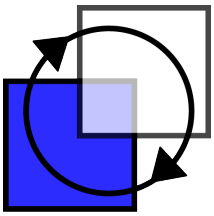
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



# Programmierung 1 (WS 2020/21)
# Zusatztutorium 2 (Lösungsvorschläge)
# Bäume

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

## Reine Bäume

**Aufgabe Z2.1** *(Ein Baum, viele Eigenschaften)*
Gegeben sei der folgende Baum: `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) Zeichnen Sie alle linearen Teilbäume.

(e) Geben Sie die Adressen aller Blätter an.

(f) Wie viele innere Knoten hat der Baum?

(g) Geben Sie einen Ausdruck an, der den gespiegelten Baum beschreibt.

(h) Welche Tiefe hat der Baum?

(i) Welche Breite hat der Baum?

(j) Welchen Grad hat der Baum?

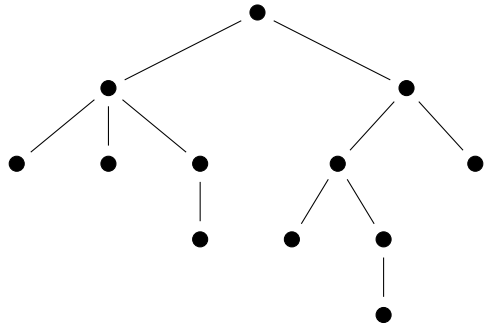
(k) Sei $a$ der durch die Adresse [2] bezeichnete Knoten.

  (i) Geben Sie die Adresse des 1. Nachfolgers von $a$ an.

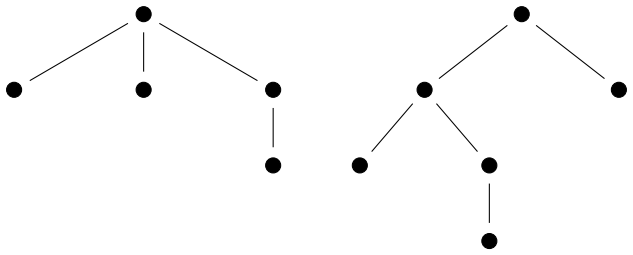
  (ii) Geben Sie die Adressen aller Knoten an, die dem Knoten $a$ untergeordnet sind.

*Lösungsvorschlag Z2.1*

(a)

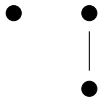


(b) Es gibt 2 Unterbäume:



(c) Es gibt 4 binäre Teilbäume:

(d) Es gibt 2 lineare Teilbäume:



(e)
- `[1,1]`
- `[1,2]`
- `[1,3,1]`
- `[2,1,1]`
- `[2,1,2,1]`
- `[2,2]`

(f) Der Baum hat 6 innere Knoten.

(g) `T [T [T [], T [T [T []], T []]], T [T [T []], T [], T []]]`

(h) Der Baum hat die Tiefe 4.

(i) Der Baum hat die Breite 6.

(j) Der Baum hat den Grad 3.

(k) (i) `[2,1]`

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

**Aufgabe Z2.2** *(Rechnen auf Bäumen 1)*
Schreiben Sie folgende Prozeduren für Bäume. Verwenden Sie dabei `foldl` und `map`, aber nicht `fold`.

(a) `breadth : tree → int`, für die Breite des Baumes.

(b) `size : tree → int`, für die Größe des Baumes.

(c) `depth : tree → int`, für die Tiefe, d.h. maximale Länge der gültigen Adressen.

(d) `degree : tree → int`, für den Grad, also die maximale Stelligkeit des Baumes.

*Lösungsvorschlag Z2.2*

(a)
```
fun breadth (T nil) = 1
  | breadth (T ts)  = foldl op+ 0 (map breadth ts)
```

(b)
```
fun size (T ts) = foldl op+ 1 (map size ts)
```

(c)
```
fun depth (T ts) = 1 + foldl Int.max (~1) (map depth ts)
```

(d)
```
fun degree (T ts) = foldl Int.max (List.length ts) (map degree ts)
```

**Aufgabe Z2.3** *(Teilbäume)*
Deklarieren Sie folgende Prozeduren:

(a) `subtree : tree → tree → bool`, die testet, ob ein Baum Teilbaum eines anderen Baums ist.

(b) `getsub : tree → int list → tree`, die zu einer Adresse den entsprechenden Teilbaum liefert.

*Lösungsvorschlag Z2.3*

(a)
```
fun subtree t (T ts) = (t = T ts) orelse List.exists (subtree t) ts
```

(b)
```
fun getsub (T ts) xs = foldl (fn (x, T ts') ⇒ List.nth(ts', x - 1)) (T ts) xs
```

## Baumfaltung

**Aufgabe Z2.4** *(Rechnen auf Bäumen 2)*
Schreiben Sie Prozeduren `breadth`, `size`, `depth` und `degree`, die den Prozeduren aus Aufgabe Z2.2 entsprechen, mithilfe von `fold`.

