Prof. Dr. Ralf Hinze
Cass Alexandru, M.Sc.
Alexander Dinges, M.Sc.
Felix Winkler, M.Sc.

RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 5: Konzepte der Programmierung (WS 2025/26)

Listen in F# Auf dem letzten Übungsblatt haben wir bereits den Typ `Nats` für Listen natürlicher Zahlen verwendet. In der Vorlesung haben wir nun auch parametrisierte Typen kennengelernt. Unter anderem wurde der folgende Typ für Listen definiert: `type List<'a> = | Nil | Cons of 'a * List<'a>`

Diese Typdefinition kommt mit den bislang bekannten Sprachelementen aus: Rekursive Varianten und Paare. Wir können diese Typdefinition auch genauso in F# verwenden und z.B. die Liste der Zahlen `1N` und `2N` als `List<Nat>` durch `Cons (1N, Cons (2N, Nil))` darstellen. In F# gibt es jedoch auch einen vordefinierten Typ für Listen, für den es drei Notationen gibt: `List<'a>`, `list<'a>` sowie `List<'a>` (unglücklicherweise schreibt sich der in F# eingebaute Typ `List<'a>` gleich wie der Listentyp aus der Vorlesung, jedoch hat der F# Typ die Konstruktoren `[]` und `::`). Dieser Typ ist nicht kompatibel mit dem selbst definierten Typ `List<'a>` aus der Vorlesung. Es ist daher wichtig zu wissen, welchen Typ man gerade verwendet. Die Unterschiede finden Sie auf Vorlesungsfolie 402.

In den Übungen verwenden wir überwiegend den in F# vordefinierten Typ für Listen. Dies hat den Vorteil, dass wir eine kürzere Notation für die Darstellung der Listen nutzen können: `[1N; 2N; 3N]` statt `Cons (1N, Cons (2N, Cons (3N, Nil)))`. Außerdem werden wir einige vordefinierte Funktionen (sogenannte Bibliotheksfunktionen) kennenlernen, die auch nur mit dem vordefinierten Listen-Typ funktionieren.

Typparameter Einschränkungen in F# In der Vorlesung haben wir polymorphe Funktionen kennengelernt. Der Typ enthält hierbei einen Typparameter, sodass die Funktion für verschiedene Typen nutzbar wird. Oft ist man jedoch nicht ganz frei in der Wahl des Typs: Wenn Elemente des Typs miteinander verglichen werden (=, <, <=, usw.), dann muss der Typ diese Vergleichsoperation unterstützen. Eine polymorphe `contains` Funktion, die überprüft ob ein gegebener Wert vom Typ `'a` in einer Liste vom Typ `List<'a>` enthalten ist, setzt voraus, dass der Typ `'a` die Gleichheit unterstützt. Die meisten Typen unterstützen sowohl Gleichheit als auch Ordnungsvergleiche; Strings und Listen sind beispielweise lexikografisch geordnet. Es gibt aber auch Typen, die diese Vergleichsoperationen nicht unterstützen. Das sind insbesondere die Funktionstypen. Der Ausdruck `(fun (x: Nat) -> x + x) = (fun (x: Nat) -> 2N * x)` ist also nicht wohlgetypt, da der Typ `Nat -> Nat` die Gleichheit nicht unterstützt. Der Typparameter für polymorphe Funktionen kann wie folgt eingeschränkt werden: `let contains<'a when 'a : equality> (x: 'a) (xs: List<'a>): Bool = ...`. Dabei ist `equality` die Einschränkung, dass die Gleichheit (=) unterstützt werden muss. Für Ordnungsvergleiche (<, <=, `min`, ...) heißt die Einschränkung `comparison` und beinhaltet automatisch auch die Gleichheit. Sie brauchen sich nicht weiter damit zu befassen, jedoch werden manche Vorlagen derartige Typeinschränkungen enthalten. Diese müssen Sie so in der Vorlage stehen lassen, da der Code ansonsten nicht mehr kompilieren wird.

