

Ein Speicher ist eine endliche Abbildung von Adressen auf Werte.

$a \in \text{Addr}$	*Adressen*
$\sigma \in \text{Addr} \to_{\text{fin}} \text{Val}$	*Speicher*

☞ Den Bereich der Adressen lassen wir abstrakt; wir fordern nur, dass es unendlich viele Adressen gibt.

35. Dynamische Semantik

- Die Auswertung verändert sich mit dem Einzug von Effekten.
- Wie müssen wir die Auswertungsregeln modifizieren, um Speichermanipulationen modellieren zu können?
- Ein Ausdruck hat neben einem *Wert* zusätzlich einen *Effekt*. Die dreistellige Relation

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

taugt nicht mehr für dieses Szenario.

- *Idee:* Wir erweitern die Auswertungsrelation zu einer *fünfstelligen* 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.

- (Im Interesse der Lesbarkeit lassen wir die Umgebung δ unter den Tisch fallen — ebenso ignorieren wir Ein- und Ausgaben.)

35. Dynamische Semantik

Wir müssen den Bereich der Werte um Adressen erweitern.

$\nu ::= \cdots$	*Werte:*
$\mid a$	Adresse

Auswertungsregeln:

$$\frac{\sigma \parallel e \Downarrow \nu \parallel \sigma'}{\sigma \parallel \textbf{\textit{ref}}\ e \Downarrow a \parallel \sigma', \{a \mapsto \nu\}} \qquad a \notin dom\ \sigma'$$

$$\frac{\sigma \parallel e \Downarrow a \parallel \sigma'}{\sigma \parallel\ !e \Downarrow \sigma'(a) \parallel \sigma'}$$

$$\frac{\sigma \parallel e_1 \Downarrow a \parallel \sigma_1 \qquad \sigma_1 \parallel e_2 \Downarrow \nu \parallel \sigma_2}{\sigma \parallel e_1 := e_2 \Downarrow () \parallel \sigma_2, \{a \mapsto \nu\}}$$

☞ $!e$ ist ein *lesender Speicherzugriff*; $e_1 := e_2$ ein *schreibender Speicherzugriff*. *Beachte:* $\sigma'(a)$ ist stets definiert.

35. Beispiel

$$\dfrac{\dfrac{\emptyset \parallel 0 \Downarrow 0 \parallel \emptyset}{\emptyset \parallel \textbf{\textit{ref}}\ 0 \Downarrow a_1 \parallel \{a_1 \mapsto 0\}}}{\emptyset \parallel \textbf{\textit{ref}}\ (\textbf{\textit{ref}}\ 0) \Downarrow a_2 \parallel \{a_1 \mapsto 0, a_2 \mapsto a_1\}}$$

☞ 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 $\{a_2 \mapsto a_1\}$.

35. Auswertungsregeln

☞ Da wir die Auswertungsrelation um zwei Argumente erweitert haben, müssen wir *alle* bisher aufgeführten Auswertungsregeln anpassen!

Jeder Teilausdruck kann einen Effekt haben:

$$(\mathit{trace} := \mathit{true}; \mathit{factorial}\ 9) + (\mathit{trace} := \mathit{false}; \mathit{factorial}\ 10)$$

Die Auswertungsregel wird wie folgt abgeändert.

$$\frac{\sigma_0 \parallel e_1 \Downarrow \nu_1 \parallel \sigma_1 \qquad \sigma_1 \parallel e_2 \Downarrow \nu_2 \parallel \sigma_2}{\sigma_0 \parallel e_1 + e_2 \Downarrow \nu_1 + \nu_2 \parallel \sigma_2}$$

☞ Beide Teilausdrücke verändern potentiell den Speicher: e_1 ändert σ_0 zu σ_1, e_2 „sieht“ den modifizierten Speicher und ändert ihn zu σ_2.

35. Auswertungsregeln

☞ Allgemein werden Ausdrücke von *links nach rechts* abgearbeitet und Speicherveränderungen werden in dieser Reihenfolge sichtbar. Die Auswertungsregel

$$\frac{e_1 \Downarrow \nu_1 \qquad e_2 \Downarrow \nu_2 \qquad \cdots \qquad e_n \Downarrow \nu_n}{e \Downarrow \nu}$$

wird wie folgt angepasst:

$$\frac{\sigma_0 \parallel e_1 \Downarrow \nu_1 \parallel \sigma_1 \qquad \sigma_1 \parallel e_2 \Downarrow \nu_2 \parallel \sigma_2 \qquad \cdots \qquad \sigma_{n-1} \parallel e_n \Downarrow \nu_n \parallel \sigma_n}{\sigma_0 \parallel e \Downarrow \nu \parallel \sigma_n}$$

☞ Der Zustand wird jeweils von Teilausdruck zu Teilausdruck weitergereicht. Man sagt auch, der Zustand wird *durchgefädelt* (engl. threading).

35. Vertiefung

☞ Eine Speicherzelle vom Typ $\mathit{Ref}\,\langle \mathit{Nat} \rangle$ können wir verwenden, um zu zählen, wie oft eine bestimmte Funktion aufgerufen wird. (Zum Zwecke der Programmoptimierung.)

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{counter} = \textbf{\textit{ref}}\ 0 \\
\textbf{\textit{let rec}}\ \mathit{fibonacci}\ (n : \mathit{Nat}) : \mathit{Nat} = \\
\quad \mathit{counter} := \ !\mathit{counter} + 1; \\
\quad \textbf{\textit{if}}\ n \leqslant 1\ \textbf{\textit{then}} \\
\qquad n \\
\quad \textbf{\textit{else}} \\
\qquad \mathit{fibonacci}\ (n \dot{-} 1) + \mathit{fibonacci}\ (n \dot{-} 2)
\end{array}
$$

35. Demo

Die Berechnung von *fibonacci n* ist aufwändig:

Mini⟩ $counter := 0; \textbf{\textit{let}}\ f = fibonacci\ 10\ \textbf{\textit{in}}\ (f,\ !counter)$
$(55, 177)$
Mini⟩ $counter := 0; \textbf{\textit{let}}\ f = fibonacci\ 20\ \textbf{\textit{in}}\ (f,\ !counter)$
$(6765, 21891)$

☞ Die Anzahl der Aufrufe übersteigt den Wert der Fibonaccifunktion in beiden Fällen.

35. Vertiefung

Wir können auch effektfrei zählen:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \textit{counting-fibonacci}\ (n : \textit{Nat}) : \textit{Nat} * \textit{Nat} = \\
\quad \textbf{\textit{if}}\ n \leqslant 1\ \textbf{\textit{then}} \\
\qquad (n, 1) \\
\quad \textbf{\textit{else}} \\
\qquad \textbf{\textit{let}}\ (f_1, c_1) = \textit{counting-fibonacci}\ (n \dot{-} 1) \\
\qquad \textbf{\textit{let}}\ (f_2, c_2) = \textit{counting-fibonacci}\ (n \dot{-} 2) \\
\qquad (f_1 + f_2, c_1 + c_2 + 1)
\end{array}
$$

☞ Die Zahl c der Aufrufe für die Berechnung von *fibonacci n* ist fast doppelt so groß wie der Funktionswert von *fibonacci* $(n + 1)$, es gilt: $c = 2 \cdot \textit{fibonacci}\ (n + 1) - 1$.

35. Memoisierung

Problem: Die gleichen Funktionswerte werden wiederholt berechnet: *fibonacci* 8 wird bei der Berechnung von *fibonacci* 20 insgesamt 233 mal ($=$ *fibonacci* 12) neu ausgerechnet.

Idee: Wir merken uns die Aufrufe und greifen später auf die memorierten Werte zurück.

let *memo-fibonacci* $=$ $[|$ ***for*** *i* ***in*** $0\,..\,99 \to$ *fibonacci i* $|]$

☞ Statt *fibonacci n* verwenden wir *memo-fibonacci*.$[n]$.

35. Memoisierung

Abstrahieren wir von $0\,..\,99$ und von *fibonacci*, erhalten wir

```
let memo (dom : Nat, func : Nat → 'v) : Nat → 'v =
  let
    memo-table = [| for i in 0 .. dom - 1 → func i |]
  in
    fun (n : Nat) → memo-table.[n]
let memo-fibonacci = memo (100, fibonacci)
```

35. Demo

fsi Memo.fs
...

☞ Unglücklicherweise dauert die Auswertung von *fibonacci* extrem (!) lange, da zunächst die Tabelle vollständig gefüllt wird.

☞ Eigentlich schwebt uns eine *bedarfsgetriebene* Füllung der Tabelle vor: erst wenn ein Funktionswert angefordert wird, berechnet *memo* den entsprechenden Tabelleneintrag.

35. Memoisierung

☞ Wenn wir Tabelleneinträge ändern wollen, müssen wir statt einem Array von Zahlen ein Array von Speicherzellen verwenden.

Jede Speicherzelle nimmt 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 $\mathit{Array}\,\langle \mathit{Ref}\,\langle \mathit{Option}\,\langle 'a \rangle \rangle \rangle$ statt $\mathit{Array}\,\langle 'a \rangle$ wie bisher.

35. Memoisierung

```
let memo (dom : Nat, func : Nat → 'v) : Nat → 'v =
  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
                  do memo-table.[n] := Some v
                  v
       | Some v → v
let memo-fibonacci = memo (100, fibonacci)
```

35. Demo

fsi Memo.fs
Mini⟩ *memo-fibonacci* 99
...

☞ Die Auswertung der beiden Definitionen dauert nunmehr keinen Fingerschnips.

☞ Jetzt tritt eine lange (!) Stille ein, wenn wir *memo-fibonacci* 99 aufrufen: die *rekursiven* Aufrufe der Fibonacci Funktion werden *nicht* memoisiert.

35. Memoisierung

Was ist zu tun?

- Wertedefinitionen dürfen nicht rekursiv sein.
- Die Wertedefinition in eine Funktionsdefinition zu überführen,

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{fibonacci}\ (n : \mathit{Nat}) : \mathit{Nat} = \\ \quad \mathit{memo}\ (100, \textbf{\textit{fun}}\ \mathit{fib}\ (n : \mathit{Nat}) \to \\ \quad\quad \textbf{\textit{if}}\ n \leqslant 1\ \textbf{\textit{then}}\ n \\ \quad\quad\quad\quad \textbf{\textit{else}}\ \ \mathit{fibonacci}\ (n \dot{-} 1) + \mathit{fibonacci}\ (n \dot{-} 2))\ n\end{array}$$

ist verführerisch, aber nicht sinnvoll: Für jeden rekursiven Aufruf wird eine *neue* Memotabelle angelegt!

- Wir können die rekursiven Aufrufe memoisieren, indem wir *memo* keine Funktion des Typs $\mathit{Nat} \to {}'v$ übergeben, sondern eine Funktion des Typs

$$(\mathit{Nat} \to {}'v) \to (\mathit{Nat} \to {}'v)$$

die über die rekursiven Aufrufe abstrahiert.

35. Memoisierung

```
let rec-memo (dom : Nat, functional : (Nat → 'v) → (Nat → 'v)) : Nat → 'v =
  let memo-table = [| for i in 0 .. dom - 1 → ref None |]
  let rec memo-f (n : Nat) : 'v =
    match !memo-table.[n] with
    | None   → let v = functional memo-f n
               do 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))
```

35. Memoisierung

fsi Memo.fs
Mini⟩ *memo-fibonacci* 99
35422484817926191507 5

☞ Der Lohn der Anstrengungen: sowohl die Definition von *memo-fibonacci* als auch alle Aufrufe von *memo-fibonacci* sind in Windeseile ausgerechnet; auch die rekursiven Aufrufe füllen die Tabelle.

35. Knobelaufgabe #20

Knut Don ist auf die Idee gekommen, natürliche Zahlen durch Bäume zu repräsentieren.

$$\textbf{\textit{type}}\ \mathit{Tree} = \mid \mathit{Tip} \mid \mathit{Fork}\ \textbf{\textit{of}}\ \mathit{Tree} * \mathit{Tree}$$

Der leere Baum stellt die Zahl 0 dar, der Baum $\mathit{Fork}\ t_1\ t_2$ repräsentiert $2^{n_1} + n_2$, wobei n_i die Bedeutung von t_i ist. Als Mini–F# Programm:

```
let rec nat (t : Tree) : Nat =
  match t with
    | Tip          → 0
    | Fork (l, r) → power (2, nat l) + nat r
```

Eine natürliche Zahl hat viele Darstellungen: $\mathit{Fork}\ l_1\ (\mathit{Fork}\ l_2\ r)$ und $\mathit{Fork}\ l_2\ (\mathit{Fork}\ l_1\ r)$ bezeichnen z. B. stets die gleiche Zahl. Für das Rechnen ist es vorteilhaft, eine *eindeutige* Repräsentation zu haben. Deswegen fordern wir, dass stets $2^{n_1} > n_2$ gilt.

Implementieren Sie die arithmetischen und die Vergleichsoperationen für diese merkwürdige Zahlendarstellung.

35. Pascal und C

☞ In den Programmiersprachen Pascal (1970 geboren) und C (1972 geboren) spielen Speicherzellen eine weitaus größere Rolle als in Mini–F#.

Es wird gerechnet, indem Inhalte von Speicherzellen schrittweise verändert werden.

Schauen wir uns die Unterschiede etwas genauer an ...

35. Pascal und C: Variablen

Mini–F#:

```
let i = ref 0
```

Pascal:

```
var i:integer;
```

☞ Initialisierung nicht möglich!

C:

```
int i = 0;
```

☞ Initialisierung nicht nötig!

35. Pascal und C: Konstanten

Mini–F#:

```
let n = 4711
```

Pascal:

```
const n = 4711;
```

C:

```
const int n = 0;
```

35. Pascal und C: Funktionen

Mini–F#:

$$\textbf{\textit{let}}\ succ\ (n : Nat) = n + 1$$

Pascal:

```
function succ (n : integer):integer;
begin
  succ := n + 1
end;
```

C:

```
int succ (int n) {
  return (n + 1);
}
```

☞ Funktionen können nur deklariert werden. In C dürfen Funktionen nicht lokal deklariert werden; in Pascal dürfen Funktionen nicht als Ergebnis zurückgegeben werden.

35. Pascal und C: Zuweisung

Mini–F#:

$$i := !i + 4711$$

Pascal:

```
i := i + 4711;
```

C:

```
i = i + 4711
```

☞ In Pascal und C werden auf der rechten Seite Speicherzellen automatisch dereferenziert: man spricht auch von L- und R-Werten einer Variablen.

Mini–F# hat übrigens eine alternative Notation für Speicherzellen, die Harry vielleicht entgegenkommt.

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*

Ein veränderlicher Bezeichner wird automatisch dereferenziert. *Lies: funds* wird zu *funds* $+$ *amount* (engl. *funds* becomes *funds* $+$ *amount*). In Kürze mehr dazu.

35. Pascal und C

☞ In Pascal und C dürfen Funktionen keine aggregierten Daten wie Paare oder Records zurückgeben.

Ausweg: Das Funktionsergebnis wird nicht über den Rückgabewert, sondern über das Argument kommuniziert!

Mini–F#:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{factorial}\ (n : \mathit{Nat}, \mathit{result} : \mathit{Ref}\langle \mathit{Nat}\rangle) = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}} \\
\qquad \mathit{result} := 1 \\
\quad \textbf{\textit{else}} \\
\qquad \mathit{factorial}\ (n \dot{-} 1, \mathit{result}); \\
\qquad \mathit{result} :=\ !\mathit{result} * n
\end{array}
$$

☞ Der Funktion wird die Adresse übergeben, unter der sie das Ergebnis ablegen soll — ähnlich einem Postfach.

35. Demo

```
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.

35. Pascal und C: call by reference

Pascal:

```
procedure factorial (n : integer, var result : integer);
begin
  if n=0 then
    result := 1
  else
    begin
      factorial (n - 1, result);
      result := result * n
    end
end;
```

☞ Der zweite Parameter ist ein *Variablen-* oder *Referenzparameter*.

```
factorial (pob, pob);
```

☞ Der erste Parameter wird *call by value* (R-Wert) übergeben, der zweite *call by reference* (L-Wert).

35. Pascal und C

C:

```
void factorial (int n, int* result) {
  if (n == 0)
    *result = 1;
  else {
    factorial (n - 1, result);
    *result *= n;
  }
}
```

☞ Der zweite Parameter ist ein Zeiger auf eine Zahl — Cs Terminologie für Adresse; * dereferenziert eine Adresse.

```
factorial (pob, &pob);
```

☞ & ist der Adressoperator; dieser macht die automatische Dereferenzierung rückgängig.

35. Lebensdauer

- Speicherzellen leben in Mini–F# 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.
- *Hoffnung:* eine Speicherzelle wird nicht mehr benötigt, wenn ihre Adresse nicht mehr sichtbar ist.
- In Mini–F# gilt dies *nicht*: *funds* ist nach Abarbeitung des lokalen Moduls nicht mehr sichtbar, wird aber benötigt.
- In Pascal ist die Hoffnung berechtigt. *Aber:* Verlust an Ausdruckskraft.
- In C gilt dies *nicht* wegen des Adressoperators. *Trotzdem wird deallokiert.* ☞ dangling pointer.

35. Garbage collection

- Wann und wie werden in Mini–F# Speicherzellen deallokiert?
- Speicherzellen werden deallokiert, wenn sie tatsächlich nicht mehr benötigt werden und wenn der Speicherplatz knapp wird.
- Diese Aufgabe übernimmt eine sogenannter *Garbage collector* (engl. Müllmann), ein wichtiger Bestandteil des Mini–F# Interpreters bzw. des Laufzeitsystems.
- 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.

35. Pascal und C: Listen

☞ Da in Pascal und C das Konzept des Speichers dominierend ist, gehen diese Sprache auch die Implementierung von Datenstrukturen wie Listen oder Bäumen anders an. Wir programmieren eine typische Listenimplementierung in Mini–F# nach.

$$\begin{array}{lll} \textbf{type}\ \mathit{List}\ \langle'a\rangle & = & \mathit{Ref}\ \langle \mathit{Item}\ \langle'a\rangle\rangle \\ \textbf{and}\ \ \mathit{Item}\ \langle'a\rangle & = & \mid \mathit{Nil} \\ & & \mid \mathit{Cons}\ \textbf{of}\ 'a * \mathit{List}\ \langle'a\rangle \end{array}$$

☞ Listen sowie sämtliche Restlisten sind Speicherzellen.

Zum Vergleich: die Listenimplementierung aus Kapitel 4.

$$\textbf{type}\ \mathit{List}\ \langle'a\rangle = \mathit{Nil} \mid \mathit{Cons}\ \textbf{of}\ 'a * \mathit{List}\ \langle'a\rangle$$

35. Pascal und C: Listen

Listenoperationen arbeiten typischerweise „in situ“ (lat. am Platz) durch Modifikation der bestehenden Strukturen.