*Lösungsvorschlag Z2.4*

(a)
```
fun breadth t = fold (fn nil ⇒ 1 | xs ⇒ foldl op+ 0 xs) t
```

(b)
```
fun size t = fold (foldl op+ 1) t
```

(c)
```
fun depth t = fold (fn xs ⇒ 1 + foldl Int.max (~1) xs) t
```

(d)
```
fun degree t = fold (fn xs ⇒ foldl Int.max (List.length xs) xs) t
```

**Aufgabe Z2.5** *(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 Z2.5*

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

**Aufgabe Z2.6** *(Das Wachsen eines Baums)*
Deklarieren Sie die Prozedur `grow : tree → tree`, welche einen Baum an den Blättern durch eine Kopie des Baums selbst erweitert. Verwenden Sie dabei die Prozedur `fold`.

*Lösungsvorschlag Z2.6*

```
fun grow t = fold (fn nil ⇒ t | ts ⇒ T ts) t
```

**Aufgabe Z2.7** *(Der Wald brennt)*

(a) Der Baum brennt. Nach dem Feuer sind alle Blätter des Baumes verschwunden, nur noch die verkohlten Äste sind übrig. Schreiben sie eine Prozedur $feuer : tree \to tree$, die alle Blätter eines Baumes entfernt. Lediglich der atomare Baum darf unverändert bleiben. Beispiel:

(b) Hat der Baum nach einer Anwendung von feuer noch Blätter? Ist es möglich, dass der Baum nach einer Anwendung mehr Blätter als vorher hat?

(c) Wie oft müssen sie feuer anwenden, um den Baum komplett zu zerstören? Der Baum ist komplett zerstört, wenn nur noch die Wurzel übrig ist, also T[]. Welche wichtige Größe spielt hier eine Rolle?

(d) Schreiben sie eine Prozedur $size : tree \to int$ , die mithilfe von feuer diese Größe bestimmt!

*Lösungsvorschlag Z2.7*

(a)
```
fun feuer (T ts) = T (foldr (fn (T [],a) ⇒ a | (t,a) ⇒ (feuer t)::a) nil ts)
```

(b) Der Baum hat noch mindestens ein Blatt und maximal so viele, wie er vorher hatte, da durch jedes zerstörte Blatt maximal ein Knoten (der Vater) zu einem neuen Blatt wird.

(c) *depth t* mal. Jeder Aufruf von feuer verringert die Tiefe des Baumes um 1.

(d)
```
fun size (T []) = 0
  | size     t      = d (feuer t) + 1
```

**Aufgabe Z2.8** *(Äste Hinzufügen)*
Schreiben Sie eine Prozedur `addb : tree → tree`, die an jedem inneren Knoten eines Baumes einen zusätzlichen Ast hinzufügt. Angewendet auf einen Binärbaum soll `addb` also einen Baum liefern, dessen zusammngesetzte Teilbäume alle dreistellig sind. Verwenden Sie `fold`.

*Lösungsvorschlag Z2.8*

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

**Aufgabe Z2.9** *(Bau(m)anleitungen wie bei IKEA)*
Ein Baum kann durch die Menge seiner möglichen Adressen beschrieben werden, also durch eine `int list list`. Ein Beispiel für eine solche Liste ist:

```
[ [] , [1] , [1,1] , [1,2] , [2] ]
```

Schreiben Sie eine Prozedur `buildTree : int list list → tree`, mit der Sie die Adressenlisten-Baumdarstellung in die Ihnen vertraute Darstellung konvertieren können. Sie können davon ausgehen, dass Sie nur gültige Bäume bekommen und dass die Adressen strikt sortiert sind.

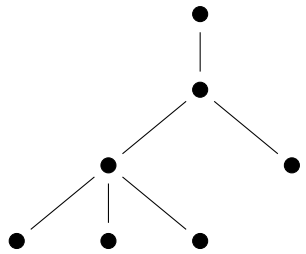
*Lösungsvorschlag Z2.9*
`replaceNth xs n y` ersetzt das $n$-te Listenelement durch $y$. `insertInTree (a, t)` fügt einen neuen Knoten, der als Adresse $a$ übergeben wird, in den Baum $t$ ein. Wir bauen den Baum, indem wir alle Adressen einfügen.

```
fun replaceNth   nil   n y   = raise Subscript
|     replaceNth (x::xr) 0 y = y::xr
|     replaceNth (x::xr) n y = x::(replaceNth xr (n - 1) y)

fun insertInTree (nil, t)    = t
| insertInTree ([a], T ts)   = T (ts @ [T[]])
| insertInTree (a::ar, T ts) =
T (replaceNth ts (a - 1) (insertInTree(ar, List.nth(ts, a - 1))))

fun buildTree adrlist = foldl insertInTree (T[]) adrlist
```