Aufrufen von polymorphen Funktionen in F# Wie auf Vorlesungsfolie 398 beschrieben, kann der Typparameter beim Aufrufen polymorpher Funktionen meist weggelassen werden. Allerdings gilt dies nicht immer, wenn der Typparameter eingeschränkt ist (siehe vorheriger Abschnitt). Die vordefinierte Funktion `max<'a when 'a : comparison>: 'a -> 'a -> 'a` gibt das größere der beiden Argumente zurück. Wir können `max<List<Nat>> [5N; 6N] [7N; 1N]` ohne Typparameter aufrufen (`max [5N; 6N] [7N; 1N]`), da F# den Typparameter `List<Nat>` aus den Argumenten bestimmen kann. Sind die beiden Eingabelisten jedoch leer, dann ist diese Verkürzung nicht möglich: `max [] []` gibt einen Fehler, während `max<List<Nat>> [] []` funktioniert. Wenn Sie also den Fehler `FS0030: Value restriction` erhalten, müssen Sie beim Funktionsaufruf den Typparameter explizit angeben.

Vorbereitung auf die Klausur Wir legen Ihnen ans Herz, mit der Klausurvorbereitung rechtzeitig zu beginnen. Sie können sich mit Hilfe der alten GdP Klausuren im KAI System[1] einen Eindruck vom Aufbau der Klausur verschaffen.

Als Hilfsmittel für die Klausuren sind zwei beidseitig handschriftlich beschriebene DIN A4 Blätter zugelassen. Beginnen Sie möglichst schon jetzt damit diese vorzubereiten. Schreiben Sie Dinge auf, die Sie nicht auswendig lernen möchten, aber dennoch hilfreich bei der Bearbeitung von Klausuraufgaben sein könnten. Dies sind zum Beispiel die Regeln der statischen und dynamischen Semantik. Ansonsten könnten noch die Parameter- und Rückgabetypen einiger nützlicher Bibliotheksfunktionen, die Sie zum Lösen der Übungsaufgaben bereits benutzt haben, hilfreich sein. Beachten Sie, dass bereits das Erstellen dieser "Spickzettel" einen Lernprozess darstellt. Sie sollten sich also Ihre eigenen Blätter konzipieren und nicht von Kommilitoninnen und Kommilitonen abschreiben.

Aufgabe 1 Parametrische Listen (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit den in F# eingebauten parametrischen Listen vertraut machen. Sie können sich an den Vorlesungsfolien 378 bis 403 sowie am Skript Kapitel 4.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Lists.fs` *aus der Vorlage* `Aufgabe-5-1.zip`.

Wir betrachten unter anderem einige aus Übungsblatt 4, Aufgabe 1 und 3 bekannte Funktionen noch einmal und verallgemeinern diese. Dazu werden die in F# eingebauten parametrischen Listen verwendet. Bitte beachten Sie die Hinweise zu Listen in F# auf der ersten Seite.

Hinweis: Verwenden Sie in Ihrer Lösung ***nicht*** *das List-Modul aus der Standardbibliothek.*

Wir verwenden bei einigen Teilaufgaben folgende Beispielliste:

```
let ex = [2N; 4N; 3N; 4N; 2N; 1N]
```

a) Schreiben Sie eine Funktion `plusOne: List<Nat> -> List<Nat>`, die eine Liste natürlicher Zahlen nimmt und zu jeder Zahl in der Liste die Zahl `1` addiert. Vergleichen Sie mit der gleichnamigen Funktion von Übungsblatt 4, Aufgabe 1.

Beispiele:

```
plusOne [] = []
plusOne ex = [3N; 5N; 4N; 5N; 3N; 2N]
```

```
let rec plusOne (xs: List<Nat>): List<Nat> =
    match xs with
    | []    -> []
    | x::ys -> (x + 1N)::(plusOne ys)
```

Gegenüber Aufgabe 1 von Übungsblatt 4 müssen nur die Konstruktoren ausgetauscht werden. Statt `Nil` haben wir jetzt `[]` und anstelle von `Cons (x, xs)` (Präfix) schreiben wir `x::xs` (Infix).

[1] https://kai.informatik.uni-kl.de/, Abruf nur aus dem Uni-Netz bzw. VPN https://rz.rptu.de/vpn/.

b) Schreiben Sie eine Funktion `filter<'a>: ('a -> Bool) -> List<'a> -> List<'a>`, die eine Funktion `p` und eine Liste `xs` nimmt und die Liste der Elemente aus `xs` zurückgibt, für die `p` **`true`** zurückgibt.

