CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY
Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group

Programmierung 1 (WS 2020/21)
Aufgaben für die Übungsgruppe D (Lösungsvorschläge)

Hinweis: Diese Aufgaben wurden von den Tutoren für die Übungsgruppe erstellt. Sie sind für die Klausur weder relevant noch irrelevant. 🤔 markiert potentiell schwerere Aufgaben.

Bezeichnerauftreten

Aufgabe TD.1 *(Spätjahrsputz)*
Bereinigen Sie die folgenden Teilprogramme: Überstreichen Sie dazu die definierenden Bezeichnerauftreten (z.B. $\overline{\texttt{a}}$), stellen Sie die lexikalischen Bindungen mit Pfeilen dar und unterstreichen Sie alle freien Bezeichner (z.B. $\underline{\texttt{a}}$). Schlussendlich benennen Sie noch die gebundenen Bezeichner konsistent durch Indizieren um (z.B. $\texttt{a}_1$).

(a)

```
val x = 3
val y = 3*x
fun p (a:int*int) x = let
                          val y = 2*x
                      in
                          (#1(a) + y * x)
                      end
val z = p a y
```

(b)

```
val a = let
            val (b,c) = a
            val a = c
            val b = 3 * b
        in
            (fn a => (b + d)* a) a
        end
```

(c)

```
fun foo x y = fn x => y * ( (fn z => z + x) x + (fn z => (fn y => y * z) ) z y)
```

Lösungsvorschlag TD.1

(a)

val $\overline{x_1}$ = 3
val $\overline{y_1}$ = 3*x_1
fun $\overline{p_1}$ ($\overline{a_1}$:int*int) $\overline{x_2}$ =let
val $\overline{y_2}$ = 2*x_2
in
(#1(a_1) + y_2 * x_2)
end
val $\overline{z_1}$ = p_1 $\underline{a}$ y_1

(b)

val $\overline{a_1}$ =let
val ($\overline{b_1}$, $\overline{c_1}$) = $\underline{a}$
val $\overline{a_2}$ = c_1
val $\overline{b_2}$ = 3 * b_1
in
(fn $\overline{a_3}$ => (b_2 + $\underline{d}$) * a_3) a_2
end

(c)

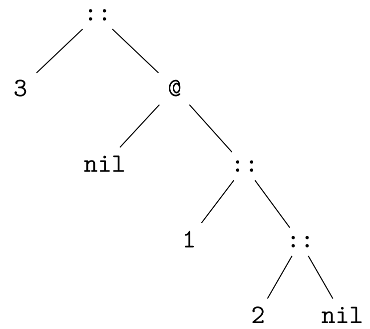
$$\texttt{fun}\ \overline{\texttt{foo}_1}\ \overline{\texttt{x}_1}\ \overline{\texttt{y}_1}\ \texttt{=fn}\ \overline{\texttt{x}_2}\ \Rightarrow \texttt{y}_1\ \texttt{*}\ \texttt{(}\ \texttt{(fn}\ \overline{\texttt{z}_1}\ \Rightarrow \texttt{z}_1\ \texttt{+x}_2\texttt{)}\ \texttt{x}_2\ \texttt{+(fn}\ \overline{\texttt{z}_2}\ \Rightarrow \texttt{(fn}\ \overline{\texttt{y}_2}\ \Rightarrow \texttt{y}_2\ \texttt{*}\ \texttt{z}_2\texttt{)}\ \texttt{)}\ \underline{\texttt{z}}\ \texttt{y}_1\texttt{)}$$

Lexikalische Bindungen: Ziehe einen Pfeil von jedem indizierten, benutzenden Bezeichnerauftreten zu dem gleich indizierten, überstrichenen Bezeichnerauftreten.

Listen-Basics

Aufgabe TD.2 *(Baumdarstellung)*
Geben Sie die Baumdarstellung des Ausdrucks `3 :: nil @ 1 :: (2 :: nil)` an.

Lösungsvorschlag TD.2



Aufgabe TD.3 *(Listentypen)*
Welchen Typ haben die folgenden Listen?

(a) `nil`

(b) `1 :: 2 :: 3 :: nil`

(c) `1 :: [2] @ [3] @ nil`

(d) `([1] @ [2]) :: nil`

(e) `nil :: nil :: nil`

(f) `nil @ nil @ nil`

Lösungsvorschlag TD.3

(a) $\forall\alpha$: α `list`

(b) `int list`

(c) `int list`

(d) `int list list`

(e) $\forall\alpha$: α `list list`

(f) α `list`, wobei α frei ist.