```
let rec append (list1 : List ⟨'a⟩, list2 : List ⟨'a⟩) =
  match !list1 with
      | Nil          → list1 := !list2
      | Cons (x, xs) → append (xs, list2)
```

☞ Es wird keine neue Liste konstruiert.

35. Pascal und C: Listen

Zum Vergleich: die Listenimplementierung aus Kapitel 4.

```
let rec append (list1 : List ⟨'a⟩, list2 : List ⟨'a⟩) : List ⟨'a⟩ =
  match list1 with
    | Nil          → list2
    | Cons (x, xs) → Cons (x, append (xs, list2))
```

☞ *append* konstruiert eine Ergebnisliste.

35. Demo

```
let primes = ref (Cons (2, ref (Cons (3, ref Nil))))
val primes : List ⟨Nat⟩
Mini⟩ primes
{contents = Cons (2, {contents = Cons (3, {contents = Nil})})}
Mini⟩ append (primes, ref (Cons (5, ref (Cons (7, ref Nil)))))
()
Mini⟩ primes
{contents = Cons (2, {contents = Cons (3, {contents =
    Cons (5, {contents = Cons (7, {contents = Nil})})})})}
```

☞ *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).

35. Pascal und C: Listen

Zum Vergleich: die Listenimplementierung aus Kapitel 4:

```
Mini⟩ let primes = Cons (2, Cons (3, Nil))
val primes : List ⟨Nat⟩
Mini⟩ primes
Cons (2, Cons (3, Nil))
Mini⟩ append (primes, Cons (5, Cons (7, Nil)))
Cons (2, Cons (3, Cons (5, Cons (7, Nil))))
Mini⟩ primes
Cons (2, Cons (3, Nil))
```

☞ *append* ist nicht destruktiv: *append* (*list1*, *list2*) konkateniert die Listen *list1* und *list2*; die Argumentlisten sind nach dem Aufruf unverändert, sie überdauern den Aufruf.

☞ Aus diesem Grund spricht man auch von einer *persistenten Datenstruktur* (lat. anhaltend, beharrlich).

☞ Persistente Datenstrukturen sind in der Regel ephemeren Datenstrukturen vorzuziehen: da sie effektfrei sind, sind sie leichter zu verstehen, zu verwenden und darüber hinaus auch flexibler.

35. Pascal und C: Listen

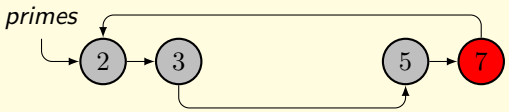
☞ Die Gefahr von Effekten illustriert die folgende Sitzung.

```
Mini⟩ primes
{contents = Cons (2, {contents = Cons (3, {contents =
    Cons (5, {contents = Cons (7, {contents = Nil})})})})}
Mini⟩ append (primes, primes)
()
Mini⟩ primes
...
```

☞ Wir hängen die Liste der ersten fünf Primzahlen an die Liste der ersten fünf Primzahlen und erhalten — ja, was eigentlich?

35. Pascal und C: Listen

Die Auswertung terminiert nicht! Der Aufruf *append* (*primes*, *primes*) hat eine *zyklische Liste* konstruiert.



☞ Das Struktur Entwurfsmuster garantiert nicht länger, dass die nach dem Muster gestrickten Funktionen auch terminieren!

☞ Aber es kommt noch schlimmer ...

35. Lösung Knobelaufgabe #18

☞ Selbst die Terminierung von nicht-rekursiven Funktionen ist nicht mehr gewährleistet!

$$
\begin{array}{l}
\textbf{let}\ fac = \textbf{ref}\ (\textbf{fun}\ (n : Nat) \to 0) \\
\textbf{let}\ factorial = \\
\quad fac := (\textbf{fun}\ (n : Nat) \to \\
\qquad \textbf{if}\ n = 0\ \textbf{then}\ 1 \\
\qquad\qquad \textbf{else}\ \ n * (\ !fac)\ (n \dot{-} 1)) \\
\quad !fac
\end{array}
$$

☞ Die erste Wertebindung allokiert eine Speicherzelle des Typs $Ref\langle Nat \to Nat\rangle$. 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.

☞ Ersetzen wir die Alternative durch $(\ !fac)\ n$, dann terminiert *factorial* nicht.

35. Veränderliche

Neben Speicherzellen bietet F# noch ein weiteres Konstrukt an, um Programme mit einem Gedächtnis auszustatten: *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*. Wie in C oder Pascal wird ein veränderlicher Bezeichner automatisch dereferenziert. Der Bezeichner *funds* hat den Typ *Nat*.

35. Speicherzellen versus Veränderliche

	Allokation	Dereferenzierung	Zuweisung
Speicherzelle	***ref*** e	$!e$	$e_1 := e_2$
Veränderliche	***let mutable*** $x = e$	x	$x \leftarrow e$

☞ Die Vor- und Nachteile der automatischen Dereferenzierung haben wir bereits diskutiert; sie gelten in gleicher Weise für ***mutable***s.

35. Speicherzellen, da capo

Nicht nur Bezeichner können als veränderlich gekennzeichnet werden, auch Komponenten von Records.

Implementierung von Speicherzellen:

```
type Ref⟨'value⟩ =
  {mutable contents : 'value}
let ref value =
  {contents = value}
let (!) cell =
  cell.contents
let (:=) cell value =
  cell.contents ← value
```

☞ Der Unterschied zwischen einer Speicherzelle, einer Adresse, und ihrem Inhalt wird deutlich: *cell* ist die Speicherzelle und *cell.contents* ihr Inhalt.

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."

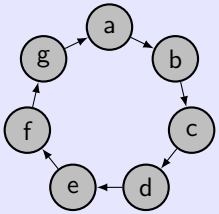
— J. R. Ross, Constraints on Variables in Syntax

35. Perlenketten

Besteht ein Record aus mehreren Komponenten, kann man selektiv für jede Komponente entscheiden, ob sie veränderlich sein soll.

```
type Necklace ⟨'elem⟩ =
  {bead            : 'elem
   mutable next : Necklace ⟨'elem⟩}
```

```
type Necklace ⟨'elem⟩ =
  {mutable bead : 'elem
   mutable next : Necklace ⟨'elem⟩}
```



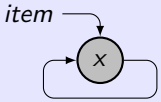
☞ Der Typ ist rekursiv definiert: Eine Kette besteht aus einer Perle und einer Kette, die aus einer Perle besteht und einer Kette, die ...

☞ Wo ist der Basisfall?

35. Konstruktion einer Perlenkette

Wie konstruieren wir ein zyklisches Geflecht? Durch eine rekursive *Werte*definition!

```
let single x =
  let rec item =
    { bead = x
      next = item }
  in item
```



☞ Rekursive Wertedefinitionen ergeben im Allgemeinen keinen Sinn: ***let rec*** $n = n + 1$.

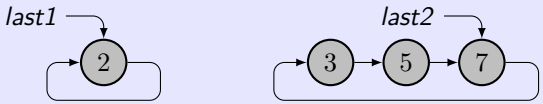
☞ Die Auswertung von *item* ist trickreich:

- 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.

Aus einer rekursiven Definition wird so ein zyklisches Geflecht.

35. Nicht-leere Sequenzen

Wir können Perlenketten verwenden, um nicht-leere, *endliche* Sequenzen von Elementen darzustellen.



☞ Wir verweisen jeweils auf das *letzte* Element der zu repräsentierenden Folge. (Warum?)

35. Nicht-leere Sequenzen: *head* und *tail*

Kopfelement und Restliste:

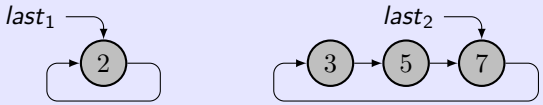
```
let head (last : Necklace ⟨'elem⟩) : 'elem =
  last.next.bead
let tail (last : Necklace ⟨'elem⟩) : Necklace ⟨'elem⟩ =
  last.next ← last.next.next
  last
```

☞ Perlenketten sind ephemeral: Die Funktion *tail* verändert ihr Argument.

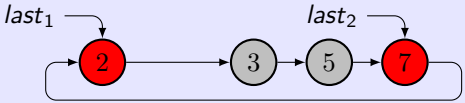
35. Nicht-leere Sequenzen: Konkatenation

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.




```
let swap (p : Necklace ⟨'elem⟩, q : Necklace ⟨'elem⟩) =
  let tmp = p.next
  p.next ← q.next
  q.next ← tmp
```

35. Nicht-leere Sequenzen: Konkatenation

Das Ergebnis der Konkatenation ist die zweite Folge.

let *append* $(last_1 : Necklace\ \langle 'elem \rangle, last_2 : Necklace\ \langle 'elem \rangle) : Necklace\ \langle 'elem \rangle =$
  swap $(last_1, last_2)$
  $last_2$

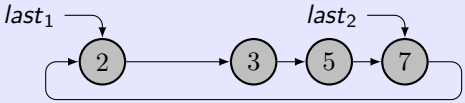
☞ Auch *append* verändert ihre Argumente: repräsentiert $last_i$ die Folge xs_i, dann

- repräsentiert $last_2$ *nach* dem Aufruf von *append* die Folge $xs_1 @ xs_2$,
- während $last_1$ die Folge $xs_2 @ xs_1$ darstellt.

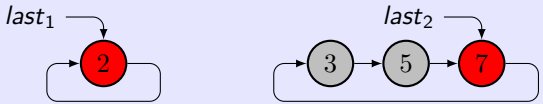
☞ Eine perfekte Symmetrie: *swap* implementiert so etwas wie die „symmetrische Konkatenation“ zweier Folgen.

35. Nicht-leere Sequenzen: Aufspaltung

Mit Hilfe von *swap* konkatenieren wir zwei Listen; mit Hilfe von *swap* können wir aber auch eine Liste aufspalten!



Wir müssen lediglich $last_1.next$ und $last_2.next$ vertauschen.



☞ Mit dem zweiten *swap* kehren wir zum alten Zustand zurück!

Verbesserte Definition von *tail*:

let $tail\ last = swap\ (last, last.next); last$

35. Nicht-leere Sequenzen: Demo

```
Mini⟩ let xs = append (single 1, single 2)
Mini⟩ let ys = append (single 815, single 4711)
Mini⟩ swap (xs, ys)
()
Mini⟩ xs
{ bead = 2; next = { bead = 815; next = { bead = 4711;
    next = { bead = 1; next = ...}}}}
Mini⟩ ys
{ bead = 4711; next = { bead = 1; next = { bead = 2;
    next = { bead = 815; next = ...}}}}
Mini⟩ swap (xs, ys)
()
Mini⟩ xs
{ bead = 2; next = { bead = 1; next = ...}}
Mini⟩ ys
{ bead = 4711; next = { bead = 815; next = ...}}
```

35. Anwendung: Ringpuffer

Aufgabe: schreibe einen Akzeptor für die Sprache

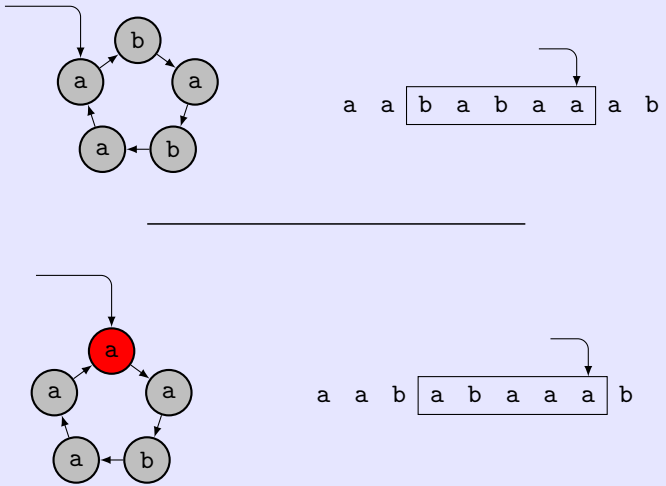
$$(\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b})^n$$

Wieviele Rechtsfaktoren hat die Sprache?

Idee: wir merken uns die letzten $n+1$ Zeichen in einem *Ringpuffer*. (Eigentlich müssten wir uns nur merken, *ob* das $(i+1)$-te Zeichen von hinten ein b oder ein anderes Zeichen ist.)

35. Anwendung: Ringpuffer — Beispiel

Mit Hilfe einer Perlenkette bewegen wir ein Sichtfenster über die Eingabefolge.



35. Anwendung: Ringpuffer — Programm

Allokation eines Ringpuffers der Größe $n + 1$:

```
let rec replicate (n : Nat) (a : 'a) : Necklace ⟨'a⟩ =
  if n = 0 then single a
           else append (replicate (n − 1) a, single a)
```

Der eigentliche Akzeptor:

```
let accept (n : Nat) : String → Bool =
  let mutable buffer = replicate n 'a'
  let rec loop = function
    | []      → buffer.next.bead = 'b'
    | x :: xs → buffer ← buffer.next
                buffer.bead ← x
                loop xs
  explode ≫ loop
```

35. Anwendung: Ringpuffer — Demo

```
Mini⟩ accept 3 "bb"
false
Mini⟩ accept 3 "bb"
false
Mini⟩ accept 3 "aa"
false
Mini⟩ accept 3 "aa"
false
```

```
Mini⟩ let accept3 = accept 3
Mini⟩ accept3 "bb"
false
Mini⟩ accept3 "bb"
true
Mini⟩ accept3 "aa"
true
Mini⟩ accept3 "aa"
false
```

Merkwürdig?

Der Aufruf *accept* 3 erzeugt jeweils einen *neuen* Ringpuffer.

In *accept3* ist *ein* Ringpuffer fest verdrahtet, der sich die vorherigen Eingaben merkt! So wird die Eingabe sozusagen kontinuierlich fortgeschrieben.

35. Persistente versus ephemerale Listen: Laufzeit

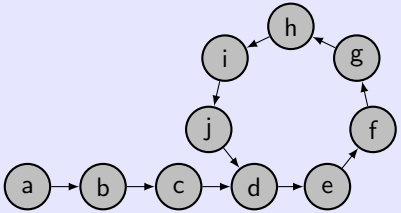
	persistent	ephemeral
head	konstant	konstant
tail	konstant	konstant
cons	konstant	konstant
append	linear zur Länge der ersten Liste	konstant

☞ Mit fortgeschrittenen Datenstrukturen lassen sich auch persistente Sequenzen implementieren, die die Konkatenation in konstanter Zeit unterstützen.

36. Knobelaufgabe #21

Nicht alle Elemente von *Necklace* $\langle 'a \rangle$ sind tatsächlich Perlenketten. Im Allgemeinen haben die Elemente die Form eines Loopings:

```
type Necklace ⟨'a⟩ =
  {mutable bead : 'a
   mutable next : Necklace ⟨'a⟩}
```



Wie lässt sich der Anfang der „Loop“ bestimmen? (Im obigen Beispiel die Perle *d*, ausgehend von *a*.)

Benötigen Sie zusätzliche Sprachfeatures? Kommt Ihr Programm mit konstantem Speicherplatz aus?

36. Motivation

Zur Erinnerung: Arrays können auf zwei Weisen konstruiert werden:

- durch Aufzählung der Elemente: $[|2; 3; 5; 7; 11|]$;
- durch Angabe einer Bildungsvorschrift: $[|\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,99 \to i * i|]$.

☞ Gilt in gleicher Weise für Listen.

☞ $[\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ e_1 \to e_2]$ ist tatsächlich eine Abkürzung für $[\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ e_1\ \textbf{\textit{do yield}}\ e_2]$.

☞ *Array-* bzw. *Listenbeschreibungen* sind sehr viel allgemeiner:

Mini⟩ $[\textbf{\textit{yield}}\ 4711; \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,5\ \textbf{\textit{do yield}}\ i * i]$
$[4711; 0; 1; 4; 9; 16; 25]$

Mini⟩ $[\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,9\ \textbf{\textit{do if}}\ i\,\%\,2 = 1\ \textbf{\textit{then yield}}\ i * i]$
$[1; 9; 25; 49; 81]$

☞ Mit einer einarmigen Alternative können Elemente gefiltert werden.

36. Motivation

☞ Generatoren dürfen geschachtelt werden.

Mini⟩ [*for* i *in* $1..8$ *do*
for j *in* $1..8$ *do*
if $(i+j)\,\%\,2 = 0$ *then*
yield (i,j)]

$[(1,1);(1,3);(1,5);(1,7);(2,2);(2,4);(2,6);(2,8);(3,1);(3,3);(3,5);$
$(3,7);(4,2);(4,4);(4,6);(4,8);(5,1);(5,3);(5,5);(5,7);(6,2);(6,4);$
$(6,6);(6,8);(7,1);(7,3);(7,5);(7,7);(8,2);(8,4);(8,6);(8,8)]$

☞ Positionen aller schwarzen Felder auf einem 8×8-Schachbrett.

36. Abstrakte Syntax

Wir erweitern Ausdrücke um *Listenbeschreibungen* (analog für Arrays).

$e \in \text{Expr} ::=$		*Ausdrücke:*
	$\mid [\, se \,]$	Listenbeschreibung

Innerhalb der Klammern steht ein sogenannter *Sequenzausdruck*, eine neue syntaktische Kategorie.

$se \in \text{SExpr} ::=$		*Sequenzausdrücke:*
	$\mid$ ***yield*** e	Element
	$\mid se_1 ; se_2$	Konkatenation
	$\mid$ ***if*** e ***then*** se	Filter
	$\mid$ ***for*** x ***in*** e ***do*** se	Generator

☞ ***yield*** e ist *kein* Ausdruck.

☞ Statische Semantik, siehe Skript.

36. Dynamische Semantik

☞ Listenbeschreibungen sind „syntaktischer Zucker“: Sie versüßen das Leben beim Programmieren, sind aber nicht lebensnotwendig.

☞ Die Bedeutung der Konstrukte können wir erklären, indem wir sie entzuckern, in uns bekannte Ausdrücke *übersetzen*. Zum Beispiel:

$$[\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,9\ \textbf{\textit{do yield}}\ i*i] \;=\; \mathit{map}\ (\textbf{\textit{fun}}\ i \to i*i)\ [0\,..\,9]$$

$$[\textbf{\textit{if}}\ i\,\%\,2 = 1\ \textbf{\textit{then yield}}\ i*i] = \textbf{\textit{if}}\ i\,\%\,2 = 1\ \textbf{\textit{then}}\ [i*i]\ \textbf{\textit{else}}\ [\,]$$

36. Dynamische Semantik

☞ Die Listenbeschreibung $[se]$ wird in den Ausdruck $[\![se]\!]$ übersetzt, wobei $[\![se]\!]$ wie folgt definiert ist.

$$
\begin{array}{ll}
[\![\textbf{\textit{yield}}\ e]\!] & = [e] \\
[\![se_1; se_2]\!] & = [\![se_1]\!] \mathbin{@} [\![se_2]\!] \\
[\![\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ se]\!] & = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ [\![se]\!]\ \textbf{\textit{else}}\ [\,] \\
[\![\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e\ \textbf{\textit{do}}\ se]\!] & = \mathit{collect}\ (\textbf{\textit{fun}}\ x \to [\![se]\!])\ e
\end{array}
$$

Die Funktion $\mathit{collect} : ('a \to 'b\ \mathit{list}) \to ('a\ \mathit{list} \to 'b\ \mathit{list})$ ist vordefiniert:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{collect}\ f = \textbf{\textit{function}} \\
\quad \mid [\,] \quad \to [\,] \\
\quad \mid x :: xs \to f\ x \mathbin{@} \mathit{collect}\ f\ xs
\end{array}
$$

☞ $\mathit{collect}\ f\ \mathit{list}$ wendet f auf jedes Element von list an und konkateniert die resultierenden Listen.

36. Dynamische Semantik: Beispiel

$$
\begin{aligned}
& [\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ [0\,..\,9]\ \textbf{\textit{do if}}\ i\,\%\,2 = 1\ \textbf{\textit{then yield}}\ i * i] \\
= \quad & \{\ \text{Übersetzung Listenbeschreibung}\ \} \\
& [\![\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ [0\,..\,9]\ \textbf{\textit{do if}}\ i\,\%\,2 = 1\ \textbf{\textit{then yield}}\ i * i]\!] \\
= \quad & \{\ \text{Übersetzung Generator}\ \} \\
& \mathit{collect}\ (\textbf{\textit{fun}}\ i \to [\![\ \textbf{\textit{if}}\ i\,\%\,2 = 1\ \textbf{\textit{then yield}}\ i * i]\!])\ [0\,..\,9] \\
= \quad & \{\ \text{Übersetzung Filter}\ \} \\
& \mathit{collect}\ (\textbf{\textit{fun}}\ i \to \textbf{\textit{if}}\ i\,\%\,2 = 1\ \textbf{\textit{then}}\ [\![\textbf{\textit{yield}}\ i * i]\!]\ \textbf{\textit{else}}\ [])\ [0\,..\,9] \\
= \quad & \{\ \text{Übersetzung}\ \textbf{\textit{yield}}\ \} \\
& \mathit{collect}\ (\textbf{\textit{fun}}\ i \to \textbf{\textit{if}}\ i\,\%\,2 = 1\ \textbf{\textit{then}}\ [i * i]\ \textbf{\textit{else}}\ [])\ [0\,..\,9] \\
= \quad & \{\ \text{Definition von}\ [l\,..\,u]\ \} \\
& \mathit{collect}\ (\textbf{\textit{fun}}\ i \to \textbf{\textit{if}}\ i\,\%\,2 = 1\ \textbf{\textit{then}}\ [i * i]\ \textbf{\textit{else}}\ [])\ [0;1;2;3;4;5;6;7;8;9] \\
= \quad & \{\ \text{Definition von}\ \mathit{collect}\ \} \\
& [] \mathbin{@} [1] \mathbin{@} [] \mathbin{@} [9] \mathbin{@} [] \mathbin{@} [25] \mathbin{@} [] \mathbin{@} [49] \mathbin{@} [] \mathbin{@} [81] \\
= \quad & \{\ \text{Definition von `@'}\ \} \\
& [1;9;25;49;81]
\end{aligned}
$$

36. Vertiefung

☞ Mit Listenbeschreibungen lassen sich bequem „Datenbankanfragen“ formulieren. *Beispiel:* eine Prüfungsdatenbank.