Beispiele:

```
filter (fun x -> x > 3N) [] = []
filter (fun x -> x > 3N) ex = [4N; 4N]
filter (fun x -> x <= 3N) ex = [2N; 3N; 2N; 1N]
```

```
let rec filter<'a> (p: 'a -> Bool) (xs: List<'a>): List<'a> =
    match xs with
    | []    -> []
    | x::xs ->
        if p x
          then x::(filter p xs)
          else filter p xs
```

c) Schreiben Sie eine Funktion `concat<'a>: List<'a> -> List<'a> -> List<'a>`, die zwei parametrische Listen `xs` und `ys` nimmt und deren Konkatenation berechnet, also die Liste in der zuerst alle Elemente aus `xs` und dann die Elemente aus `ys` kommen. Verwenden Sie nicht den in F# eingebauten Konkatenationsoperator `@`.

Beispiele:

```
concat [] ex = ex
concat ex [] = ex
concat [1N] [2N] = [1N; 2N]
```

```
let rec concat<'a> (xs: List<'a>) (ys: List<'a>): List<'a> =
    match xs with
    | [] -> ys
    | x::zs -> x::(concat zs ys)
```

d) Schreiben Sie eine Funktion `mirror<'a>: List<'a> -> List<'a>`, die eine Liste nimmt und die gespiegelte Liste berechnet, also eine Liste in der die Elemente in umgekehrter Reihenfolge enthalten sind.

Beispiele:

```
mirror [] = []
mirror ex = [1N; 2N; 4N; 3N; 4N; 2N]
```

```
// Lösung mit Laufzeit quadratisch in der Länge von xs
let rec mirror<'a> (xs: List<'a>): List<'a> =
    match xs with
    | [] -> []
    | x::ys -> concat (mirror ys) [x]

// Effizientere Lösung (lineare Laufzeit)
let mirror'<'a> (xs: List<'a>): List<'a> =
    // Hilfsfunktion berechnet concat (mirror xs) zs
    let rec mirrorConcat (xs: List<'a>) (zs: List<'a>): List<'a> =
        match xs with
        | [] -> zs
        | x::ys -> mirrorConcat ys (x::zs)
    mirrorConcat xs []
```

e) Schreiben Sie eine Funktion `sum: List<Nat> -> Nat`, die eine Liste natürlicher Zahlen nimmt und die Summe der Zahlen zurückgibt.

Beispiele:

```
sum [] = 0N                                    sum ex = 16N
```

```
let rec sum (xs: List<Nat>): Nat =
    match xs with
    | [] -> 0N
    | x::ys -> x + sum ys
```

Aufgabe 2 Warteschlangen (Einreichaufgabe, 12 Punkte)

Motivation: In dieser Aufgabe sollen Sie das Programmieren mit Records und parametrisierten Typen einüben. Sie können sich an den Vorlesungsfolien 280 bis 304 und 378 bis 403 sowie am Skript Kapitel 4.1 und 4.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Queue.fs` *aus der Vorlage* `Aufgabe-5-2.zip`.

Eine Warteschlange (*engl. queue*) ist eine Datenstruktur, welche eine Sammlung von Elementen verwaltet und es in möglichst effizienter Weise erlaubt, sowohl Elemente ans Ende anzufügen als auch vom Anfang zu entfernen.

Würden wir eine einfache Liste als Warteschlange verwenden, müssten wir für eine der genannten Operationen ganz durch die Liste laufen. Um ein besseres Laufzeitverhalten zu erzielen, modellieren wir in dieser Aufgabe Warteschlangen mit Hilfe zweier Listen. Dabei repräsentiert die erste Liste die Vorderseite der Warteschlange, ihr erstes Element steht ganz vorne in der Warteschlange. Die zweite Liste repräsentiert das Ende der Warteschlange in umgekehrter Reihenfolge, ihr erstes Element korrespondiert also zum letzten Element der Warteschlange. Die gesamte Warteschlange erhält man, wenn man beide Listen aneinander hängt und eine dabei spiegelt.