Aufgabe TD.4 *(Listen-Typen)*
Entscheiden Sie, ob die folgenden Ausdrücke wohlgetypt sind. Wenn ja - welche Typen haben diese?

(a) `nil`

(b) `nil :: nil`

(c) `(nil :: nil) :: nil`

(d) `(1 :: nil) :: ((1 :: nil) :: nil) :: nil`

(e) `nil :: nil @ nil`

(f) `(nil :: nil) :: (nil :: nil)`

(g) `nil @ (nil :: nil)`

Lösungsvorschlag TD.4

(a) $\forall\alpha$: α `list`

(b) $\forall\alpha$: α `list list`

(c) $\forall\alpha$: α `list list list`

(d) ↯

(e) $\forall\alpha$: α `list list`

(f) $\forall\alpha$: α `list list list`

(g) α `list list`, wobei α frei ist.

Aufgabe TD.5 *(Patternmatching ist kein Glücksspiel)*
Implementieren Sie eine regelbasierte Prozedur, die eine Liste aus Integer-Tupeln bekommt und die Anzahl an Nullen in den Tupeln ausgibt.
`howmany : (int * int) list → int`

Lösungsvorschlag TD.5

```
fun howmany nil = 0
  | howmany ((0,0)::xr) = 2 + howmany xr
  | howmany ((0,_)::xr) = 1 + howmany xr
  | howmany ((_,0)::xr) = 1 + howmany xr
  | howmany (x::xr) = howmany xr
```

Aufgabe TD.6 *(Range)*

(a) Schreiben sie eine Prozedur `range : int → int → int list`, welche eine Liste ausgibt, die alle ganzen Zahlen zwischen dem ersten Argument und einschließlich des zweiten Arguments in aufsteigender Reihenfolge enthält.

(b) Modifizieren Sie ihre Prozedur `range` zu `rangeStepsize : int → int → int → int list`, welche jetzt eine Liste aller ganzen Zahlen zwischen dem Ersten und dem zweiten Argument ausgibt, wobei der Abstand zwischen der jeweiligen Zahl und dem ersten Argument ein Vielfaches des dritten Arguments ist.

Beispiel: `rangeStepsize 1 7 2 = [1,3,5,7]`

Lösungsvorschlag TD.6

(a) `fun range m n = if(m<=n) then m::range (m+1) n else nil`

(b) `fun rangeStepsize m n s = if(m<=n) then m:: rangeStepsize (m+s) n s else nil`

Aufgabe TD.7 *(How To primes)*
Das Sieb des Eratosthenes ist ein Algorithmus zur Berechnung von Primzahlen bis einschließlich einem gegebenen Index n. Dazu wird eine Liste oder Tabelle von Zahlen von 2 bis n verwendet, aus der nach und nach mehr Zahlen rausgestrichen werden. Solange es noch Elemente in der Liste gibt, betrachtet man jeweils die kleinste Zahl, die in der Liste ist, nimmt sie aus der Liste heraus und markiert sie als Primzahl. Dann streicht man alle Vielfachen von ihr aus der Liste. Die Menge der markierten Zahlen ist dann gerade die Menge aller Primzahlen zwischen 2 und n. Im Folgenden soll dieses Prinzip in SML umgesetzt werden.

(a) Schreiben Sie eine endrekursive Prozedur `listevonzahlen : int → int → int list`, die von einem Startindex m an alle Zahlen einschließlich n aufsteigend geordnet in eine Liste schreibt.

(b) Schreiben Sie eine Prozedur `loesche : int → int list → int list`, die aus einer gegebenen Liste alle Zahlen entfernt, die durch eine gebene Zahl n teilbar sind.

(c) Schreiben Sie nun eine Prozedur `sieb' : int list → int list → int list`, die aus der einer Liste xs das erste Element x entfernt, es an die andere Liste ys anhängt und alle Zahlen aus xs entfernt, die durch x teilbar sind.

(d) Fügen Sie nun die drei Hilfsprozeduren zum Sieb des Erathostenes zusammen.

