

## Lösungshinweise/-vorschläge zum Übungsblatt 4: Grundlagen der Programmierung (WS 2023/24)

### 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 281 bis 373 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 übergebenen 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 281 bis 327 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<sup>1</sup> ist.

Beispiele:

`isLeapYear 2023 = false`

`isLeapYear 2024 = true`

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

<sup>1</sup>[https://de.wikipedia.org/wiki/Schaltjahr#Gregorianischer\\_Kalender](https://de.wikipedia.org/wiki/Schaltjahr#Gregorianischer_Kalender)

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

Beispiel:

```
daysInMonth 2021 2 = 28
daysInMonth 2020 2 = 29
daysInMonth 2023 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 = 2023N; month = 11N; day = 21N; weekday = Tuesday } =
  { year = 2023N; month = 11N; day = 22N; weekday = Wednesday }
nextDate { year = 2023N; month = 12N; day = 31N; weekday = Sunday } =
  { year = 2024N; month = 1N; day = 1N; weekday = Monday }
```

```
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 = 2023N; month = 11N; day = 21N; weekday = Tuesday } = true
validateWeekday { year = 2023N; month = 11N; day = 21N; 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, 6 Punkte)

*Motivation:* In dieser Aufgabe sollen Sie sich mit rekursiven Variantentypen beschäftigen. Sie können sich an den Vorlesungsfolien 332 bis 373 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 `double: Nats -> Nats`, die jeden Eintrag der gegebenen Liste verdoppelt. Beispiele:

```
double Nil = Nil
```

```
double (Cons (1N, Cons (2N, Nil))) = Cons (2N,Cons (4N,Nil))
```

```
double ex = Cons (4N, Cons (8N, Cons (6N, Cons (8N, Cons (4N, Cons (2N, Nil))))))
```

```
let rec double (xs: Nats): Nats =  
    match xs with  
    | Nil -> Nil  
    | Cons (x, xs) -> Cons (2N*x, double xs)
```

- b) Schreiben Sie eine Funktion `isSorted: Nats -> Bool`, die prüft, ob die gegebene List *aufsteigend* sortiert ist.

Beispiele:

```
isSorted Nil = true
```

```
isSorted (Cons (1N, Cons (2N, Nil))) = true
```

```
isSorted (Cons (1N, Cons (1N, Nil))) = true
```

```
isSorted (Cons (2N, Cons (1N, Nil))) = false
```

```
isSorted ex = false
```

```
let rec isSorted (xs: Nats): Bool =  
    match xs with  
    | Nil -> true  
    | Cons (x, Nil) -> true  
    | Cons (x, Cons (y, xs)) -> x <= y && isSorted (Cons (y, xs))
```

- 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 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, xs) -> if p x then Cons (x, filter p xs) else filter p xs
```

## Aufgabe 4 Statische Semantik von rekursiven Funktionen (Einreichaufgabe, 3 Punkte)

Wir betrachten die rekursive Funktionsdefinition

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

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

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

$$\frac{\frac{\frac{\frac{\frac{\Sigma_1 \vdash \text{not} : \text{Bool} \rightarrow \text{Bool}}{\Sigma_1 \vdash \text{not} : \text{Bool} \rightarrow \text{Bool}}}{\Sigma_1 \vdash \text{not } x : \text{Bool}}}{\Sigma_1 \vdash f : \text{Bool} \rightarrow \text{Bool}}}{\Sigma_1 \vdash f (\text{not } x) : \text{Bool}}}{\Sigma_1 \vdash \text{if } x \text{ then } x \text{ else } f (\text{not } x) : \text{Bool}}}{\Sigma \vdash \text{let rec } f (x: \text{Bool}): \text{Bool} = \text{if } x \text{ then } x \text{ else } f (\text{not } x) : \{f \mapsto \text{Bool} \rightarrow \text{Bool}\}}$$





e)  $((\text{fun } (c: \text{Nat}) \rightarrow (\text{fun } (x: \text{Nat}) \rightarrow c)) \ 42) \ 815$

Statische Semantik

$$\frac{\frac{\frac{\frac{}{\{c \mapsto \text{Nat}, x \mapsto \text{Nat}\} \vdash c : \text{Nat}}{\{c \mapsto \text{Nat}\} \vdash \text{fun } x \rightarrow c : (\text{Nat} \rightarrow \text{Nat})}}{\emptyset \vdash \text{fun } c \rightarrow (\text{fun } x \rightarrow c) : (\text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat}))} \quad \frac{}{\emptyset \vdash 42 : \text{Nat}}}{\emptyset \vdash (\text{fun } c \rightarrow (\text{fun } x \rightarrow c)) \ 42 : (\text{Nat} \rightarrow \text{Nat})} \quad \frac{}{\emptyset \vdash 815 : \text{Nat}}}{\emptyset \vdash ((\text{fun } c \rightarrow (\text{fun } x \rightarrow c)) \ 42) \ 815 : \text{Nat}}$$

Dynamische Semantik

Definiere aus Platzgründen

$f := \text{fun } c \rightarrow (\text{fun } x \rightarrow c)$

$$\frac{\frac{\frac{}{\emptyset \vdash f \Downarrow \langle \emptyset, c, (\text{fun } x \rightarrow c) \rangle} \quad \frac{}{\emptyset \vdash 42 \Downarrow 42} \quad \frac{}{\{c \mapsto 42\} \vdash \text{fun } x \rightarrow c \Downarrow \langle \{c \mapsto 42\}, x, c \rangle}}{\emptyset \vdash f \ 42 \Downarrow \langle \{c \mapsto 42\}, x, c \rangle} \quad \frac{}{\emptyset \vdash 815 \Downarrow 815} \quad \frac{}{\{c \mapsto 42, x \mapsto 815\} \vdash c \Downarrow 42}}{\emptyset \vdash (f \ 42) \ 815 \Downarrow 42}$$