```
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}
```

```
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};...]
```

36. Vertiefung: Datenbanken

Wie heißt der Student mit der Matrikelnummer 4711?

```
let query1 : Name list =
  [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]
```

36. Vertiefung: Datenbanken

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]
```

37. Motivation

☞ Mit der Einführung von Operationen zur Ein- und Ausgabe und zur Speichermanipulation wird es notwendig, Effekte 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?“) weicht
- einem *maschinennahen*, *präskriptiven* Stil („Wie?“).

Das „Wie“, der Kontrollfluss, kann mit speziellen Sprachmitteln, den Kontrollstrukturen, präzisiert werden.

37. Motivation

☞ Eine ***for***-Schleife kann verwendet werden, um effektvolle Ausdrücke aneinanderzureihen:

```
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
```

☞ Kommt uns die Syntax bekannt vor?

37. Motivation

Die Syntax erinnert stark an Sequenzausdrücke; lediglich ***yield*** ist durch *print* ersetzt worden.

```
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)
```

37. Motivation

Es ist 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* summiert die Elemente der angegebenen Liste.

☞ Eine ***for***-Schleife wiederholt einen Effekt für *alle* Elemente einer Liste. Das ist nicht für alle Aufgabenstellungen adäquat ...

37. Motivation

Mit Hilfe einer ***while***-Schleife kann man die lineare Suche *iterativ* implementieren.

```
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
```

☞ Der Rumpf der ***while***-Schleife wird wiederholt, solange die Schleifenbedingung wahr ist.

☞ ***while***-Schleifen sind *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.

37. Motivation

Auch die binäre Suche lässt sich iterativ formulieren:

```
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 Intervallgrenzen l und u und den Suchstatus *found*. Die Schleifenbedingung $l \leqslant u$ && *found* = *None* involviert alle drei Veränderliche.

37. Abstrakte Syntax

Ausdrücke vom Typ *Unit*, die *nur* ihres Effektes wegen ausgerechnet werden, heißen auch *Anweisungen*.

$e ::= \cdots$	*Kontrollstrukturen:*
$\mid\ e_1 ; e_2$	Sequenz
$\mid\ \textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2$	einarmige Alternative
$\mid\ \textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3$	zweiarmige Alternative
$\mid\ \textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1\ \textbf{\textit{do}}\ e_2$	beschränkte Wiederholung
$\mid\ \textbf{\textit{while}}\ e_1\ \textbf{\textit{do}}\ e_2$	unbeschränkte Wiederholung

37. Statische Semantik

Die ***for***-Schleife ist ein „Binder“: Die Schleifenvariable ist im Rumpf der Schleife sichtbar.

$$\frac{\Sigma \vdash e_1 : \mathit{List}\,\langle t_1 \rangle \qquad \Sigma, \{x_1 \mapsto t_1\} \vdash e : \mathit{Unit}}{\Sigma \vdash \textbf{\textit{for}}\ x_1\ \textbf{\textit{in}}\ e_1\ \textbf{\textit{do}}\ e : \mathit{Unit}}$$

$$\frac{\Sigma \vdash e_1 : \mathit{Bool} \qquad \Sigma \vdash e_2 : \mathit{Unit}}{\Sigma \vdash \textbf{\textit{while}}\ e_1\ \textbf{\textit{do}}\ e_2 : \mathit{Unit}}$$

☞ Der Schleifenrumpf ist jeweils eine Anweisung.

37. Dynamische Semantik

Schleifen sind wie Listenbeschreibungen syntaktischer Zucker.

☞ Die Schleife ***for*** x ***in*** e_1 ***do*** e_2 entspricht dem Aufruf *foreach* e_1 (***fun*** $x \to e_2$), wobei *foreach* wie folgt definiert ist.

```
let rec foreach (list : List ⟨'a⟩) (body : 'a → Unit) : Unit =
  match list with
    | []      → ()
    | x :: xs → body x; foreach xs body
```

☞ *foreach* ist *endrekursiv*: der rekursive Aufruf ist die „letzte Aktion“ im Funktionsrumpf.

37. Dynamische Semantik

☞ Die Schleife ***while*** e_1 ***do*** e_2 wird in den Ausdruck *whiledo* (***fun*** $() \to e_1$) (***fun*** $() \to e_2$) übersetzt.

```
let rec whiledo (test : Unit → Bool) (body : Unit → Unit) : Unit =
  if test () then
    body ()
    whiledo test body
```

☞ Warum hat *test* den Typ $Unit \to Bool$?

☞ Auch *whiledo* ist *endrekursiv*.

Verhältnis Rekursion und Iteration:

- eine Schleife kann in eine endrekursive Funktion übersetzt werden; und umgekehrt
- kann eine endrekursive Funktion in eine Schleife übersetzt werden.

☞ Mehr dazu im Skript.

37. Vertiefung

Aufgabe: Sortieren eines Arrays an „Ort und Stelle“ (engl. in place, lat. in situ), das heißt, *ohne* zusätzlichen Speicher zu allokieren.

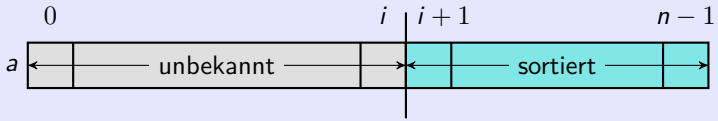
```
Mini⟩ let revenues = [| 815; 4; 7; 1; 1 |]
Mini⟩ selection-sort revenues
()
Mini⟩ revenues
[| 1; 1; 4; 7; 815 |]
```

Die Sortierfunktion hat kein interessantes Ergebnis, aber einen bemerkenswerten Effekt: Nach dem Aufruf ist das Eingabearray sortiert.

☞ 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.

37. Vertiefung: Sortieren durch Einfügen

Idee: Wir teilen das zu sortierende Array gedanklich in zwei Zonen auf.



Über die linke, graue Zone wissen wir nichts; die rechte, grüne Zone ist sortiert. Wir lassen sie schrittweise wachsen, indem wir jeweils ein graues Element in die grüne Zone einfügen.

37. Vertiefung: Sortieren durch Einfügen

```
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
```

☞ $l \,..\, d \,..\, r$ generiert die Elemente l, $l + d$, $l + 2 \cdot d$, ..., r.

37. Vertiefung: Sortieren durch Einfügen

Schleifeninvariante: $a.[i+1] \leqslant a.[i+2] \leqslant \cdots \leqslant a.[n-1]$

- die Initialisierung der Laufvariable etabliert die Invariante:

  $i = n - 2$

- der Schleifenrumpf erhält die Invariante:

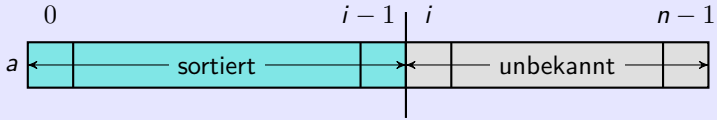
  let $key = a.[i]$
  let mutable $j = i + 1$
  while $j < n$ && $key > a.[j]$ ***do***
  $\quad a.[j-1] \leftarrow a.[j]$
  $\quad j \leftarrow j + 1$
  $a.[j-1] \leftarrow key$

- nach dem letzten Schleifendurchlauf impliziert die Invariante die Nachbedingung:

  $i = -1$

37. Vertiefung: Sortieren durch Auswählen

Idee: Wir teilen das zu sortierende Array gedanklich in zwei Zonen auf.



Die linke, grüne Zone ist sortiert; über die rechte, graue Zone wissen wir nichts. Die grüne Zone wird schrittweise nach rechts ausgedehnt, indem jeweils das Minimum der grauen Zone mit dem ersten Element der grauen Zone ausgetauscht wird.

Schleifeninvariante: Die grüne Zone ist sortiert *und* die Elemente der grünen Zone sind höchstens so groß wie die Elemente der grauen Zone.

37. Vertiefung: Sortieren durch Auswählen

```
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
```

37. Vertiefung: Sortieren durch Zählen

Zur Erinnerung: Jedes Sortierverfahren, das auf dem Vergleichen von Elementen basiert, benötigt im schlechtesten Fall mindestens $n \lg n$ Vergleiche.

☞ Es nicht immer zwingend zu vergleichen; um 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. ☞ Auch Arrays sind ***mutable***:

```
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 |]
```

☞ Die Laufzeit ist proportional zu $\max\{m, n\}$, wobei n die Größe des Eingabearrays ist.

38. Motivation

Nicht jede Rechnung lässt sich einem erfolgreichen Ende zuführen:

$$815 \;/\; (4711 - 7 * 673) + 1$$

☞ Die Auswertungsregel für '/' ist nicht anwendbar.

- Die Teilrechnung $815 \;/\; (4711 - 7 * 673)$ kann nicht weitergeführt werden.
- Daraus folgt aber nicht, dass wir auch die Gesamtrechnung abbrechen sollten.
- *Im Gegenteil:* resultiert der Divisor zum Beispiel aus einer interaktiven Eingabe, dann sollte das Programm die Benutzer*in auf den Fehler hinweisen.
- *Wunsch:* ein allgemeiner Mechanismus, um Ausnahmesituationen zu signalisieren, Rechnungen abzubrechen und an anderer Stelle wieder aufzunehmen.

38. Motivation: Werfen einer Ausnahme

Idee: 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 fortgesetzt.

Bildlicher: eine Ausnahme wird *geworfen* und *gefangen*.

Die Division wirft 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.

38. Motivation: Werfen einer Ausnahme

☞ Eine Ausnahme kann auch explizit mit ***raise*** geworfen werden.

Mini⟩ ***raise*** *Div* $+ 1$
uncaught exception : *Div*

☞ Dieser Ausdruck hat den gleichen Effekt wie $815 \;/\; (4711 - 7 * 673) + 1$.

☞ Da die Rechnung abgebrochen wird, besitzt der Ausdruck ***raise*** *Div* $+ 1$ *keinen* Wert, sondern hat nur einen Effekt.

38. Motivation: Fangen einer Ausnahme

Eine geworfene Ausnahme kann mit dem ***try***-Konstrukt gefangen werden.

```
Mini⟩ try show (815 / (4711 − 6 ∗ 773) + 1) with | Div → "?"
"12"
Mini⟩ try show (815 / (4711 − 7 ∗ 673) + 1) with | Div → "?"
"?"
```

☞ 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 Schlüsselwörtern ***with*** zum Einsatz.

38. Motivation: Fangen einer Ausnahme

☞ Passt die Ausnahme auf die linke Seite einer Regel, so wird mit der Auswertung der rechten Seite fortgefahren.

```
Mini⟩ try show (815 / (4711 − 7 ∗ 673) + 1) with | Div → "?"
"?"
```

☞ Man sagt auch, die Ausnahme wird behandelt.

☞ Passt kein Muster, so wird die Ausnahme weitergeworfen.

```
Mini⟩ try show (815 / (4711 − 7 ∗ 673) + 1) with | Match → "?"
uncaught exception : Div
```

38. Motivation

☞ Ausnahmen sind normale Werte, Elemente des Typs *Exception*, und können wie Elemente eines Variantentyps verwendet werden.

```
Mini⟩ let exns = [|Div; Match|]
val exns : Array ⟨Exception⟩
Mini⟩ match exns.[1] with | Div → "div" | Match → "match"
"div"
```

Zunächst wird ein Array von Ausnahmen konstruiert; anschließend wird das zweite Arrayelement mit einer Fallunterscheidung analysiert.

38. Motivation

Die Ausnahme *Match* signalisiert, dass eine Fallunterscheidung fehlgeschlagen ist, dass keines der angegebenen Muster auf den analysierten Wert gepasst hat.

```
let head (list : List ⟨'a⟩) : 'a =
  match list with
  | x :: _ → x
```

Die Funktion *head* bestimmt das erste Element einer Liste; der Aufruf *head e* wirft die Ausnahme *Match*, wenn *e* zur leeren Liste auswertet. ☞ Mathematisch gesehen ist *head* eine *partielle Funktion*.

```
Mini⟩  head (head ([] :: [4711] :: []))
uncaught exception : Match
Mini⟩  try head (head ([] :: [4711] :: [])) with | Match → 0
0
```

(☞ Dichtung und Wahrheit: In F# wird tatsächlich eine *Microsoft.FSharp.Core.MatchFailureException* geworfen, siehe Skript.)

38. Motivation: Definition von Ausnahmen

☞ Man kann auch eigene Ausnahmen definieren, um Fehlerfälle genauer beschreiben zu können.

```
exception Head

let head (list : List ⟨'a⟩) : 'a =
  match list with
  | []     → raise Head
  | x :: _ → x
```

☞ ***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).

38. Motivation

☞ Ausnahmen können auch ein Argument haben, um Informationen vom Ort des Abbruchs zum Ort der Wiederaufnahme zu schicken.

```
exception Insufficient of Nat

module Account =
  let private funds = ref 0

  let deposit (amount : Nat) =
        funds := !funds + amount

  let withdraw (amount : Nat) =
    if !funds ⩾ amount then
        funds := !funds ∸ amount
    else
        raise (Insufficient (!funds))

  let balance () = !funds
```

☞ Die Funktion *withdraw* bucht den gewünschten Betrag nur ab, wenn das Konto gedeckt ist. Die Ausnahme gibt den maximal abhebbaren Betrag an.

38. Motivation

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
    Account.withdraw amount; amount
  with
  | Insufficient rest → Account.withdraw rest; rest
```

38. Abstrakte Syntax

Wir erweitern Definitionen um Ausnahmedefinitionen.

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***exception*** C ***of*** t	Definition einer Ausnahme

☞ Wird bei einer ***exception*** Definition kein Typargument angegeben, dann fassen wir das, wie bei „normalen“ Konstruktoren, als Abkürzung für C ***of*** $Unit$ auf.

Wir erweitern Ausdrücke um Konstrukte zum Werfen und Fangen von Ausnahmen.

$e ::= \cdots$	*Ausdrücke:*
$\mid$ ***raise*** e	Werfen einer Ausnahme
$\mid$ ***try*** e ***with*** m	Fangen einer Ausnahme

☞ Syntaktisch entspricht ***try*** e ***with*** m einer erweiterten Fallunterscheidung: e ist ein Ausdruck und m eine Folge von Regeln der Form $p \to e$.

38. 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*** Definitionen eingeführt werden.

Typregeln:

exception *C* ***of*** *t*

☞ Eine Ausnahmedefinition wird ähnlich wie eine Variantentypdefinition gehandhabt: der Ausnahmekonstruktor korrespondiert zu einer Funktion vom Typ $t \to$ *Exception*. Wir erlauben es *nicht*, Ausnahmekonstruktoren zu redefinieren oder lokal zu definieren.

38. Statische Semantik

Typregeln:

$$\frac{\Sigma \vdash e : \mathit{Exception}}{\Sigma \vdash \textbf{\textit{raise}}\ e : t}$$

$$\frac{\Sigma \vdash e : t \qquad \Sigma \vdash m\ (\mathit{Exception}) : t}{\Sigma \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m : t}$$

☞ Die Regel m muss, angewendet auf eine Ausnahme, zu einem Element des Typs t auswerten, siehe Skript.

38. Statische Semantik

☞ ***raise*** *e* hat einen beliebigen Typ!

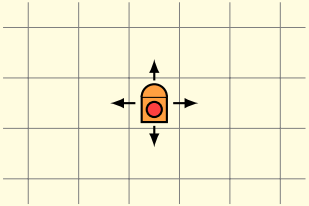
Somit kann ***raise*** *e* überall verwendet werden:

- als Boolescher Wert: ***raise*** *Div* && *e*,
- als natürliche Zahl: ***raise*** *Div* $+ 4711$,
- als Funktion: (***raise*** *Div*) 4711,
- als Paar: *snd* (***raise*** *Div*),
- usw.

☞ ***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.

38. Knobelaufgabe #22

Ein kleiner Roboter wird in einem 2-dimensionalen Gitter ausgesetzt, das sich in alle vier Richtungen unendlich ausdehnt. Der Roboter wird in einem Schritt in eine der vier Himmelrichtungen bewegt; er meldet jeweils zurück, ob er die Zielposition bereits vorher besucht hat.