Lösungsvorschlag TD.7

(a)

```
fun listehelp m n xs = if n < m then xs
else listehelp m (n - 1) (n :: xs)
fun listevonzahlen m n = listehelp m n nil
```

(b)

```
fun loesche n    nil     = nil
| loesche n (x :: xr) = if (x mod n = 0) then loesche n xr
else x :: (loesche n xr)
```

(c)

```
fun sieb' xs    nil     = xs
| sieb' xs (y :: yr)  = sieb' (xs @ [y]) (loesche y yr)
```

(d)

```
fun siebdeserathostenes n = sieb' nil (listevonzahlen 2 n)
```

Aufgabe TD.8 *(Potenzmengen)*

(a) Schreiben Sie eine Prozedur `power : α list → α list list`, die zu einer gegebenen Liste die Potenzmenge, also die Liste aller Teillisten liefert.

> **Beispiel:** `power [1, 2, 3] = [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, ↳ 3]]`

Die Reihenfolge der Teillisten in der Ergebnisliste soll dabei keine Rolle spielen.

(b) Schreiben Sie eine Prozedur `kpower : int → α list → α list list`, die zu einer Liste *xs* die Liste aller *k*-elementigen Teilmengen liefert.

> **Beispiel:** `kpower 2 [1, 2, 3] = [[1, 2], [1, 3], [2, 3]]`

Lösungsvorschlag TD.8

(a)

```
fun power nil = [nil]
| power (x::xr) = let val ps = power xr
in map (fn ys => x::ys) ps @ ps end
```

(b)

```
fun kpower k xs = List.filter (fn xs => length xs = k) (power xs)
```

Höherstufige Listenprozeduren

Aufgabe TD.9 *(Nur für Mitglieder)*

Schreiben Sie eine polymorphe Prozedur `member : 'α → 'α list → bool`, die testet, ob ein Wert als Element in einer Liste vorkommt. Lösen Sie dies auf vier Arten:

(a) durch eine regelbasierte Prozedurdeklaration,

(b) mit Hilfe der vordeklarierten Prozedur `List.exists`.

(c) mit Hilfe der vordeklarierten Prozedur `List.filter`.

(d) mit Hilfe der Prozedur `foldl`.

Lösungsvorschlag TD.9

(a)

```
fun member x   nil      = false
     | member x (y :: yr) = x = y orelse member x yr
```

(b)

```
fun member x = List.exists (fn y => x = y)
```

(c)

```
fun member x xs = length (List.filter (fn y => x = y) xs) > 0
```

(d)

```
fun member x = foldl (fn (y, b) => b orelse x = y) false
```

Aufgabe TD.10 *(Doublemap)*
Schreiben Sie eine Prozedur `doublemap : (α * β → γ) → α list → β list → γ list`, die eine übergebene Prozedur `f` auf ein Tupel bestehend aus je einem Element der beiden übergebenen Listen anwendet. Sind die Listen unterschiedlich groß, soll die Ausnahme `Subscript` geworfen werden.

Beispiel: `doublemap f [2, 3] [true, false] = [f (2, true), f (3, false)]`

Lösungsvorschlag TD.10

```
fun doublemap f    nil        nil     = nil
  | doublemap f (x :: xr) (y :: yr) = f(x, y) :: doublemap f xr yr
  | doublemap f     _          _      = raise Subscript
```

Aufgabe TD.11 *(Skalarprodukt)*
Schreiben Sie eine Prozedur `prod : int list → int list → int`, die, gegeben zwei gleich lange Listen, das Skalarprodukt berechnet. Für zwei Listen

$$[x_1, x_2, x_3, \ldots, x_k] \text{ und } [y_1, y_2, y_3, \ldots, y_k],$$

soll `prod` also

$$x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3 + \ldots + x_k \cdot y_k.$$

berechnen.
Falls die Listen unterschiedlich lang sind, soll die Ausnahme `Subscript` geworfen werden.

Hinweis: Sie dürfen Ihre Prozedur `doublemap` aus der vorherigen Aufgabe verwenden.

Lösungsvorschlag TD.11

```
fun prod xs ys = foldl op+ 0 (doublemap op* xs ys)
```

Faltung

Aufgabe TD.12 *(Map mit Faltung)*
Schreiben Sie eine Prozedur `foldmap : (α → β) → α list → β list`, die `map` mithilfe von Faltung implementiert.

Beispiel: Für `(fn x => x div 2) [2, 5, 6, 9]` liefert `foldmap`: `[1, 2, 3, 4]`.

