



Programmierung 1 (WS 2020/21)

Aufgaben für die Übungsgruppe F (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.

Aus gegebenem Anlass

Aufgabe TF.1 (*true story*)

Vereinfachen Sie folgende Konditionale:

- (a) `if b = true then true else false`
- (b) `if b then false else true`
- (c) `if x = true then akku else false`
- (d) `if ord c >= 97 then if ord c <= 122 then if a then true
else false else false else false`
- (e) `if cmp(x, p) = LESS orelse cmp(x, p) = EQUAL then true else false`
- (f) `if #1(a, b, c) then #2(a, b, c) else #3(a, b, c)`

Lösungsvorschlag TF.1

- (a) `b`
- (b) `not b`
- (c) `akku andalso b`
- (d) `a andalso ord c >= 97 andalso ord c <= 122`
- (e) `cmp(x, p) <> GREATER`
- (f) `if a then b else c`

Optionen

Aufgabe TF.2 (*Er suchte immer so lange, bis er vergaß, was er suchte.*)

Schreiben Sie eine Prozedur $find : (\alpha \rightarrow \text{bool}) \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ option}$, die zu einer Prozedur und einer Liste das erste Element der Liste liefert, für das die Prozedur `true` liefert.

Lösungsvorschlag TF.2

```
1 fun find p (x::xs) = if p x then SOME x else find p xs
2 | find p nil      = NONE
```

Aufgabe TF.3 (*Letztes Element einer Liste*)

Schreiben Sie eine Prozedur $last : \alpha \text{ list} \rightarrow \alpha \text{ option}$, die das letzte Element einer Liste liefert.

Lösungsvorschlag TF.3

```

1 fun last nil = NONE
2   | last (x::nil) = SOME x
3   | last (x::xs) = last xs

```

Aufgabe TF.4 (*Wrapper*)

- (a) Schreiben Sie eine Prozedur $toOption : \alpha \rightarrow \alpha \text{ option}$, die ein Element in eine eingelöste Option umwandelt.
- (b) Deklarieren Sie nun eine Prozedur $toOptionList : \alpha \text{ list} \rightarrow \alpha \text{ option list}$, die aus jedem Element einer Liste eine eingelöste Option macht.

Lösungsvorschlag TF.4

- (a)

```
1 fun toOption x = SOME x
```

- (b)

```
1 fun toOptionList xs = map toOption xs
```

Aufgabe TF.5 (*Matrjoschkas sind immer eine gute Option*)

Eine Matrjoschka ist eine (aus Holz gefertigte und bunt bemalte) ineinander schachtelbare, eiförmige russische Puppe. Eine Matrjoschka besteht also aus einer Reihe von Holzpuppen, die bis auf die innerste Puppe alle hohl sind und kleinere Puppen aufnehmen können.

- (a) Deklarieren Sie einen Datentyp in SML, der eine so zusammengesetzte Matrjoschka Puppe *mit Hilfe von Optionen* darstellen kann. Dabei soll `M NONE` die Darstellung der innersten Puppe sein.
- (b) Schreiben Sie eine Prozedur $count : \text{matrjoschka} \rightarrow \text{int}$, die eine gegebene Matrjoschka durchzählt, also feststellt, aus wie viel Puppen sie besteht.

Lösungsvorschlag TF.5

- (a)

```
1 datatype matrjoschka = M of matrjoschka option
```

- (b)

```
1 fun count (M NONE) = 1
2   | count (M (SOME x)) = 1 + count x
```

Aufgabe TF.6 (*Wörterbuchweisheiten*)

Schreiben Sie eine Prozedur $get : (\text{int} * \alpha) \text{ list} \rightarrow \text{int} \rightarrow \alpha \text{ option}$, die zu einem gegebenen Index aus einer Liste mit Index-Wert-Tupeln den zugehörigen Wert findet. Sollte der übergebene Index nicht existieren, soll `NONE` zurückgegeben werden.