Zum Beispiel: auf die Eingaben NEESWNNWSS anwortet der Roboter mit NNNNNYNNYY.

type *Direction* $=|$ *N* | *E* | *S* | *W*

robot : *Direction* $\rightarrow$ *Bool*

Programmieren Sie den Roboter so, dass er zu jeder Eingabe die Ausgabe in *konstanter* Zeit ermittelt.

38. Dynamische Semantik

- Die Auswertung verändert sich mit dem Einzug von Effekten.
- Wie müssen wir die Auswertungsregeln modifizieren, um Ausnahmen modellieren zu können?
- Eine Auswertung hat jetzt zwei mögliche Ergebnisse:
  - einen „gewöhnlichen“ Wert wie 4711 oder
  - einen „außergewöhnlichen“ Wert wie *Div*.
- *Idee:* Wir modifizieren die Auswertungsrelation

$$\delta \vdash e \Downarrow r$$

wobei r ein *Resultat* ist, ein Wert oder eine Ausnahme.

38. Dynamische Semantik

Die geworfene Ausnahme nennt man auch *Paket*.

$r \in \text{Result}$	*Resultate*
$r ::= \nu$	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*.

38. Dynamische Semantik

Auswertungsregeln:

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash \textbf{\textit{raise}}\ e \Downarrow \boxed{\nu}} \qquad \frac{\delta \vdash e \Downarrow \boxed{\nu}}{\delta \vdash \textbf{\textit{raise}}\ e \Downarrow \boxed{\nu}}$$

☞ Wenn bei der Auswertung von e bereits ein Paket geworfen wird (!), so wirft ***raise*** dieses Paket weiter.

☞ Werte werden niemals doppelt verpackt: ein Ergebnis der Form $\boxed{\boxed{\nu}}$ gibt es *nicht*.

38. Dynamische Semantik

Auswertungsregeln:

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m \Downarrow \nu}$$

$$\frac{\delta \vdash e \Downarrow \boxed{\nu} \qquad \delta \vdash m\,(\nu) \Downarrow r}{\delta \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m \Downarrow r}$$

$$\frac{\delta \vdash e \Downarrow \boxed{\nu} \qquad \delta \vdash m\,(\nu) \Downarrow \lightning}{\delta \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m \Downarrow \boxed{\nu}}$$

☞ 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 das Paket $\boxed{\nu}$ geworfen („die Auswertung misslingt“), so fängt ***with*** das Paket auf und packt es aus ($\delta \vdash m\,(\nu) \Downarrow r$ beschreibt das „pattern matching“, siehe Skript).

38. Dynamische Semantik

☞ Insgesamt gibt es vier unterschiedliche Konstellationen:

$\mathit{try}\ 4711\ \mathit{with} \mid \mathit{Div} \to 815 \Downarrow 4711$

☞ Die „versuchte“ Auswertung ist erfolgreich.

$\mathit{try}\ 4711\ /\ 0\ \mathit{with} \mid \mathit{Div} \to 815 \Downarrow 815$

☞ Die Auswertung misslingt; die Ausnahme wird gefangen und behandelt; die Ausnahmebehandlung resultiert in einem Wert.

$\mathit{try}\ 4711\ /\ 0\ \mathit{with} \mid \mathit{Div} \to \mathit{raise}\ \mathit{Match} \Downarrow \boxed{\mathit{Match}}$

☞ Die Auswertung misslingt; die Ausnahme wird gefangen und behandelt; die Ausnahmebehandlung resultiert in einem Paket.

$\mathit{try}\ 4711\ /\ 0\ \mathit{with} \mid \mathit{Match} \to 815 \Downarrow \boxed{\mathit{Div}}$

☞ Die Auswertung misslingt; die Ausnahme wird gefangen, aber nicht behandelt; die Ausnahme wird weitergeworfen.

38. Dynamische Semantik

☞ Mit Hilfe von Ausnahmen können wir die fehlenden Auswertungsregeln für die Division usw. nachtragen.

$$\frac{\delta \vdash e_2 \Downarrow 0}{\delta \vdash e_1 \,/\, e_2 \Downarrow \boxed{\mathit{Div}}} \qquad \frac{\delta \vdash e_2 \Downarrow 0}{\delta \vdash e_1 \,\%\, e_2 \Downarrow \boxed{\mathit{Mod}}}$$

$$\frac{\delta \vdash e \Downarrow \nu \qquad \delta \vdash m\,(\nu) \Downarrow \lightning}{\delta \vdash \textbf{\textit{match}}\ e\ \textbf{\textit{with}}\ m \Downarrow \boxed{\mathit{Match}}}$$

$$\frac{\delta \vdash e_1 \Downarrow s \qquad \delta \vdash e_2 \Downarrow i}{\delta \vdash e_1.[e_2] \Downarrow \boxed{\mathit{Subscript}}} \quad i \notin \mathit{dom}\ s$$

38. Dynamische Semantik

☞ Wie immer müssen wir *alle* bisher aufgeführten Auswertungsregeln anpassen.

raise *Div* + ***raise*** *Match*

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 einer der beiden Summanden eine Ausnahme wirft.

38. Dynamische Semantik

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*:

$$\frac{\delta_1 \vdash e_1 \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$

$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \qquad \delta_2 \vdash e_2 \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$

$$\vdots$$

$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \qquad \delta_2 \vdash e_2 \Downarrow \nu_2 \qquad \cdots \qquad \delta_n \vdash e_n \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$

38. Vertiefung: Panik

☞ Nicht immer lässt sich eine abgebrochene Rechnung sinnvoll weiterführen.

```
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. (Warum?)

38. Vertiefung: *panic*

☞ Wenn man möchte, kann man *Panic* Ausnahmen abfangen — am besten mit einem ***try***-Ausdruck um das komplette Programm.

```
try
  ...
with
| Panic msg →
  putline ("panic! (the 'impossible' happened)\n" ^ msg ^
           "\nPlease report it as a bug to" ^
           "support@harry-hacker.org.")
```

38. Vertiefung

☞ 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.

Aufgabe: Produkt einer Liste von Zahlen berechnen. Mit dem Peano Entwurfsmuster erhalten wir die folgende Definition.

```
let product (list : List ⟨Nat⟩) : Nat =
  match list with
    | []       → 1
    | n :: ns → n ∗ product ns
```

☞ Enthält die Liste irgendwo eine Null, dann wird unnötige Arbeit verrichtet.

38. Vertiefung

```
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
```

☞ Mit ***raise*** *Zero* „springen wir aus der Rekursion heraus“.

38. Vertiefung: einfache Parser

☞ 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
  in
    if s = "" then raise Read else nat (List.rev (explode s))
```

☞ Diese Version beklagt sich, wenn der String leer ist oder wenn andere Zeichen als Ziffern vorkommen.

38. Vertiefung: Validatoren

☞ Diese Version von *read-nat* können wir zum Beispiel verwenden, um die Funktion *is-nat* etwas kürzer zu definieren.

```
let is-nat (s : String) : Result ⟨Nat⟩ =
  try
    Okay (read-nat s)
  with
  | Read → Error "natural number expected"
```

38. Vertiefung: *Option* und *Result*

☞ Die Datentypen *Option* und *Result* dienen dem gleichen Zweck wie Ausnahmen.

Option	*Result*	Ausnahmen	
Some e	*Okay* e	e	ν
None	*Error* e	***raise*** e	$\boxed{\nu}$
match $\ldots$ ***with***	***match*** $\ldots$ ***with***	***try*** $\ldots$ ***with***	

☞ Mit Hilfe von *Option* und *Result* können/müssen wir das Werfen und Fangen von Ausnahmen selbst programmieren!

38. Vertiefung: Eingabe mit Validierung

☞ Reimplementieren wir *checked-query* mit Hilfe von Ausnahmen.

Ohne Ausnahmen: Ein Validator hat den Typ $t_1 \to \mathit{Result}\,\langle t_2 \rangle$.

Mit Ausnahmen: Ein Validator hat den Typ $t_1 \to t_2$. Eine fehlerhafte Eingabe wird jetzt durch das Werfen einer Ausnahme signalisiert.

```
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.

38. Vertiefung: Eingabe mit Validierung

☞ Die Validatoren haben einen einfacheren Typ: sie geben entweder den semantischen Wert zurück oder werfen eine *Panic* 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)
```

38. Vertiefung: Eingabe mit Validierung

☞ Die Funktion *both* ist jetzt schlicht und einfach die (Vorwärts-) Komposition von Funktionen.

$$\begin{array}{l}\textbf{\textit{let}}\ \mathit{both}\ (\mathit{first} : {'a} \to {'b}, \mathit{second} : {'b} \to {'c}) : {'a} \to {'c} = \\ \quad \textbf{\textit{fun}}\ x \to \mathit{second}\ (\mathit{first}\ x)\end{array}$$

38. Vertiefung

☞ Für den Anwender von *checked-query* ändert sich sich nichts; die Aufrufe funktionieren unverändert:

```
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
```

38. Vertiefung: *Option* und *Result*

Vorteile von Option und Result:

- Der Typ von Ausdrücken und Funktionen macht explizit, wer „Ausnahmen wirft“ und wer nicht.
- Man kann nicht vergessen, Ausnahmen zu behandeln.
- Man kommt nicht in Versuchung, Ausnahmen zu fangen, die gar nicht geworfen werden.

Nachteil von Option und Result:

- Die Auswertungsregeln für Ausnahmen müssen im Prinzip nachprogrammiert werden.

38. Vertiefung: *Option* und *Result*

Mit Ausnahmen:

$e_1 + e_2$

Mit Result:

```
match e1 with
| Okay n1 → match e2 with
             | Okay n2    → Okay (n1 + n2)
             | Error msg → Error msg
| Error msg → Error msg
end
```

☞ Jeder der drei Zweige korrespondiert zu einer Auswertungsregel.

38. Vertiefung: Taschenrechner

Aufgabe: Programmierung eines interaktiven Taschenrechners.

Variante 1: ein UPN-Rechner. *UPN* steht für Umgekehrte Polnische Notation, ein Synonym für Postfixnotation.

```
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 $4711 + 815 * 2765$.

38. Vertiefung: UPN-Rechner

☞ Ausdrücke in Postfixnotation lassen sich besonders leicht ausrechnen: Wir merken uns die eingegebenen Zahlen; jede Operation ersetzt die letzten beiden Zahlen durch das Ergebnis.

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.

38. Vertiefung: UPN-Rechner

☞ Einen Stapel 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
...
```

38. Vertiefung: UPN-Rechner

```
...
  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 Speicherzelle, auf der die Funktionen arbeiten, ist gekapselt, also nur lokal sichtbar.

38. Vertiefung: UPN-Rechner

☞ Der UPN-Rechner implementiert eine einfache Kommandozeile.

```
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 ()
```

38. Vertiefung: Mini^2-F# Rechner

Variante 2: ein Mini^2-F# Rechner, der Infixnotation unterstützt.

Mini$^2\rangle$ $4711 + 815 * 2765$
2258186
Mini$^2\rangle$ $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$
55
Mini$^2\rangle$ $1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10$
3628800
Mini$^2\rangle$ *Hello*, *world* !
lexical error : *illegal character*
Mini$^2\rangle$ $4711\ 815\ 2765 * +$
syntax error

38. Vertiefung: Mini^2-F# Rechner

```
let rec read-eval-print-loop () =
  try
    let s = query "Mini-BPL > "
    if s = "" then
      ()
    elif contains s ["q"; "quit"; "halt"; "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 ()
```

☞ Typisch für einen Kommandozeileninterpreter (REPL) ist der rekursive Aufruf am Ende.

38. Vertiefung: Mini^2-F# Rechner

```
let mini² () =
  try
    read-eval-print-loop ()
  with
  | EOF → putline "bye bye"
```

☞ Signalisiert die Benutzer*in das Ende der Eingabe, dann wirft *query* eine *EOF* Ausnahme (end of file).

38. Paradigmen: imperative Programmierung

imperare
1. befehlen, gebieten, anordnen;
2. auferlegen, zu liefern befehlen;
3. herrschen, den Oberbefehl haben.

C is quirky, flawed, and an enormous success.
— Dennis M. Ritchie

B can be thought of C without types; more accurately,
it is BCPL squeezed into 8K bytes of memory and
filtered through Thompson's brain.
— Dennis M. Ritchie

38. Lösung Knobelaufgabe #17

Welche Sprache bezeichnet der folgende reguläre Ausdruck?

$$\mathit{riddle} = (\mathtt{a}\ \mathtt{a} \mid \mathtt{b}\ \mathtt{b})^*\ ((\mathtt{a}\ \mathtt{b} \mid \mathtt{b}\ \mathtt{a})\ (\mathtt{a}\ \mathtt{a} \mid \mathtt{b}\ \mathtt{b})^*\ (\mathtt{a}\ \mathtt{b} \mid \mathtt{b}\ \mathtt{a})\ (\mathtt{a}\ \mathtt{a} \mid \mathtt{b}\ \mathtt{b})^*)^*$$

Beobachtung: jedes abgeleitete Wort hat eine gerade Länge.

Beobachtung: der Ausdruck *riddle* hat die gleiche Struktur wie

$$\mathtt{0}^*\ (\mathtt{1}\ \mathtt{0}^*\ \mathtt{1}\ \mathtt{0}^*)^*$$

☞ Der Ausdruck bezeichnet die Sprache aller Wörter mit einer geraden Anzahl von Einsen und beliebig vielen Nullen.

38. Lösung Knobelaufgabe #17

$$riddle = (\mathtt{a\ a} \mid \mathtt{b\ b})^* \ ((\mathtt{a\ b} \mid \mathtt{b\ a})\ (\mathtt{a\ a} \mid \mathtt{b\ b})^*\ (\mathtt{a\ b} \mid \mathtt{b\ a})\ (\mathtt{a\ a} \mid \mathtt{b\ b})^*)^*$$

bezeichnet die Sprache aller Wörter mit einer geraden Anzahl von Wörter der Form a b | b a und beliebig vielen Wörtern der Form a a | b b.

☞ *Also:* eine gerade Anzahl von Wörtern, die eine ungerade Anzahl as *und* eine ungerade Anzahl bs enthalten, und eine beliebige Anzahl von Wörtern, die eine gerade Anzahl as *und* eine gerade Anzahl bs enthalten.

☞ *Mit anderen Worten:* die Wörter enthalten eine gerade Anzahl as *und* eine gerade Anzahl bs.

☞ *riddle* / a enthält entsprechend eine ungerade Anzahl as *und* eine gerade Anzahl bs; *riddle* / b enthält eine gerade Anzahl as *und* eine ungerade Anzahl bs.

Teil VIII

Objekte

38. Knobelaufgabe #23

Harry Hacker hat programmiert; was sein Programm wohl macht?

```
let mutable cur = root
while cur <> null do
  if cur.Left = null then
    accept cur.Key
    cur ← cur.Right
  else
    let mutable pre = cur.Left;
    while pre.Right <> null && pre.Right <> cur do
      pre ← pre.Right;
    if pre.Right = null then
      pre.Right ← cur
      cur ← cur.Left
    else
      pre.Right ← null
      accept cur.Key
      cur ← cur.Right
```

Lisa Lista vermutet, dass es um Binärbäume geht ...

38. Gliederung

38. Überblick

Dieses Kapitel ist dem Studium von Objekten gewidmet.

Bisher: Programmierung im Kleinen.

Jetzt: Programmierung im Großen.

☞ Es gibt im wesentlichen zwei Mechanismen, um größere Programme oder Softwaresysteme zu strukturieren:

- *Modulsysteme* und
- *Objektsysteme*.

Das Modulsystem von Mini–F# ist einfach gestrickt: Verwaltung von Namensräumen. Objektsysteme können auf Prototypen oder auf Klassen basieren. Beide Ansätze schauen wir uns im folgenden an ...

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

38. Charakteristika — Fortsetzung

Was macht eine objektorientierte Sprache aus?

- *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). Manchmal sagt man auch, dem Objekt wird ein Nachricht geschickt.
- *Kapselung*: Die interne Repräsentation eines Objekts ist außen nicht sichtbar und kann somit lokal geändert werden.

38. Charakteristika — Fortsetzung

- *Untertypen* (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* (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.

39. Exkurs

☞ Der Aufruf *monus* (e_1, e_2) wirft eine Ausnahme, wenn e_2 größer ist als e_1.

exception *Insufficient* ***of*** *Nat*

let *monus* $(n_1 : Nat, n_2 : Nat) : Nat =$
  if $n_1 \geqslant n_2$ ***then***
    $n_1 \dot{-} n_2$
  else
    raise $(Insufficient\ n_1)$

39. Kapselung

Zur Erinnerung: Kapselung eines Zustandes.

module *Account* $=$
  let mutable private *funds* $= 0$

  let *deposit* $(amount : Nat) =$
    $funds \leftarrow funds + amount$

  let *withdraw* $(amount : Nat) =$
    $funds \leftarrow monus\ (funds, amount)$

  let *balance* $() = funds$

☞ Die Repräsentation eines Bankkontos, die Speicherzelle *funds*, ist nach außen nicht sichtbar.

☞ *Beobachtung: deposit*, *withdraw* und *balance* gehören konzeptionell zusammen, sind aber nur lose gekoppelt.

39. Schnittstellen

Wunsch: Mehrere Operationen zu einer Einheit zusammenfassen.

```
type IAccount =
  interface
    abstract member Deposit  : Nat → Unit
    abstract member Withdraw : Nat → Unit
    abstract member Balance  : Nat
  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; *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.

39. Jargon: Objekte, Methoden und Eigenschaften

Eine Schnittstelle heißt auch Objekttyp; Elemente eines Objekttyps heißen entsprechend *Objekte*.

Funktionen werden zu *Methoden* (engl. methods), andere Werte zu *Eigenschaften* (engl. properties):

- *Deposit* und *Withdraw* sind Methoden;
- *Balance* ist eine Eigenschaft.

39. Objekte

Ein Bankkonto:

```
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 =
        funds
  }
```

☞ {*new IAccount with* ...} ist ein Objektausdruck; er erzeugt ein Element des Typs *IAccount*, ein Objekt. Die Methoden und Eigenschaften der Schnittstelle werden implementiert. Das Objekt hat einen internen Zustand, die Speicherzelle *funds*, um den aktuellen Kontostand zu repräsentieren.

39. Aufruf von Methoden

☞ Zugriff auf die Methoden und Eigenschaften mit der Punktnotation.

```
Mini⟩ lisas.Deposit 4711
()
Mini⟩ lisas.Withdraw 815
()
Mini⟩ lisas.Withdraw 2765
()
Mini⟩ lisas.Balance
1131
```

Jargon: 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 von Nachrichten.