Lösungsvorschlag TD.12

```
fun foldmap f xs = foldr (fn (x, a) => (f x) :: a) nil xs
```

Aufgabe TD.13 *(Listenprozeduren mit Faltung)*
Unter den folgenden Prozeduraufrufen ist je ein linker zu genau einem rechten semantisch äquivalent. Ordnen Sie diese einander zu.

`map f x` - 1	A - `foldr op:: nil xs`
`filter f xs` - 2	B - `foldr f s (foldr (fn (x, a) => a @ [x]) nil xs)`
`xs @ ys` - 3	C - `foldl op:: nil xs`
`id xs` - 4	D - `foldr op:: ys xs`
`foldl f s xs` - 5	E - `foldr (fn (x, a) => f x :: a) nil xs`
`length xs` - 6	F - `foldr (fn (x, a) => if f x then x::a else a) nil xs`
`foldr f s xs` - 7	G - `foldl (fn (_, a) => a + 1) 0 xs`
`rev xs` - 8	H - `foldl f s (foldl op:: nil xs)`

Dabei ist `id` die Prozedur, die die Identitätsfunktion implementiert. Sie ist definiert als `fun id x = x`.

Lösungsvorschlag TD.13
1 - E, 2 - F, 3 - D, 4 - A, 5 - B, 6 - G, 7 - H, 8 - C

Aufgabe TD.14 *(Haarspalterei)*
Schreiben Sie mit Faltung eine nichtrekursive Prozedur `split : int → int list → int list * int list`, die gegeben ein sogenanntes Pivotelement k eine Liste in zwei Teillisten zerlegt, wobei die erste nur die Zahlen enthält, die kleiner als k sind, und die zweite nur die Zahlen, die größer oder gleich k sind. Die Reihenfolge, in der die Elemente in der Liste geordnet sein sollen, ist dabei nicht relevant.

> **Beispiel:** `split 6 [7, 3, 6, 5, 4, 10, 1, 2, 9, 8]` soll zu `([2, 1, 4, 5, 3], [8, 9, 10, 6, 7])` auswerten.

Lösungsvorschlag TD.14

```
fun split pvt xs = foldl
    (fn (x, (smaller, larger)) => if x < pvt then (x::smaller, larger)
                                               else (smaller, x::larger))
    (nil, nil) xs
```

Aufgabe TD.15 *(Listenkomprimierung)*
Schreiben Sie mithilfe von Faltung eine Prozedur `zip : α list → β list → (α * β) list`, die gleichlange Listen zu einer Liste wie aus dem Typschema ablesbar zusammenfügt.

> **Beispiel:** `zip [1, 2] [true, false]` soll zu `[(1, true), (2, false)]` auswerten.

Lösungsvorschlag TD.15

```
fun zip xs ys = rev (#1(foldl (fn (x, (a, ys)) => ((x, hd ys) :: a, tl ys)) (nil, ys) xs))
```

Aufgabe TD.16 *(Prozeduren Umdenken)*
Schreiben Sie die Listenprozedur `foldl` mithilfe von `iter`. Sie dürfen dafür die Prozedur `List.length : α list → int` verwenden, die die Länge einer Liste liefert.

Lösungsvorschlag TD.16

```
fun myFoldl f s xs = #1 (iter (List.length xs) (s, xs)
                        (fn(s', xr) => (f(hd xr, s'), tl xr)))
```

Aufgabe TD.17 *(howMany * 3)*
Schreiben Sie eine Prozedur `howMany : α list → (α → bool) → int`, welche testet, wie viele Elemente einer Liste eine bestimmte Bedingung erfüllen, auf drei Arten:

(a) ohne jegliche Hilfsprozeduren

(b) nicht rekursiv und ohne Faltung, aber mithilfe der vordeklarierten Listenprozeduren

(c) nicht rekursiv und mit Faltung

Lösungsvorschlag TD.17

(a)

```
fun howMany   nil      p = 0
  | howMany (x :: xr) p = (if p x then 1 else 0) + howmany xr p
```

(b)

```
fun howMany ls p = List.length (List.filter p ls)
```

(c)

```
fun howMany ls p =  foldl (fn (x, a) => if p x then a + 1 else a) 0 ls
```

Aufgabe TD.18 *(Zick Zack)*
Eine Liste ganzer Zahlen hat die *Zick-Zack-Eigenschaft*, wenn ihre Einträge abwechselnd größer und kleiner als der Vorgänger sind. Zum Beispiel haben `[3, 7, ~6, 0, ~1]`, `[42, 7, 15]`, `[7]` die Zick-Zack-Eigenschaft, die Listen `[5, 2, 7, 42, 13]` und `[3, 2, 1]` hingegen nicht.