Die Warteschlange selbst wird durch den Typ `DEQ<'a>` (für engl. *double ended queue*) modelliert.

```
type DEQ<'a> = // Die Warteschlange
    { frontLength: Nat
      front: List<'a>
      rearLength: Nat
      rear: List<'a>
    }
```

Wie Sie sehen, merken wir uns zusätzlich die Längen beider Listen. **Wir legen fest, dass die hintere Liste nie mehr Elemente enthalten darf, als die vordere Liste.** Wäre dies beim Anfügen eines neuen Elements der Fall, so hängen wir die Elemente der hinteren Liste in umgekehrter Reihenfolge an das Ende der vorderen Liste an. Dadurch folgt, dass die vordere Liste nur leer ist, wenn auch die hintere Liste leer ist.

Hinweis: Beachten Sie die Hinweise auf der ersten Seite.

Hinweis: Wenn Sie möchten, können Sie in Ihrer Lösung das List-Modul aus der Standardbibliothek verwenden.

a) Schreiben Sie eine Funktion `isEmpty<'a>: DEQ<'a> -> Bool`, die eine Warteschlange als Argument erwartet und zurückgibt, ob die Warteschlange leer ist oder nicht.

```
let isEmpty<'a> (q: DEQ<'a>): Bool =
    q.frontLength = 0N
```

Aufgrund der im Aufgabentext genannten Invariante genügt es zu prüfen, ob die vordere Liste leer ist. Dies ist der Fall, wenn diese die Länge `0N` hat. Wir können alternativ mit Hilfe von **`match`** eine entsprechende Fallunterscheidung der Liste selbst durchführen. Wenn wir versuchen mit `q.front = []` zu prüfen, ob die Liste leer ist, erhalten wir die Fehlermeldung `FS0001: Einem Typparameter fehlt die Einschränkung "when 'a : equality"`. Die Prüfung auf Gleichheit ist nur möglich, wenn dies explizit für den Typparameter gefordert wird (s. Seite 1).

b) Schreiben Sie eine Funktion `repair<'a>: DEQ<'a> -> DEQ<'a>`, die prüft, ob die Warteschlange die Invariante erfüllt. Ist dies der Fall, wird die Warteschlange unverändert zurückgegeben. Andernfalls wird die Invariante hergestellt und die resultierende Warteschlange zurückgegeben.

Hinweis: Sie müssen nicht extra die Längen der Listen mit `frontLength` und `rearLength` abgleichen (indem Sie z.B. `q.front.Length` verwenden). Stattdessen werden die Längen `frontLength` und `rearLength` in jeder manipulierenden Operation so angepasst, dass sie die Längen der Listen korrekt angeben.

```
let rec repair<'a> (q: DEQ<'a>): DEQ<'a> =
    if q.rearLength <= q.frontLength then
        q
    else
        { frontLength = q.frontLength + q.rearLength
        ; front = q.front @ List.rev q.rear
        ; rearLength = 0N
        ; rear = []}
```

Wenn die Invariante erfüllt ist (die hintere Liste ist kürzer oder gleich lang wie die vordere Liste), wird die Warteschlange unverändert zurückgegeben. Ist die Invariante verletzt, so wird die hintere Liste in umgekehrter Reihenfolge an die vordere Liste angehängt. Die Längen werden entsprechend angepasst.

c) Schreiben Sie eine Funktion `enqueue<'a>: 'a -> DEQ<'a> -> DEQ<'a>`, welche ein Element `x` sowie eine Warteschlange `q` nimmt. Das Element `x` soll am Ende der Warteschlange `q` eingefügt und die resultierende Warteschlange als Ergebnis zurückgegeben werden.

Hinweis: Verwenden Sie die `repair` Funktion.

```
let rec enqueue<'a> (x: 'a) (q: DEQ<'a>): DEQ<'a> =
    repair { q with rearLength = q.rearLength + 1N ; rear = x::q.rear}
```

Da sich das letzte Element der Warteschlange ganz vorne in der `rear` Liste befindet, können wir `x` einfach dort anfügen. Die Länge der hinteren Liste wird entsprechend um 1 erhöht. Die Schreibweise `{q with ...}` erlaubt uns, ein neues Record zu konstruieren, welches standardmäßig mit den Inhalten aus `q` befüllt wird, aber an den in `...` genannten Feldern geändert wird (damit sparen wir uns etwas Schreibarbeit). Zu guter Letzt verwenden wir `repair`, um die Einhaltung der Invariante sicherzustellen.

d) Schreiben Sie eine Funktion `dequeue<'a>: DEQ<'a> -> Option<'a * DEQ<'a>>`, die das vorderste Element der Warteschlange entnimmt. Zurückgegeben wird ein Paar bestehend aus diesem vordersten Element sowie der restlichen Warteschlange. Wenn die Warteschlange leer ist gibt es kein vorderstes Element; daher kommt der `Option`-Typ zum Einsatz, sodass dann `None` zurückgegeben werden kann.

