

Programmierung 1

Vorlesung 10

Livestream beginnt um 10:20 Uhr

*Konstrukturen und
Ausnahmen, Teil 2
Bäume*

Programmierung 1

Darstellung arithmetischer Ausdrücke

Mit Hilfe von Konstruktortypen können **syntaktische Objekte** als **Werte dargestellt werden**.

Beispiel: Arithmetische Ausdrücke (gebildet aus Konstanten, Variablen, Summe und Produkt)

```
type var = string
```

```
datatype exp = C of int  
              | V of var  
              | A of exp * exp  
              | M of exp * exp
```

Darstellung von Umgebungen

- ▶ **Umgebungen** liefern den **Wert** von **Variablen**

```
type env = var -> int
```

- ▶ **Ausnahme** falls Wert **unbekannt**

```
exception Unbound
```

- ▶ **Beispiel:** [x:=0, y:=3]

```
val env = fn "x" => 0  
          | "y" => 3  
          | _  => raise Unbound
```

Evaluation von arithmetischen Ausdrücken

```
fun eval env (C c)      = c
  | eval env (V v)      = env v
  | eval env (A(e,e')) = eval env e + eval env e'
  | eval env (M(e,e')) = eval env e * eval env e'
```

val eval : env → exp → int

Ausnahmen

- ▶ **Ausnahmen** sind Werte des Konstruktortyps *exn*

Empty

Empty : exn

- ▶ Der Konstruktortyp *exn* kann **erweitert** werden:

exception New

exn New : exn

exception Newer of int

exn Newer : int → exn

fun test New = 0

| test (Newer x) = x

| test _ = ~1

Werfen von Ausnahmen

- ▶ **Ausnahmen** werden mit Raise-Ausdrücken **geworfen**:

raise $\langle \text{Ausdruck} \rangle$

raise New

!Uncaught exception: New

- ▶ Raise-Ausdrücke können **jeden Typ** annehmen:

fun f x y = if x then y else *raise* New

val f : $bool \rightarrow \alpha \rightarrow \alpha$

- ▶ **Konvention:** Wenn eine Prozedur Ausnahmen wirft, geben wir den **Ausnahmekonstrukt** als **Kommentar** an:

hd : $\alpha \text{ list} \rightarrow \alpha$ (* Empty *)

List.nth : $\alpha \text{ list} * \text{int} \rightarrow \alpha$ (* Subscript *)

Ausführungsreihenfolge

- ▶ StandardML schreibt eine strikte **links-nach-rechts Ausführung** vor

Beispiel: `(raise Overflow, raise Subscript)`
wirft die Ausnahme Overflow.

- ▶ **Sequenzialisierungen** sind Ausdrücke der Form

$(\langle \textit{Ausdruck} \rangle ; \dots ; \langle \textit{Ausdruck} \rangle)$

$(e_1; \dots; e_n)$ ist syntaktischer Zucker für $\#n (e_1, \dots, e_n)$

Beispiele:

`(5 ; 7)`

`7 : int`

`(raise New ; 7)`

`! Uncaught exception: New`

Fangen von Ausnahmen

- **Ausnahmen** werden mit Handle-Ausdrücken **gefangen**:

⟨Ausdruck⟩ handle ⟨Regel⟩ | ... | ⟨Regel⟩

```
(raise New) handle New => ()
```

() : unit

```
(raise Newer 7) handle Newer x => x
```

7 : int

```
fun test f = f() handle Newer x => x | Overflow => ~1
```

val test : (unit → int) → int

```
test (fn () => raise Newer 6)
```

6 : int

```
fun fac n = if n < 1 then 1 else n * fac(n-1)
```

val fac : int → int

```
fac 15
```

! Uncaught exception: Overflow

```
test (fn () => fac 15)
```

~1 : int

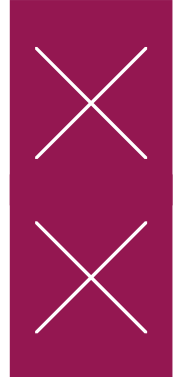
Frage

Die Prozedur f sei wie folgt deklariert:

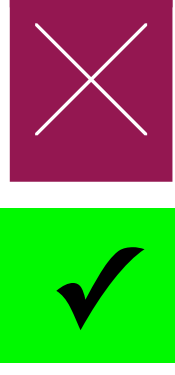
```
fun f x = if (x < 0) then 1 else 2 div x  
          handle Div => 3 handle Empty => 4
```

Was ist das Ergebnis von f 0?