Deklarieren Sie mithilfe der Faltungsprozeduren eine Prozedur `zigzag : int list → bool`, die überprüft, ob eine gegebene Liste die Zick-Zack-Eigenschaft erfüllt. Verwenden Sie außer Faltung keine Hilfsprozeduren.

Lösungsvorschlag TD.18

```
fun zigzag nil = true
  | zigzag (x::nil) = true
  | zigzag (x::y::zs) = #1 (foldl
      (fn (u, (ok, gt, pre)) =>
          if gt then (ok andalso u > pre, false, u)
                else (ok andalso u < pre, true, u))
      (true, y < x, y) zs)
```

Aufgabe TD.19 *(Faltung mit Kombinierungsfunktion)*
Deklarieren Sie mithilfe von `foldl` eine Prozedur `apply: (α * α → β) → α list → β list` so, dass `apply f xs` die übergebene Prozedur `f` auf je zwei aufeinanderfolgende Listenelemente aus `xs` anwendet und eine Liste mit den Ergebnissen zurückgibt.

Beispiel: `apply op* [5, 6, 3, 1, 7]` soll zu `[30, 18, 3, 7]` auswerten.

Verwenden Sie außer `foldl` und `rev` keine rekursiven Hilfsprozeduren. Sie dürfen annehmen, dass die übergebene Liste mindestens zwei Elemente enthält.

Lösungsvorschlag TD.19

```
fun apply f (x::xs) =
    rev (#1 (foldl (fn (u, (rs, pre)) => (f(pre, u) :: rs, u)) ([], x) xs))
```

Aufgabe TD.20 *(Potenzliste)*
Schreiben Sie mithilfe von Faltung eine nichtrekursive Prozedur `powerlist : α list → α list list`, die die Potenzliste einer gegebenen Liste berechnet. Unter der Potenzliste wird dabei die Liste verstanden, die alle Teillisten der vorher gegebenen Liste enthält, wobei die Reihenfolge der Elemente egal ist.

Beispiel: `powerlist [1,2,3]` soll zu `[[], [1], [2], [2, 1], [3], [3, 1], [3, 2], [3, 2, 1]]` auswerten.

Lösungsvorschlag TD.20

```
fun powerlist xs = foldl (fn (x, a) => a @ map (fn k => x::k) a) [nil] xs
```

Aufgabe TD.21 *(Was sind Monaden?)*

(a) Schreiben Sie mit Faltung eine nichtrekursive Prozedur `flatmap : (α → β list) → α list → β list`, die für eine Prozedur `p` und eine Liste `xs` zum gleichen Ergebnis wie `concat (map p xs)` auswertet.

(b) Schreiben Sie nur mit Hilfe von `flatmap` eine Prozedur `dup`, die jedes Auftreten eines Elementes einer Liste verdoppelt. So soll beispielsweise `dup [1, 2, 3] = [1, 1, 2, 2, 3, 3]` gelten.

(c) Können Sie auch die Prozeduren `map`, `filter` oder `concat` mit `flatmap` schreiben?

Lösungsvorschlag TD.21

(a) `fun flatmap f xs = foldl (fn (x,a) => a @ f x) nil xs`

(b) `fun dup xs = flatmap (fn k => [k, k]) xs`

(c)

```
fun map f xs = flatmap (fn k => [f k]) xs
fun filter s xs = flatmap (fn k => if k = s then [] else [k]) xs
fun concat xs = flatmap id xs
```

Aufgabe TD.22 *(Longest Increasing Subsequence)*

Deklarieren Sie mithilfe von `foldl` eine Prozedur `lis : int list → int`, die zu einer Liste von ganzen Zahlen die Länge der längsten aufsteigenden Folge von Zahlen berechnet, die sich durch Entfernen beliebig vieler Zahlen aus der ursprünglichen Liste ergibt.

Formal: Sind $x_1, x_2, \ldots, x_n$ die Zahlen in der Liste, dann ist die größte Zahl m zu berechnen, sodass es Indizes $i_1 < i_2 < \cdots < i_m$ mit $x_{i_1} < x_{i_2} < \cdots < x_{i_m}$ gibt.