Hinweis: Sie haben den Optionstyp `Option<'a>` in der Vorlesung auf Folie 398 bzw. im Skript auf Seite 135 kennengelernt. Tatsächlich ist dieser Typ genau so in F# bereits eingebaut.

```
let dequeue<'a> (q: DEQ<'a>): Option<'a * DEQ<'a>> =
    match q.front with
    | [] -> None
    | y::ys -> Some (
        y,
        repair {q with frontLength = q.frontLength - 1N; front = ys}
    )
```

Aufgrund der Invariante genügt es, die vordere Liste zu analysieren. Ist diese leer, so gibt die Funktion `None` zurück. Ansonsten verwenden wir wieder die Idee aus der vorherigen Teilaufgabe: Das vorderste Element wird entfernt, die Länge entsprechend angepasst und schließlich wird mit `repair` die Einhaltung der Invariante sichergestellt.

Aufgabe 3 Ausdrücke vereinfachen (Einreichaufgabe, 3 Punkte)

Motivation: Wir haben in den Klausuren die Erfahrung gemacht, dass Studierende häufig unnötig komplexe Ausdrücke schreiben. Einerseits vermindert ein solch komplexer Ausdruck die Lesbarkeit, andererseits kostet er wertvolle Zeit beim Aufschreiben. Daher wollen wir zu diesem frühen Zeitpunkt schon einüben, wie man gängige Ausdrücke möglichst kurz darstellen kann.

Schreiben Sie Ihre Lösungen in die Datei `Simplify.fs` *aus der Vorlage* `Aufgabe-5-3.zip`.

Geben Sie für die folgenden Ausdrücke jeweils einen vereinfachten (also möglichst kurzen) Ausdruck an, der auf jeden Fall zu demselben Wert wie der ursprüngliche Ausdruck auswertet.

Beispiel: `false = (a = true)` lässt sich vereinfachen zu `not a`.

a) `if a then b else false`

```
a && b
```

b) `if (a = true) then 2N else 3N`

```
if a then 2N else 3N
```

c) `if (x <> 0N) then false else true`

```
x = 0N
```

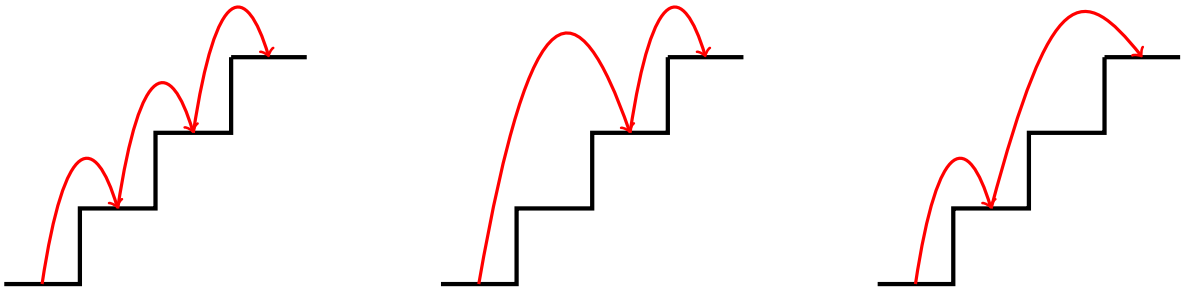
Aufgabe 4 Stufenproblem (Einreichaufgabe, 12 Punkte)

Motivation: In dieser Aufgabe sollen Sie mit den in F# eingebauten parametrischen Listen ein komplexeres Problem lösen.

Schreiben Sie Ihre Lösungen in die Datei `Steps.fs` *aus der Vorlage* `Aufgabe-5-4.zip`.