39. Objekte

Eine alternative Implementierung eines Bankkontos:

```
let ludwigs : IAccount =
  let size          = 10
  let history       = [| for k in 0 .. size − 1 → 0 |]
  let mutable i     = 0
  let next ()       = (i + 1) % size
  {
    new 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]
  }
```

39. Demo

```
Mini⟩ lisas.Balance
1131
Mini⟩ ludwigs.Deposit 4711
()
Mini⟩ lisas.Withdraw 1000; ludwigs.Deposit 1000
()
Mini⟩ lisas.Balance
131
Mini⟩ ludwigs.Balance
5711
```

☞ Die beiden Objekte verstehen 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 (dynamic dispatch).

39. Objekte und Schnittstellen

☞ Objekte sind normale Werte; sie können insbesondere Argument oder Ergebnis von Funktionen sein.

let *transfer* $(account_1 : IAccount, amount : Nat, account_2 : IAccount) =$
  $account_1.Withdraw\ amount;$
  $account_2.Deposit\ amount$

Wir können eine Überweisung tätigen, auch wenn die beteiligten Objekte unterschiedlich implementiert sind.

```
Mini⟩ (lisas.Balance, ludwigs.Balance)
(131, 5711)
Mini⟩ transfer (ludwigs, 815, lisas)
()
Mini⟩ (lisas.Balance, ludwigs.Balance)
(946, 4896)
```

(☞ Good SE practice: programming against an interface.)

39. „Objektfabriken“

☞ Von einem Bankkonto zu vielen Bankkonten, sprich einer Bank:

```
module TrustMe =
  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 =
          funds
    }
```

☞ die Funktion *TrustMe.account* heißt auch *Konstruktor*, da sie Objekte konstruiert.

39. Module und Objekte

Module und Objekte können sinnvoll kombiniert werden.

```
module TrustMe =
  do putline "TrustMe is founded."
  let BIC = 4711    // Bank Identifier Code
  let mutable private no = 0
  let no-of-accounts () = no
  let account (seed : Nat) : IAccount =
    do no ← no + 1
    let mutable funds = seed
    {
      new IAccount with ...
    }
```

Modul: bankspezifische, kontoübergreifende Operationen (*BIC*, Gesamtzahl der Konten: *no*).
Objekt: kontospezifische Operationen (*Deposit* etc).

39. Delegation

Aufgaben können an andere Objekte delegiert werden (Komposition):

```
module TrustMe =
  ...
  exception 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
    }
```

☞ Die Nachrichten *Deposit*, *Withdraw* und *Balance* werden an das Basisobjekt *basic* delegiert.

39. Demo

```
Mini⟩ TrustMe.BIC
val it : Nat = 4711
Mini⟩ let herberts = TrustMe.account 8150
Mini⟩ let harrys = TrustMe.student-account 0
Mini⟩ TrustMe.no-of-accounts ()
val it : Nat = 2
Mini⟩ transfer (herberts, 2000, harrys)
Mini⟩ transfer (harrys, 2000, herberts)
uncaught exception : Limit
```

39. Abstrakte Datentypen versus Objekte

Die Zusammenfassung von Operationen zu einer Schnittstelle kennen wir von den abstrakten Datentypen (Teil V).

- Dort: Realisierung einer Schnittstelle mit Hilfe des Modulsystems.
- Hier: Realisierung einer Schnittstelle mit Hilfe des Objektsystems.

☞ Ein ADT kann auch mit dem Objektsystem realisiert werden.

Unterschiede:

- Verschiedene Implementierungen eines ADTs zeigen exakt das gleiche Verhalten; sie sind austauschbar.
- (Modulsystem: eine Schnittstelle wird zu einem Zeitpunkt durch *eine* Implementierung realisiert.)
- Objekte, die die gleiche Schnittstelle unterstützen, zeigen ein verwandtes, nicht aber notwendigerweise das gleiche Verhalten (Standardkonto versus Studierendenkonto).
- (Objektsystem: eine Schnittstelle kann zu einem Zeitpunkt durch *mehrere* Objekte realisiert werden.)

39. Abstrakte Syntax

☞ Der Übersichtlichkeit halber formalisieren wir nur Schnittstellen mit genau 2 Methoden.

Wir erweitern Typen um Schnittstellentypen (engl.interface types).

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***type*** $T =$	Schnittstellentypdefinition ($\ell_1 \neq \ell_2$)
interface	
abstract member $\ell_1 : t_1$	
abstract member $\ell_2 : t_2$	
end	

39. Abstrakte Syntax

Wir erweitern Ausdrücke um Objektausdrücke (engl. object expressions) und Methodenaufrufe (engl. method invocations).

$e ::= \cdots$	*Ausdrücke:*
$\mid \{\textbf{\textit{new}}\ T\ \textbf{\textit{with}}$	Objektausdruck ($\ell_1 \neq \ell_2$)
$\quad \textbf{\textit{member}}\ s_1.\ell_1 = e_1$	
$\quad \textbf{\textit{member}}\ s_2.\ell_2 = e_2$	
$\}$	
$\mid e.\ell$	Methodenaufruf

☞ $\{\textbf{\textit{new}}\ T\ \textbf{\textit{with}} \ldots\}$ ist ein anonymes Objekt; $e.\ell$ ist ein Methodenaufruf: dem Objekt e wird die Nachricht ℓ geschickt.

☞ s_1 bzw. s_2 ist ein beliebiger Bezeichner, der den Empfänger der Nachricht repräsentiert; typische Namen: *self* oder *this*.

☞ Ein Objekt ist eine Sammlung von Methoden; die Methoden legen fest, was mit einem Objekt gemacht werden kann. Ein Objekt ist die Summe seines Verhaltens.

39. Statische Semantik — Objekte

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.
Typregeln:

$$
\frac{\Sigma, \{ s_1 \mapsto T \} \vdash e_1 : t_1 \qquad \Sigma, \{ s_2 \mapsto T \} \vdash e_2 : t_2}{\begin{array}{l}\Sigma \vdash \{\ \textbf{\textit{new}}\ T\ \textbf{\textit{with}} \\ \qquad \textbf{\textit{member}}\ s_1.\ell_1 = e_1 \\ \qquad \textbf{\textit{member}}\ s_2.\ell_2 = e_2 \} : T\end{array}}
$$

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

39. Dynamische Semantik — Objekte

Wir erweitern Werte um Methodentabellen und Methodenabschlüsse.

$\mu \in \mathsf{Lab} \to_{\mathrm{fin}} \mathsf{Val}$	*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 Labels auf Werte ab. Ein Methodenabschluss entspricht einem rekursiven Funktionsabschluss *ohne* Parameter.

39. Dynamische Semantik — Objekte

Auswertungsregeln (dynamische Bindung):

$$\delta \vdash \{\ \textbf{\textit{new}}\ T\ \textbf{\textit{with}}\ \textbf{\textit{member}}\ s_1.\ell_1 = e_1\ \textbf{\textit{member}}\ s_2.\ell_2 = e_2\} \Downarrow \{\ell_1 \mapsto \langle\!\langle \delta,\ s_1,\ e_1 \rangle\!\rangle, \ell_2 \mapsto \langle\!\langle \delta,\ s_2,\ e_2 \rangle\!\rangle\}$$

☞ Ein Objekt wertet im Wesentlichen zu sich selbst aus.

$$\frac{\delta \vdash e \Downarrow \mu \qquad \delta', \{s_i \mapsto \mu\} \vdash e_i \Downarrow \nu_i}{\delta \vdash e.\ell_i \Downarrow \nu_i} \qquad \text{mit } \mu(\ell_i) = \langle\!\langle \delta',\ s_i,\ e_i \rangle\!\rangle$$

☞ Die statische Semantik stellt sicher, dass $\ell \in \mathit{dom}\ \mu$. Das Label ℓ wird zur Laufzeit in der zu dem Objekt e gehörigen Methodentabelle nachgeschlagen (dynamic dispatch). Der „*self*-Bezeichner“ s_i wird an das Objekt selbst (engl. self) gebunden.

39. Knobelaufgabe #15 — da capo

Erinnern wir uns an die Lösung der Knobelaufgabe #15:

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

☞ Rekursion wird durch Selbstapplikation simuliert.

39. Knobelaufgabe #15 — da capo

☞ Genau das Gleiche passiert bei einem Methodenaufruf!

```
type Factorial =
  interface
    abstract member factorial : Nat → Nat
  end
let fac =
  {new Factorial with
    member self.factorial (n : Nat) : Nat =
      if n = 0 then 1
               else self.factorial (n − 1) ∗ n
  }
let factorial (n : Nat) : Nat =
  fac.factorial n
```

39. „Getter“ und „Setter“

```
type IAccount =
  ...
  abstract member Balance : Nat

let lisas : IAccount = ...
  { new IAccount with ...
      member self.Balance = funds
  }
```

Die Definition der Eigenschaft *Balance* ist eine Abkürzung für:

```
type IAccount =
  ...
  abstract member Balance : Nat with get

let lisas : IAccount = ...
  { new IAccount with ...
      member self.Balance with get () = funds
  }
```

39. „Getter“ und „Setter“: Sichten

```
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
  }
```

39. „Getter“ und „Setter“: Demo

```
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
```

☞ Die Eigenschaft *Name* kombiniert „Rohdaten“; sie definiert eine Sicht auf die Daten.

☞ Die Zuweisung *dolittle.Age* ← 2 *erhöht* das Alter um den angegebenen Wert (C: `dolittle.Age += 2`.)

39. Vertiefung: Ausdrücke

☞ Objekte müssen nicht notwendigerweise einen internen Zustand haben.

☞ Alternative Darstellung von Ausdrücken:

```
type IExpr =
  interface
    abstract member Value : Nat
    abstract member Show : String
  end
```

☞ Was kann/will ich mit einem Ausdruck machen? Auswerten und in einen String überführen!

☞ Ein Objekt ist die Summe seines Verhaltens.

39. Demo

```
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)))"
```

39. Vertiefung: Ausdrücke

```
let constant (n : Nat) : IExpr =
  { new IExpr with
      member self.Value = n
      member self.Show = show n
  }
let add (expr_1 : IExpr, expr_2 : IExpr) : IExpr =
  { new IExpr with
      member self.Value = expr_1.Value + expr_2.Value
      member self.Show = "(" ^ expr_1.Show ^ " + " ^ expr_2.Show ^ ")"
  }
let mul (expr_1 : IExpr, expr_2 : IExpr) : IExpr =
  { new IExpr with
      member self.Value = expr_1.Value * expr_2.Value
      member self.Show = "(" ^ expr_1.Show ^ " * " ^ expr_2.Show ^ ")"
  }
```

☞ *constant*, *add* und *mul* heißen auch *Konstruktoren*, da sie Objekte konstruieren (so wie Datenkonstruktoren Daten konstruieren).

39. Variantentypen versus Objekttypen

Objekte definieren sich durch das Verhalten: was kann mit dem Objekt gemacht werden (Idee des abstrakten Datentyps).

Im Gegensatz zu Varianten- und Recordtypen, die durch die Angabe der Elemente definiert werden (konkreter Datentyp).

Variantentyp: Wie ist ein Ausdruck aufgebaut?

- Einfach: neue Funktionalität hinzufügen. Es wird einfach eine neue Funktion definiert.
- Schwierig: neue Datenkonstruktoren hinzufügen. *Jede* bestehende Funktion muss erweitert werden.

Objekte: Was mache ich mit einem Ausdruck?

- Einfach: neue Objektkonstruktoren hinzufügen. Es wird einfach eine neue Konstruktorfunktion definiert.
- Schwierig: neue Funktionalität hinzufügen. *Jede* bestehende Konstruktorfunktion muss erweitert werden.

39. Generische Schnittstellen

☞ Wie Datentypen können auch Schnittstellen mit einem oder mehreren Typen parametrisiert werden.

```
type IStack ⟨'elem⟩ =
  interface
    abstract member Push : 'elem → Unit
    abstract member Pop  : Unit → 'elem
    abstract member Top  : 'elem
  end
```

☞ Stacks können Elemente beliebigen Typs enthalten; jeder einzelne Stack ist aber homogen: die Elemente besitzen den gleichen Typ.

39. Polymorphe Objektkonstruktoren

```
exception Pop
exception Top

let stack ⟨'elem⟩ () =
  let mutable stack = []
  {
    new IStack ⟨'elem⟩ with
      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
  }
```

39. UPN-Taschenrechner Deluxe

```
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"
```

40. Knobelaufgabe #24

Die Queen feiert in Kürze ihren 100. Geburtstag. Aus Anlass des runden Jubiläums plant sie, ihren gutsortierten Weinkeller zu öffnen und die edlen Tropfen ihren Gästen zu kredenzen.

Drei Wochen vor den Feierlichkeiten teilt ihr der Geheimdienst mit, dass eine ihrer rund tausend Weinflaschen vergiftet wurde — man wisse, aber nicht welche. Aber man kenne das Gift: Ein winziger Schluck führt zum Tode; allerdings nicht unmittelbar, sondern mit einer Verzögerung von zwei Wochen.

Die Queen ist bereit, Opfer zu bringen und einige ihrer Diener zum Probetrinken abzustellen. Für die Vorbereitung der Feierlichkeiten wird aber eigentlich das gesamte Personal benötigt. Wieviele Diener muss die Queen höchstens abstellen, um die vergiftete Weinflasche zu identifizieren? Tausend Diener reichen sicherlich aus, aber kommt sie auch mit weniger aus?

40. Motivation

Zusätzliche Funktionalität: Kontoauszug.

```
type IAccountPlus =
  interface
    inherit IAccount
    abstract member Statement : Array ⟨Nat⟩
  end
```

☞ *IAccountPlus* bietet zusätzlich die Möglichkeit an, einen Kontoauszug anzufordern.

Lies: die Schnittstelle *IAccountPlus* *erweitert* die Schnittstelle *IAccount*. (Das Schlüsselwort ***inherit*** ist etwas unglücklich gewählt.)

40. Motivation

Das Objekt *ludwigs* lässt sich zu einem Plus-Konto erweitern.

```
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] |]
  }
```

40. Motivation

Beobachtung: Das Objekt *ludwigs* kann nicht an die Funktion *transfer* übergeben werden!

transfer : $\mathit{IAccount} * \mathit{Nat} * \mathit{IAccount} \to \mathit{Unit}$

ludwigs : $\mathit{IAccountPlus}$

☞ Der Typ von $\mathit{IAccountPlus}$ ist nicht *gleich* dem Typ $\mathit{IAccount}$.

40. Motivation

Beobachtung: Der Aufruf

$$\mathit{transfer}\ (\mathit{lisas}, 1000, \mathit{ludwigs})$$

kann aber problemlos ausgerechnet werden, da *ludwigs* mehr Funktionalität als gefordert anbietet — *ludwigs* ist sozusagen überqualifiziert.

☞ Das Typsystem von Mini–F# kann mit Hilfe von *Untertypen* flexibler gemacht werden.

40. Motivation

Idee: ein Element eines Untertyps kann überall da verwendet werden, wo ein Element eines Obertyps verlangt wird. Wir führen eine Untertypbeziehung auf Typen ein.

Zum Beispiel: die Schnittstelle *IAccountPlus* erweitert die Schnittstelle *IAccount* (Schlüsselwort ***inherit***).

$$\mathit{IAccountPlus} \preccurlyeq \mathit{IAccount}$$

Mit Hilfe einer sogenannten Typanpassung (engl. cast) lässt sich ein Element eines Untertyps in ein Element eines Obertyps überführen.

$$\mathit{transfer}\ (\mathit{lisas}, 1000, \mathit{ludwigs} :> \mathit{IAccount})$$

40. Motivation

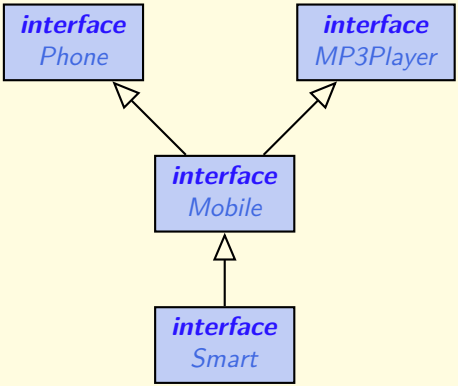
Mehrfachvererbung ist möglich:

```
type Phone =
  abstract member Dial    : Nat → Unit
type MP3Player =
  abstract member Play    : String → Unit
type Mobile =
  inherit Phone
  inherit MP3Player
  abstract member Lookup : String → Nat
  abstract member Call    : String → Unit
type Smart =
  inherit Mobile
  abstract member Browse : String → Unit
```

(Die ***interface*** ... ***end*** Klammer kann weggelassen werden.)

40. Motivation

Schnittstellen als Diagramm:



☞ Bestandteil der „Unified Modeling Language“ (UML), einer Modellierungssprache für Software.

40. Abstrakte Syntax

Deklarationen:

$d ::= \cdots$	*Deklarationen:*
$\quad \mid$ ***type*** $U =$	erweiterte Schnittstellentypdefinition
$\quad\quad$ ***interface***	
$\quad\quad\quad$ ***inherit*** T_1	
$\quad\quad\quad$ ***inherit*** T_2	
$\quad\quad\quad$ ***abstract member*** $\ell : t$	
$\quad\quad$ ***end***	

☞ Es dürfen beliebig viele ***inherit*** Klauseln aufgeführt werden und beliebig viele neue Mitglieder hinzukommen.

$e ::= \cdots$	*Ausdrücke:*
$\quad \mid e :> t$	Typumwandlung$\diagdown$Typeinschränkung

40. Statische Semantik

Typregel:

$$\frac{\Sigma \vdash e : t' \qquad t' \preccurlyeq t}{\Sigma \vdash (e :> t) : t}$$

☞ Informationsverlust: das Verhalten wird eingeschränkt. (Ein Objekt ist die Summe seines Verhaltens.)

40. Statische Semantik

Gute Nachricht: Typanpassungen werden auch automatisch vorgenommen (engl. automatic upcast).

$$\frac{\Sigma \vdash e : t \qquad t \preccurlyeq \tau'}{\Sigma \vdash e : \tau'}$$

Schlechte Nachricht: die automatische Typanpassung gelingt nicht immer. Problemzonen: Zweige einer Alternative.

☞ Die Relation '$\preccurlyeq$' muss mit Leben gefüllt werden.

40. Untertypen: Quasiordnung

Die Untertypbeziehung ist *reflexiv* und *transitiv*.

$$\frac{}{t \preccurlyeq t}$$

$$\frac{t_1 \preccurlyeq t_2 \qquad t_2 \preccurlyeq t_3}{t_1 \preccurlyeq t_3}$$

☞ '$\preccurlyeq$' ist eine sogenannte *Quasiordnung*.

40. Untertypen: *inherit*

Eine Typdefinition mit ***inherit*** Klauseln

type $U =$
  interface
    inherit T_1
    inherit T_2
    abstract member $\ell : t$
  end

legt *explizit* die folgenden Beziehungen fest (Axiome):

$$\frac{}{U \preccurlyeq T_1} \qquad \frac{}{U \preccurlyeq T_2}$$

40. Untertypen: Paare

Wie ist '$\preccurlyeq$' auf strukturierten Typen wie zum Beispiel dem Paartyp $t_1 * t_2$ definiert?

$$\frac{t_1 \preccurlyeq t_1' \qquad t_2 \preccurlyeq t_2'}{t_1 * t_2 \preccurlyeq t_1' * t_2'}$$

☞ Warum ist das eine sinnvolle Festlegung?

