Kapitel 7: Benutzerdefinierte Datentypen

Andreas Abel

LFE Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München

10. Juni 2011

Quelle: Martin Wirsing, *Benutzerdefinierte Datentypen*
Foliensatz ProMo SS 2010

Einführende Fragen

- Wie definiere ich komplexe Datenstrukturen in SML?
- Z.B. Eine Datei-Verzeichnis-Struktur?
- Ist `list` in SML fest eingebaut oder kann ich Listen selbst definieren?
- Welche Datenstruktur verwende ich zur Repräsentation von abstrakter Syntax (z.B. SML-Typausdrücke)?

Inhalt

Aufzählungstypen

- Eine Vergleichfunktion hat drei mögliche Ergebnisse: *kleiner*, *gleich*, oder *größer*.
- Diese Alternativen repräsentieren wir durch einen Aufzählungstypen:

```
datatype CMP = LT | EQ | GT
```

- Diese Anweisung erzeugt eine neue **Typkonstante** `CMP` und drei neue Werte (**Konstruktoren**) `LT`, `EQ`, `GT` vom Typ `CMP`.
- Damit implementieren wir eine Vergleichsfunktion auf Zeichen:

```
fun compareChar(x : char, y : char) : CMP =
  if x > y then GT
  else if x = y then EQ
  else LT
```

Verarbeitung von Aufzählungen

- Die Konstruktoren von Aufzählungen werden mit **pattern matching** unterschieden.

```
type 'a comparison = 'a * 'a -> CMP

fun compareLex(cmpA : 'a comparison,
               cmpB : 'b comparison)
             : ('a * 'b) comparison =
  fn ((a1,b1),(a2,b2)) =>
    case cmpA(a1,a2)
      of GT => GT
       | LT => LT
       | EQ => cmpB(b1,b2)
```

- Diese Funktion höherer Ordnung vergleicht ein Paar `(a1,b1)` mit einem Paar `(a2,b2)` lexikographisch, wobei die ersten Komponenten mit Vergleichsfunktion `cmpA` und die zweiten mit `cmpB` verglichen werden.

Selbstgestrickter Listenvergleich

- Eine polymorphe Vergleichsfunktion für Listen:

```
fun compareList(cmp : 'a comparison)
                : 'a list comparison =
  fn ([],     []   ) => EQ
   | ([],     ys   ) => LT
   | (xs,     []   ) => GT
   | (x::xs, y::ys) =>
     compareLex (cmp,compareList cmp) ((x,xs),(y,ys))
```

- Vergleichsfunktion für Strings als Zeichenlisten:

```
val compareCharList = compareList compareChar

fun compareString(s,t) = compareCharList
      (String.explode s, String.explode t)
```

Optionale Ergebnisse

- Folgender Typ ist in der Standardbibliothek von SML vordefiniert:

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

- Erzeugt Konstruktoren: `NONE : 'a option` und `SOME : 'a -> 'a option`.
- Man verwenden ihn als Rückgabetyp für partielle Funktionen (anstatt eine Ausnahme zu werfen). Z.B. Holen eines Wertes aus einer Assoziationsliste:

```
fun lookup (key: ''a, assocList: (''a * 'b) list): 'b option =
  let fun loop []          = NONE
        | loop ((k,v)::l) =
            if k = key then SOME v else loop l
  in loop assocList
  end
```

- Achtung! Wegen des Gleichheitstests `k = key` muss der Schlüsseltyp ein **Gleichheitstyp** (engl. *equality type*) `''a` sein.

Verarbeitung von optionalen Werten

- Undefiniertheit `NONE` kann mittels pattern matching abgefangen werden:

```
fun lookupWithDefault
      (default   : 'b,
       key       : ''a,
       assocList : (''a * 'b) list) : 'b =
  case lookup(key, assocList)
    of SOME v => v
     | NONE   => default
```

Disjunkte Vereinigung

- Polymorphe Datentypen können auch **mehrere Typparameter** haben. Z.B. Typ der disjunkten Vereinigung:

```
datatype ('a,'b) either = Left of 'a | Right of 'b
```

- Hiermit können wir informative Fehlermeldungen zurückgeben:

```
fun division (n:int, m:int) : (string,int) either =
  if m = 0 then Left "division by zero"
  else Right (n div m)

fun resultToString (Left err) = err
  | resultToString (Right n ) = Int.toString n
```

Rekursive Datentypen

- Datentyp-Deklarationen dürfen **rekursiv** sein, d.h. der gerade definierte Typ darf in den Typen seiner Konstruktoren vorkommen.
- Bekanntes Beispiel ist der Typ der Listen:

```
datatype 'a list = nil | cons of 'a * 'a list
```