Verwenden Sie ausschließlich `foldl` als Hilfsprozedur. Ihre Prozedur selbst sollte nicht rekursiv sein!

> **Beispiel:** `lis [5, 4, 9, 10, 14, 3]` soll zu 4 auswerten, denn durch Entfernen der Zahlen 4 und 3 ergibt sich die aufsteigende Folge `5, 9, 10, 14` der Länge 4. Eine längere aufsteigende Folge gibt es nicht.

Lösungsvorschlag TD.22

Es bezeichne $f(k)$ die Länge einer LIS, die x_k als letztes Element hat. Wir beobachten, dass die Gleichung

$$f(k) = 1 + \max_{\substack{1 \leq j < k \\ x_j < x_k}} f(j)$$

gilt, falls das Maximum existiert. Ansonsten ist $f(k) = 1$. Unser Ziel ist es, $f(k)$ für alle $k \in \{1, \ldots, n\}$ zu berechnen. Dafür gehen wir mit einer Faltung über die Liste und merken uns im Akkumulator eine Liste von Paaren (e_k, ℓ_k), wobei e die zugehörige Zahl $e = x_k$ in der ursprünglichen Liste und $\ell = f(k)$ die Länge einer LIS mit x_k als letztes Element bezeichnet. Um einen neuen Eintrag (e_{k+1}, ℓ_{k+1}) zu berechnen, gehen wir mit einer weiteren Faltung über die uns bekannte Liste von partiellen Ergebnissen (e_j, ℓ_j) für $1 \leq j \leq k$ und verwalten ein Maximum m, das verändert wird, falls $e_j < e_{k+1}$ und $\ell_j > m$ gilt. Zum Schluss gehen wir mit einer dritten Faltung über die Liste dieser Paare und berechnen uns die maximale Länge.

```
fun lis xs =
    foldl (fn ((e, len), rs) => if len > rs then len else rs) 0
        (foldl (fn (u, partial) =>
            (u, 1 + (foldl (fn ((e, len), m) =>
                if e < u andalso len > m then len else m) 0 partial)) :: partial) [] xs)
```

Aufgabe TD.23 *(Faltung ist toll, nicht?)*
Dieter Schlau hat sich viele Gedanken zu Listenprozeduren mit Faltung gemacht. Dummerweise hat er seinen Prozeduren keine vernünftigen Namen gegeben. Konkret geht es ihm um folgenden Code:

```
fun f (xs, n) = hd (#1(foldl (fn (x, (ls, i)) => (if i = n then [x] else ls, i + 1))
                             ([], 0) xs))
```

(a) Helfen Sie Dieter. Was berechnet `f`? Welcher Listenprozedur entspricht `f`?

(b) Dieters Prozedur scheint etwas zu komplex für ihren Zweck zu sein. Können Sie `f` in eine endrekursive Prozedur umschreiben, die besser lesbar ist?

(c) Geben Sie es zu, als Sie diese Aufgabenstellung gelesen haben, dachten Sie: „Nicht schon wieder Dieter. Was hat er dieses Mal verbrochen?“ Und ja, Dieter bzw. seine Prozedur macht tatsächlich unnötigen Kram. Wie viel schneller ist Ihre Prozedur im Mittel?

Lösungsvorschlag TD.23

(a) `f` entspricht `List.nth`, gibt also das n-te Element einer Liste aus. Der einzige Unterschied ist, dass `f` statt `Subscript` `Empty` wirft, wenn das Element nicht in der Liste enthalten ist.

(b)
```
fun nth ([], _) = raise Empty
  | nth (x::_, 0) = x
  | nth (_::xs, n) = nth (xs, n - 1)
```

(c) Wenn mit `nth` auf das erste/nullte Listenelement zugegriffen werden soll, dann wird genau eine Anwendungsgleichung benötigt. Wird hingegen auf das letzte Element einer Liste mit n Elementen zugegriffen, so sind n Anwendungsgleichungen nötig. Sie brauchen also im Mittel $\frac{n}{2}$ Anwendungsgleichungen.

Bei `f` wird unabhängig von n die Abstraktion auf jedes Listenelement angewandt. Lässt man außer Acht, dass `f` an sich auch schon durch das `foldl` komplexer ist, ist `nth` also im Mittel doppelt so schnell.

