Programmierung 1

Vorlesung 12

Livestream beginnt um 10:20 Uhr

Bäume,
Teil 3

Programmierung 1

Faltung

$$fold : (\alpha\ list \to \alpha) \to tree \to \alpha$$

Schrittprozedur Baum

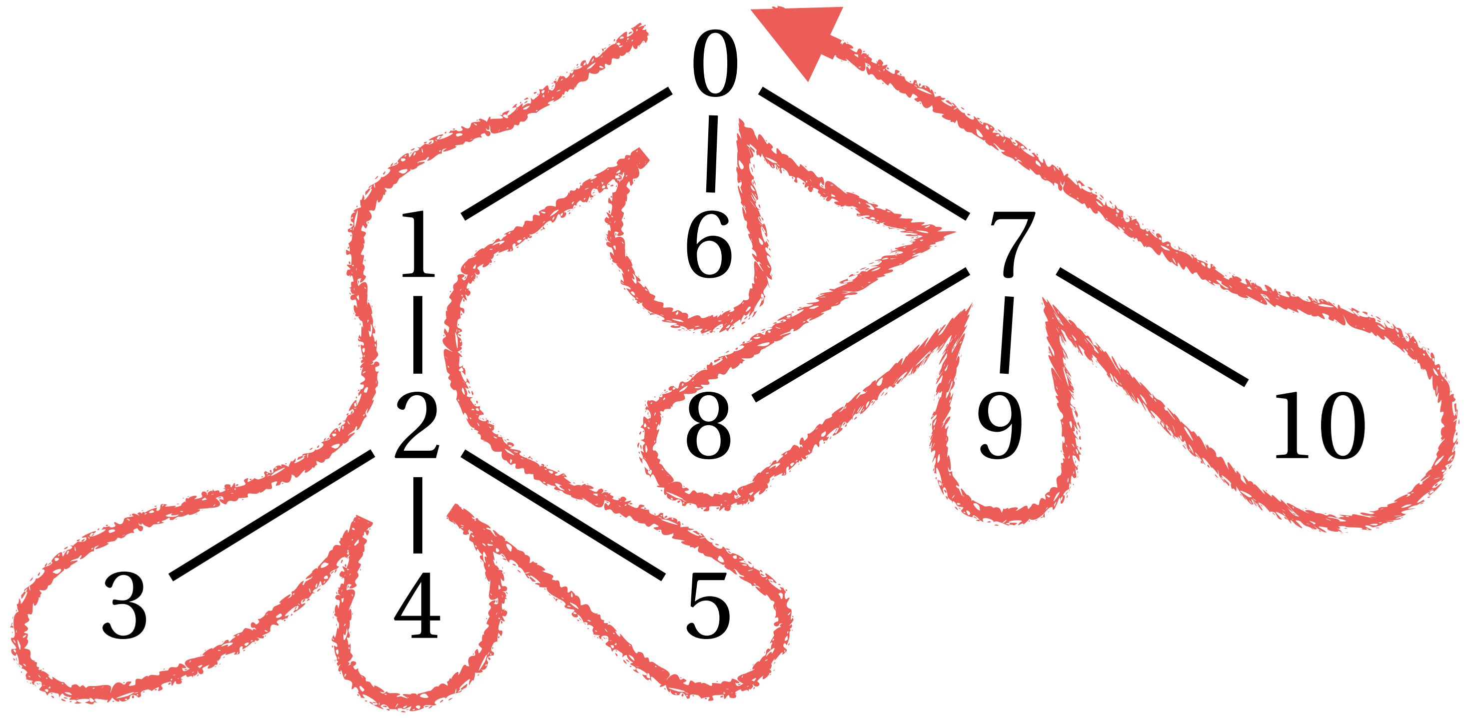
- Die **Schrittprozedur** bestimmt den **Wert eines Baums** aus den **Werten der Unterbäume**

```
fun fold f (T ts) = f (map (fold f) ts)
```

$val\ fold : (\alpha\ list \to \alpha) \to tree \to \alpha$

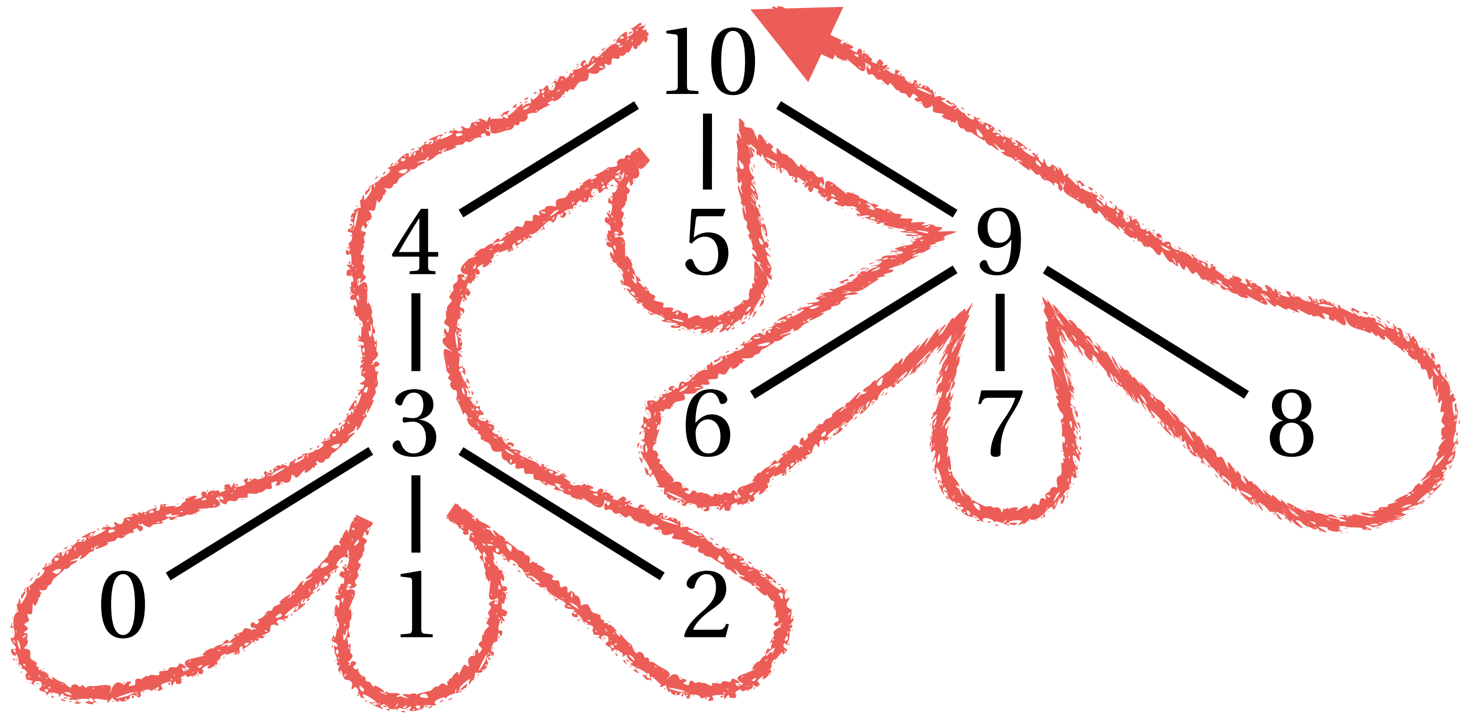
Pränummerierung vs. Postnummerierung

- **Pränummerierung**
  induziert **Präordnung**



Knoten wird beim **ersten Besuch** nummeriert.

- **Postnummerierung**
  induziert **Postordnung**



Knoten wird beim **letzten Besuch** nummeriert.

Frage

Welcher Teilbaum von T[T[T[]],T[]] hat Nummer 3 gemäß Pränummerierung?

- T[] ✓
- T[T[]]
- T[T[T[]],T[]]
- keiner

Teilbaumzugriff mit Pränummern

- **Auflisten aller Teilbäume** in **Präordnung**

```
fun presubtrees (T ts) =
foldl (fn (t,tr) => tr @ (presubtrees t))
            [T ts] ts
```

- **Zugriff auf Teilbaum** mit **Pränummer**

```
fun prest t k = List.nth(presubtrees t,k)
```

oder (besser:)

```
fun prest t =
let
  fun prel nil k = raise Subscript
    | prel (t::tr) 0 = t
    | prel ((T ts)::tr) k = prel (ts@tr) (k-1)
in
  prel [t]
end
```

Teilbaumzugriff mit Postnummern

- **Auflisten aller Teilbäume** in **Postordnung**

```
fun postsubtrees (T ts) =
foldr (fn (t,tr) => (postsubtrees t) @ tr) [T ts] ts
```

- **Zugriff auf Teilbaum** mit **Postnummer**

```
fun postst t k = List.nth(postsubtrees t,k)
```

wie besser?

Teilbaumzugriff mit Postnummern

- **Agenda:** tree entry list

```
datatype ‘a entry = I of ‘a | F of ‘a
```

`I`: “Muss noch komplett bearbeitet werden”

`F`: “Unterbäume befinden sich bereits weiter links auf der Agenda”

- **Zugriff auf Teilbaum** mit **Postnummer**

```
fun postst t =
let
  fun postl nil k = raise Subscript
    | postl (I (T ts)::es) k =
      postl ((map (fn t => (I t)) ts) @ ((F (T ts))::es)) k
    | postl (F t::es) 0 = t
    | postl (F t::es) k = postl es (k-1)
in
  postl [I t]
end
```

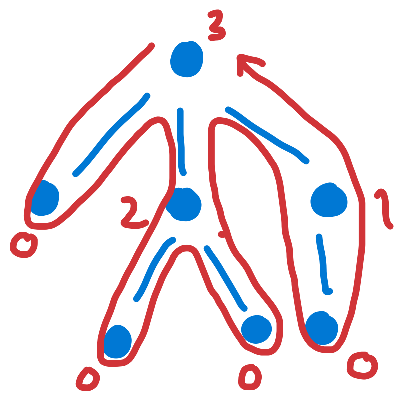
Linearisierungen

- **Bäume** können als **Listen von natürlichen Zahlen** dargestellt werden. Dabei wird jeder **Knoten** durch seine **Stelligkeit** dargestellt.

- **Prälinearisierung:** Anordnung nach **Präordnung**

```
fun pre (T ts) = length ts :: List.concat(map pre ts)
```

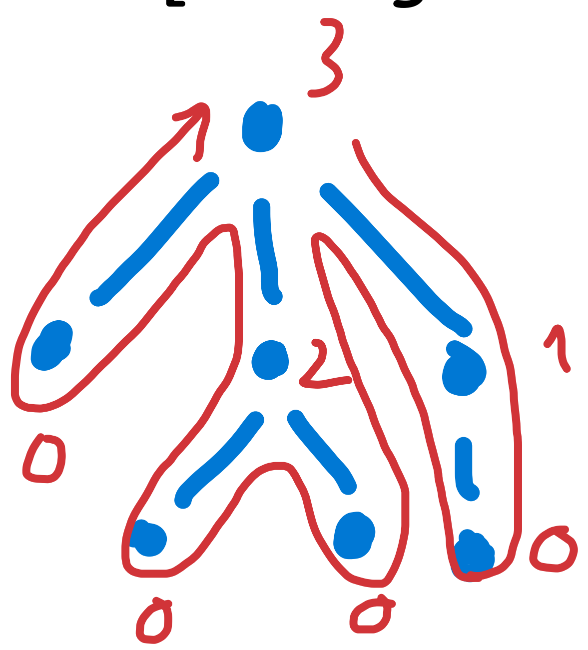
$$\begin{array}{rcl} T[\,] & \rightsquigarrow & [0] \\ T[T[\,]] & \rightsquigarrow & [1,0] \\ T[T[\,],T[\,]] & \rightsquigarrow & [2,0,0] \\ T[T[\,],\ T[T[\,],T[\,]],\ T[T[\,]]] & \rightsquigarrow & [3,0,2,0,0,1,0] \end{array}$$



- **Postlinearisierung:** Anordnung nach **Postordnung**

```
fun post (T ts) = List.concat(map post ts) @ [length ts]
```

$$\begin{array}{rcl} T[\,] & \rightsquigarrow & [0] \\ T[T[\,]] & \rightsquigarrow & [0,1] \\ T[T[\,],T[\,]] & \rightsquigarrow & [0,0,2] \\ T[T[\,],\ T[T[\,],T[\,]],\ T[T[\,]]] & \rightsquigarrow & [0,0,0,2,0,1,3] \end{array}$$



Frage

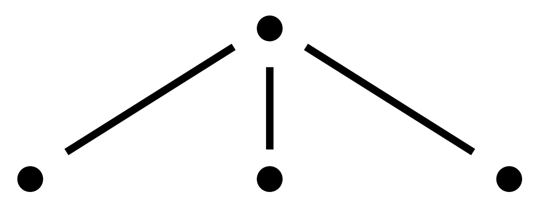
Welcher der folgenden Listen stellt keine Prä- oder Postlinearisierung eines Baums dar?

- [1] ✓
- [1,0] ✗
- [0,1] ✗
- [0,1,0] ✓

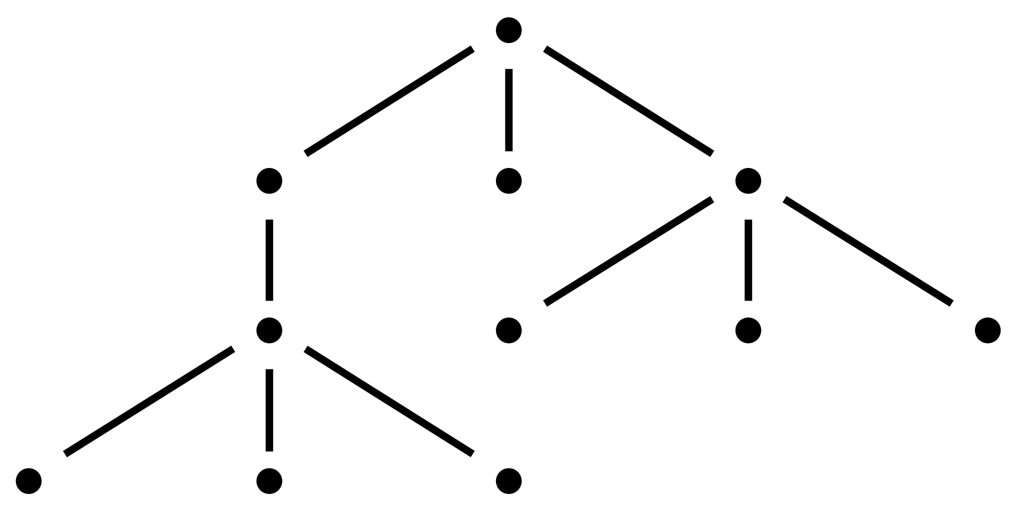
Balancierte Bäume

Ein Baum wird als **balanciert** bezeichnet, wenn die **Adressen** seiner **Blätter** alle die **gleiche Länge** haben.

balanciert



balanciert



nicht balanciert

Test auf Balance

Proposition: Ein Baum ist genau dann **balanciert**, wenn seine **Unterbäume** alle **balanciert** sind **und** alle die **gleiche Tiefe** haben.

```
exception Unbalanced

fun check (n,m) = if n=m then n else raise Unbalanced

fun depthb (T nil) = 0
  | depthb (T(t::tr)) = 1+foldl check (depthb t) (map depthb tr)
```

val depthb : tree → int `(* Unbalanced *)`

```
fun balanced t = (depthb t ; true) handle Unbalanced => false
```

val balanced : tree → bool

Frage

Welche der folgenden Aussagen sind korrekt?

- Alle Unterbäume eines balancierten Baums sind balanciert. ✓
- Alle Unterbäume eines balancierten Baums haben die gleiche Tiefe. ✓
- Alle Teilbäume eines balancierten Baums sind balanciert. ✓
- Alle Teilbäume eines balancierten Baums haben die gleiche Tiefe. ✗

Finitäre Mengen

- Eine **Menge** heißt **rein,**
  wenn **jedes Ihrer Elemente** eine reine Menge ist.
- Eine **Menge** heißt **finitär,**
  wenn sie **endlich** ist **und**
  jedes Ihrer Elemente eine finitäre Menge ist.

- **Beispiele:**
  $\{\emptyset, \{\emptyset\}\}$: rein und finitär
  $\{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \ldots\}$: rein aber nicht finitär

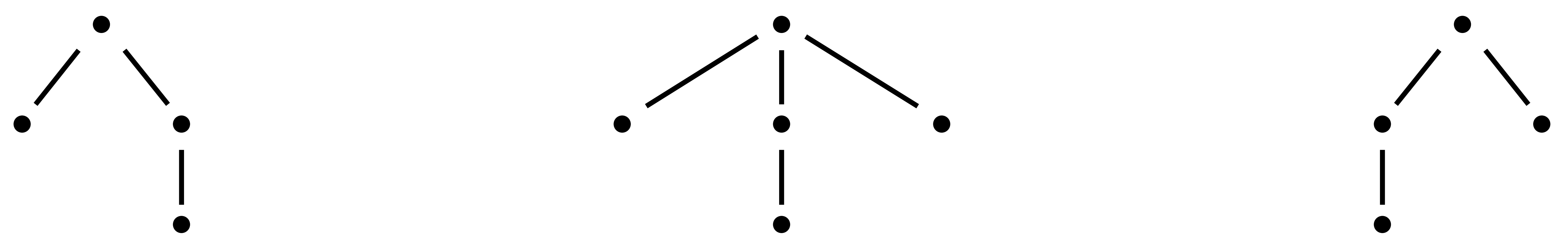
- Reine Mengen sind eine universelle Datenstruktur für die Darstellung mathematischer Objekte!

Darstellung finitärer Mengen

- Wir benutzen **reine Bäume** zur Darstellung finitärer Mengen:
  $\mathrm{Set}(\,T\,[t_1,\ldots,t_n]\,) = \{\,\mathrm{Set}\ t_1,\ldots,\ \mathrm{Set}\ t_n\,\}$

- Die Darstellung ist **nicht eindeutig**.

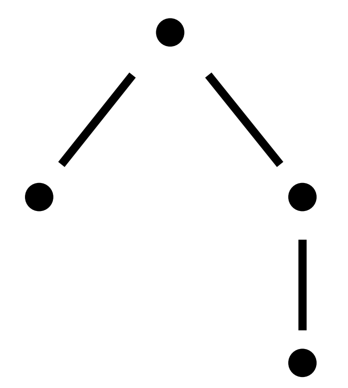
- **Beispiel** $\{\emptyset, \{\emptyset\}\}$:



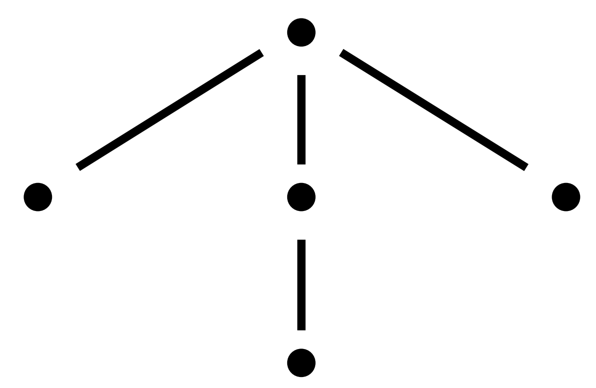
Gerichtete Bäume

- Ein Baum ist **gerichtet,** wenn seine Unterbaumlisten **strikt sortiert** sind gemäß der lexikalischen Baumordnung.

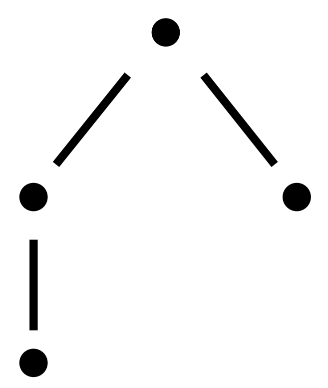
- **Beispiel:**



gerichtet



nicht gerichtet

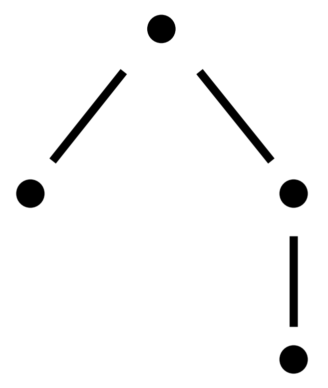


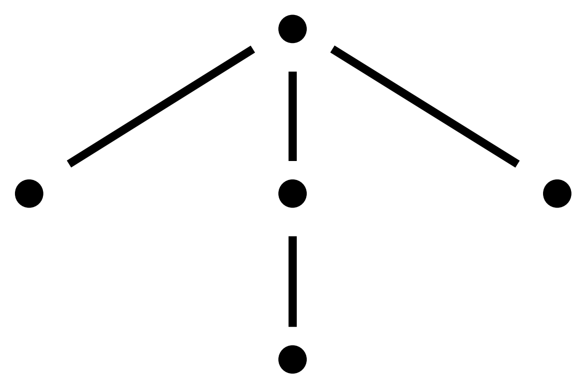
nicht gerichtet

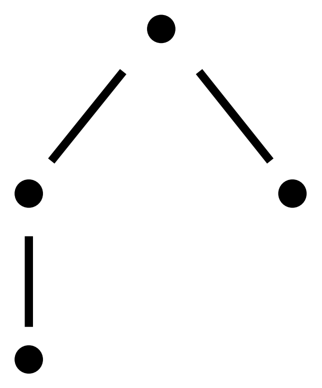
```
fun strict(t::t'::tr) = compareTree(t,t')=LESS andalso strict(t'::tr)
  | strict    _         = true

fun directed (T ts) = strict ts andalso List.all directed ts
```

val directed : tree → bool

Mengengleichheit







Unter welchen Umständen stellen
zwei **reine Bäume** die **gleiche Menge dar?**

1. **Set t_1 = Set t_2** genau dann,
   wenn Set $t_1 \subseteq$ Set t_2 **und** Set $t_2 \subseteq$ Set t_1.
2. **Set t_1 $\subseteq$ Set t_2** genau dann,
   wenn Set $t_1' \in$ Set t_2 für **jeden Unterbaum** t_1' von t_1.
3. **Set t_1 $\in$ Set t_2** genau dann,
   wenn Set $t_1 =$ Set t_2' für **einen Unterbaum** t_2' von t_2.

Verschränkte Rekursion

1. **Set t_1 = Set t_2** genau dann,
   wenn Set $t_1 \subseteq$ Set t_2 **und** Set $t_2 \subseteq$ Set t_1.
2. **Set t_1 $\subseteq$ Set t_2** genau dann,
   wenn Set $t_1{}' \in$ Set t_2 für **jeden Unterbaum** $t_1{}'$ von t_1.
3. **Set t_1 $\in$ Set t_2** genau dann,
   wenn Set t_1 = Set $t_2{}'$ für **einen Unterbaum** $t_2{}'$ von t_2.

```
fun eqset x y = subset x y andalso subset y x

and subset (T xs) y = List.all (fn x => member x y) xs

and member x (T ys) = List.exists (eqset x) ys
```

Verschränkte Rekursion

- **Beispiel:** Die **Größe eines Baums**

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

val size : tree → int

```
val size = fold (foldl op+ 1)
```

```
fun size (T ts) = foldl accusize 1 ts

and accusize (t,a) = size t + a
```

Von verschränkter zu einfacher Rekursion

Verschränkte Rekursion kann stets auf **einfache Rekursion** **zurückgeführt** werden.

Beispiel:

- ```
  fun f (x,y) = if x>0 then f(x-1,y) + g(x,y-1) else 1
  and g (x,y) = if y>0 then f(x-1,y) + g(x,y-1) else 1
  ```

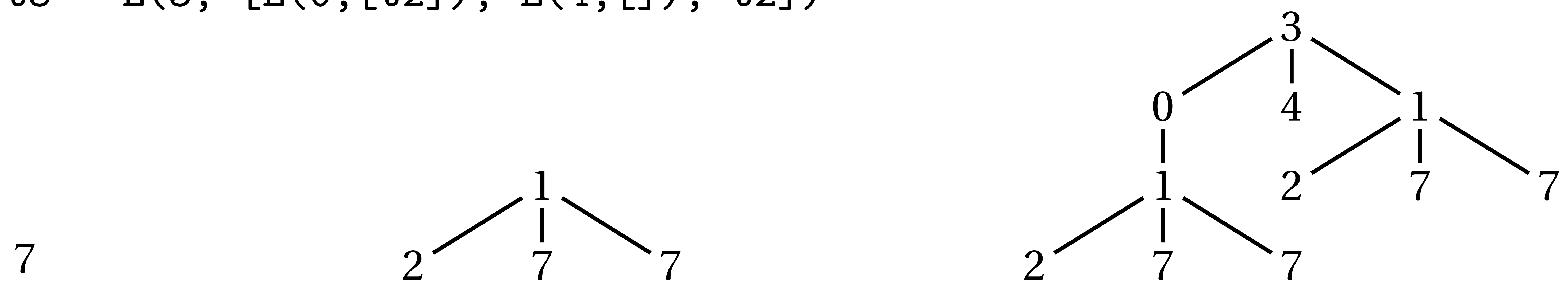
- ```
  fun f’ g (x,y) = if x>0 then f’ g (x-1,y) + g(x,y-1) else 1
  fun g (x,y) = if y>0 then f’ g (x-1,y) + g(x,y-1) else 1
  fun f (x,y) = f’ g (x,y)
  ```

Markierte Bäume

- In einem **markierten Baum** ist jeder **Knoten** mit einer **Marke** (einem **Wert** aus einem **Grundtyp)** versehen.

```
datatype 'a ltr = L of 'a * 'a ltr list
```

```
val t1 = L(7,[])
val t2 = L(1, [L(2,[]), t1, t1])
val t3 = L(3, [L(0,[t2]), L(4,[]), t2])
```



$t1 = L(7, [])$ $t2 = L(1, [L(2, []), t1, t1])$ $t3 = L(3, [L(0, [t2]), L(4, []), t2])$

Kopf und Gestalt

- Die Marke der Wurzel heißt der **Kopf** des **markierten Baums.**

  ```
  fun head (L(x,_)) = x
  ```

  $val\ head : \alpha\ ltr \to \alpha$

- Die **Gestalt** ist der **reine Baum**, den man durch **Löschen der Marken** erhält.

  ```
  fun shape (L(_,ts)) = T(map shape ts)
  ```

  $val\ shape : \alpha\ ltr \to tree$

- Über die Gestalt übertragen sich alle **Begriffe für reine Bäume** (Stelligkeit, linear, binär, balanciert, Größe, Tiefe,…) **auf markierte Bäume**.

Prozeduren auf markierten Bäumen

- **Suche** nach der **ersten Marke** im Baum **gemäß Präordnung**, für die eine gegebene Prozedur *true* liefert.

```
fun find p t = let
  fun find’ nil = NONE
    | find’ (L(x, ts)::tr) =
          if p x then SOME x else find’ (ts @ tr)
in
  find’ [t]
end
```

- **Anwenden** einer **Prozedur** auf **alle Marken** im Baum.

```
fun lmap f (L (x, ts)) = L (f x, map (lmap f) ts)
```

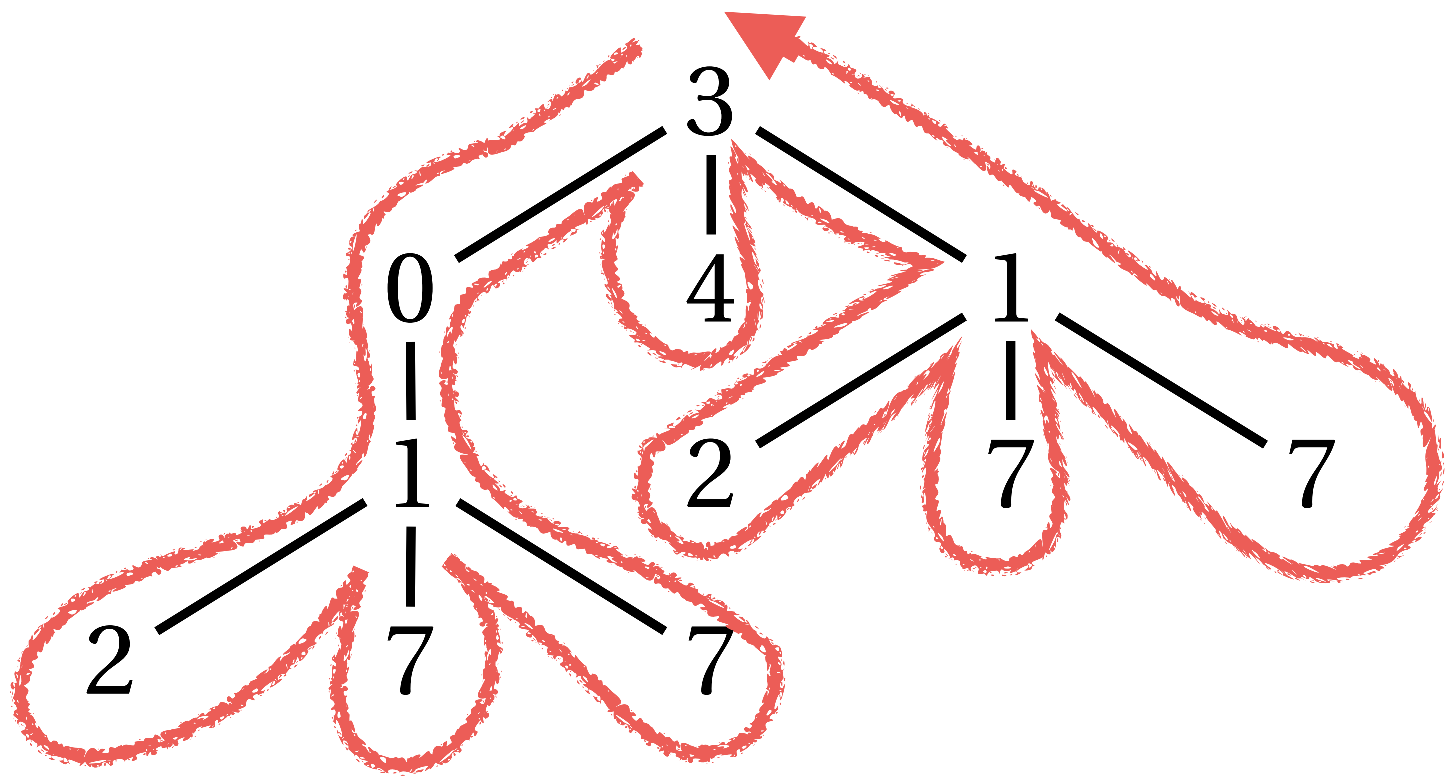
Projektionen

- Die **Präprojektion** eines **markierten Baums** ist die **gemäß Präordnung** geordnete Liste seiner Marken.

  $$prep(L(x,\ [t_1,\ldots,t_n])) = [x] \mathbin{@} prep\ t_1 \mathbin{@} \ldots \mathbin{@} prep\ t_n$$

- Die **Postprojektion** eines **markierten Baums** ist die **gemäß Postordnung** geordnete Liste seiner Marken.

  $$pop(L(x,\ [t_1,\ldots,t_n])) = pop\ t_1 \mathbin{@} \ldots \mathbin{@} pop\ t_n \mathbin{@} [x]$$



$prep\ t3 = [3,0,1,2,7,7,4,1,2,7,7]$

$pop\ t3 = [2,7,7,1,0,4,2,7,7,1,3]$

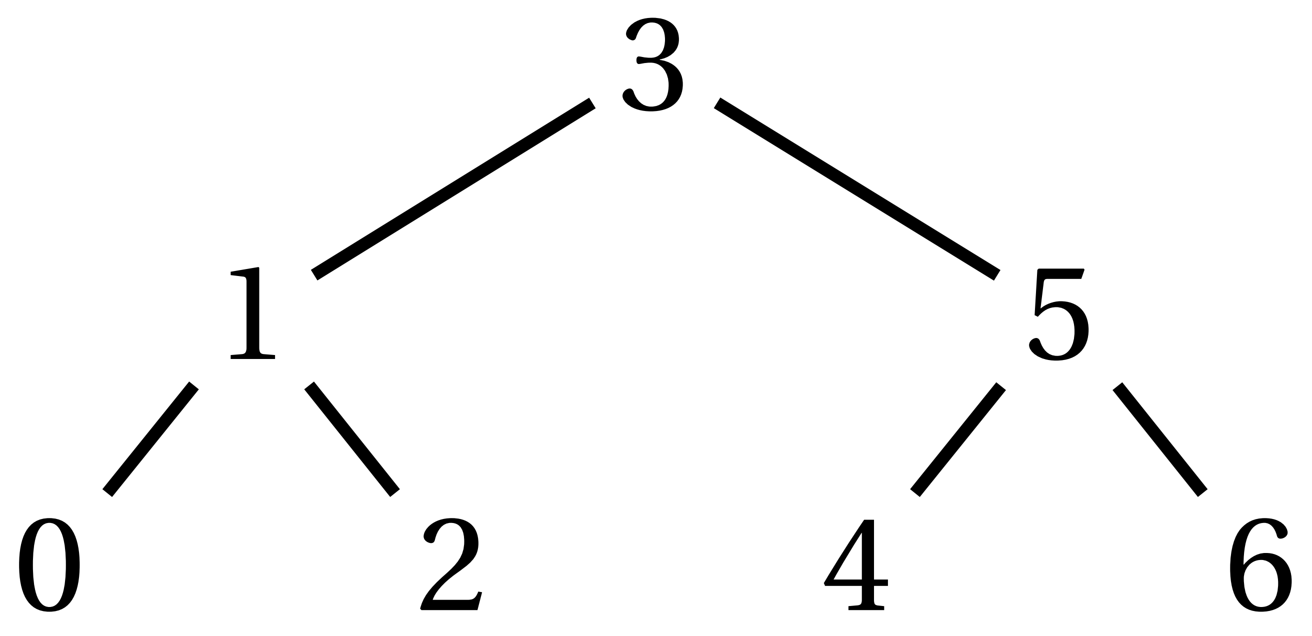
Frage

Was ist die Präprojektion von
L("B",[L("A",[L("D",[])]),L("C",[])]) ?

- ABDC ✗
- BADC ✓
- CDAB ✗
- DCAB ✗
- DACB ✗

Projektionen

- Bei der **Inprojektion** eines **binären** markierten **Baums** erscheint die Marke eines **inneren Knoten**
  nach den Marken des **linken Unterbaums** und
  vor den Marken des **rechten Unterbaums.**



$[0, 1, 2, 3, 4, 5, 6]$

www.prog1.saarland