▶ 1



▶ 2



▶ 3



▶ 4



Beispiel: Test auf Mehrfachauftreten

```
exception Double;
```

```
fun mask compare p = case compare p of  
    EQUAL => raise Double | v => v
```

```
fun testDouble compare xs =  
    (pinsert (mask compare) xs; false)  
  handle Double => true;
```

*val testDouble : (α * $\alpha \rightarrow \text{order}$) \rightarrow α list \rightarrow bool*

Adjunktion

```
exception Unbound
```

```
fun adjoin oldenv newenv x = newenv x  
    handle Unbound => oldenv x
```

Beispiel:

```
fun umgebung1 s = if s="banane" then 2 else  
    if s="apfel" then 4 else  
    raise Unbound
```

```
fun umgebung2 s = if s="aprikose" then 6 else  
    if s="banane" then 3 else  
    raise Unbound
```

```
val umgebung3 = adjoin umgebung1 umgebung2
```

Optionen

- ▶ Der Typkonstruktor **option** ist vordeklariert:

```
datatype 'a option = NONE | SOME of 'a
```

- ▶ Die mit option beschriebenen Typen heißen **Optionstypen**, ihre Werte **Optionen**.
- ▶ Die mit SOME konstruierten Optionen heißen **eingelöst**, die mit NONE konstruierten Optionen heißen **uneingelöst**.

```
fun nth _ nil = NONE  
  | nth n (x::xr) = if n<1 then SOME x else nth (n-1) xr
```

```
val nth : int → α list → α option
```

```
nth 2 [3,4,5]
```

```
SOME 5 : int option
```

```
nth 3 [3,4,5]
```

```
NONE : int option
```

Optionen

► valOf:

```
fun valOf (SOME x) = x  
  | valOf NONE      = raise Option.Option
```

val valOf: α option \rightarrow α

```
valOf (nth 2 [3,4,5])
```

5: int

► isSome:

```
fun isSome NONE      = false  
  | isSome (SOME _) = true
```

val isSome: α option \rightarrow bool

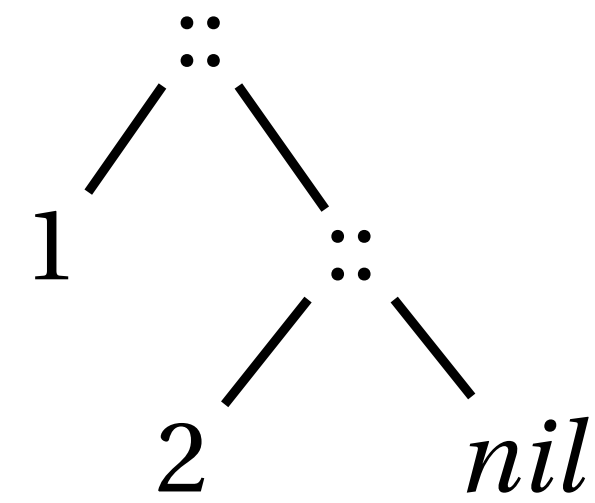
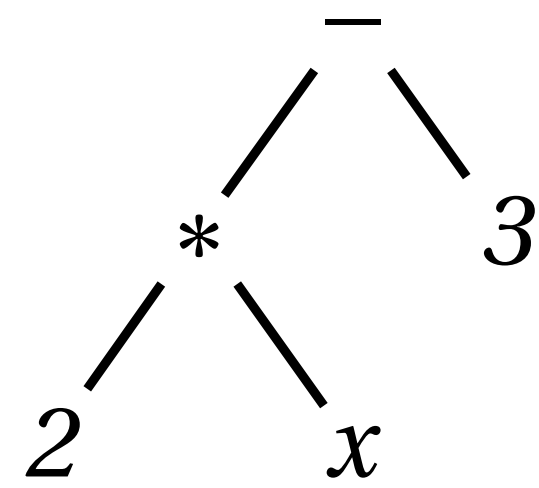
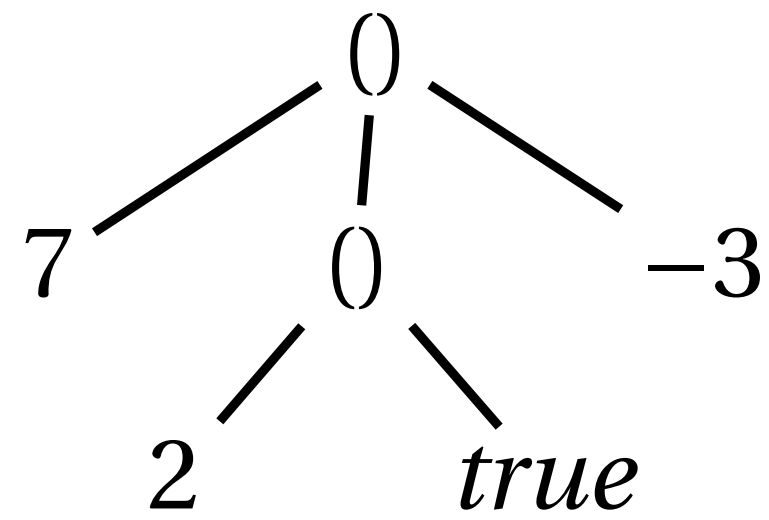
Beispiel: Test auf Mehrfachauftreten

```
fun findDouble compare xs = let
  exception Double of 'a
  fun compare' (x,y) = case compare (x,y) of
    EQUAL => raise Double x | v => v
in
  (pinsert compare' xs; NONE)
  handle Double x => SOME x
end
```