Aufgabe TD.24 *(Muskeln falten)*
Ihr Freund Dieter Schlau trainiert gerade ~~in der Muckibude~~ Zuhause. Neben ihm steht eine Dose Proteinpulver. Als er nach dem Training die Dose in die Hand nimmt, fällt sein Blick auf die Liste von Aminosäuren, die auf der Packung aufgedruckt ist. Wie soll das seine Muskeln nur zum Wachsen bringen?!

Das Thema lässt Dieter keine Ruhe. Er beschließt seinen Freund und Bioinformatik-Studenten Andreas[1] um Hilfe zu fragen, der mit ihm die Vorlesung Programmierung 1 hört.

Andreas erzählt von der sogenannten “Proteinbiosynthese”, die aus der mRNA, welche aus der DNA abgelesen wird, diese Aminosäuren in die neu gebauten Proteine für deine Muskeln einbaut. Dieter erfährt, dass die DNA doppelsträngig und die mRNA einzelsträngig ist und aus einer Folge von Nukleinbasen bestehen, welche einen Code beschreiben, der ausgelesen werden kann und möchte jetzt diese als `char`-Listen darstellen.

Für die Basen (Cytosin Guanin Adenin Thymin Uracil) nimmt er die Buchstaben “C”, “’G”, “A”, “T”, “U”.

(a) **Code validieren**
DNA Code besteht nur aus den Basen C G A T und mRNA-Code besteht nur aus den Basen C G A U. Deklarieren Sie zwei Prozeduren `isDNA` und `ismRNA` vom Typ `char list → bool`, die gegeben einer (beliebigen) `char`-Liste überprüfen, ob der Code gültig ist.

(b) **Komplementäre Base ermitteln**
Jede Base hat eine Partnerbase:

- Umwandlung von DNA-Code zu DNA-Zweitstrang: `C -> G, G -> C, T -> A, A -> T`
- Umwandlung von DNA-Code zu mRNA-Code: `C -> G, G -> C, T -> A, A -> U`

Deklarieren Sie eine Prozedur `complementary : bool * char → char`, die die passende Partnerbase zuordnet. Die erste Komponente im Tupel gibt an, ob die Ausgangsbase in eine Base des DNA-Gegenstrang

[1] Andreas hat diese Aufgabe erstellt. Vielen Dank!

oder der mRNA umcodiert werden soll.
Werfen Sie bei einer ungültigen Base eine Ausnahme. Verwenden Sie kein Konditional.

(c) **DNA und mRNA Stränge erstellen**
Deklarieren Sie nun, mit Hilfe von `complementary` zwei Prozeduren vom Typ `char list → char list`:

`transcription` soll aus einer Liste von DNA-Code die dazugehörige mRNA-Liste zurück geben.
z.B. `[#"C", #"G", #"T", #"A", #"A", #"G"] → [#"G", #"C", #"A", #"U", #"U", #"C"]`

`complementaryStrang` soll aus einer Liste von DNA-Code den dazugehörige Gegenstrang zurück geben.
z.B. `[#"C", #"G", #"T", #"A", #"A", #"G"] → [#"G", #"C", #"A", #"T", #"T", #"C"]`

Tipp: Verwenden Sie Faltung. Überlegen Sie sich zuerst, ob `foldl` oder `foldr` besser geeignet ist.

(d) **Den mRNA Strang entlang laufen**
Ein Ribosom läuft (vereinfacht erklärt) den Strang entlang, erkennt mit Hilfe seiner tRNA Basen-Tripletts (Codons) und baut eine Aminosäurekette auf.

Spielen Sie Ribosom und schreiben Sie eine Prozedur `triplet : char list → string list` welche immer drei Basen zusammenfasst. z.B.:
`[#"C", #"C", #"G", #"U", #"A", #"G", #"A", #"G", #"C"] → ["CCG", "UAG", "AGC"]`

Da eine Aminosäure immer aus drei Basen besteht, können Sie davon ausgehen, dass die Länge der mRNA-Liste ein vielfaches von 3 ist.

Hinweis: Ihre Prozedur soll nicht rekursiv sein. Verwenden Sie Faltung.