40. Untertypen: Paare

Welche Operationen werden auf Paare angewendet? Die Projektionsfunktionen *fst* und *snd*.

Die Untertypregel für Paare lässt sich herleiten, wenn man die Typregeln für *fst* und *snd* studiert.

$$\dfrac{\dfrac{\dfrac{\dots}{\Sigma \vdash e : t_1 * t_2}}{\Sigma \vdash \mathit{fst}\ e : t_1} \qquad \dfrac{}{t_1 \preccurlyeq t_1'}}{\Sigma \vdash \mathit{fst}\ e : t_1'} \qquad \dfrac{\dfrac{\dfrac{\dots}{\Sigma \vdash e : t_1 * t_2}}{\Sigma \vdash \mathit{snd}\ e : t_2} \qquad \dfrac{}{t_2 \preccurlyeq t_2'}}{\Sigma \vdash \mathit{snd}\ e : t_2'}$$

Die Paarregel erlaubt es alternativ, direkt das Paar anzupassen:

$$\dfrac{\dfrac{\dfrac{\dots}{\Sigma \vdash e : t_1 * t_2} \qquad \dfrac{}{t_1 * t_2 \preccurlyeq t_1' * t_2'}}{\Sigma \vdash e : t_1' * t_2'}}{\Sigma \vdash \mathit{fst}\ e : t_1'} \qquad \dfrac{\dfrac{\dfrac{\dots}{\Sigma \vdash e : t_1 * t_2} \qquad \dfrac{}{t_1 * t_2 \preccurlyeq t_1' * t_2'}}{\Sigma \vdash e : t_1' * t_2'}}{\Sigma \vdash \mathit{snd}\ e : t_2'}$$

40. Untertypen: Funktionen

$$\frac{t_1' \preccurlyeq t_1 \qquad t_2 \preccurlyeq t_2'}{t_1 \to t_2 \preccurlyeq t_1' \to t_2'}$$

Der Funktionstyp ist

- *kontravariant* im Argumenttyp ($t_1' \preccurlyeq t_1$) und
- *kovariant* im Ergebnistyp ($t_2 \preccurlyeq t_2'$).

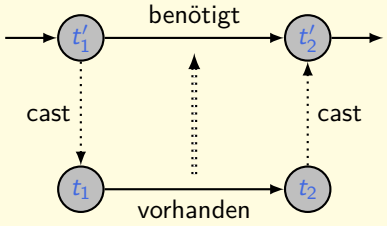
40. Untertypen: Funktionen

Die Varianz des Funktionstyps lässt sich herleiten, wenn man die Typregel der Funktionsapplikation studiert.

$$\dfrac{\dfrac{}{\Sigma \vdash e : t_1 \to t_2} \qquad \dfrac{\dfrac{}{\Sigma \vdash e_1 : t_1'} \qquad \dfrac{}{t_1' \preccurlyeq t_1}}{\Sigma \vdash e_1 : t_1}}{\dfrac{\Sigma \vdash e\ e_1 : t_2 \qquad \dfrac{}{t_2 \preccurlyeq t_2'}}{\Sigma \vdash e\ e_1 : t_2'}}$$

40. Untertypen: Funktionen

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.

40. Untertypen: Funktionen – Beispiele

$$
\begin{gathered}
\mathit{Smart} \to \mathit{Smart} \preccurlyeq \mathit{Smart} \to \mathit{Phone} \\
\mathit{Phone} \to \mathit{Smart} \preccurlyeq \mathit{Smart} \to \mathit{Smart} \\
\mathit{Phone} \to \mathit{Smart} \preccurlyeq \mathit{Smart} \to \mathit{Phone} \\
(\mathit{Smart} \to \mathit{Phone}) \to \mathit{Smart} \preccurlyeq (\mathit{Phone} \to \mathit{Smart}) \to \mathit{Phone} \\
((\mathit{Phone} \to \mathit{Smart}) \to \mathit{Phone}) \to \mathit{Smart} \preccurlyeq ((\mathit{Smart} \to \mathit{Phone}) \to \mathit{Smart}) \to \mathit{Phone}
\end{gathered}
$$

☞ In dem Typ $((t_1 \to t_2) \to t_3) \to t_4$ stehen t_1 und t_3 an kontravarianten und t_2 und t_4 an kovarianten Positionen.

Die Richtung der Quasiordnung wechselt mit jeder Schachtelung im Argumenttyp.

40. Untertypen: Speicherzellen

Eine Speicherzelle kann mit ‘ !’ gelesen und mit ‘:=’ geschrieben werden.
Der Lesezugriff ist kovariant, der Schreibzugriff ist kontravariant.

Da Speicherzellen beide Zugriffsarten unterstützen, sind sie summa summarum *invariant*.

$$\frac{t' \preccurlyeq t \qquad t \preccurlyeq t'}{\mathit{Ref}\langle t' \rangle \preccurlyeq \mathit{Ref}\langle t \rangle}$$

☞ Entsprechendes gilt für alle modifizierbaren Datenstrukturen (engl. mutable data structures) wie zum Beispiel Arrays.

40. Untertypen: Speicherzellen

Die Invarianz von *Ref* lässt sich herleiten, wenn man die Typregeln für die Dereferenzierung ‘ !’ und die Zuweisung ‘:=’ studiert.

$$\dfrac{\dfrac{\overline{\Sigma \vdash e : \mathit{Ref}\langle t\rangle}}{\Sigma \vdash\ !e : t} \qquad \overline{t \preccurlyeq t'}}{\Sigma \vdash\ !e : t'}$$

$$\dfrac{\overline{\Sigma \vdash e_1 : \mathit{Ref}\langle t\rangle} \qquad \dfrac{\overline{\Sigma \vdash e_2 : t'} \qquad \overline{t' \preccurlyeq t}}{\Sigma \vdash e_2 : t}}{\Sigma \vdash (e_1 := e_2) : \mathit{Unit}}$$

40. Untertypen: Speicherzellen — da capo⋆

☞ Unterschiede man mit Hilfe des Typsystems zwischen lesendem und schreibendem Zugriff, ergäbe sich ein verfeinertes Bild.

$$\frac{\Sigma \vdash e : t}{\Sigma \vdash \textbf{\textit{ref}}\ e : \mathit{Ref}\langle t\rangle}$$

$$\frac{\Sigma \vdash e : \mathit{Read\text{-}Ref}\langle t\rangle}{\Sigma \vdash\ !e : t}$$

$$\frac{\Sigma \vdash e_1 : \mathit{Write\text{-}Ref}\langle t\rangle \qquad \Sigma \vdash e_2 : t}{\Sigma \vdash e_1 := e_2 : \mathit{Unit}}$$

☞ *Read-Ref* ist der Typ einer ‘read-only’ Speicherzelle; *Write-Ref* ist der Typ einer ‘write-only’ Speicherzelle.

40. Untertypen: Speicherzellen — da capo⋆

Der Typ *Read-Ref* ist kovariant, *Write-Ref* hingegen kontravariant.

$$\frac{t' \preccurlyeq t}{\mathit{Read\text{-}Ref}\langle t' \rangle \preccurlyeq \mathit{Read\text{-}Ref}\langle t \rangle} \qquad \frac{t \preccurlyeq t'}{\mathit{Write\text{-}Ref}\langle t' \rangle \preccurlyeq \mathit{Write\text{-}Ref}\langle t \rangle}$$

Der Typ *Ref* ist der Durchschnitt von *Read-Ref* und *Write-Ref*.

$$\frac{}{\mathit{Ref}\langle t' \rangle \preccurlyeq \mathit{Read\text{-}Ref}\langle t \rangle} \qquad \frac{}{\mathit{Ref}\langle t' \rangle \preccurlyeq \mathit{Write\text{-}Ref}\langle t \rangle}$$

☞ Eine les- und schreibbare Speicherzelle lässt sich zu einer ‘read-only’ oder einer ‘write-only’ Speicherzelle herabstufen.

☞ Wozu sind ‘write-only’ Speicherzellen gut?

40. Dynamische Semantik

Die Auswertung von Programmen ändert sich nicht.

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash (e :> t) \Downarrow \nu}$$

☞ Eine Typanpassung könnte prinzipiell mit einer Umwandlung des Wertes einhergehen.
Zum Beispiel:

- $Nat \preccurlyeq Int$: eine natürliche Zahl wird in der Regel anders repräsentiert als eine ganze Zahl (ohne und mit Vorzeichen);
- $Ref\langle t\rangle \preccurlyeq t$: Speicherzellen werden automatisch dereferenziert.

☞ Sämtliche Auswertungsregeln müssten dann angepasst werden!

40. Vertiefung

☞ Untertypen sind eine Spielart der *Polymorphie*.

Zur Erinnerung: 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.

Man unterscheidet zwischen 4 verschiedenen Arten von Polymorphie:

	universelle Polymorphie	ad-hoc Polymorphie
konjunktive Polymorphie	parametrische Polymorphie ($\forall$) (engl. generics) *length*	Überladung ($\wedge$) (engl. overloading) $=$, $+$
Inklusions-polymorphie	Untertypen ($\preccurlyeq$) (engl. subtyping) *transfer*	Konversion (engl. coercion) —

40. Parametrische Polymorphie

Die polymorphe Funktion $\mathit{length} : \mathit{List}\,\langle'a\rangle \to \mathit{Nat}$ kann zum Beispiel die Länge einer Liste eines beliebigen Typs bestimmen; sie besitzt im Prinzip unendliche viele Typen.

$$
\begin{aligned}
&\mathit{length} : \mathit{List}\,\langle \mathit{Nat}\rangle \to \mathit{Nat}\\
&\mathit{length} : \mathit{List}\,\langle \mathit{Bool}\rangle \to \mathit{Nat}\\
&\mathit{length} : \mathit{List}\,\langle \mathit{List}\,\langle \mathit{Nat}\rangle\rangle \to \mathit{Nat}\\
&\mathit{length} : \mathit{List}\,\langle \mathit{List}\,\langle \mathit{Bool}\rangle\rangle \to \mathit{Nat}\\
&\ldots
\end{aligned}
$$

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).

40. Überladung

Von *Überladung* (engl. overloading) spricht man, wenn der *gleiche* Bezeichner für *unterschiedliche* Funktionen für unterschiedliche Werte verwendet wird.

$$
\begin{aligned}
(+) &: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
(+) &: \mathit{float} \to \mathit{float} \to \mathit{float} \\
(+) &: \mathit{String} \to \mathit{String} \to \mathit{String}
\end{aligned}
$$

Überladung ist die kleine Schwester der parametrischen Polymorphie.

40. Überladung

Methodennamen dürfen überladen werden:

```
type IStack ⟨'elem⟩ =
  interface
    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.

☞ Um Mehrdeutigkeiten zu vermeiden, müssen sich die *Argumenttypen* der überladenen Methoden unterscheiden.

40. Untertypen

Die Funktion *transfer* ist ebenfalls polymorph; sie kann auf Elemente eines beliebigen Untertyps von *IAccount* angewendet werden.

$$
\begin{aligned}
&\mathit{transfer} : \mathit{IAccount} * \mathit{Nat} * \mathit{IAccount} && \to \mathit{Unit}\\
&\mathit{transfer} : \mathit{IAccount} * \mathit{Nat} * \mathit{IAccountPlus} && \to \mathit{Unit}\\
&\mathit{transfer} : \mathit{IAccountPlus} * \mathit{Nat} * \mathit{IAccount} && \to \mathit{Unit}\\
&\mathit{transfer} : \mathit{IAccountPlus} * \mathit{Nat} * \mathit{IAccountPlus} && \to \mathit{Unit}\\
&\ldots
\end{aligned}
$$

Die Zahl der Untertypen von *IAccount* ist zwar endlich, aber unbegrenzt, da die Schnittstellenhierarchie zu jedem Zeitpunkt beliebig erweitert werden kann.

40. Konversion

Von *Konversion* (engl. coercion) spricht man, wenn ein Element eines Typs automatisch in ein Element eines anderen Typs umgewandelt wird. Wird *nicht* unterstützt.

☞ Konversionen müssen *explizit* durchgeführt werden:

```
Mini⟩ float32 123456789
val it : float32 = 123456792.0f
Mini⟩ int it
val it : int = 123456792
```

Weitere böse Überraschungen:

```
Mini⟩ int 1e10f
val it : int = -2147483648
Mini⟩ float32 it
val it : float32 = -2147483650f
```

Konversion ist die kleine Schwester des Untertyppolymorphismus.

41. Knobelaufgabe #25

Zwei Listen sind *Permutationen* voneinander, wenn sie die gleichen Elemente enthalten — die Reihenfolge der Elemente spielt dabei keine Rolle. Eine Liste mit n verschiedenen Elementen besitzt insgesamt $n!$ Permutationen.

Eine Liste heißt *Megapermutation* von x, wenn sie jede Permutation von x als zusammenhängende Teilliste enthält.

x	Megapermutation von x
abc	abcabacba
abcd	abcdabcadbcabdcabacdbacbdabcadcba

Schreiben Sie ein Programm, das zu einer gegebenen Liste eine möglichst kurze Megapermutation bestimmt.

Hinweis: Verwenden Sie das Strukturentwurfsmuster für Listen.

41. Klassen

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
```

☞ Eine Klasse ist ein *Zwitterwesen*: *TrustMe* ist sowohl ein Typ, genauer: eine Schnittstelle, als auch ein Wert, der die Schnittstelle implementiert, ein Objektkonstruktor.

41. Objekterzeugung

Mit ***new*** *TrustMe* 4711 oder kurz *TrustMe* 4711 wird ein neues Konto eröffnet, in das ein initialer Betrag von 4711€ eingezahlt wird.

```
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
```

☞ Jargon: *TrustMe* 4711 ist eine Instanz der Klasse *TrustMe*.

☞ Eine Klassendefinition ***type*** T (x) $=$ ***class*** $\ldots$ ***end*** lässt sich als Schablone (engl. template) auffassen; mit ***new*** T e wird die Schablone instantiiert; das Ergebnis ist ein Objekt vom Typ T.

41. Klassenvariablen und -methoden

```
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 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
```

41. Klassenvariablen und -methoden

Klassenmethoden und -eigenschaften können unabhängig von der Existenz eines Objekts sprich eines Bankkontos verwendet werden.

☞ Zum Zeitpunkt der Gründung einer Bank existieren keine Konten.

```
Mini⟩ TrustMe.BIC
4711
Mini⟩ let lisas = TrustMe 4711
val lisas : TrustMe
Mini⟩ TrustMe.no-of-accounts
1
```

☞ Die jeweilige Nachricht wird an die Klasse in unserem Beispiel an die Bank *TrustMe* geschickt.

41. Felder, Methoden und Eigenschaften

- ***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* (engl. field) genannt. Auch eine Instanzvariable ist stets privat.
- ***do*** e
  kennzeichnet einen Ausdruck, der bei der Erzeugung eines Objekts mit ***new*** ausgeführt wird.

41. Felder, Methoden und Eigenschaften

- ***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.

41. Felder, Methoden und Eigenschaften

- ***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.
- ***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.

41. Felder, Methoden und Eigenschaften

- ***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.

41. Module versus Klassen

Sichtbarkeit (engl. visibility, auch accessibility):

- *Modul:* Position des Bezeichners: global oder lokal;
- *Klasse:* durch vorhandene oder fehlende ***static*** Kennzeichnung.

41. Klassen und Schnittstellen

Eine alternative Implementierung einer Bank:

```
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
```

41. Klassen und Schnittstellen

Beobachtung: Weder das Objekt *TrustMe* 4711 noch das Objekt *Cheap'N'Easy* 815 kann an die Funktion *transfer* übergeben werden!

transfer : $\mathit{IAccount} * \mathit{Nat} * \mathit{IAccount} \to \mathit{Unit}$

$\mathit{TrustMe}\quad 4711 : \mathit{TrustMe}$
$\mathit{Cheap'N'Easy}\ 815 : \mathit{Cheap'N'Easy}$

☞ Alle drei Typen, *IAccount*, *TrustMe*, *Cheap'N'Easy*, sind verschieden und damit inkompatibel.

„*All types are created unequal.*“

☞ 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?).

41. Klassen und Schnittstellen

```
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
```

☞ *Lies:* *Cheap'N'Easy* implementiert die Schnittstelle *IAccount*. Damit ist die Klasse ein Untertyp der Schnittstelle: $\mathit{Cheap'N'Easy} \preccurlyeq \mathit{IAccount}$.

41. Klassen und Schnittstellen

Jetzt gelingt die Überweisung: Beim Aufruf der Funktion *transfer* wird automatisch der Typ mit Hilfe der Subsumptionsregel angepasst.

```
Mini⟩ let herberts = Cheap'N'Easy 4711
Mini⟩ let harrys = Cheap'N'Easy 10
Mini⟩ transfer (herberts, 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 später, warum das sinnvoll ist.)

41. Klassen und Schnittstellen

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.

☞ Lässt man die Typanpassung weg, erfolgt ein rekursiver Aufruf mit der Konsequenz der Nichtterminierung!

41. Klassen und Schnittstellen

Man kann auch den umgekehrten Weg beschreiten und die Schnittstellenmethoden durch Methoden der Klasse definieren.

```
type TrustMe (seed : Nat) =
  class
    ...
    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.

41. Klassen und Schnittstellen

Nach diesen Vorarbeiten lassen sich sowohl Banküberweisungen tätigen als auch die Kontostände direkt ohne Umweg über die Schnittstelle abrufen.

```
Mini⟩ let herberts = TrustMe 4711
Mini⟩ let harrys = Cheap'N'Easy 0
transfer (herberts, 100, harrys)
()
Mini⟩ herberts.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.

41. Klassen und Schnittstellen

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
```

41. Klassen und Schnittstellen

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.

41. Generische Klassen

In Teil V haben wir endliche Abbildungen mit Hilfe von Listen, Suchlisten und Suchbäumen implementiert, siehe Skript.

Ein Auszug aus der Schnittstelle:

type $Map\,\langle 'key, 'value\rangle$ ***when*** $'key : comparison\rangle$

val *empty* : $Map\,\langle 'key, 'value\rangle$
val *add* : $'key * 'value \to Map\,\langle 'key, 'value\rangle \to Map\,\langle 'key, 'value\rangle$
val *look-up* : $'key \to Map\,\langle 'key, 'value\rangle \to 'value\ option$

☞ Ein Wörterbuch assoziiert mit einem Schlüssel einen bestimmten Wert. Sowohl Schlüssel als auch Wert können einen beliebigen Typ haben.

☞ Die oben genannten Implementierungen sind *persistent*.

☞ Im Folgenden implementieren wir eine ephemerale Variante von endlichen Abbildungen: Beim Erweitern wird die ursprüngliche Abbildung überschrieben; sie ist danach *nicht* mehr verfügbar.