- **Wechselseitige Rekursion** mittels **datatype ... and ...**

```
type name = string
type url  = string

datatype file
  = File   of name
  | Dir    of directory
and directory
  = Local  of name * file list
  | Remote of url
```

Typen mit nur einem Konstruktor

- SML expandiert Typsynonyme wie **type** name = string.

```
datatype file = Dir of directory | File of string
datatype directory = Local of string * file list | Remote of string
```

- Um die Unterscheidung von name und url zu forcieren, verwenden wir einen **wrapper** ("Umschlag").

```
datatype name  = Name of string
datatype url   = Url  of string
datatype file  = ...
```

- Nun lautet die Antwort von SML:

```
datatype file = Dir of directory | File of name
datatype directory = Local of name * file list | Remote of url
```

Datentypdeklaration (allgemein)

- Die allgemeine Form einer **Datentypdeklaration** in SML is

$$\begin{array}{lll} \textbf{datatype} & D_1 = cd_{1,1} \mid \ldots \mid cd_{1,n_1} \\ \textbf{and} & \ldots \\ \textbf{and} & D_m = cd_{m,1} \mid \ldots \mid cd_{m,n_m} \end{array}$$

  wobei jedes D_i eine der Formen
  1. `identifier`,
  2. `'a identifier`, oder
  3. `('a_1,...,'a_k) identifier`

  annimmt und jedes $cd_{i,j}$ eine der Formen
  1. `c` (Konstruktor ohne Argument), oder
  2. `c` **`of`** `typ` (Konstruktor mit Argument vom Typ `typ`).
- Einfaches Beispiel: Listen

```
datatype 'a list = nil | cons of 'a * 'a list
```

Binärbäume

- Ist A eine Menge, so wird die Menge $A^{\triangle}$ der *Binärbäume mit Knotenmarkierungen in* A (oder einfach **Binärbäume über** A) induktiv definiert durch:
  1. Der leere Baum ε ist in $A^{\triangle}$.
  2. Sind l und r in $A^{\triangle}$ und x in A, so ist das Tripel (l, x, r) in $A^{\triangle}$.
- Alternativ kann man die Mengen $A_n^{\triangle}$ der **Binärbäume mit Höhe (Tiefe) kleiner** n rekursiv definieren durch:
  1. $A_0^{\triangle} = \emptyset$ (kein Baum hat negative Höhe).
  2. $A_{n+1}^{\triangle} = \{\varepsilon\} \cup \{(l, x, r) \mid x \in A,\ l, r \in A_n^{\triangle}\}$.
- Man beweist leicht $A_n^{\triangle} \subseteq A_{n+1}^{\triangle}$ (Kumulativität.
- $A^{\triangle} = \bigcup_{n \in \mathbb{N}} A_n^{\triangle}$ ist der Limes der Folge $A_0^{\triangle}, A_1^{\triangle}, \ldots$
- Die **Höhe (Tiefe)** eines Baums t ist das kleinste n, so dass $t \in A_{n+1}^{\triangle}$.