Lösungsvorschlag TF.6

```

1 fun get list key =
2   foldr (fn ((index, value), s) => if index = key then SOME value else s) NONE list

```

Rekursive Konstruktoren

Aufgabe TF.7 (Expressions)

Sie haben folgenden Konstruktor:

- (a) Schreiben Sie eine Prozedur `vars : exp → var list`, die zu einem Ausdruck eine Liste liefert, die die in dem Ausdruck vorkommenden Variablen enthält.
- (b) Schreiben Sie eine Prozedur `count : var → exp → int`, die zählt, wie oft eine Variable in einem Ausdruck auftritt. Beispielsweise tritt x in $x + x$ zweimal auf.
- (c) Schreiben Sie eine Prozedur `check : exp → exp → bool`, die für zwei Ausdrücke e und e' testet, ob e ein Teilausdruck von e' ist.
- (d) Schreiben Sie eine Prozedur `instantiate : env → exp → exp`, die zu einer Umgebung U und einem Ausdruck e den Ausdruck liefert, den man aus e erhält, indem man die in e vorkommenden Variablen gemäß U durch Konstanten ersetzt. Beispielsweise soll für die Umgebung $[x := 5, y := 3]$ und den Ausdruck $A (V \text{"x"}, V \text{"y"})$ der Ausdruck $A (C 5, C 3)$ geliefert werden.
- (e) Schreiben Sie eine Prozedur `eval : env → exp → int` die arithmetische Ausdrücke auswertet.

Lösungsvorschlag TF.7

(a)

```
1 fun vars (V s)      = [s]
2   | vars (A (e, e')) = vars e @ vars e'
3   | vars (M (e, e')) = vars e @ vars e'
4   | vars _          = nil
```

Alternative mit `components` aus §6.4.1

```
1 fun vars (V s) = [s]
2   | vars e = List.concat (map vars (components e))
```

(b)

```
1 fun count s (V s') = if s=s' then 1 else 0
2   | count s (A (e, e')) = count s e + count s e'
3   | count s (M (e, e')) = count s e + count s e'
4   | count _ _       = 0
```

Alternative:

```
1 fun count s (V s') = if s = s' then 1 else 0
2   | count s e = foldl op+ 0 (map (count s) (components e))
```

(c)

```
1 fun check e e' = e==e' orelse
2   case e' of
3     A (e1, e2) ⇒ check e e1 orelse check e e2
4     | M (e1, e2) ⇒ check e e1 orelse check e e2
5     | _        ⇒ false
```

(d)

```
1 fun instantiate env (V s)      = C (env s)
2   | instantiate env (A (e, e')) = A (instantiate env e, instantiate env e')
3   | instantiate env (M (e, e')) = M (instantiate env e, instantiate env e')
4   | instantiate env e          = e
```

(e)

```
1 fun eval f (C n) = n
2   | eval f (V x) = f x
3   | eval f (A (e1, e2)) = eval f e1 + eval f e2
4   | eval f (M (e1, e2)) = eval f e1 * eval f e2
```

Freie Bäume

Aufgabe TF.8 (Aufwärmen)

(a) Geben Sie die grafische Darstellung folgender Bäume an:

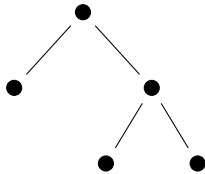
(i) $T[T[]]$, $T[]$

(ii) $T[T[]]$, $T[T[]]$, $T[]$

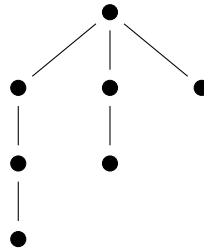
(iii) $T[T[T[]]]$, $T[]$, $T[T[]]$, $T[T[]]$

(b) Leiten Sie die Zeichendarstellung folgender Bäume ab:

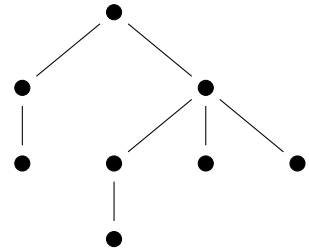
(i)



(ii)



(iii)

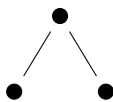


(c) Betrachten Sie Baum (iii) aus Aufgabenteil (b). Wie viele innere Knoten und wie viele Blätter besitzt er?