41. Generische Klassen

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"
```

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*.

41. Generische Klassen

Für beide Operationen bieten wir syntaktischen Zucker an:

```
Mini⟩ perfume.[5]
"Chanel"
Mini⟩ perfume.[0] ← "Eau"
()
Mini⟩ perfume.[0]
"Eau"
Mini⟩ perfume.[0] ← "Wasser"
()
Mini⟩ perfume.[0]
"Wasser"
```

☞ *perfume*.[*k*] ← *v* fügt ein neues Schlüssel-Wert Paar hinzu (bzw. aktualisiert einen bestehenden Eintrag für den Schlüssel *k*); *perfume*.[*k*] schlägt den angegebenen Schlüssel nach.

41. Generische Klassen

☞ Wie Record-, Varianten- und Schnittstellentypen können auch Klassentypen mit einem oder mehreren Typen parametrisiert werden.

```
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 look-up key mapping with
        | None       → raise (KeyNotFoundException ())
        | Some value → value

    ...
  end
```

☞ Eine Klasse ist ein *Zwitterwesen*: *SearchTree* ist ein *parametrisierter* Typ und ein *polymorpher* Objektkonstruktor.

41. Generische Klassen

```
type SearchTree ⟨'key, 'value when 'key : comparison⟩ () =
  class
    ...
    member self.Item
      with get key       = self.Lookup key
      and  set key value = self.Add (key, value)
  end
```

Der syntaktische Zucker für Erweiterung und Zugriff wird mittels der Eigenschaft *Item* realisiert. Es handelt sich um eine sogenannte *indizierte Eigenschaft* (engl. indexed property).

☞ 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 $\ell = \mathit{Item}$, dann kann man ℓ auch auslassen und kurz $e.[i]$ und $e.[i] \leftarrow e'$ schreiben.

41. Smalltalk und Java

☞ In den Programmiersprachen Smalltalk (1972 geboren) und Java (1995 geboren) spielen Objekte und Klassen eine weitaus größere Rolle als in Mini–F#.

- *Smalltalk:* Alles ist ein Objekt, auch Wahrheitswerte und Zahlen. Jedes Objekt ist Instanz einer Klasse; auch Klassen sind Objekte. Im Gegensatz zu Mini–F# ist Smalltalk *dynamisch getypt*.
- *Java:* Fast alles ist ein Objekt; neben Klassen gibt es auch primitive Typen wie Wahrheitswerte und Zahlen. Wie Mini–F# ist Java *statisch getypt*.

41. Java

One of the most important influences on the design of Java was a much earlier language called Simula.
— James Gosling

Your development cycle is much faster because Java is interpreted. The compile-link-load-test-crash-debug cycle is obsolete.
— James Gosling

Like the creators of sitcoms or junk food or package tours, Java's designers were consciously designing a product for people not as smart as them.
— Paul Graham

41. Java: Klassen

```
class TrustMe {
    private long funds = 0;

    TrustMe (long seed) {
        this.funds = seed;
    }

    void deposit (long amount) {
        funds = funds + amount;
    }
    void withdraw (long amount) throws Insufficient {
        if (amount > funds)
            throw new Insufficient (funds);
        else
            funds = funds - amount;
    }
    long balance () { return funds; }
}
```

☞ `TrustMe (long seed)` ist ein *Konstruktor*. Die Methodenrümpfe verwenden C-Syntax.

41. Java: Klassen

```
class TrustMe {
    ...
    static private int no = 0;
    static int BIC        = 4711;
    static {
        System.out.println ("TrustMe is founded.");
    }

    TrustMe (long seed) {
        this.funds = seed;
        no++;
    }
    ...
    static long no_of_accounts () { return no; }
}
```

41. Java: Schnittstellen

```
interface IAccount {
    void deposit  (long amount);
    void withdraw (long amount) throws Insufficient;
    long balance  ();
}
```

☞ Die Tatsache, dass `withdraw` möglicherweise eine Ausnahme wirft, ist im Typ vermerkt.

41. Java: Klassen, da capo

```
class TrustMe implements IAccount {
    private long funds    = 0;
    static private int no = 0;
    static int BIC        = 4711;

    ...
    public void deposit (long amount) {
      ...
    }

    public void withdraw (long amount) throws Insufficient {
      ...
    }

    public long balance () {
      ...
    }
    ...
}
```

41. Java: Module und Funktionen

```
import java.math.*;

public final class Factorial2 {
    public static BigInteger factorial (BigInteger n) {
        if (n.compareTo (BigInteger.ZERO) == 0)
            return BigInteger.ONE;
        else
            return n.multiply (factorial (
                    n.subtract (BigInteger.ONE)));
    }

    public static void main (String[] args) {
        BigInteger n = new BigInteger("99");

        System.out.println (factorial (n));
    }
}
```

☞ Funktionen sind Klassenmethoden; Module sind Klassen (die nur Klassenvariablen und -methoden enthalten).

41. Lösung Knobelaufgabe #16

Kann man einem regulären Ausdruck ansehen, ob alle in der von dem regulären Ausdruck bezeichneten Sprache enthaltenen Wörter eine gerade Anzahl von as enthalten?

- $\emptyset$
- ϵ
- $(\mathtt{aa})^*$
- $\mathtt{a}\,(\mathtt{aa})^*$
- $(\epsilon \mid \mathtt{aa})\,(\epsilon \mid \mathtt{aa})$
- $(\mathtt{a} \mid \mathtt{aaa})\,(\mathtt{a} \mid \mathtt{aaa})$
- $(\epsilon \mid \mathtt{aa})\,(\mathtt{a} \mid \mathtt{aaa})$
- $(\mathtt{ab})^*$
- $(\mathtt{a}\,\emptyset\,\mathtt{b})^*$

41. Lösung Knobelaufgabe #16

- Wir nennen eine Sprache *gerade*, wenn alle in der Sprache enthaltenen Wörter eine gerade Anzahl von as enthalten.
- Wir nennen eine Sprache *ungerade*, wenn alle in der Sprache enthaltenen Wörter eine ungerade Anzahl von as enthalten.
- Wir sind daran interessiert, ob eine Sprache gerade ist.
- Um das festzustellen, benötigen wir auch die Information, ob eine Sprache ungerade ist.

$$(a \mid aaa)\ (a \mid aaa)$$

- Einige Sprachen sind weder gerade noch ungerade.

$$(\epsilon \mid aa) \mid (a \mid aaa)$$

- Eine Sprache ist sowohl gerade als auch ungerade!

$$\emptyset$$

41. Lösung Knobelaufgabe #16

```
type Abstract =
  | Both   // Durchschnitt von Even und Odd (nur ∅).
  | Even   // Sprachen, deren Wörter eine gerade # as enthalten.
  | Odd    // Sprachen, deren Wörter eine ungerade # as enthalten.
  | None   // Vereinigung von Even und Odd.

let rec even-odd (reg : Reg) : Abstract =
  match reg with
  | Eps          → Even
  | Sym x        → match x with A → Odd | B → Even
  | Cat (r1, r2) → cat (even-odd r1, even-odd r2)
  | Empty        → Both
  | Alt (r1, r2) → alt (even-odd r1, even-odd r2)
  | Rep r        → rep (even-odd r)
```

☞ Wir müssen noch *cat*, *alt* und *rep* definieren.

41. Lösung Knobelaufgabe #16

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{cat}\ (a_1 : \mathit{Abstract}, a_2 : \mathit{Abstract}) : \mathit{Abstract} = \\
\quad \textbf{\textit{match}}\ (a_1, a_2)\ \textbf{\textit{with}} \\
\quad \mid (\mathit{Both}, \_) \quad \mid (\_, \mathit{Both}) \quad \to \mathit{Both} \\
\quad \mid (\mathit{None}, \_) \quad \mid (\_, \mathit{None}) \quad \to \mathit{None} \\
\quad \mid (\mathit{Even}, \mathit{Even}) \mid (\mathit{Odd}, \mathit{Odd}) \to \mathit{Even} \\
\quad \mid (\mathit{Even}, \mathit{Odd}) \mid (\mathit{Odd}, \mathit{Even}) \to \mathit{Odd}
\end{array}
$$

☞ *Both* ist das Nullelement der Konkatenation, *Even* ist das Einselement.

41. Lösung Knobelaufgabe #16

```
let alt (a1 : Abstract, a2 : Abstract) : Abstract =
  match (a1, a2) with
  | (Both, a) | (a, Both) → a
  | (Even, Even)          → Even
  | (Odd, Odd)            → Odd
  | _                     → None
```

☞ *Both* ist das Einselement der Alternative.

41. Lösung Knobelaufgabe #16

```
let rep (a : Abstract) : Abstract =
  match a with
  | Both | Even → Even
  | _           → None
```

☞ $\emptyset^* = \epsilon$, deswegen ist *rep Both* = *Even*.

42. Knobelaufgabe #26

$$P = NP\ ?$$

http://www.claymath.org/millennium-problems/p-vs-np-problem

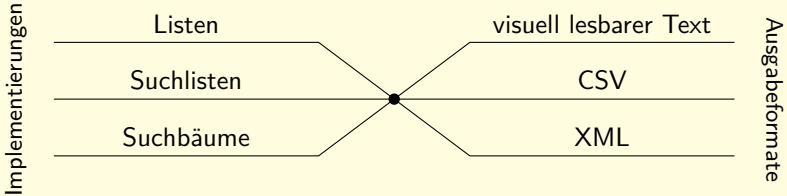
42. Motivation

Aufgabe: Das Wörterbuch soll um die Möglichkeit erweitert werden, die verwalteten Schlüssel-Wert Paare auszugeben — entweder flüchtig auf dem Bildschirm oder dauerhaft in eine Datei.

Möglichkeit A: Wir erweitern die Klasse *SearchTree* um eine maßgeschneiderte Methode.

member *Print* : *Unit* → *Unit*

42. Motivation



Vorteil:

- Einfache und direkte Umsetzung der Aufgabe.

Nachteil:

- Kombinatorische Explosion von Programmieraufgaben.

42. Motivation

Möglichkeit B: Wir überführen die Schlüssel-Wert Paare in ein einheitliches Zwischenformat, einen Mediator.

member $Keys : Unit \to List\ \langle 'key \rangle$

Vorteil:

- Kombinatorische Explosion von Programmieraufgaben wird vermieden.

Nachteil:

- Zusätzlicher Speicherplatz wird benötigt: Die Daten, die bereits im Wörterbuch abgelegt sind, werden faktisch dupliziert.

42. Motivation

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 $Keys : Unit \to IEnumerator\ \langle 'key \rangle$

Der Sequenztyp *IEnumerator* firmiert unter vielen verschiedenen Namen:

- „Aufzähler“ (engl. enumerator),
- „Wiederholer“ (engl. iterator),
- „Cursor“ (engl. cursor),
- „faule Liste“ (engl. lazy list) und
- Generator.

42. Motivation: Aufzähler

Wir ersetzen eine *Datenstruktur* durch eine *Kontrollstruktur*.

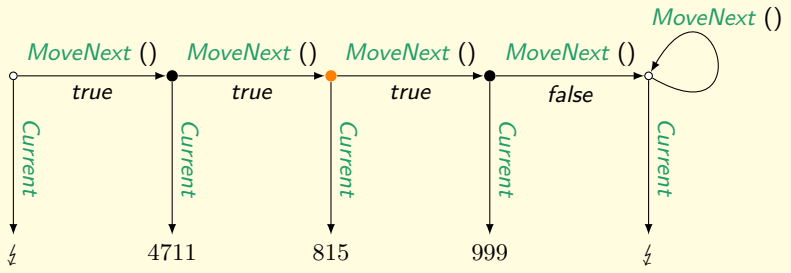
Aus einem *Datum*, das alle Elemente der repräsentierten Sequenz umfasst (eine Liste), wird ein *Programm* (ein Aufzähler), das es erlaubt, alle Elemente nacheinander zu generieren.

```
type IEnumerator ⟨'elem⟩ =
  interface
    abstract member Current  : 'elem
    abstract member MoveNext : Unit → Bool
  end
```

☞ *Current* gibt das aktuelle Element der Aufzählung zurück; *MoveNext* () rückt zum nächsten Element vor und signalisiert, ob ein Folgeelement tatsächlich existiert.

42. Motivation: Aufzähler

Bei der Verwendung der Schnittstelle *IEnumerator* muss man sich an ein striktes *Protokoll* halten.



☞ Der „Cursor“ der Aufzählung kann entweder auf ein Element zeigen oder vor dem ersten bzw. nach dem letzten Element stehen.

42. Motivation: Aufzähler

Darstellung des Intervalls *lower* . . *upper*.

```
let range (lower, upper) =
  let mutable started = false
  let mutable current = lower
  {new IEnumerator ⟨Nat⟩ 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 Funktion *range* ist *defensiv programmiert*.

42. Motivation: Aufzähler

Die implizite Darstellung einer Sequenz durch ein Programm bietet die Möglichkeit, *unendliche* Sequenzen zu repräsentieren.

```
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
  }
```

☞ *MoveNext* gibt immer *true* zurück; die Aufzählung terminiert nie.

42. Motivation: Aufzähler

Die Funktion *from-list* : $List\ \langle 'elem \rangle \to IEnumerator\ \langle 'elem \rangle$ implementiert einen *Repräsentationswechsel*.

```
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
  }
```

42. Motivation: Aufzählbare Objekte

☞ Viele „Dinge“ sind aufzählbar, die Elemente einer Zahlenfolge (die Primzahlen, die Catalanzahlen), die Elemente eines Containers (eines Arrays, einer Liste, eines Stacks).

Allgemeine Schnittstelle für *aufzählbare Objekte*:

```
type IEnumerable<'elem> =
  interface
    abstract member GetEnumerator : Unit → IEnumerator<'elem>
  end
```

Möglichkeit D: Die Klasse *SearchTree* implementiert die Schnittstelle.

```
interface IEnumerable<'key> with
  member self.GetEnumerator () = ...
```

42. Motivation: Aufzählbare Objekte

Syntaktischer Zucker: Die ***for***-Schleife zur Linken ist eine Abkürzung für die ***while***-Schleife zur Rechten.

```
for x in xs do
    body
```

```
let enumerator = xs.GetEnumerator ()
while enumerator.MoveNext () do
    let x = enumerator.Current
    body
```

☞ ***for***-Schleifen geben nicht länger ein Terminierungsversprechen :-(. Die Aufzählung muss ja nicht endlich sein (siehe *squares*).

42. Motivation: Aufzählbare Objekte

Die beiden Schnittstellen 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
```

☞ *sum* funktioniert für beliebige aufzählbare Objekte, insbesondere für Listen und Arrays.

42. Motivation: Sequenzbeschreibungen

Die obigen Programme haben einen *präskriptiven* Charakter. Es wird genau detailliert,

- *wie* das nächste Element in Abhängigkeit vom aktuellen Zustand generiert und
- der Zustand aktualisiert wird.

Alternativ können Sequenzen *deskriptiv* mit Hilfe von *Sequenzbeschreibungen* programmiert werden. Wir geben an,

- *was* die Elemente der Sequenz sind.

Die Syntax kennen wir schon von Listen- und Arraybeschreibungen.

Redefinition von *squares*:

$$\textbf{\textit{let}}\ \mathit{squares} = \mathit{seq}\ \{\ \textbf{\textit{for}}\ n\ \textbf{\textit{in}}\ \mathit{nats}\ \textbf{\textit{do yield}}\ n * n\ \}$$

42. Motivation: Sequenzbeschreibungen

Die Folge der natürlichen Zahlen selbst lässt sich entweder präskriptiv mit einem Objektausdruck oder deskriptiv mit Hilfe einer rekursiven *Werte*definition programmieren.

```
let rec nats =
  seq {yield 0
       for n in nats do
         yield n + 1}
```

☞ Giuseppe Peanos Beschreibung der natürlichen Zahlen wird eingefangen: Eine natürliche Zahl ist entweder 0 oder der Nachfolger $n + 1$ einer natürlichen Zahl n.

42. Motivation: Sequenzbeschreibungen

Die Definition des Aufzählers für die Schlüssel des Wörterbuchs steht noch aus. Im Wesentlichen ein Inorder-Durchlauf:

```
let rec inorder = function
    | Leaf           → Seq.empty
    | Node (l, x, r) → seq { yield! inorder l; yield x; yield! inorder r }
```

☞ *Seq.empty* generiert die leere Sequenz.

☞ ***yield***! *e* ist eine Abkürzung für ***for*** *x* ***in*** *e* ***do yield*** *x*.

42. Abstrakte Syntax

Wir erweitern Ausdrücke um Sequenzbeschreibungen.

$e \in \text{Expr} ::=$	*Ausdrücke:*
$\quad \mid \; \mathit{seq}\,\{\mathit{se}\}$	Sequenzbeschreibungen

Innerhalb der Klammern steht ein *Sequenzausdruck*, dessen Syntax wir schon von Listen- und Arraybeschreibungen kennen.

42. Statische Semantik

Sequenzbeschreibungen besitzen den Typ $seq\,\langle t\rangle$, ein Alias für den Schnittstellentyp $IEnumerable\,\langle t\rangle$.

$t \in \text{Type} ::=$	*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 $seq\,\langle t\rangle$.

$$\frac{}{List\,\langle t\rangle \preccurlyeq seq\,\langle t\rangle} \qquad \frac{}{Array\,\langle t\rangle \preccurlyeq seq\,\langle t\rangle}$$

☞ Typregeln, siehe Skript.

42. Dynamische Semantik

Die Sequenzbeschreibung $seq\,\{se\}$ wird in den Ausdruck $[\![se]\!]$ übersetzt, wobei $[\![se]\!]$ wie folgt definiert ist.

$$
\begin{array}{ll}
[\![\textbf{\textit{yield}}\ e]\!] & = \mathit{Seq.singleton}\ e \\
[\![\textbf{\textit{yield}}!\ e]\!] & = e \\
[\![se_1; se_2]\!] & = \mathit{Seq.append}\ ([\![se_1]\!])\ ([\![se_2]\!]) \\
[\![\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ se]\!] & = \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ [\![se]\!]\ \textbf{\textit{else}}\ \mathit{Seq.empty} \\
[\![\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e\ \textbf{\textit{do}}\ se]\!] & = \mathit{Seq.collect}\ (\textbf{\textit{fun}}\ x \rightarrow [\![se]\!])\ e
\end{array}
$$

☞ Wir beschränken uns auf die Definition der Konkatenation.

42. Dynamische Semantik

Konkatenation von Aufzählern:

```
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, ob der erste Aufzähler aktiv ist. Wenn nicht, werden die Nachrichten an den zweiten Aufzähler delegiert.

42. Dynamische Semantik

Konkatenation von aufzählbaren Objekten:

```
let append (first : IEnumerable<'elem>)
           (second : IEnumerable<'elem>) =
  { new IEnumerable<'elem> with
    member self.GetEnumerator () =
           append-enumerators (first.GetEnumerator ())
                              (second.GetEnumerator ())
  }
```

42. Vertiefung: Testen

Harry Hacker hat einen neuen Sortieralgorithmus entwickelt: *lightning-sort*. Lisa Lista schlägt vor, das Programm vor dem produktiven Einsatz systematisch zu testen.