(e) **Tripletts suchen**
Deklarieren Sie eine Prozedur `amino : string → string list → bool`, die ein gesuchtes Triplett in einer Triplett-Liste sucht.
z.B. `amino "AUA" ["AUU","AUA","GUA","ACA"] = true`

(f) **Proteinfaltung**
Jedes Triplett kodiert eine Aminosäure. Hier ein Auszug[2]:

```
val AminoAcids = [("Start","AUG"),("Stop","UAA"),("Leucin","CUC"),("Serin","UCA"),("Alanin","UCA"), ("Cystein","UGA"),("Asparagin","GAG"),("Histidin","CAU")]
```

(i) Deklarieren Sie eine Prozedur `checkAS : (string * string) list → string → bool`, die überprüft, ob ein Triplett in der Liste ist.

(ii) Deklarieren Sie nun eine Prozedur `nameAS : (string * string) → string → string`, die den Namen eines Tripletts nachschlägt.

(iii) Deklarieren Sie eine Prozedur `tRNA : (string * string) list → string list → string list`, die aus einer Triplett-Liste eine Liste mit Namen der Aminosäuren erstellt. Da wir nur an den in der Liste aufgeführen Aminoräuren interessiert sich, sollen alle anderen ignoriert werden.

Lösungsvorschlag TD.24

(a)

```
fun isRNA code = foldl (fn (#"C", true) => true | (#"G", true) => true | (#"A", true) => true | (#"U", true) => true | _ => false) true code

fun isDNA code = foldl (fn (#"C", true) => true | (#"G", true) => true | (#"A", true) => true | (#"T", true) => true | _ => false) true code
```

(b)

```
exception WrongBase

fun complementary ( _ ,#"G") = #"C"
  | complementary ( _ ,#"C") = #"G"
  | complementary ( _ ,#"T") = #"A"
  | complementary ( true ,#"A") = #"U"
  | complementary ( _ ,#"A") = #"T"
  | complementary _ = raise WrongBase
```

[2] Es gibt noch viel mehr Aminosäuren: https://de.wikipedia.org/wiki/Code-Sonne

(c)

```
fun transcription code = foldr (fn (x,s) => complementary(true,x)::s ) nil code

fun complementaryStrand code = foldr (fn (x,s) => complementary(false,x)::s ) nil code
```

(d)

```
fun triplet mRNA = #3(foldr
 (fn (nb,(n,T,L)) => if n < 2 then (n+1,nb::T,L) else (0, nil, implode (nb:: T)::L))
 (0,nil,nil) mRNA)
```

(e)

```
fun amino AS TP = foldl (fn (x,b) => b andalso x = AS) false TP
```

(f)

```
fun checkAS (AS_List: (string*string) list) AS = foldl (fn (x,b) => b andalso #2x = AS) false AS_List

fun nameAS (AS_List: (string*string) list) AS = foldr (fn (T,y) => if (#2 T) = AS then (#1 T) else y) "Not Found" AS_List

fun tRNA (AS_List: (string*string) list) mRNA = foldr (fn (x,y) => if checkAS AS_List x then (nameAS AS_List x)::y else y) nil mRNA
```

Strings und Chars

Aufgabe TD.25 *(Error Unknown Character)*
Schreiben Sie eine Prozedur `clean : char list -> string list -> string list` die, gegeben eine Liste von verbotenen Zeichen und eine Liste von Strings, alle Strings rausfiltert, die verbotene Zeichen enthalten.

Beispiel: Es soll `clean [#"A", #"S"] ["ABBA", "INXS", "ACDC", "REM"] = ["REM"]` gelten.

Sie können wie folgt vorgehen:

(a) Schreiben Sie eine Prozedur `isAllowed : char list -> string -> bool` die, gegeben eine Liste von verbotenen Zeichen und einen String, entscheidet, ob der String ein verbotenes Zeichen enthält.

(b) Schreiben Sie nun mit Hilfe von `isAllowed` die Prozedur `clean`

Lösungsvorschlag TD.25

(a)

```
fun isAllowed forbiddenChars s =
    let
        val stringChars = explode s
        fun charForbidden c = List.exists (fn f => f = c) forbiddenChars
    in
        not (List.exists (fn c1 => charForbidden c1) stringChars)
    end
```

(b)

```
fun clean forbidden stringList =  List.filter (isAllowed forbidden) stringList
```

Aufgabe TD.26 *(Zahltag)*
Schreiben Sie eine Prozedur `number: string -> int`, die eine Zeichenkette als Ganzzahl interpretiert.

Beispiel: `number("1234")` soll zu 1234 auswerten.

Lösungsvorschlag TD.26

```
fun number x = foldl (fn (c, n) => n * 10 + ord c - ord #"0") 0 (explode x)
```