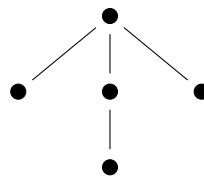
Lösungsvorschlag TF.8

(a) Die grafischen Darstellungen ergeben sich wie folgt:

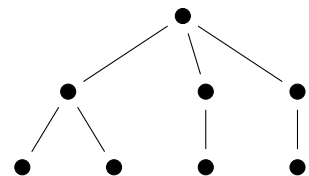
(i)



(ii)



(iii)



(b) (i) $T[T[]]$, $T[T[]]$, $T[]$

(ii) $T[T[T[T[]]]]$, $T[T[]]$, $T[]$

(iii) $T[T[T[]]]$, $T[T[T[]]]$, $T[]$, $T[]$

(c) 4 innere Knoten und 4 Blätter

Aufgabe TF.9 (Baumschule)

Gibt es Bäume, die die folgenden Bedingungen erfüllen? Falls nein: Begründen Sie, warum dies nicht möglich ist. Falls ja: Geben Sie ein Beispiel an. Kann es noch weitere Beispiele geben?

(a) Der Baum hat keinen inneren Knoten.

(b) Der Baum hat nur einen Unterbaum und nur die Größe 4, aber 4 Teilbäume.

(c) Der Baum ist *balanciert* und hat die Größe 6.

(d) Der Baum ist binär und hat mehr innere Knoten als Blätter.

(e) Der Baum ist binär und hat die Größe 5.

(f) Der Baum ist binär und balanciert, und hat die Größe 6.

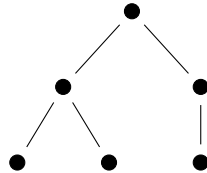
Lösungsvorschlag TF.9

(a) eindeutig: ●

(b) eindeutig:

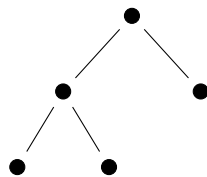


(c) mehrdeutig:



(d) Nicht möglich. Jedem inneren Knoten lässt sich immer mindestens ein Blatt (injektiv) zuordnen, das „unter“ ihm ist.

(e) mehrdeutig:



(f) Nicht möglich. Binäre und balancierte Bäume können nur die Größen $1, 3, 7, 15, \dots$ haben.

Aufgabe TF.10 (Ternärbäume)

Wir nennen einen Baum ternär, wenn jeder seiner (zusammengesetzten) Teilbäume dreistellig ist.

- (a) Schreiben Sie eine Prozedur `test : tree → bool`, die prüft, ob ein Baum ternär ist.
- (b) Schreiben Sie eine Prozedur `ternary : int → tree`, die zu $n > 0$ einen Ternärbaum der Tiefe n liefert.

Lösungsvorschlag TF.10

(a)

```
1 fun test (T nil) = true
2   | test (T [t1, t2, t3]) = test t1 andalso test t2 andalso test t3
3   | test _ = false
```

(b)

```
1 fun ternary 0 = T nil
2   | ternary n = let val t = ternary (n - 1) in T [t, t, t] end
```

Alternative:

```
1 fun ternary n = iter n (T nil) (fn t ⇒ T [t, t, t])
```

Aufgabe TF.11 (Ein Baum, viele Eigenschaften)

Gegeben sei der folgende Baum:

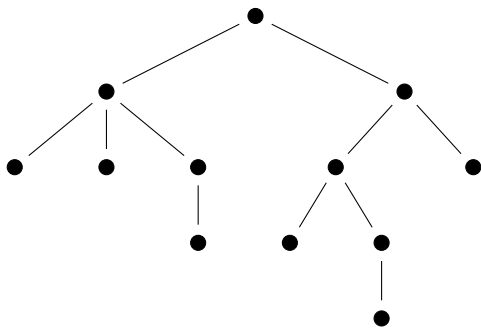
$T[T[T[T[]], T[]], T[T[T[]]], T[T[T[]], T[T[]]], T[]]$

- (a) Zeichnen Sie die grafische Darstellung des Baumes.
- (b) Zeichnen Sie alle Unterbäume des Baumes.
- (c) Zeichnen Sie alle binären Teilbäume.

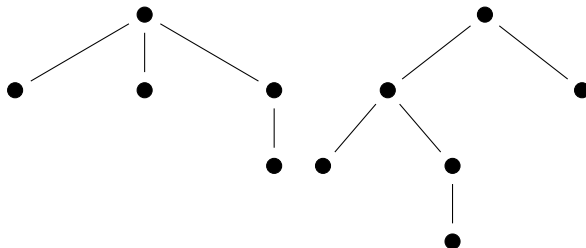
- (d) Geben Sie die Adressen aller Blätter an.
- (e) Wie viele innere Knoten hat der Baum?
- (f) Geben Sie einen Ausdruck an, der den gespiegelten Baum beschreibt.
- (g) Welche Tiefe hat der Baum?
- (h) Welche Breite hat der Baum?
- (i) Welchen Grad hat der Baum?
- (j) Sei a der durch die Adresse [2] bezeichnete Knoten.
- (i) Geben Sie die Adresse des 1. Nachfolgers von a an.
- (ii) Geben Sie die Adressen aller Knoten an, die dem Knoten a untergeordnet sind.

Lösungsvorschlag TF.11

(a)



(b) Es gibt 2 Unterbäume:



(c) Es gibt 4 binäre Teilbäume:



- (d)
- [1,1]
 - [1,2]
 - [1,3,1]
 - [2,1,1]
 - [2,1,2,1]
 - [2,2]

(e) Der Baum hat 6 innere Knoten.

(f) $T[T[T[], T[T[T[]], T[]], T[T[T[]], T[], T[]]]$

(g) Der Baum hat die Tiefe 4.

(h) Der Baum hat die Breite 6.

(i) Der Baum hat den Grad 3.

(j) (i) [2,1]

(ii) [2], [2,1], [2,2], [2,1,1], [2,1,2], [2,1,2,1]

Aufgabe TF.12 (Blätter hinzufügen)

Schreiben Sie eine Prozedur `addb : tree → tree`, die an jedem inneren Knoten eines Baumes ein zusätzliches Blatt hinzufügt. Verwenden Sie `fold`.

Lösungsvorschlag TF.12

```
1 fun addb t = fold (fn nil ⇒ (T nil) | ts ⇒ T ((T[]) :: ts)) t
```

Aufgabe TF.13 (Binärologie)

Schreiben Sie eine Prozedur `zweierTeilbaum : tree → int`, die testet, wie viele Teilbäume eines Baumes zweistellig sind (d.h. wie viele Teilbäume genau zwei Unterbäume haben). Verwenden Sie `fold`.

■ **Beispiel:** `zweierTeilbaum (T[T[T nil, T nil], T nil]) = 2`

Lösungsvorschlag TF.13

```
1 fun zweierTeilbaum t = fold (fn nil ⇒ 0 | [x, y] ⇒ 1 + x + y | xs ⇒ foldl op+ 0 xs) t
```

Aufgabe TF.14 (n-ologie)

Wir wollen nur die Prozedur `zweierTeilbaum` aus der obigen Aufgabe “Binärologie” verallgemeinern. Schreiben Sie eine Prozedur `nTeilbaum : int → tree → int`, die testet, wie viele Teilbäume eines Baumes n-stellig sind. Verwenden Sie `fold`.

■ **Beispiel:** `nTeilbaum 2 (T[T[T nil, T nil], T nil]) = 2`

Lösungsvorschlag TF.14

```
1 fun nTeilbaum n t = fold (fn xs ⇒ if List.length xs = n then foldl op+ 1 xs else foldl op+ 0  
2   ↳ xs) t
```

Aufgabe TF.15 (Shallow)

Schreiben Sie eine Prozedur `shallow : tree → int`, die die Länge des kürzesten Pfades zu einem Blatt bestimmt (wohingegen `depth` die Länge des längsten Pfades zu einem Blatt bestimmt). Verwenden Sie `fold`. 🤔

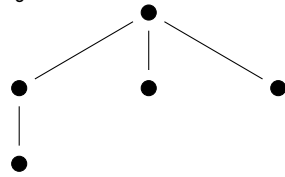
Lösungsvorschlag TF.15

```
1 fun shallow t = fold (fn nil ⇒ 0  
2   ↳ (y::ys) ⇒ 1 + foldl Int.min y ys) t
```

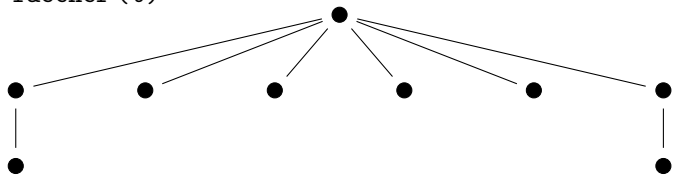
Aufgabe TF.16 (Fächern)

Schreiben Sie eine Prozedur `faecher : tree → tree`, die einen Baum auffächert. Ein aufgefächelter Baum sei ein Baum, bei dem alle Unterbäume aufgefächert sind und bei dem sich die Unterbäume aus der Liste `ts` der (nun aufgefächerten) Unterbäume durch `ts @ (rev ts)` ergeben. Verwenden Sie `fold`.

Beispiel: t



$\text{faecher}(t)$



Lösungsvorschlag TF.16

```
1 fun faecher (T ts) = T ((map faecher ts) @ (rev (map faecher ts)))
```

Alternative:

```
1 fun faecher t = fold (fn ts => T(ts @ (rev ts))) t
```

Aufgabe TF.17 (Achtung, nicht ganz nach Schema!)

Deklarieren Sie eine Prozedur `even : tree → bool`, die für einen Baum `true` ausgibt, wenn die Anzahl der Knoten gerade ist und ansonsten `false` ausgibt. Verwenden Sie nur Werte des Typen `bool` und Prozeduren. Verwenden Sie `fold`.



(a) Deklarieren Sie `even` ohne `fold`.

(b) Deklarieren Sie `even` mit `fold`.

Hinweis: Für b) : Verwenden Sie den Gleichheits-Operator (`=`). Zwei Bäume zusammen betrachtet (ohne den Vorgänger, der beide verbindet) haben genau dann eine gerade Anzahl an Knoten, wenn beide Bäume eine gerade Anzahl Knoten haben oder beide eine ungerade Anzahl Knoten haben.

Lösungsvorschlag TF.17

```
(a)
1 fun even t = let
2     fun even' (T ts, b) = foldl even' (not b) ts
3     in
4     even' (t, true)
5     end
```

`not b` berücksichtigt den Knoten, auf den `even` gerade aufgerufen wird.

```
(b)
1 val even = fold (foldl op= false)
```

Der Startwert ist `false`, da er den Knoten, auf dem `even` aufgerufen wird, repräsentiert. Dieser ist ein atomarer Baum und somit ungerade, also `false`.

Aufgabe TF.18 (Fraktal)

Schreiben Sie eine Prozedur `fraktal : tree → tree`, die die Blätter eines Baumes durch eine Kopie des Eingabebaums ersetzt. Verwenden Sie `fold`.

Lösungsvorschlag TF.18

```
1 fun fraktal' t (T nil) = t
2   | fraktal' t (T ts) = T (map (fraktal' t) ts)
3 fun fraktal t = fraktal' t t
```

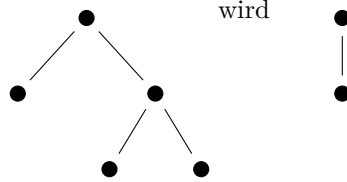
Alternative:

```
1 fun fraktal t = fold (fn nil => t | ts => T ts) t
```


Aufgabe TF.19 (Waldbrand)

Der Baum brennt! Nach dem Feuer sind alle Blätter des Baumes verschwunden, nur noch die verkohlten Äste sind übrig. Schreiben Sie eine Prozedur `feuer : tree → tree`, die alle Blätter eines Baumes entfernt. Lediglich der atomare Baum darf unverändert bleiben.

Beispiel: Aus dem Baum



Lösungsvorschlag TF.19

```
1 fun feuer (T ts) = T (foldr (fn (T [], a) => a
2                               | ( x , a) => feuer x :: a) nil ts)
```

Aufgabe TF.20 (Blätter hinzufügen)

Schreiben Sie eine Prozedur `addb : tree → tree`, die an jedem inneren Knoten eines Baumes ein zusätzliches Blatt hinzufügt. Verwenden Sie `fold`.

Lösungsvorschlag TF.20

```
1 fun addb t = fold (fn nil => (T nil) | ts => T ((T[]) :: ts)) t
```

Aufgabe TF.21 (Fraktal)

Schreiben Sie eine Prozedur `fraktal : tree → tree`, die die Blätter eines Baumes durch eine Kopie des Eingabebaums ersetzt. 🤔

Lösungsvorschlag TF.21

```
1 fun fraktal' t (T nil) = t
2   | fraktal' t (T ts) = T (map (fraktal' t) ts)
3 fun fraktal t = fraktal' t t
```

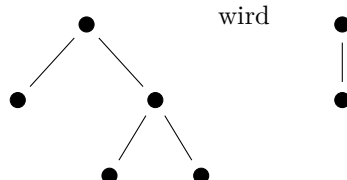
Alternative:

```
1 fun fraktal t = fold (fn nil => t | ts => T ts) t
```

Aufgabe TF.22 (Waldbrand)

Der Baum brennt! Nach dem Feuer sind alle Blätter des Baumes verschwunden, nur noch die verkohlten Äste sind übrig. Schreiben Sie eine Prozedur `feuer : tree → tree`, die alle Blätter eines Baumes entfernt. Lediglich der atomare Baum darf unverändert bleiben. 🤔

Beispiel: Aus dem Baum



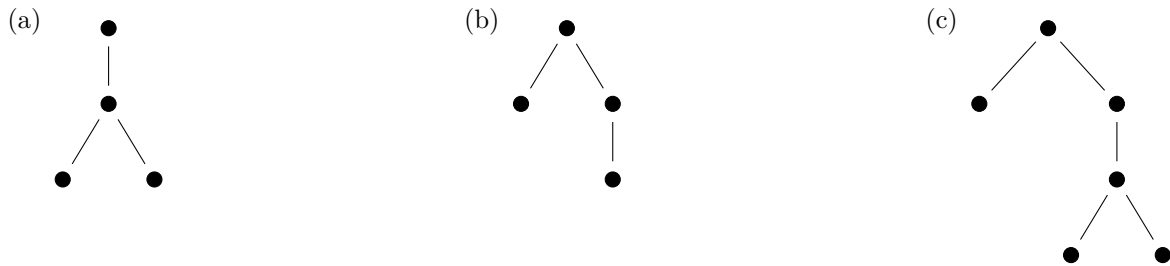
Lösungsvorschlag TF.22

```
1 fun feuer (T ts) = T (foldr (fn (T [], a) => a
2                               | ( x , a) => feuer x :: a) nil ts)
```

Baumkonzepte

Aufgabe TF.23 ("Ortung muss sein" - Google)

Ordnen Sie die folgenden Bäume gemäß der lexikalischen Baumordnung:



Lösungsvorschlag TF.23

$b \prec c \prec a$

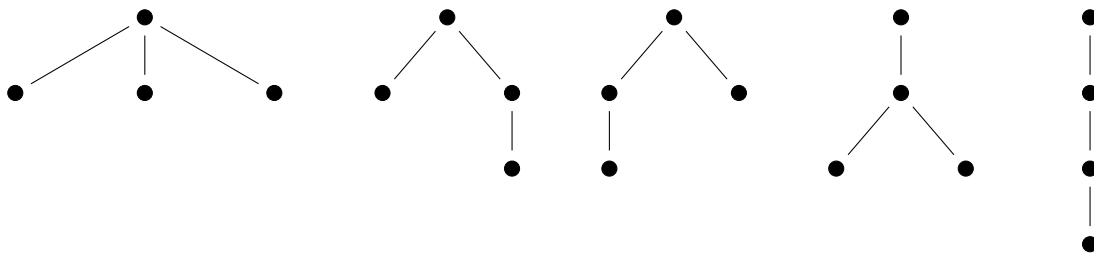
Aufgabe TF.24 (Aufbäumen!)

Zeichnen Sie alle Bäume der Größe 4 und ordnen Sie diese lexikalisch.



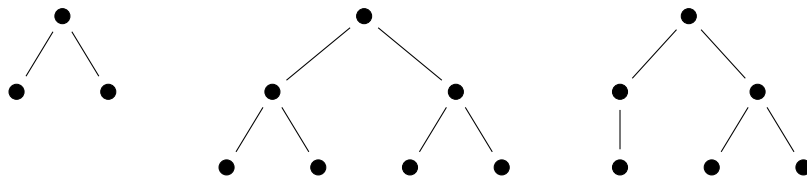
Lösungsvorschlag TF.24

Die Bäume sind von links nach rechts lexikalisch angeordnet:



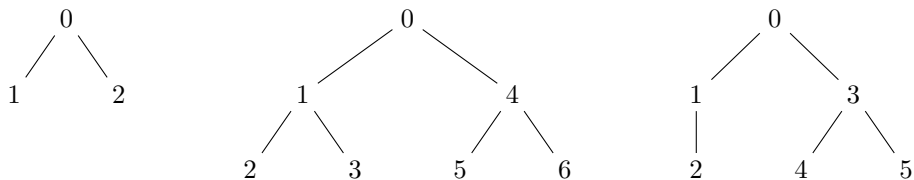
Aufgabe TF.25 (Tour de arbre)

Geben Sie für die folgenden Bäume jeweils die Prä- und die Postnummerierung an:

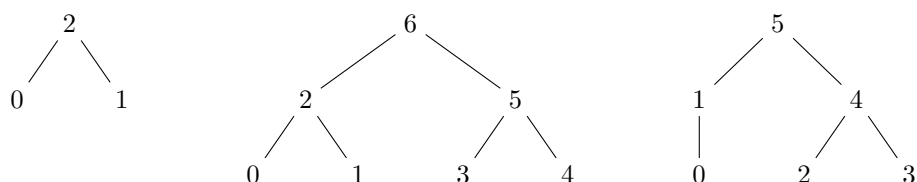


Lösungsvorschlag TF.25

Pränummerierung:

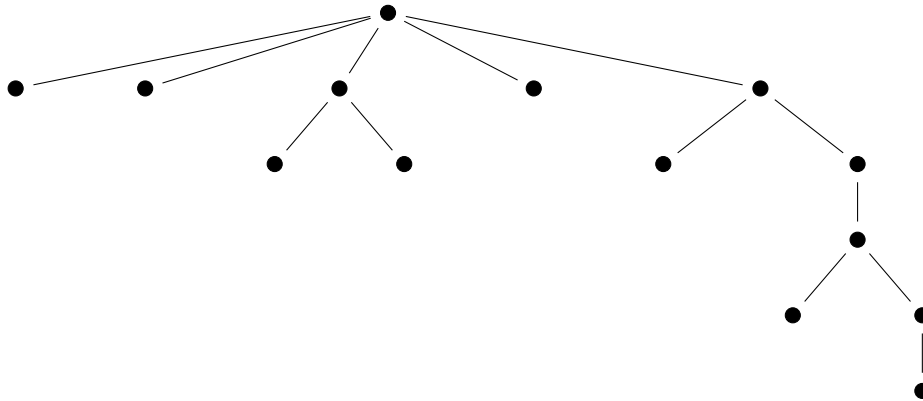


Postnummerierung:



Aufgabe TF.26 (*Linearisierungen schreiben*)

Geben Sie die Prä- und Postlinearisierung des folgenden Baumes an:

*Lösungsvorschlag TF.26*

Prälinearisierung: [5, 0, 0, 2, 0, 0, 0, 2, 0, 1, 2, 0, 1, 0]

Postlinearisierung: [0, 0, 0, 0, 2, 0, 0, 0, 0, 1, 2, 1, 2, 5]


Aufgabe TF.27 (*Prä- und Postlinearisierung*)

- Schreiben Sie eine Prozedur `prelin : tree → int list`, die die Prälinearisierung eines Baumes bestimmt.
- Schreiben Sie eine Prozedur `postlin : tree → int list`, die die Postlinearisierung eines Baumes bestimmt.

Lösungsvorschlag TF.27

- `fun prelin (T tr) = length tr :: (List.concat (map prelin tr))`
- `fun postlin (T tr) = (List.concat (map postlin tr)) @ [length tr]`

Aufgabe TF.28 (*Bäume zählen*)

Schreiben Sie eine Prozedur `count : tree → tree → int`, die zählt, wie oft ein Baum in einem anderen vorkommt. Benutzen sie `fold`. 

Hinweis: Arbeiten Sie mit einem Tupel (aktueller Baum, Anzahl bisher gezählter Bäume) innerhalb von `fold`. Der Gleichheits-Operator `=` funktioniert auch auf Bäumen.

Lösungsvorschlag TF.28

```
1 fun count x t = #2(fold (foldr (fn ((T p,q), (T ap,aq)) => (T((T p)::ap),if (T p)=x then aq+1+
  ↳ q else aq+q)) (T nil,0)) t)
```

Aufgabe TF.29 (*Im Baum nachschlagen*)

Schreiben Sie eine Prozedur `lookup : α ltr → int list → α` , welche für einen markierten Baum die Marke an dem durch die Adresse beschriebenen Knoten zurückgibt. Liegt die Adresse nicht im Baum, so soll eine Ausnahme geworfen werden.

```
1 fun lookup (L (a,xs)) nil = a
2 | lookup (L (a,xs)) (y::yr) = lookup (List.nth (xs,y)) yr
```

Liegt die Adresse nicht im Baum, so wirft List.nth die gefragte Ausnahme.

Aufgabe TF.30 (Gärtnern für Informatiker)

Wir wollen im Folgenden eine Prozedur `reshape : tree → int list → tree → tree` deklarieren, welche in einem Baum, den Teilbaum, welcher an einer übergebenen Adresse liegt, durch einen anderen Baum ersetzt.



- Schreiben Sie die Prozedur `getListNth : α list → int → α` , welche das n-te Element einer Liste liefert. Schreiben Sie außerdem `setListNth : α list → int → α → α list`, welche das n-te Element einer Liste ersetzen soll. Bei ungültigen Eingaben soll eine Ausnahme geworfen werden
- Nutzen Sie nun `getListNth` und `setListNth` um `reshape` zu schreiben. Bei ungültigen Eingaben soll eine Ausnahme geworfen werden.

Lösungsvorschlag TF.30

(a)

```
1 fun getListNth nil _ = raise Subscript
2 | getListNth (x::xr) 0 = x
3 | getListNth (x::xr) n = getListNth xr (n - 1)
4 fun setListNth nil _ _ = raise Subscript
5 | setListNth (x::xr) 0 y = y::xr
6 | setListNth (x::xr) n y = setListNth xr (n - 1) y
```

(b)

```
1 fun reshape t nil newtree = newtree
2 | reshape (T xs) (y::ys) newtree =
3 T (setListNth xs y (reshape (getListNth xs y) ys newtree))
```