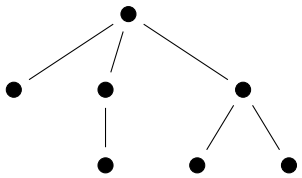
## Nummerierung und Linearisierung

**Aufgabe Z2.10** *(Linearisierung und Nummerierung)*
Annotieren Sie jeweils die Prä- und Postnummerierung an den Graphen. Geben Sie anschließend die Prä- und Postlinearisierung an.

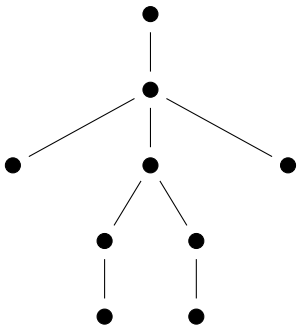
(a)



(b)



(c)



*Lösungsvorschlag Z2.10*

(a) Prä: `[1,2,3,0,0,0,0]`
Post: `[0,0,0,3,0,2,1]`

(b) Prä: `[3,0,1,0,2,0,0]`
Post: `[0,0,1,0,0,2,3]`

(c) Prä: `[1,3,0,2,1,0,1,0,0]`
Post: `[0,0,1,0,1,2,0,3,1]`

**Aufgabe Z2.11** *(Linearisierungsprozeduren)*
Schreiben Sie folgende Prozeduren:

(a) `prelin : tree → int list`, die zu einem Baum die Prälinearisierung liefert.

(b) `postlin : tree → int list`, die die Postlinearisierung eines Baumes liefert.

*Lösungsvorschlag Z2.11*

(a)
```
1 fun prelin (T ts) = (List.length ts) :: List.concat(map (prelin) ts)
```

(b)
```
1 fun postlin (T ts) = (List.concat (map (postlin) ts)) @ [List.length ts]
```

## Markierte Bäume und Projektionen

**Aufgabe Z2.12** *(Supersumme)*
Schreiben Sie eine Prozedur `sum : int ltree → int`, die die Summe aller Marken eines markierten Baumes berechnet.

*Lösungsvorschlag Z2.12*
```
fun sum (L (a, xs)) = foldl op+ a (map sum xs)
```

**Aufgabe Z2.13** *(Gestalt)*
Schreiben Sie eine Prozedur `shape : α ltree → tree`, die die Gestalt eines markierten Baumes zurückgibt.

*Lösungsvorschlag Z2.13*
```
fun shape (L (_, xs)) = T (map shape xs)
```

**Aufgabe Z2.14** *(filter auf markierten Bäumen)*
Schreiben Sie eine Prozedur `filter : ( α → bool ) → α ltr → α list`, die zu einem Prädikat `p` und einem markierten Baum alle Marken in einer Liste liefert, für die das Prädikat zu `true` auswertet. Die Marken sollen gemäß der Präordnung angeordnet sein.

*Lösungsvorschlag Z2.14*

```
fun filter p (L(x, ts)) = (if p x then [x] else nil) @ List.concat (map (filter p) ts)
```

**Aufgabe Z2.15** *(Baum-foltr)*

(a) Schreiben Sie eine Prozedur `foltr : ( α * β list → β ) → α ltr → β`, die Faltung für markierte Bäume realisiert.

(b) Schreiben Sie anschließend mit Hilfe von `foltr` eine Prozedur `sum : int ltr → int`, die alle Marken eines markierten Baumes aufsummiert.

(c) Schreiben Sie nun mit Hilfe von `foltr` die Prozedur `filter` aus Aufgabe Z2.14.

*Lösungsvorschlag Z2.15*

(a)
```
fun foltr f (L (m, ts)) = f (m, map (foltr f) ts)
```

(b)
```
val sum = foltr (fn (m, a) ⇒ foldl op+ m a)
```

(c)
```
val filter = foltr (fn (x, ys) ⇒ (if p x then [x] else nil) @ List.concat ys)
```

**Aufgabe Z2.16** *(Find)*
Schreiben Sie eine Prozedur `find : ( α → bool ) → α ltr → α option`, die falls vorhanden eine Marke eines Baumes zurückgibt, die ein gegebenes Prädikat erfüllt. 🤔

*Lösungsvorschlag Z2.16*

```
fun find p (L(x, ts)) = if p x then SOME x
                        else foldl (fn (SOME x, _) ⇒ SOME x
                                     | (NONE, a) ⇒ a) NONE
                          (map (find p) ts)
```

Alternativ:

```
fun find p (L(x, ts)) = if p x then SOME x
                        else valOf (List.find isSome (map (find p) ts)) handle Option ⇒ NONE
```

**Aufgabe Z2.17** *(Binary Matching)*
Schreiben Sie eine Prozedur `tCheck : ('α * 'α → bool) → 'α ltr → bool`, die auf Binärbäumen ein Prädikat für alle Knoten prüft. Das Prädikat erhält als Argumente die derzeitige Marke des Knoten und die Marke eines der Kinder. `tcheck` soll `true` zurückgeben, wenn das Prädikat für alle Knoten mit seinen Kindern `true` liefert. Für Blätter wertet die Prozedur immer zu `true` aus, da diese keine Kinder haben.

Sollte es sich bei dem Baum nicht um einen Binärbaum handeln, soll die `exception NotABinaryTree` geworfen werden.

**Hinweis:** Überlegen Sie sich, wie Sie sinnvoll Pattern-Matchen können.

*Lösungsvorschlag Z2.17*

```
exception NotABinaryTree;

fun tCheck p (L(x, nil)) = true
  | tCheck p (L(x, [L(y,t1),T(z,t2)])) = p(x, y) andalso p(x, z) andalso checker p  (L(y, t1))
                                                 andalso checker p (L(z,t2))
  | tCheck p (L(x, [L(y,t),L(z,nil)])) = p(x, y) andalso p(x, z) andalso checker p  (L(y, t))
  | tCheck p (L(x, [L(y,nil),L(z,t)])) = p(x, y) andalso p(x, z) andalso checker p (L(z, t))
  | tCheck p (L(x, [L(y,nil),L(z,nil)])) = p(x, y) andalso p(x, z)
  | tCheck p _ = raise NotABinaryTree;
```

**Aufgabe Z2.18** *(Suchbäume)*
Suchbäume sind markierte Bäume, für die gilt:

1. Jeder Teilbaum hat maximal 2 Unterbäume.
2. Alle Marken im linken Unterbaum jedes Teilbaums sind kleiner als die Marke der Wurzel des Teilbaums
3. Alle Marken im rechten Unterbaum jedes Teilbaums sind größer als die Marke der Wurzel des Teilbaums.
4. Doppelauftreten von Marken gibt es nicht.

Wir beschränken uns hier auf Marken des Typs `int`.

(a) Überlegen Sie sich, wie der Suchbaum aussieht, der sich aus den Marken 4, 8, 5, 7, 1, 9 aufbaut, wenn diese nach und nach in den Suchbaum, der nur aus der Wurzel 6 besteht, eingefügt werden.

(b) Schreiben Sie eine Prozedur `insert : int ltr → int → int ltr`, die in einen Suchbaum die übergebene Marke so einfügt, dass der Baum ein Suchbaum bleibt. Sie dürfen annehmen, dass der übergebene Baum binär ist, alsojeder Teilbaum entweder 2 oder 0 Unterbäume hat.

(c) Welchen Vorteil bietet ein Suchbaum, wenn nach dem Auftreten einer Marke gesucht wird?

(d) Schreiben Sie eine Prozedur `search : int ltr → int → bool`, die in einem Suchbaum nach einer Marke sucht. Gehen Sie dabei von dem Spezialfall aus, dass der Suchbaum ein binärer markierter Baum ist.

**Hinweis:** Nutzen Sie hierbei die Vorteile des binären Suchbaums aus. Das bedeutet vor allem, dass Sie meistens nicht alle Marken des Baumes besuchen müssen.

*Lösungsvorschlag Z2.18*

(a) `L(6, [L(4, [L(1,nil), L(5, nil)]), L(8, [L(7, nil), L(9, nil)])])`

(b)
```
fun insert (L(x,ts)) y =
       if x=y then L(x,ts)
       else case ts of
               nil    ⇒ L(x,[L(y,nil)])
           | [t1,t2] ⇒ if y<x then L(x,[insert t1 y, t2])
                       else L(x,[t1, insert t2 y])
```

(c) Man weiß an jeder Marke eines Teilbaums genau, in welchem Unterbaum weiter gesucht werden muss. So können viele Marken schnell ausgeschlossen werden, die gar nicht besucht werden müssen.

(d)

```
fun search (L(x,nil)) y = x=y
  | search (L(x,[t1,t2])) y = case Int.compare (y,x) of
                                  LESS    ⇒ search t1 y
                                | EQUAL   ⇒ true
                                | GREATER ⇒ search t2 y
```