Wir betrachten eine Treppe mit n Stufen und nehmen zunächst an, dass wir in einem Schritt entweder eine oder zwei Stufen hinaufgehen können. Damit ergeben sich verschiedene Schrittfolgen, wie wir die Treppe hinaufsteigen können. In der folgenden Abbildung sind alle Möglichkeiten für $n = 3$, also drei Stufen, dargestellt.

Hinweis: Sie dürfen das List-Modul aus der Standardbibliothek verwenden.



`[1N;1N;1N]` `[2N;1N]` `[1N;2N]`

a) Schreiben Sie eine Funktion `findSteps12`, die alle möglichen Schrittfolgen einer `n`-stufigen Treppe berechnet. In einem Schritt dürfen Sie eine oder zwei Treppenstufen weit hinaufsteigen.

Tipp: Verkleinern Sie das Problem und lösen Sie es dann durch rekursive Aufrufe. Überlegen Sie sich dazu zuerst die Möglichkeiten für einen ersten Schritt. Danach benötigen Sie eine (kleinere) Schrittfolge, um auf die insgesamt gewünschte Anzahl an Schritten zu kommen. Diese beiden Teillösungen (erster Schritt und die kleinere Schrittfolge) müssen Sie dann zu einer Gesamtlösung zusammensetzen. Dazu brauchen Sie gegebenenfalls noch eine Hilfsfunktion.

```
let rec prepend<'a> (x: 'a) (xs: List<List<'a>>): List<List<'a>> =
    match xs with
    | [] -> []
    | y::ys -> (x::y)::(prepend x ys)

let rec findSteps12 (n: Nat): List<List<Nat>> =
    if      n = 0N then []
    else if n = 1N then [[1N]]
    else if n = 2N then [[1N; 1N]; [2N]]
    else (prepend 1N (findSteps12 (n-1N))) @ (prepend 2N (findSteps12 (n-2N)))
```

Wir unterscheiden in `findSteps12` drei Basisfälle:

- Wenn `n = 0N`, gibt es keine Stufen, wir geben eine leere Liste zurück.
- Für `n = 1N` gibt es genau eine mögliche Lösung.
- Für `n = 2N` geben wir die beiden möglichen Lösungen zurück.

Wenn `n > 2N` ist, können wir sowohl noch einen, als auch zwei Schritte gehen. Diese beiden Möglichkeiten fügen wir jeweils an die Teillösungen der rekursiven Aufrufe an, die das Stufenproblem um eine bzw. zwei Stufen kleinere Treppe lösen. Dazu definieren wir die rekursive Hilfsfunktion `prepend`.

b) Nun betrachten wir eine Treppe mit sehr kleinen Stufen. Sie können entweder eine, drei oder fünf Stufen in einem Schritt hinaufsteigen. Implementieren Sie die Funktion `findSteps135`, welche alle möglichen Schrittfolgen mit den genannten Schrittweiten berechnet.

```
let rec findSteps135 (n: Nat) : List<List<Nat>> =
    if      n = 0N then []
    else if n = 1N then [[1N]]
    else if n = 3N then [[1N;1N;1N]; [3N]]
    else if n = 5N then [[1N;1N;1N;1N;1N]; [1N;1N;3N]
                        ;[1N;3N;1N]; [3N;1N;1N]; [5N]]
    else (prepend 1N (findSteps135 (n-1N))) @
         (prepend 3N (findSteps135 (n-3N))) @
         (prepend 5N (findSteps135 (n-5N)))
```

Wir verwenden dieselbe Idee wie in Teilaufgabe a) und passen die Basisfälle und rekursiven Aufrufe entsprechend an. Für jede der Schrittweiten schreiben wir einen Basisfall und einen rekursiven Aufruf. Für Stufenzahlen zwischen den Schrittweiten, also 2 und 4, brauchen wir keine gesonderten Basisfälle. Für `n = 2N` gibt nur der um eins kleinere rekursive Aufruf eine nicht leere Liste zurück, sodass `prepend` nur für diesen Fall eine nichtleere Liste zurückgibt (wir erhalten als Ergebnis `(prepend 1N [[1N]]) @ (prepend 3N []) @ (prepend 5N []) = [[1N; 1N]]`). Der Fall `n = 4N` funktioniert analog.

c) Da es sich hier um einen etwas umfangreicheren Algorithmus handelt, erwarten wir, dass Sie Ihren Code so kommentieren, dass Ihre Lösung einfach nachzuvollziehen ist. Außerdem sollte Ihr Code keine unnötig komplexen Ausdrücke enthalten, siehe auch Aufgabe 3. Dafür vergeben wir hier zwei Punkte.

d) *Freiwillige Zusatzaufgabe:* Schreiben Sie eine Funktion `findSteps`, die zusätzlich eine Liste von Schrittweiten erwartet und entsprechend alle damit möglichen Schrittfolgen berechnet.

```
let rec contains<'a when 'a: equality> (x: 'a) (xs: List<'a>): Bool =
    match xs with
    | [] -> false
    | y::ys -> y = x || contains x ys

let rec findSteps (n: Nat) (stepSizes: List<Nat>): List<List<Nat>> =
    let rec h (xs: List<Nat>): List<List<Nat>> =
        match xs with
        | [] -> []
        | x::xs -> (prepend x (findSteps (n-x) stepSizes)) @ (h xs)
    if n = 0N then []
    else
        let res = h stepSizes
        if contains n stepSizes then [n]::res else res

// Mit dem List Modul: List.collect f xs = List.concat (List.map f xs)
let rec findSteps' (n: Nat) (stepSizes: List<Nat>): List<List<Nat>> =
    if n = 0N then []
    else
        let res = List.collect (fun s -> prepend s (findSteps' (n-s) stepSizes))
                               stepSizes
        if List.contains n stepSizes then [n]::res else res
```

Der Basisfall für `n = 0N` bleibt bestehen. Für `n > 0N` rufen wir `findSteps` zunächst für alle um `stepSizes` kleineren Teilprobleme rekursiv auf und konkatenieren die Teilergebnisse. In der als `findSteps` bezeichneten Lösung wird dies mit Hilfe einer rekursiven Hilfsfunktion umgesetzt. Dagegen verwendet die als `findSteps'` bezeichnete Lösung Funktionen des List-Moduls und ist entsprechend etwas kürzer.

Falls es sich bei `n` um eine in `stepSizes` definierte Schrittweite handelt (das prüfen wir mit `contains` bzw. `List.contains`), fügen wir dies als zusätzliche Lösung an die Ergebnisliste an.

Aufgabe 5 Balanciertes Ternärsystem (Trainingsaufgabe)

Motivation: In dieser Aufgabe sollen Sie mit Listen und Variantentypen arbeiten. Sie können sich an den Vorlesungsfolien 306 bis 331 und 379 bis 401 bzw. am Skript Kapitel 4.2 und 4.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Ternary.fs` *aus der Vorlage* `Aufgabe-5-5.zip`.

Zur Zahlendarstellung verwenden wir in dieser Aufgabe ein balanciertes ternäres Stellenwertsystem, welches wir mit Hilfe von Listen modellieren. Als Ziffern werden `M` („minus eins“), `Z` („zero“ bzw. „null“) und `P` („plus eins“) verwendet.

```
type Ternary = | M | Z | P // -1, 0, 1
```

Wir legen fest, dass die niederwertigste Ziffer vorne in der Liste steht und entsprechend die höchstwertige Ziffer am Ende der Liste. Damit repräsentiert die Liste `[M; Z; M; P]` die Zahl $(-1)\cdot 3^0+0\cdot 3^1+(-1)\cdot 3^2+1\cdot 3^3 = 17$. Wir brauchen im balancierten Ternärsystem kein Vorzeichen, um negative Zahlen darzustellen.

Beachten Sie, dass die Darstellung einer Zahl aufgrund von führenden Nullen nicht eindeutig ist. Teilaufgabe b) stellt eine Hilfsfunktion bereit, mit der Sie das Problem in den darauffolgenden Teilaufgaben umgehen können.

Weitere Beispiele:

```
[M; P; M] // -7     [M; M] // -4      [M] // -1          [M; P] // 2       [M; M; P] // 5
[Z; P; M] // -6     [Z; M] // -3      []  //  0          [Z; P] // 3       [Z; M; P] // 6
[P; P; M] // -5     [P; M] // -2      [P] //  1          [P; P] // 4       [P; M; P] // 7
```

Hinweis: Wir verwenden in dieser Aufgabe den Typ `Int` der ganzen Zahlen. Zahl-Literale dieses Typs haben keinen `N`-Suffix, die Zahl 42 ist also einfach `42` und -42 ist `-42`.

a) Schreiben Sie eine Funktion `bedeutung: List<Ternary> -> Int`, die für eine gegebene Repräsentation im ternären Stellenwertsystem die entsprechende ganze Zahl berechnet.

```
let rec bedeutung (n: List<Ternary>): Int =
    match n with
    | [] -> 0
    | M::ns -> 3 * bedeutung ns - 1
    | Z::ns -> 3 * bedeutung ns // + 0
    | P::ns -> 3 * bedeutung ns + 1
```

b) Implementieren Sie die Funktion `zCons` (einen „smarten Konstruktor“), die eine Null (`Z`) an eine Zahl im balancierten Ternärsystem anhängt, sofern deren Darstellung nicht der leeren Liste entspricht. Verwenden Sie diesen smarten Konstruktor in den folgenden Teilaufgaben, sofern Sie ein `Z` an eine Zahl im balancierten Ternärsystem anfügen möchten.

Hinweis: Es genügt, wenn mit `zCons` nur der Fall behandelt wird, dass die übergebene Liste leer ist. Sie müssen nicht prüfen, ob es weitere führende Nullen gibt. Damit ist z. B. `zCons [Z] = [Z; Z]`. Allerdings tritt dieser Fall nicht auf, wenn statt `Z::` stets `zCons` verwendet wird. Für oben genanntes Beispiel erhalten wir also mit `zCons (zCons []) = []` das erwartete Ergebnis.

```
let zCons (ns: List<Ternary>): List<Ternary> =
    match ns with
    | [] -> []
    | _ -> Z::ns
```

c) Schreiben Sie eine Funktion `inc`, die eine Zahl im balancierten Ternärsystem um den Wert eins erhöht.

```
let rec inc (n: List<Ternary>): List<Ternary> =
    match n with
    | [] -> [P]
    | M::ns -> zCons ns
    | Z::ns -> P::ns
    | P::ns -> M::(inc ns)
```

d) Schreiben Sie eine Funktion `dec`, die eine Zahl im balancierten Ternärsystem um den Wert eins verringert.

```
let rec dec (n: List<Ternary>): List<Ternary> =
    match n with
    | [] -> [M]
    | M::ns -> P::(dec ns)
    | Z::ns -> M::ns
    | P::ns -> zCons ns
```

e) Schreiben Sie eine Funktion `fromInt: Int -> List<Ternary>`, die eine ganze Zahl ins balancierte Ternärsystem überführt. Orientieren Sie sich am Leibniz Entwurfsmuster.

```
let rec fromInt (n: Int): List<Ternary> =
    if n = 0 then []
    else if n % 3 =  2 then M::(inc (fromInt (n/3)))
    else if n % 3 =  1 then P::(fromInt (n/3))
    else if n % 3 = -1 then M::(fromInt (n/3))
    else if n % 3 = -2 then P::(dec (fromInt (n/3)))
    else (* n % 3 =  0 *)   zCons (fromInt (n/3))
```

f) Schreiben Sie eine Funktion `add: List<Ternary> -> List<Ternary> -> List<Ternary>`, die zwei Zahlen im balancierten Ternärsystem addiert.

```
let rec add (m: List<Ternary>) (n: List<Ternary>): List<Ternary> =
    match (m, n) with
    | ([], x) | (x, []) -> x
    | (M::ms, M::ns) -> P :: (add (dec ms) ns)
    | (P::ms, P::ns) -> M :: (add (inc ms) ns)
    | (M::ms, Z::ns) | (Z::ms, M::ns)                  -> M :: (add ms ns)
    | (M::ms, P::ns) | (P::ms, M::ns) | (Z::ms, Z::ns) -> zCons (add ms ns)
    | (P::ms, Z::ns) | (Z::ms, P::ns)                  -> P :: (add ms ns)
```

g) Schreiben Sie eine Funktion `negative: List<Ternary> -> List<Ternary>`, die das Vorzeichen einer Zahl im balancierten Ternärsystem umkehrt.

```
let rec negative (n: List<Ternary>): List<Ternary> =
    match n with
    | [] -> []
    | M::ns -> P::(negative ns)
    | Z::ns -> zCons (negative ns)
    | P::ns -> M::(negative ns)
```