*val findDouble : (α * $\alpha \rightarrow \text{order}$) \rightarrow α list \rightarrow α option*

Kapitel 7

Bäume

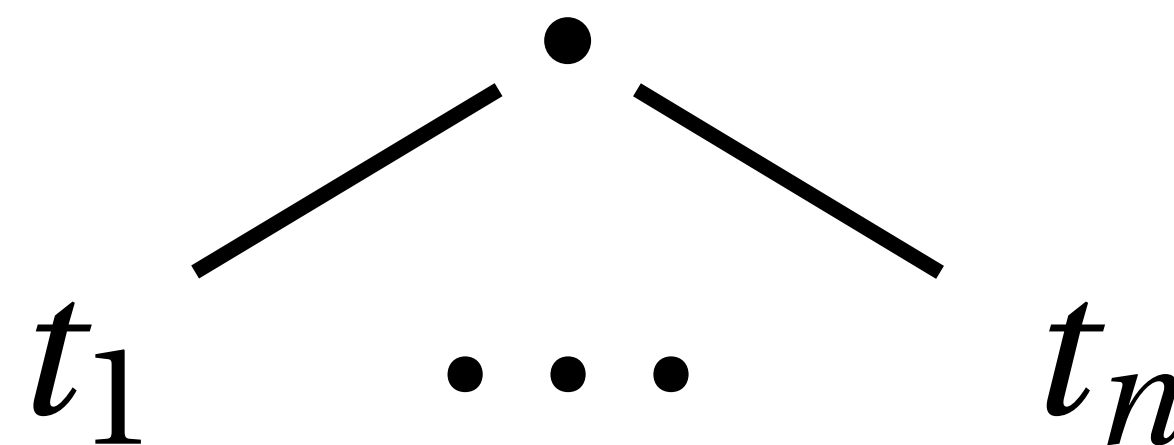


Grafische Darstellung

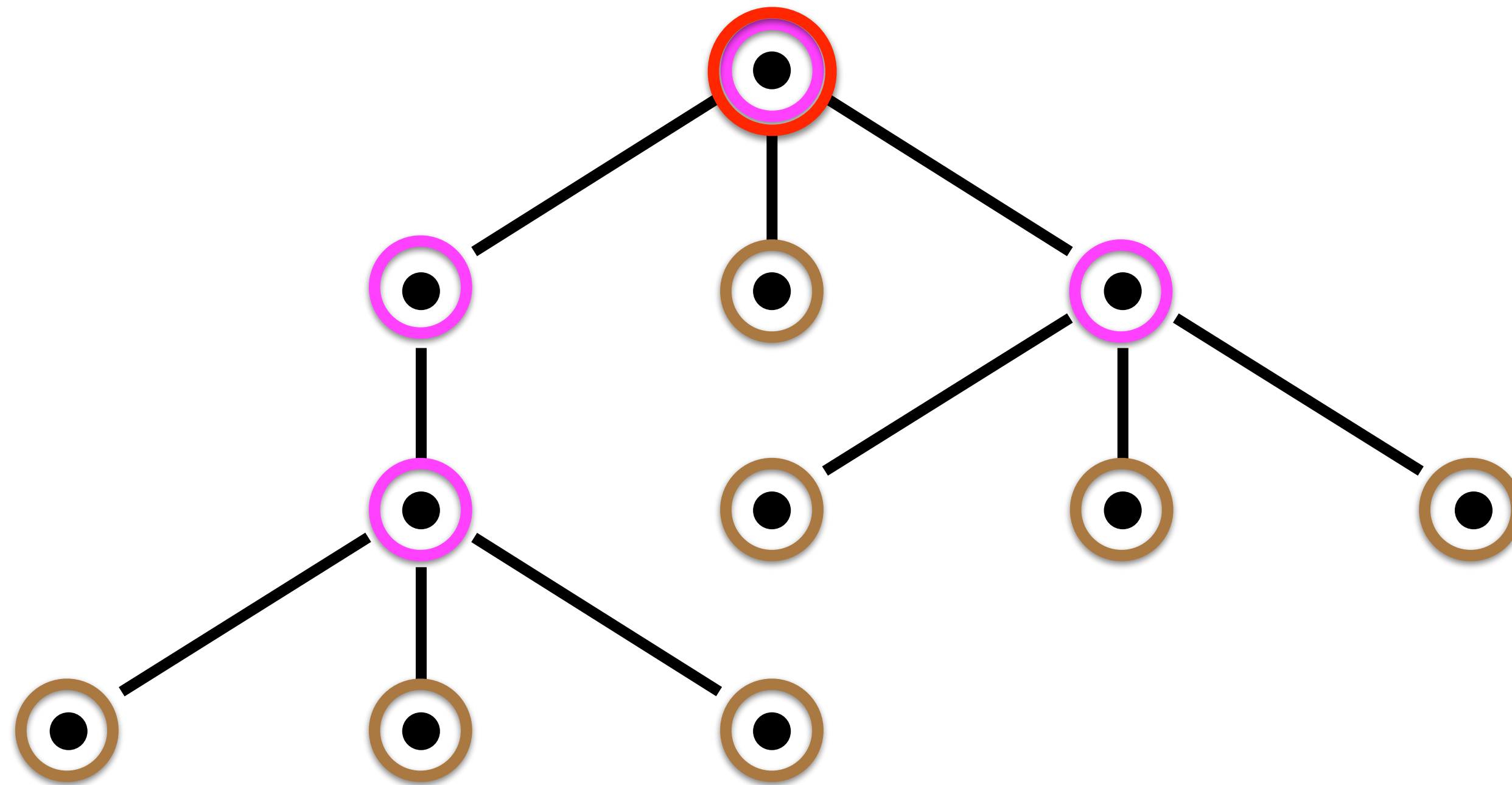
- ▶ **Atomarer Baum:** der atomare Baum wird dargestellt als ein einzelner Knoten:



- ▶ **Zusammengesetzte Bäume:** die grafische Darstellung der Unterbäume t_1, \dots, t_n wird durch n Kanten mit einem Knoten verbunden.



Grafische Darstellung



Blätter: Knoten aus denen keine Kanten zu tieferen Knoten führen

Innere Knoten: Knoten, die keine Blätter sind

Wurzel: der oberste Knoten

Reine Bäume

► **Idee:** "Ein Baum ist die Liste seiner Unterbäume."

► **In ML:**

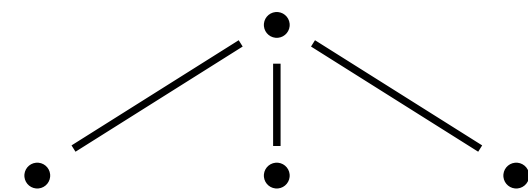
```
datatype tree = T of tree list
```

► **Beispiele:**



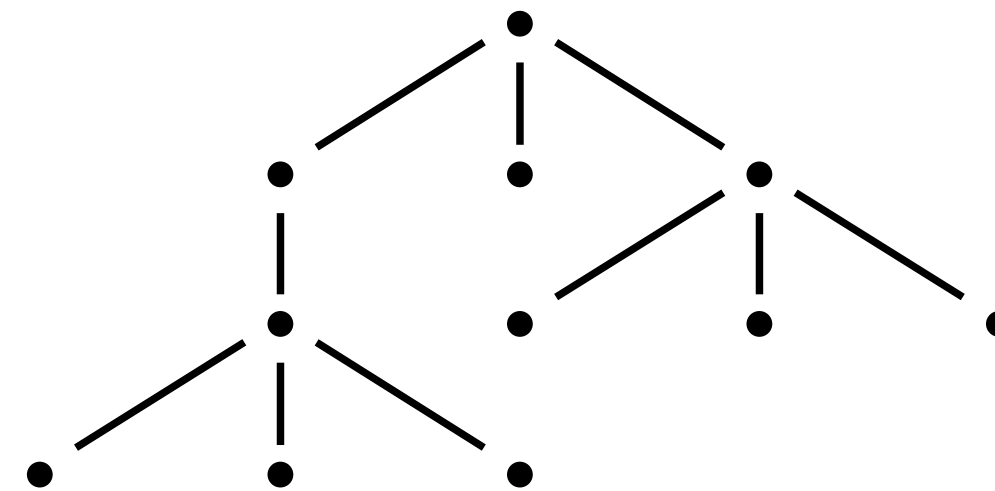
$t1 = T[]$

**der atomare
Baum**



$t2 = T[t1, t1, t1]$

**zusammengesetzter
Baum**



$t3 = T[T[t2], t1, t2]$

**zusammengesetzter
Baum**

Frage

Wieviele Knoten hat der Baum $T[T[],T[],T[]]$?

▶ 1



▶ 2



▶ 3



▶ 4



▶ 5



Unterbäume (direct subtrees)

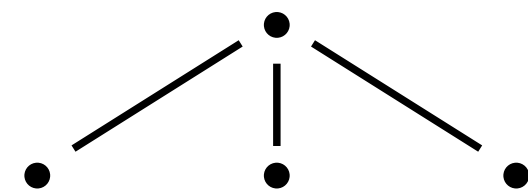
► Die **Stelligkeit** des Baums $T[t_1, \dots, t_n]$ ist die Zahl n .

► Die **Unterbäume** des Baums $T[t_1, \dots, t_n]$ sind die Bäume t_1, \dots, t_n .