☞ Sequenzbeschreibungen eignen sich wunderbar für die Generierung von 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.

Summa summarum werden 4.037.914 Tests durchgeführt.

Wie lassen sich die Permutationen einer Liste systematisch generieren?

42. Permutationsalgorithmen

- *Peano Entwurfsmuster:* Problem der Größe n wird auf Problem der Größe $n-1$ zurückgeführt.
- *Permutieren:* die Problemgröße entspricht der Anzahl der zu permutierenden Elemente.
- *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*.
- Fokussiert auf die Eingabe.
- *Permutieren durch Auswählen:*
  - *Wähle* nacheinander alle Elemente *aus*,
  - bilde jeweils alle Permutationen der restlichen Elemente,
  - setze das entfernte Element vor jede korrespondierende Permutation.
- Fokussiert auf die Ausgabe.

42. Permutieren durch Einfügen

Beispiel: Wir wollen alle Ziffern der Zahl 1234 permutieren.

Wir entfernen die erste Ziffer, die ①, und berechnen 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.

①234	2①34	23①4	234①
①324	3①24	32①4	324①
①342	3①42	34①2	342①
①243	2①43	24①3	243①
①423	4①23	42①3	423①
①432	4①32	43①2	432①

Insgesamt erhalten wir $3! \cdot 4 = 4! = 24$ Permutationen.

42. Permutieren durch Einfügen

Die Hilfsfunktion *insertions* fügt ein Element an allen Positionen einer Liste ein.

```
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 permutations = function
    | []      -> seq {yield []}
    | x :: xs -> seq {for ys in permutations xs do
                        for zs in insertions x ys do
                          yield zs}
```

42. Permutieren durch Auswählen

Beispiel: Es 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.

①234	①243	①324	①342	①423	①432
②134	②143	②314	②341	②413	②431
③214	③241	③124	③142	③421	③412
④231	④213	④321	④312	④123	④132

Wiederum erhalten wir insgesamt $4 \cdot 3! = 4! = 24$ Permutationen.

☞ Permutieren durch Auswählen generiert die Permutationen in *lexikographischer Reihenfolge*.

42. 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 permutations = function
    | [] → seq {yield []}
    | xs → seq {for (y, ys) in deletions xs do
                  for zs in permutations ys do
                    yield y :: zs}
```

42. Lösung Knobelaufgabe #19

Die *unendliche* Folge

$$0\,1\,0\,2\,0\,1\,0\,3\,0\,1\,0\,2\,0\,1\,0\,4\,0\,1\,0\,2\,0\,1\,0\,3\,0\,1\,0\,2\,0\,1\,0\,5\ldots$$

ist invariant unter der folgenden Transformation.

- Jedes Element wird um 1 erhöht.
- An den Anfang und zwischen je zwei Elemente wird eine 0 gesetzt.

Direkte Umsetzung mit Hilfe einer rekursiven *Werte*definition:

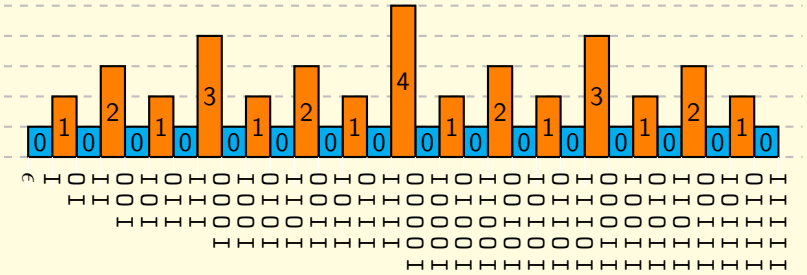
```
let rec carry =
  seq {yield 0
       for i in carry do
         yield i + 1
         yield 0}
```

42. Lösung Knobelaufgabe #19

Die *unendliche* Folge

0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 5 . . .

heißt *Übertragsequenz* (engl. binary carry or ruler sequence). Das n-te Element entspricht dem Exponenten der höchsten 2-Potenz, die n teilt.



☞ Siehe *A*007814 in der On-Line Encyclopedia of Integer Sequences (kurz OEIS, http://oeis.org/).

It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing $\langle \ldots \rangle$ is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing.

— Christopher Strachey

43. Implementierungsvererbung

Wie Schnittstellen durch „Vererbung“ verbreitert werden können, so lassen sich auch Klassen um zusätzliche Funktionalität erweitern.

```
type TrustMeGold (seed : Nat) =
  class
    inherit TrustMe (seed)
    member self.Clear () : Nat =
      let amount = self.Balance
      self.Withdraw amount
      amount
  end
```

☞ *Implementierungsvererbung*: *TrustMeGold* ist eine *Unterklasse* (engl. subclass) der Klasse *TrustMe*, der *Oberklasse* (engl. subclass) bzw. *Basisklasse* (engl. base class).

☞ *Einfachvererbung*: Eine Klasse darf nur von *einer* Oberklasse erben.

43. Redefinition

☞ Vererbung kann auch verwendet werden, um Verhalten zu ändern (Alternative zu Delegation).

```
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
```

☞ ***override*** zeigt an, dass sich über die bereits bestehende Definition der Methode *Withdraw* hinweggesetzt wird (engl. override).

☞ In der Redefinition von *Withdraw* wird auf die ursprüngliche Definition mit ***base***.*Withdraw* zugegriffen.

43. Virtuelle Methoden

Die Basisklasse muss allerdings Reimplementierungen zulassen.

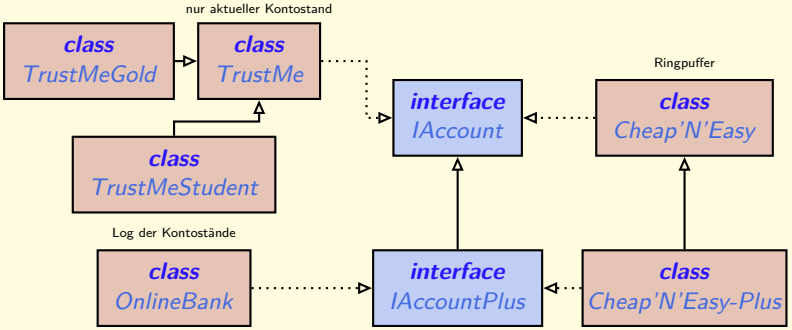
```
type TrustMe (seed : Nat) =
  class
    ...
    abstract member Withdraw : Nat → Unit
    default self.Withdraw (amount : Nat) =
      funds ← monus (funds, amount)
    ...
  end
```

☞ Neben der Signatur von *Withdraw* wird eine *Standarddefinition* (engl. default definition) angegeben — diese ist optional.

☞ Eine abstrakte Methode mit einer Standarddefinition heißt auch *virtuelle Methode* (engl. virtual method).

43. Klassendiagramm

Die Klassenhierarchie kann vollkommen unabhängig von der Schnittstellenhierarchie entwickelt werden.



☞ Programme, siehe Skript.

43. Entwurfsmuster

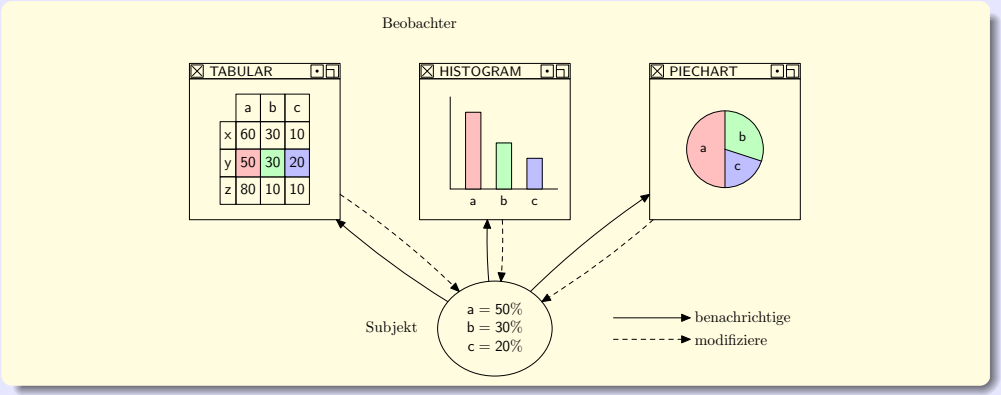
Entwurfsmuster existieren in verschiedenen „Größenordnungen“.

- Entwurfsmuster im Kleinen (small-scale):
  Organisation und Entwurf einzelner Funktionen oder Gruppen von Funktionen: Peano, Leibniz, Struktur.
- Entwurfsmuster im Größeren (medium-scale):
  Organisation und Entwurf von größeren Programmteilen oder Bibliotheken: zum Beispiel „Beobachter“ (engl. observer)
- Entwurfsmuster im Großen (large-scale):
  Organisation und Entwurf der Architektur eines Programms oder Softwaresystems.

Im Folgenden schauen wir uns zwei Beispiele für „medium-scale“ Entwurfsmuster an: Beobachter und Schablonenmethode.

43. Beobachter-Entwurfsmuster

- *Beispiel:* Trennung von Daten und deren Repräsentation.



- Subjekt hat beliebig viele Beobachter; ändert sich der Zustand des Subjekts, sollen alle Beobachter darüber informiert werden.
- *Lose Kopplung:* Das zu beobachtende Objekt soll von den Beobachtern unabhängig bleiben, ihre Schnittstelle nicht kennen.
- *Vorteil:* Ein Aspekt lässt sich unabhängig von den anderen ändern.

43. Beobachter-Entwurfsmuster: Demo

Wir illustrieren das Beobachter-Entwurfsmuster am Beispiel eines beobachtbaren Zählers.

Zähler mit einer einfachen „Fortschrittsanzeige“:

```
let steps = counter ()
do steps.Attach (fun n → if n % 1000 = 0 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)
```

Nach jeweils 1000 Schritten wird ein Asteriskus ‘*’ ausgegeben.

```
Mini⟩ fibonacci 20
**********************val it = 987
Mini⟩ steps.State
val it = 21891
```

43. Beobachter-Entwurfsmuster: Beobachter

Ein *Beobachter* ist eine Prozedur, die einen Zustand verarbeitet.

type $Observer\ \langle 'state \rangle = 'state \rightarrow Unit$

43. Beobachter-Entwurfsmuster: Subjekt

Das Subjekt, das wir beobachten wollen, wird durch eine *abstrakte Klasse* realisiert (die Eigenschaft *State* ist 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 interne Methode *Notify* ist nicht für den Endbenutzer gedacht, sondern wird lediglich Unterklassen zur Verfügung gestellt.

43. Abstrakte Klassen

Das sogenannte *Attribut* (engl. attribute) [< *AbstractClass* >] kennzeichnet die Klasse *Subject* als abstrakt: nicht alle Methoden sind implementiert.

Auf diese Weise wird sichergestellt, dass nicht aus Versehen die Bereitstellung einer ***default***-Methode versäumt wird.

```
[< AbstractClass >]
type Subject ⟨'state⟩ () =
  class
    ...
    abstract member State : 'state with get, set
    ...
  end
```

☞ Eine abstrakte Klasse hat keine Instanzen.

43. Beobachter-Entwurfsmuster: Zähler

Eine abstrakte Klasse kann durch eine Unterklasse konkretisiert werden oder durch einen Objektausdruck.

```
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 beobachten nur schreibende Zugriffe (via *Notify*).

43. Schablonen-Entwurfsmuster

- *Beispiel:* Der Ablauf eines Zwei-Personen-Spiels lässt sich in eine feste *Schablone* pressen: Die Spieler sind abwechselnd am Zug; ein Spieler verliert, wenn keine Zugmöglichkeiten mehr existieren.
- *Idee:* Skelett eines Algorithmus festlegen, aber die Teilschritte variabel halten.
- So besteht die Möglichkeit, einzelne Schritte des Algorithmus zu verändern oder zu überschreiben, ohne dass die zu Grunde liegende Struktur des Algorithmus modifiziert werden muss.

Wir illustrieren das Schablonen-Entwurfsmuster am Beispiel von *neutralen Zwei-Personen-Spielen*.

43. Schablonen-Entwurfsmuster: Demo

Das bekannteste neutrale Zwei-Personen-Spiel ist *Nim*: Von einem Haufen Streichhölzer werden in einem Spielzug 1–3 Hölzer entfernt.

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 der Rechner („`I win`“).

43. Schablonen-Entwurfsmuster

Die variablen Bestandteile des Algorithmus werden mit abstrakten Methoden modelliert. Der Algorithmus selbst ist eine konkrete Methode, die aber von den abstrakten Bestandteilen abhängt.

```
[< 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 () =
      let mutable turn = false
      while not self.Finished do
        self.Position ()
        turn ← not turn
        self.Move turn
      self.Winner turn
  end
```

43. Schablonen-Entwurfsmuster

```
type Nim (n : Nat) =
  inherit TwoPlayerGame ()

  let mutable matches = n

  let announce i = putline ("I take " ^ show i); i

  override self.Finished =
    matches = 0

  override self.Position () = putline ...

  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")
```

43. Kritik

> *I fear the 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*

Von den unzähligen Sprachfeatures, die wir im Laufe der Vorlesung eingeführt haben, ist Vererbung das wohl kontroverseste.

inheritance breaks abstraction

In vielen Fällen lässt sich Implementierungsvererbung durch einfachere und bewährte Sprachfeatures ersetzen.

- *Schablonen-Entwurfsmuster:*
  eine Schablone ist eine *Funktion höherer Ordnung*;
- *Beobachter-Entwurfsmusters:*
  Objektkonstruktoren höherer Ordnung.

43. Schablonen-Entwurfsmuster — da capo

Eine Schnittstelle für Zwei-Personen-Spiele:

```
type ITwoPlayerGame =
  interface
    abstract member Finished : Bool
    abstract member Position : Unit → Unit
    abstract member Move     : Bool → Unit
    abstract member Winner   : Bool → Unit
  end
```

43. Schablonen-Entwurfsmuster — da capo

Das Schablonen-Entwurfsmuster lässt sich alternativ mit Hilfe einer Funktion umsetzen, die mit einem Spielobjekt parametrisiert ist.

```
let play (game : ITwoPlayerGame) =
  let mutable turn = false
  while not game.Finished do
    game.Position ()
    turn ← not turn
    game.Move turn
  game.Winner turn
```

43. Beobachter-Entwurfsmuster — da capo

Eine Schnittstelle für „Zustände“.

```
type IState⟨'state⟩ =
  interface
    abstract member State : 'state with get, set
  end
```

43. Beobachter-Entwurfsmuster — da capo

Wir parametrisieren den Zähler mit dem Beobachter:

```
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 = 0 then putchar '*')
```

☞ Der Objektkonstruktor *counter* ist eine *Funktion höherer Ordnung*.

☞ „Low cost“ Version; vollständige Version, siehe Skript.

43. Delegation versus Vererbung

Um Objekte kompostional zu definieren, haben wir zwei grundlegende Techniken kennengelernt:

- *Delegation*;
- *Unterklassen*: Implementierungsvererbung.

☞ Im Allgemeinen ist Delegation die bessere Wahl. Warum? Um die Gründe zu verstehen, müssen wir etwas ausholen ...

☞ Es ist wichtig, den Mechanismus des Nachrichtensendens im Zusammenspiel von Klassen und Unterklassen genau zu verstehen.

43. Vererbung: Schrittzähler

```
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
```

☞ *Inc* und *Step* sind beide virtuelle Methoden.

43. Vererbung: Paritätszähler

```
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
```

☞ Der Paritätszähler hält nach, ob die Gesamtzahl der Schritte gerade oder ungerade ist.

43. Vererbung: Demo

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
```

☞ *Problem:* Jede Erhöhung wird doppelt gezählt!

43. Vererbung: Paritätszähler — da capo

Lösung: Wir dürfen *Step* nicht redefinieren.

```
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
```

43. Vererbung: Schrittzähler — da capo

Problem gelöst? Nicht ganz ...

Einige Zeit später optimiert Harry Hacker die Klasse *StepCounter*:

```
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
```

43. Vererbung: Demo

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
```

☞ Die Parität wird überhaupt nicht aktualisiert!

43. Vererbung: Fazit

☞ Die Änderung der Basisklasse zieht nicht-lokale Änderungen der Unterklassen nach sich — eine Klasse kann viele Unterklassen besitzen und diese können über verschiedene Module verstreut sein.

☞ Die Unterklasse *ParityCounter* lässt sich 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

☞ Das Problem tritt bei der Delegation nicht auf ...

43. Delegation: Schrittzähler

```
type IStepCounter =
  abstract member Inc  : Unit → Unit
  abstract member Step : Int  → Unit
```

```
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 ()
  }
```

43. Delegation: Paritätszähler

```
type IParityCounter =
  inherit IStepCounter
  abstract member Parity : Bool
```

```
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
  }
```

43. Delegation: Demo

```
Mini⟩ let counter = parity-counter ()
Mini⟩ counter.Step 3
Step: delegator
Step: delegatee
Inc: delegatee
Inc: delegatee
Inc: delegatee
Mini⟩ counter.Parity
false
```

☞ Die Parität wird korrekt nachgehalten.

43. Delegation: Schrittzähler — da capo

Einige Zeit später optimiert Harry Hacker den Konstruktor *step-counter*:

```
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

  }
```

43. Delegation: Demo

Die Kontrollausgaben ändern sich:

```
Mini⟩ let counter = parity-counter ()
Mini⟩ counter.Step 3
Step: delegator
Step: delegatee
Mini⟩ counter.Parity
false
```

☞ Die Parität wird weiterhin korrekt nachgehalten.

43. Delegation: Fazit

Wie erklären sich die Unterschiede zwischen den beiden Ansätzen?

- *Vererbung:* Es 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.
- *Delegation:* Es existieren *zwei* Objekte, der Paritätszähler und der Schrittzähler, die in nachvollziehbarer Weise miteinander interagieren: Einer delegiert Arbeit an den anderen.

☞ *Fazit:* virtuelle Methoden und Unterklassen sollten mit Bedacht eingesetzt werden.

43. Paradigmen: objektorientierte Programmierung

obiectare
entgegenwerfen; preisgeben; vorwerfen, vorhalten.

As if my mother, instead of saying: „Please bring a chair to the table“, had said „please, send a message to an instance of the class chair, which is a subclass of class furniture, to be brought to the very instance of the class table, being also a subclass of class furniture.“
— László Böszörményi

43. Paradigmen

To know another language is to have a second soul.
— Charlemagne (742–814)

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

Überblick:

- *funktionale Programmierung:* deskriptiv, problemnah, werte-fokussiert;
- *imperative Programmierung:* präskriptiv, maschinennah, effekt-fokussiert;
- *objektorientierte Programmierung:* Programmierung im Großen.

☞ Die Paradigmen konkurrieren nicht; sie ergänzen sich sinnvoll.


murabeho
adeus
再见
auf wiedersehen
goodbye
au revoir
görüşmek üzere
¡hasta la vista!
arrivederci
До свидания!
do widzenia
TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN