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 4: Konzepte der Programmierung (WS 2025/26)

Klausuranmeldung Denken Sie daran, die Klausur und die Vorleistung im Prüfungsamt anzumelden!

Aufgabe 1 Datentypen (Präsenzaufgabe)

Motivation: Um Programmieraufgaben im ExClaim-System testen zu können, müssen wir die darin verwendeten Typdefinitionen vorgeben. In dieser Aufgabe sollen Sie (rekursive) Varianten und Records wiederholen und selbst entsprechende Typen definieren, um Sachverhalte zu modellieren. Sie können sich an den Vorlesungsfolien 280 bis 372 sowie am Skript Kapitel 4.1 und 4.2 orientieren.

a) Varianten

1. Definieren Sie einen Variantentyp zur Modellierung von Tierarten, der die Ausprägungen `Hund`, `Katze` und `Maus` annehmen kann.

   Zusätzlich soll ein Variantentyp für Lebewesen definiert werden: Bei einem Lebewesen kann es sich entweder um ein Tier einer der oben genannten Tierarten handeln, oder um ein Lebewesen, das kein Tier ist.

```
type Tierart =
    | Hund
    | Katze
    | Maus

type Lebewesen =
    | KeinTier
    | Tier of Tierart
```

2. Schreiben Sie eine Funktion `eineMaus: Tierart -> Bool`, die prüft, ob es sich bei der übergebenen Tierart um eine Maus handelt.

```
let eineMaus (t: Tierart): Bool =
    match t with
    | Maus -> true
    | _ -> false
```

3. Schreiben Sie eine Funktion `eineKatze: Lebewesen -> Bool`, die prüft, ob es sich beim übergenenen Lebewesen um eine Katze handelt.

```
let eineKatze (l: Lebewesen): Bool =
    match l with
    | KeinTier -> false
    | Tier t ->
        match t with
        | Katze -> true
        | _ -> false

let eineKatze' (l: Lebewesen): Bool =
    match l with
    | Tier Katze -> true
    | _ -> false
```

4. Schreiben Sie eine Funktion `mindestensEinTier: Lebewesen -> Lebewesen -> Bool`, die prüft, ob es sich bei mindestens einem der beiden Argumente um ein Tier handelt.

```
let mindestensEinTier (l1: Lebewesen) (l2: Lebewesen): Bool =
    match (l1, l2) with
    | (KeinTier, KeinTier) -> false
    | _ -> true
```

b) Rekursive Varianten

1. Wiederholen Sie den Typ `Nats` für Listen natürlicher Zahlen

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

2. Schreiben Sie eine Funktion `findMax`, welche die größte Zahl in einer Liste natürlicher Zahlen zurückgibt.

```
let rec findMax (ns: Nats): Nat =
    match ns with
    | Nil -> 0N
    | Cons (x, xs) -> max x (findMax xs)
```

3. Schreiben Sie eine Funktion `plusOne`, die alle Zahlen in der Liste um eins erhöht.

```
let rec plusOne (ns: Nats): Nats =
    match ns with
    | Nil -> Nil
    | Cons (x, xs) -> Cons (x + 1, plusOne xs)
```

c) Records

1. Schreiben Sie einen Record-Typ, um Eigenschaften von Büchern zu speichern. Gesichert werden soll der Titel, der Name der Autorin/des Autors, das Veröffentlichungsjahr sowie die ISBN.

   Verwenden Sie den Record, um das Buch „Harry Potter and the Philosopher's Stone“ von „J. K. Rowling“ aus dem Jahr 1997 mit der ISBN 9780747532743 zu erfassen.

```
type Buch = {titel: String ; autor: String ; jahr: Nat ; isbn: Nat}
let harry = { titel="Harry Potter and the Philosopher's Stone"
            ; autor="J. K. Rowling" ; jahr = 1997N ; isbn = 9780747532743N }
```

2. Wo liegt der Vorteil gegenüber einem entsprechenden Quadrupel?

```
type Buch' = String * String * Nat * Nat
```

   Hier müssen wir uns merken, welche Information an welcher Position steht. Durch die Angabe von Labels ist die Reihenfolge bei Records egal. Aussagekräftige Labels schaffen zusätzliche Klarheit beim Lesen und Schreiben des Codes.

Aufgabe 2 Kalenderdaten (Einreichaufgabe, 7 Punkte)

Motivation: In dieser Aufgabe sollen Sie sich mit Variantentypen und Records beschäftigen. Sie können sich an den Vorlesungsfolien 280 bis 326 sowie am Skript Kapitel 4.1 und 4.2 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Dates.fs` *aus der Vorlage* `Aufgabe-4-2.zip`.

Wir repräsentieren ein Kalenderdatum durch folgende Datentypen:

```
type Weekday = | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
type Date = { year: Nat; month: Nat; day: Nat; weekday: Weekday }
```

Ein Datum besteht also aus je drei Zahlen für das Jahr, den Monat und den Tag, sowie dem Wochentag.

Hinweis: Einige Teilaufgaben können Sie mit Hilfe der Funktionen aus vorherigen Teilaufgaben lösen.

a) Schreiben Sie eine Funktion `nextWeekday: Weekday -> Weekday`, die einen Wochentag nimmt und den nächsten Wochentag zurückgibt.

Beispiel:

```
nextWeekday Monday = Tuesday
```

```
let nextWeekday (d: Weekday): Weekday =
    match d with
    | Monday -> Tuesday
    | Tuesday -> Wednesday
    | Wednesday -> Thursday
    | Thursday -> Friday
    | Friday -> Saturday
    | Saturday -> Sunday
    | Sunday -> Monday
```

b) Schreiben Sie eine Funktion `isLeapYear: Nat -> Bool`, die eine natürliche Zahl nimmt, die eine Jahreszahl darstellen soll, und prüft, ob diese ein Schaltjahr[1] ist.

Beispiele:

```
isLeapYear 2025 = false
isLeapYear 2024 = true
```

```
let isLeapYear (y: Nat): Bool =
    (y % 4N = 0N) && (y % 100N <> 0N || y % 400N = 0N)
```

[1] https://de.wikipedia.org/wiki/Schaltjahr#Gregorianischer_Kalender

c) Schreiben Sie eine Funktion `daysInMonth: Nat -> Nat -> Nat`, die eine Jahreszahl sowie eine Monatszahl nimmmt und zurückgibt, wie viele Tage es im Monat gibt. Ist der Monatswert nicht sinnvoll, ist es egal, was zurückgegeben wird.

Beispiel:

```
daysInMonth 2023 2 = 28
daysInMonth 2024 2 = 29
daysInMonth 2025 11 = 30
```

```
let daysInMonth (year: Nat) (month: Nat): Nat =
    if month = 2N then if isLeapYear year then 29N else 28N
    elif month = 4N || month = 6N || month = 9N || month = 11N then 30N
    else 31N
```

d) Schreiben Sie eine Funktion `nextDate: Date -> Date`, die ein Datum nimmt und das nächste Datum zurückgibt. Sie müssen nicht prüfen, ob die Eingabe ein gültiges Datum ist.

Beispiel:

```
nextDate { year = 2025N; month = 11N; day = 18N; weekday = Tuesday } =
         { year = 2025N; month = 11N; day = 19N; weekday = Wednesday }
nextDate { year = 2025N; month = 12N; day = 31N; weekday = Wednesday } =
         { year = 2026N; month = 1N;  day = 1N;  weekday = Thursday }
```

```
let nextDate (d: Date): Date =
    let nextDay = nextWeekday d.weekday
    let nextDayNum = d.day + 1N
    if nextDayNum > daysInMonth d.year d.month then
        if d.month = 12N then
            { year = d.year + 1N; month = 1N; day = 1N; weekday = nextDay }
        else
            { year = d.year; month = d.month + 1N; day = 1N; weekday = nextDay }
    else
        { year = d.year; month = d.month; day = nextDayNum; weekday = nextDay }
```

e) Schreiben Sie eine Funktion `nextDateN: Date -> Nat -> Date`, die ein Datum und eine natürliche Zahl nimmt und das Datum zurückgibt, das nach der angegebenen Anzahl von Tagen folgt. Sie müssen nicht prüfen, ob die Eingabe ein gültiges Datum ist.

Tipp: Nutzen Sie das Peano-Entwurfsmuster und Ihre Funktion aus der vorherigen Teilaufgabe.

```
let rec nextDateN (d: Date) (n: Nat): Date =
    if n = 0N then d
    else nextDateN (nextDate d) (n - 1N)
```

f) *Freiwillige Zusatzaufgabe:* Schreiben Sie eine Funktion `validateWeekday: Date -> Bool option`, die ein Datum nimmt und prüft, ob der Wochentag mit dem Datum übereinstimmt. Sie soll darüber hinaus nichts prüfen.

Beispiel:

```
validateWeekday { year = 2025N; month = 11N; day = 18N; weekday = Tuesday } = true
validateWeekday { year = 2025N; month = 11N; day = 18N; weekday = Wednesday } = false
```

Wir verwenden die Formel von Wikipedia:

```
let rec numberToWeekday (x: Nat): Weekday =
    if x = 0N then Sunday
    elif x = 1N then Monday
    elif x = 2N then Tuesday
    elif x = 3N then Wednesday
    elif x = 4N then Thursday
    elif x = 5N then Friday
    elif x = 6N then Saturday
    else numberToWeekday (x % 7N)

let monthFactor (d: Date): Nat =
    let m = d.month
    if m = 3N then 2N
    elif m = 4N then 5N
    elif m = 5N then 0N
    elif m = 6N then 3N
    elif m = 7N then 5N
    elif m = 8N then 1N
    elif m = 9N then 4N
    elif m = 10N then 6N
    elif m = 11N then 2N
    elif m = 12N then 4N
    elif m = 1N then 0N
    elif m = 2N then 3N
    else 0N

let firstDigitsOfYear (d: Date): Nat =
    if d.month <= 2N then (d.year - 1N)/100N else d.year/100N

let lastDigitsOfYear (d: Date): Nat =
    if d.month <= 2N then (d.year - 1N)%100N else d.year%100N

let validateWeekday (d: Date): Bool =
    let y = lastDigitsOfYear d
    let c = firstDigitsOfYear d
    let m = monthFactor d
    let actualWdAsNumber = d.day + m + y + y / 4N + c / 4N - 2N*c
    let actualWeekday = numberToWeekday (actualWdAsNumber % 7N)
    actualWeekday = d.weekday
```

Aufgabe 3 Listen natürlicher Zahlen (Einreichaufgabe, 8 Punkte)

Motivation: In dieser Aufgabe sollen Sie sich mit rekursiven Variantentypen beschäftigen. Sie können sich an den Vorlesungsfolien 332 bis 351 sowie am Skript Kapitel 4.2.2 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Nats.fs` *aus der Vorlage* `Aufgabe-4-3.zip`.

Bevor wir nächste Woche den in F# eingebauten Typ für Listen kennenlernen, arbeiten wir diese Woche zunächst mit einem selbst definierten Typ für Listen natürlicher Zahlen. Alles was wir dazu brauchen, sind rekursive Varianten und Tupel. Sie kennen den Typ bereits aus der Vorlesung:

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

Hinweis: Wenn Sie im Internet nach Teilen der Aufgabenstellung suchen, werden Sie vielleicht auf das `List` *F#-Modul stoßen, das allerdings mit dem in F# eingebauten Typ für Listen arbeitet. Dieses Modul werden wir auf einem späteren Übungsblatt vorstellen, hier dürfen Sie es in Ihrer Lösung jedoch* ***nicht*** *verwenden.*

Wir verwenden bei einigen Teilaufgaben die Beispielliste:

```
let ex = Cons (2N, Cons (4N, Cons (3N, Cons(4N, Cons(2N, Cons (1N, Nil))))))
```

a) Schreiben Sie eine Funktion `concat: Nats -> Nats -> Nats`, die zwei Listen natürlicher Zahlen `xs` und `ys` nimmt und deren Konkatenation berechnet, also die Liste in der zuerst alle Zahlen aus `xs` und dann die Zahlen aus `ys` kommen.

Beispiele:

```
concat Nil ex = ex
concat ex Nil = ex
concat (Cons (1N, Nil)) (Cons (2N, Nil)) = Cons (1N,Cons (2N,Nil))
```

```
let rec concat (xs: Nats) (ys: Nats): Nats =
    match xs with
    | Nil -> ys
    | Cons (x, zs) -> Cons (x, concat zs ys)
```

b) Schreiben Sie eine Funktion `mirror: Nats -> Nats`, die eine Liste natürlicher Zahlen nimmt und die gespiegelte Liste berechnet, also eine Liste in der die Zahlen in umgekehrter Reihenfolge enthalten sind.

Beispiele:

```
mirror Nil = Nil
mirror ex = Cons (1N,Cons (2N,Cons (4N,Cons (3N,Cons (4N,Cons (2N,Nil))))))
```

Freiwillig: Schaffen Sie es, die Funktion so zu schreiben, dass sie nur linear viele Schritte in der Länge der Liste benötigt?

```
// Lösung mit Laufzeit quadratisch in der Länge von xs
let rec mirror (xs: Nats): Nats =
    match xs with
    | Nil -> Nil
    | Cons (x, ys) -> concat (mirror ys) (Cons (x, Nil))

// Effizientere Lösung (lineare Laufzeit)
let mirror' (xs: Nats): Nats =
    // Hilfsfunktion berechnet concat (mirror xs) zs
    let rec mirrorConcat (xs: Nats) (zs: Nats): Nats =
        match xs with
        | Nil -> zs
        | Cons (x, ys) -> mirrorConcat ys (Cons (x, zs))
    mirrorConcat xs Nil
```

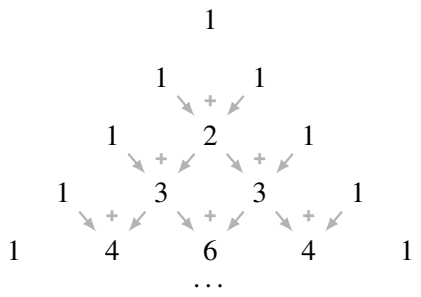
c) Schreiben Sie eine Funktion `filter: (Nat -> Bool) -> Nats -> Nats`, die ein Prädikat auf den natürlichen Zahlen (also eine Funktion, die eine natürliche Zahl nimmt und prüft, ob diese eine bestimmte Bedingung erfüllt) und eine Liste von natürlichen Zahlen nimmt und und die Zahlen zurückgibt, die das Prädikat erfüllen.

Beispiele:

```
filter p Nil = Nil
filter (fun n -> n%2=0) (Cons (2N, Cons (4N, Nil))) = Cons (2N, Cons (4N, Nil))
filter (fun n -> n%2=0) (Cons (1N, Cons (6N, Nil))) = Cons (6N, Nil)
filter (fun n -> n>3) ex = Cons (4N, (Cons (4N, Nil)))
```

```
let rec filter (p: Nat -> Bool) (xs: Nats): Nats =
    match xs with
    | Nil -> Nil
    | Cons (x, ys) ->
        if p x then
            Cons (x, filter p ys)
        else
            filter p ys
```

d) Implementieren Sie die Funktion `pascal: Nat -> Nats`, die eine Zeile des Pascalschen Dreiecks berechnet.




```
pascal 0 = Cons (1, Nil)

pascal 1 = Cons (1, Cons (1, Nil))

pascal 2 = Cons (1, Cons (2, Cons (1, Nil)))

pascal 3 = Cons (1,Cons(3,Cons(3,Cons(1,Nil))))

pascal 4 = Cons (1, Cons (4, Cons (6, Cons (4,
    Cons (1, Nil)))))
```

Wie in der Grafik gezeigt, werden immer je zwei benachbarte Elemente der vorherigen Zeile addiert und außen an den Rändern jeweils eine 1 hinzugefügt.

Tipp: Verwenden Sie das Peano-Entwurfsmuster und definieren Sie zusätzlich eine rekursive Hilfsfunktion.

```
let rec pascal (level: Nat): Nats =
    let rec step (xs: Nats): Nats =
        match xs with
        | Cons (_, Nil) -> Cons (1N, Nil)
        | Cons (x, Cons(y, zs)) -> Cons (x+y, step (Cons (y, zs)))
    if level = 0N then Cons (1N, Nil)
    else Cons (1N, (step (pascal (level - 1N))))
```

Aufgabe 4 Dynamische und Statische Semantik (Einreichaufgabe, 7 Punkte)

Motivation: In dieser Aufgabe sollen Sie die Semantikregeln rückwärts anwenden: Im ersten Teil ist die linke Seite (der Ausdruck) unbekannt, aber die rechte Seite (der Wert) vorgegeben. Im zweiten Teil arbeiten Sie wieder in der gewohnten Richtung, der Ausdruck ist nun bekannt und es ist der dazugehörige Typ gesucht. Sie benötigen die Regeln der Vorlesungsfolien 109-110, 138-139, 143-144 und 182-183.

Finden Sie einen Ausdruck e, der gemäß den Regeln der Dynamischen Semantik aus der Vorlesung zu folgendem Wert auswertet:

$$\langle \{a \mapsto 5\}, x, a + x \rangle$$

Geben Sie einen Beweisbaum mit den Regeln der Dynamischen Semantik an, der

$$\emptyset \vdash e \Downarrow \langle \{a \mapsto 5\}, x, a + x \rangle$$

zeigt. Bestimmen Sie anschließend den Typ t Ihres Ausdrucks e und geben Sie einen Beweisbaum mit den Regeln der Statischen Semantik an, der

$$\emptyset \vdash e : t$$

zeigt.

Ausdruck e := `let a = 5 in fun x → a + x` Link zum Baum

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \texttt{5} \Downarrow \texttt{5}}}{\emptyset \vdash \texttt{let a = 5} \Downarrow \{\texttt{a} \mapsto \texttt{5}\}} \quad \dfrac{}{\{\texttt{a} \mapsto \texttt{5}\} \vdash \texttt{fun x → a + x} \Downarrow \langle \{\texttt{a} \mapsto \texttt{5}\}, \texttt{x}, \texttt{a + x} \rangle}}{\emptyset \vdash \texttt{let a = 5 in fun x → a + x} \Downarrow \langle \{\texttt{a} \mapsto \texttt{5}\}, \texttt{x}, \texttt{a + x} \rangle}$$

Typ t := Nat → Nat Link zum Baum

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \texttt{5} : \text{Nat}}}{\emptyset \vdash \texttt{let a = 5} : \{\texttt{a} \mapsto \text{Nat}\}} \quad \dfrac{\dfrac{\dfrac{}{\{\texttt{a} \mapsto \text{Nat}, \texttt{x} \mapsto \text{Nat}\} \vdash \texttt{a} : \text{Nat}} \quad \dfrac{}{\{\texttt{a} \mapsto \text{Nat}, \texttt{x} \mapsto \text{Nat}\} \vdash \texttt{x} : \text{Nat}}}{\{\texttt{a} \mapsto \text{Nat}, \texttt{x} \mapsto \text{Nat}\} \vdash \texttt{a + x} : \text{Nat}}}{\{\texttt{a} \mapsto \text{Nat}\} \vdash \texttt{fun (x: Nat) → a + x} : \text{Nat} \to \text{Nat}}}{\emptyset \vdash \texttt{let a = 5 in fun (x: Nat) → a + x} : \text{Nat} \to \text{Nat}}$$

Hinweis: Es sind auch andere Ausdrücke möglich, die `5` lässt sich beispielsweise durch `2 + 3` ersetzen, dann werden die Bäume jedoch größer.

Aufgabe 5 Statische Semantik von rekursiven Funktionen (Trainingsaufgabe)

Wir betrachten die rekursive Funktionsdefinition

```
let rec f  (x: Bool): Bool = (if x then x else f (not x))
```

bezüglich der Signatur $\Sigma := \{\texttt{not} \mapsto \text{Bool} \to \text{Bool}\}$. Geben Sie einen vollständigen Beweisbaum an.

Definiere aus Platzgründen $\Sigma_1 := \{\texttt{not} \mapsto \text{Bool} \to \text{Bool}, \texttt{f} \mapsto \text{Bool} \to \text{Bool}, \texttt{x} \mapsto \text{Bool}\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_1 \vdash \texttt{x} : \text{Bool}} \quad \dfrac{}{\Sigma_1 \vdash \texttt{x} : \text{Bool}} \quad \dfrac{\dfrac{}{\Sigma_1 \vdash \texttt{f} : \text{Bool} \to \text{Bool}} \quad \dfrac{\dfrac{}{\Sigma_1 \vdash \texttt{not} : \text{Bool} \to \text{Bool}} \quad \dfrac{}{\Sigma_1 \vdash \texttt{x} : \text{Bool}}}{\Sigma_1 \vdash \texttt{not x} : \text{Bool}}}{\Sigma_1 \vdash \texttt{f (not x)} : \text{Bool}}}{\Sigma_1 \vdash \texttt{if x then x else f (not x)} : \text{Bool}}}{\Sigma \vdash \texttt{let rec f (x: Bool): Bool = if x then x else f (not x)} : \{\texttt{f} \mapsto \text{Bool} \to \text{Bool}\}}$$