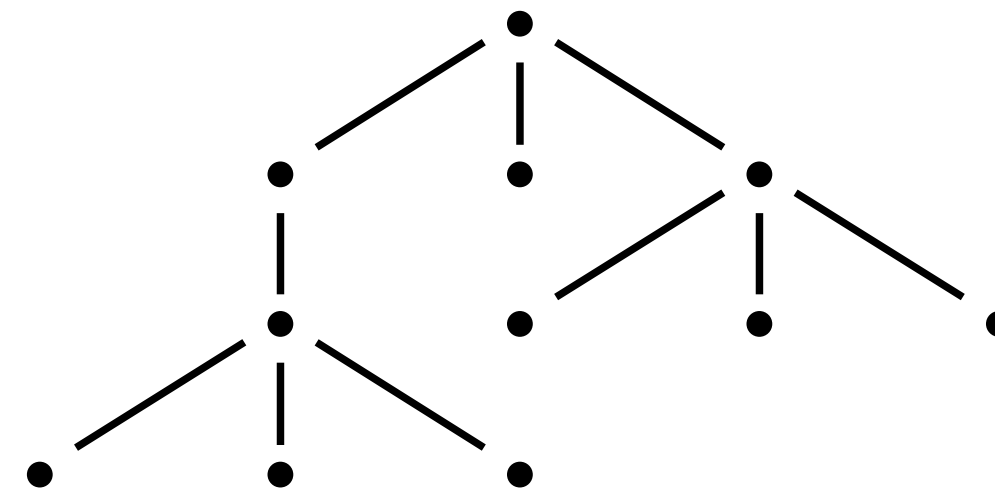
► **Beispiel:**



$t_1 = T[]$



$t_2 = T[t_1, t_1, t_1]$



$t_3 = T[T[t_2], t_1, t_2]$

Unterbäume von t_3 : $T[t_2], t_1, t_2$

Unterbäume von t_2 : t_1

Unterbäume von t_1 : keine Unterbäume

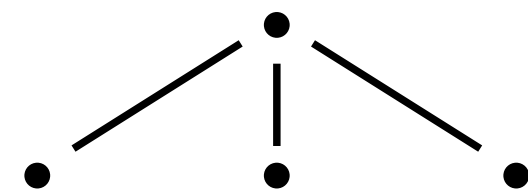
Teilbäume (subtrees)

- ▶ Wenn t ein **Baum** ist, dann ist t ein **Teilbaum** von t .
- ▶ Wenn t' ein **Unterbaum** von t ist, dann ist **jeder Teilbaum** von t' ein **Teilbaum** von t .

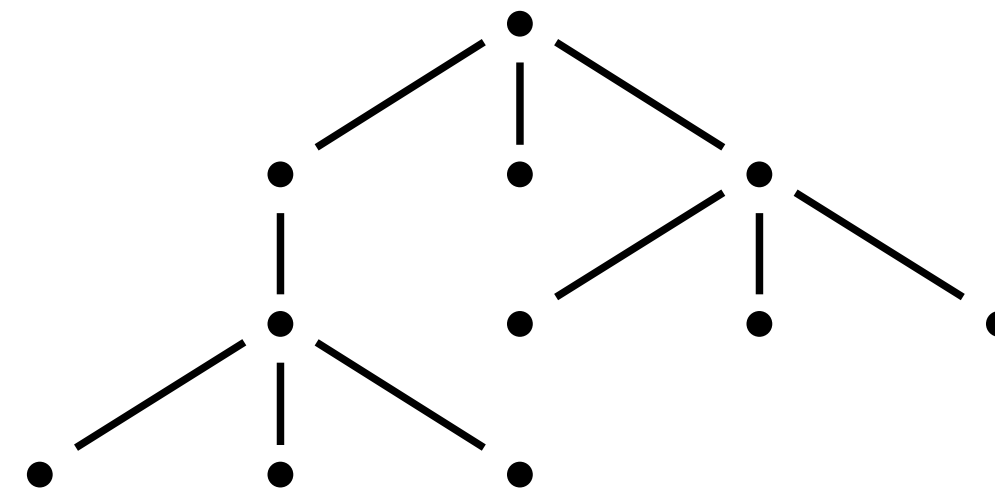
▶ Beispiel:



$t1 = T[]$



$t2 = T[t1, t1, t1]$



$t3 = T[T[t2], t1, t2]$

Teilbäume von $t3$: $t3, T[t2], t1, t2$

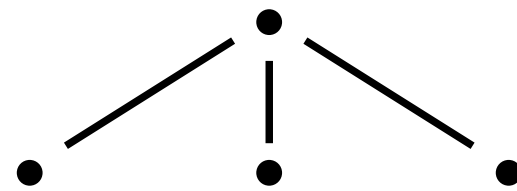
Teilbäume von $t2$: $t2, t1$

Teilbäume von $t1$: $t1$

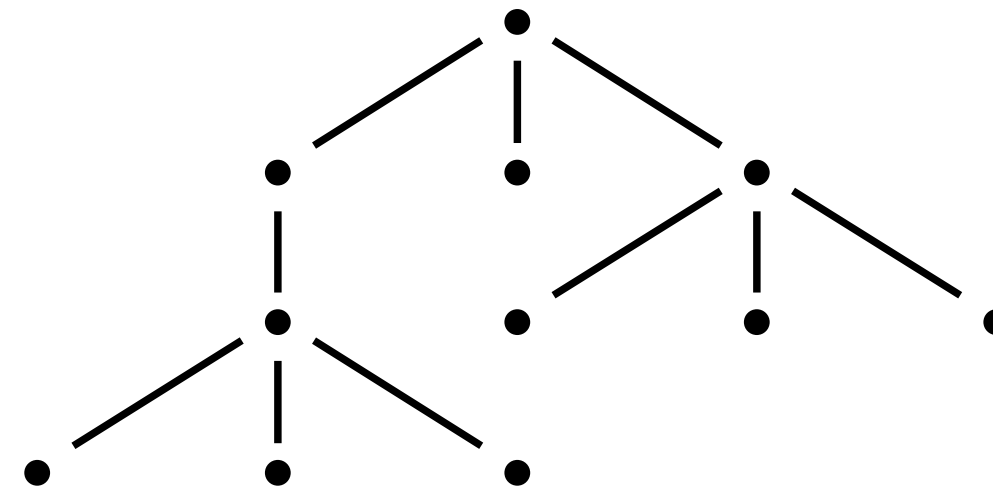
Auftreten von Teilbäumen



$t_1 = T[]$



$t_2 = T[t_1, t_1, t_1]$



$t_3 = T[T[t_2], t_1, t_2]$

- ▶ t_1 tritt dreimal als Teilbaum von t_2 auf.
- ▶ t_2 tritt zweimal als Teilbaum von t_3 auf.

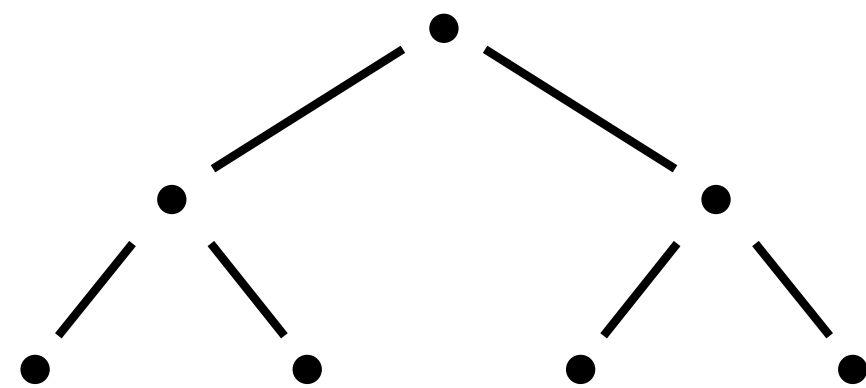
Lineare vs. binäre Bäume

- ▶ Ein Baum heißt **linear** wenn jeder seiner zusammengesetzten Teilbäume **einstellig** ist.
- ▶ Ein Baum heißt **binär** wenn jeder seiner zusammengesetzten Teilbäume **zweistellig** ist.

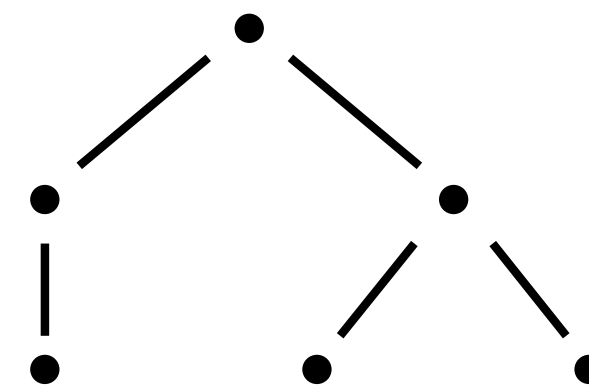
▶ Beispiele:



linear







binär



weder linear noch binär

Frage

Welche der folgenden Aussagen sind korrekt?

- ▶ Der atomare Baum ist linear. 
- ▶ Ein Unterbaum ist immer auch ein Teilbaum. 
- ▶ Ein Teilbaum ist immer auch ein Unterbaum. 
- ▶ Ein Baum hat mindestens so viele Blätter wie jeder seiner Teilbäume. 

Erste Prozeduren

```
datatype tree = T of tree list
```

► Stelligkeit:

```
fun arity (T ts) = length ts  
val arity : tree → int
```

► kter Unterbaum:

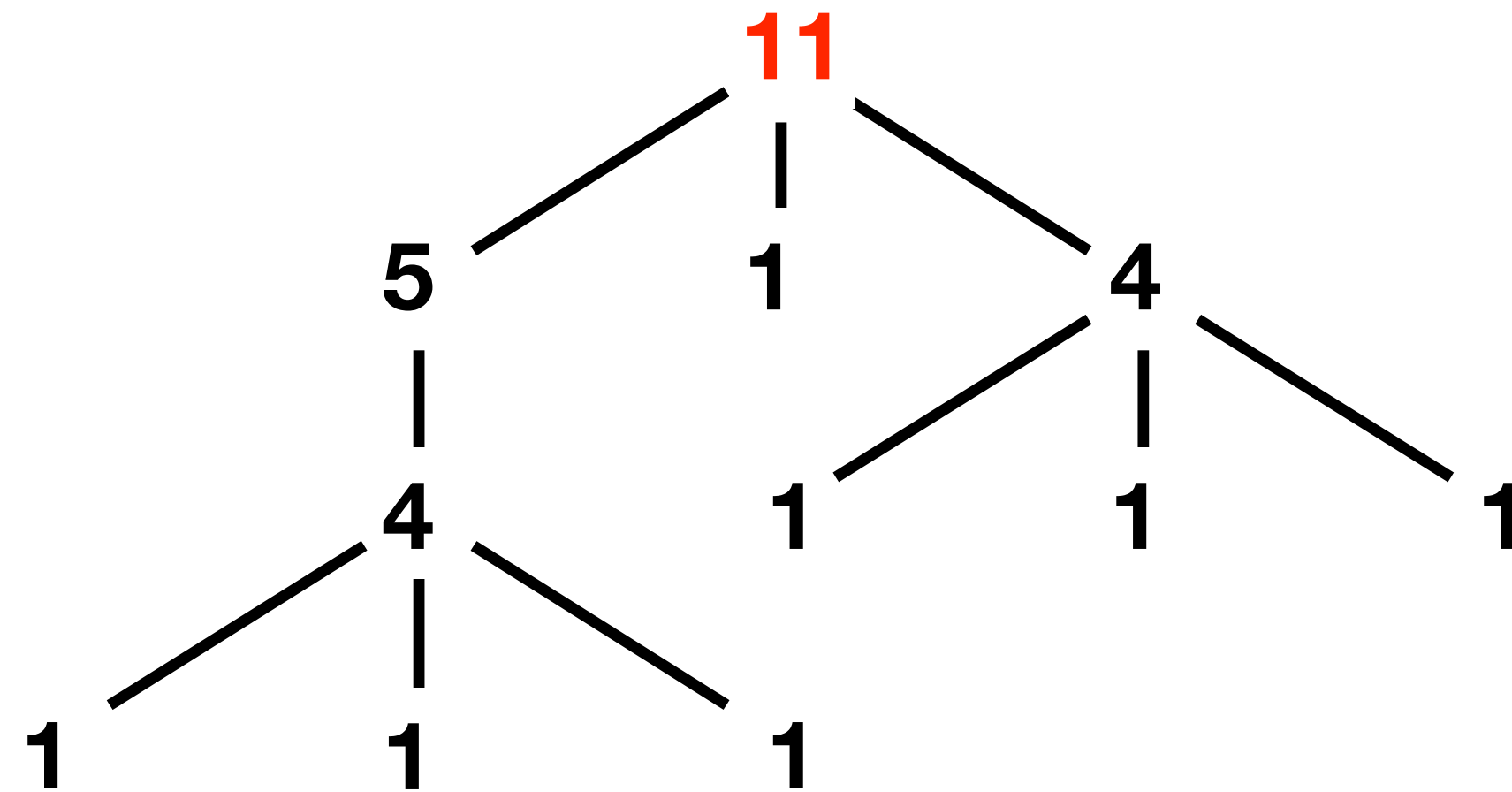
```
fun dst (T ts) k = List.nth(ts, k-1)  
val dst : tree → int → tree
```

► Test auf lineare Bäume:

```
fun linear (T nil) = true  
  | linear (T [t]) = linear t  
  | linear _ = false  
val linear : tree → bool
```

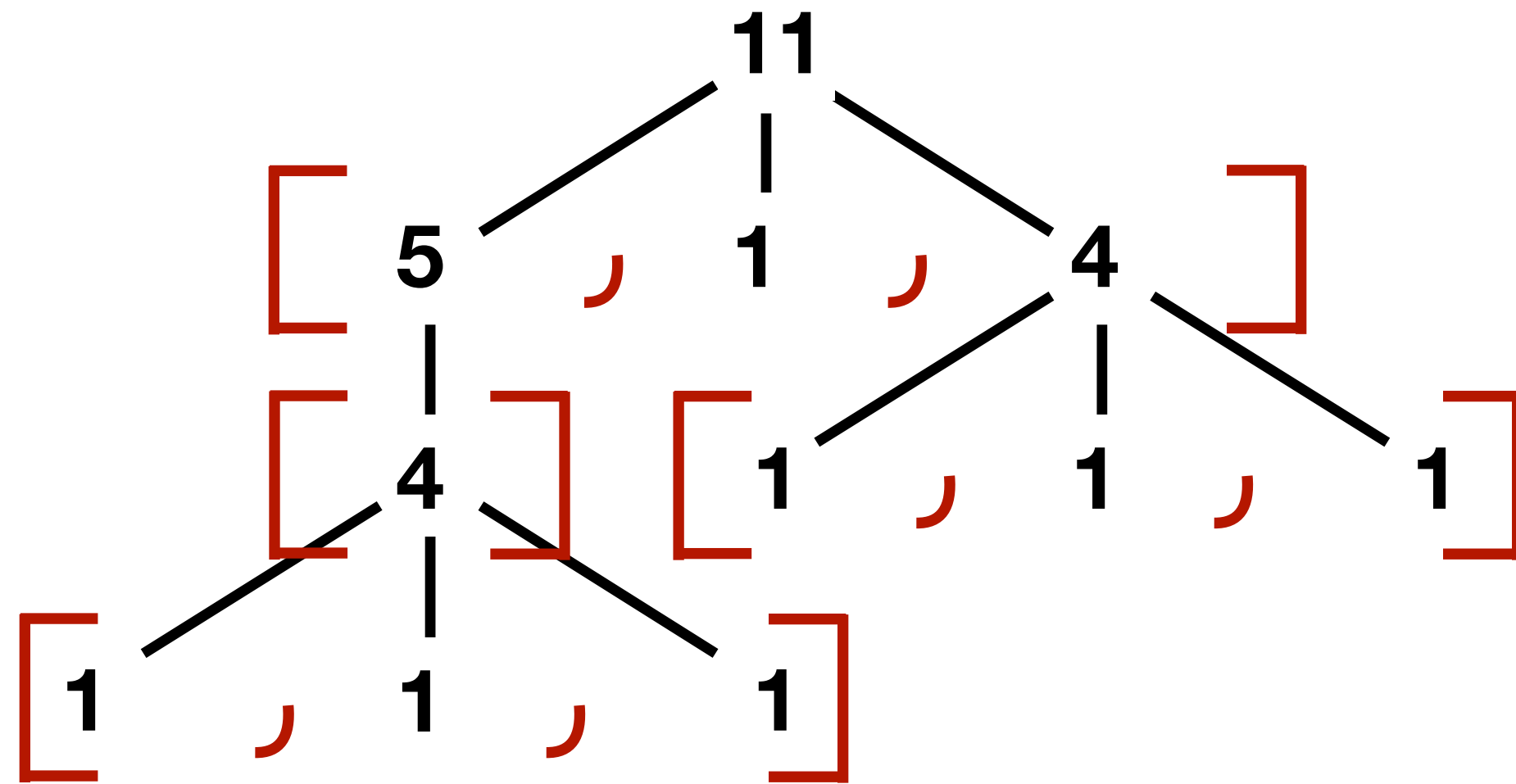
Anzahl der Knoten in einem Baum

- **Idee:** Wir berechnen den Wert für einen Baum **rekursiv** aus den Werten der Unterbäume



Anzahl der Knoten in einem Baum

- **Idee:** Wir berechnen den Wert für einen Baum aus einer **rekursiv** berechneten **Liste** der Werte der Unterbäume



Anzahl der Knoten in einem Baum



- ▶ **Idee:** Wir berechnen den Wert für einen Baum aus einer **rekursiv** berechneten **Liste** der Werte der Unterbäume
- ▶ **Idee:** Benutze map, um die Prozedur rekursiv auf die Unterbäume anzuwenden
- ▶ **Idee:** Benutze Faltung, um aus der Liste der Werte der Unterbäume den Wert für den Baum zu berechnen.

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

```
val size : tree → int
```

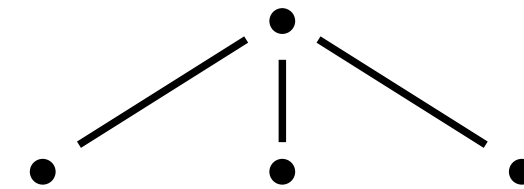
Anzahl der Knoten in einem Baum

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

```
val size : tree → int
```

A single black dot representing a tree with one node.

$t1 = T[]$



$t2 = T[t1, t1, t1]$

```
size T[]
```

```
= foldl op+ 1 (map size [])
```

```
= foldl op+ 1 []
```

```
= 1
```

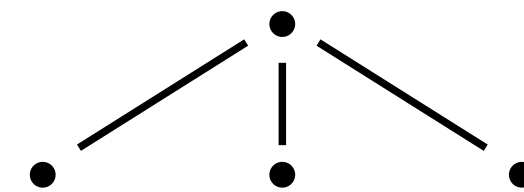
Anzahl der Knoten in einem Baum

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

```
val size : tree → int
```

•

$t1 = T[]$



$t2 = T[t1, t1, t1]$

```
size T[T[],T[],T[]]
```

```
= foldl op+ 1 (map size [T[],T[],T[]])
= foldl op+ 1 (size T[] :: map size [T[],T[]])
= foldl op+ 1 (1 :: map size [T[],T[]])
= foldl op+ 1 (1 :: size T[] :: map size [T[]])
= foldl op+ 1 (1 :: 1 :: map size [T[]])
= foldl op+ 1 (1 :: 1 :: size T[] :: map size [])
= foldl op+ 1 (1 :: 1 :: 1 :: map size [])
= foldl op+ 1 (1 :: 1 :: 1 :: nil)
= foldl op+ 1 [1,1,1]
= 4
```

www.prog1.saarland