Terminologie für Bäume

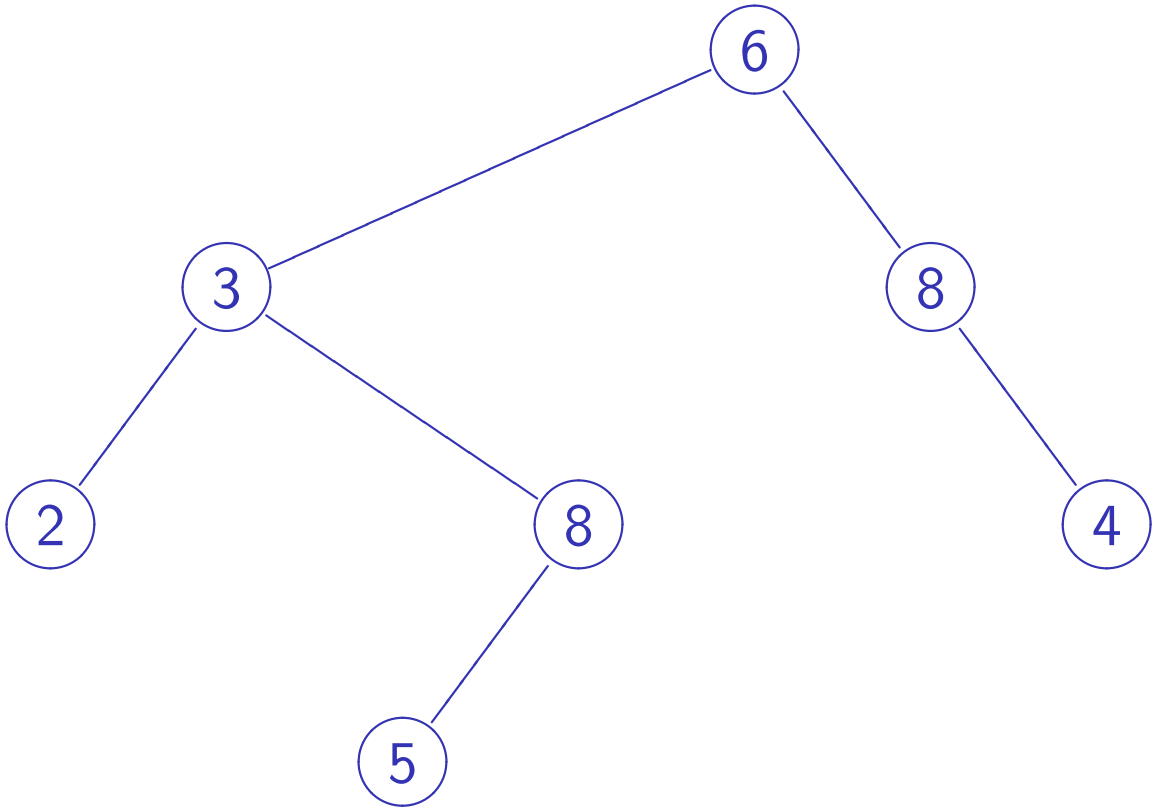
- x heißt **Wurzel** oder **(Wurzel)beschriftung** von (l, x, r).
- l heißt **linker** und r **rechter Unterbaum** von (l, x, r).
- Ein Binärbaum der Form $(\varepsilon, x, \varepsilon)$ heißt *Blatt*.
- ε heißt **leerer Baum**, jeder andere Baum ist **nichtleer**.
- Die **Knoten** und **Teilbäume** eines Binärbaums sind rekursiv definiert:
  1. Der leere Baum ε hat keine Knoten und nur sich selbst als Teilbaum.
  2. Die Knoten von (l, x, r) sind x plus die Knoten von l und r. Die Teilbäume von (l, x, r) sind (l, x, r) plus die Teilbäume von l und r.
- Sei $[x]$ definiert als Abkürzung für ein Blatt $(\varepsilon, x, \varepsilon)$ und

$$t = (([2],\ 3,\ ([5], 8, \varepsilon)),\quad 6,\quad (\varepsilon,\ 8,\ [4]))$$

- t hat Knoten $\{2, 3, 5, 8, 6, 4\}$ und Teilbäume $\{t,\ (([2],\ 3,\ ([5], 8, \varepsilon)),\ (\varepsilon,\ 8,\ [4])),\ ([5], 8, \varepsilon),\ [2],\ [4],\ [5],\ \varepsilon\}$.

Typische Darstellung eines Binärbaumes

- Wurzel: 6
- Blätter: 2, 5, 4



Binärbäume in SML

- Binärbäume sind ein Beispiel eines rekursiven Datentyps:

```
datatype 'a tree
  = Empty                                  (* leerer Baum *)
  | Node of 'a tree * 'a * 'a tree         (* Knoten      *)

fun leaf x = Node(Empty, x, Empty)         (* Blatt       *)

val t = Node(Node(leaf 2, 3, Node(leaf 5, 8, Empty)),
             6, Node(Empty, 8, leaf 4))
```

- In der Antwort von SML wird der Baum nicht vollständig ausgegeben:

```
val t = Node (Node (Node #,3,Node #),6,Node (Empty,8,Node #)) : int tree
```

- Datenstrukturen werden nur bis zur Tiefe `Control.Print.printDepth` gedruckt. Diesen Wert können wir setzen, z.B.

```
Control.Print.printDepth := 10;
```

Standardfunktionen für Binärbäume

- Projektionen (partiell, undefiniert auf `Empty`!)

```
fun label (Node(_,x,_)) = x
fun left  (Node(l,_,_)) = l
fun right (Node(_,_,r)) = r
```

- Test auf leerer Baum. Schlechte Implementierung:

```
fun isEmpty t = (t = Empty)

Warning: calling polyEqual
val isEmpty = fn : ''a tree -> bool
```

- Funktioniert nur für Gleichheitstypen `''a`. Besser:

```
fun isEmpty (Empty)  = true
  | isEmpty (Node _) = false

val isEmpty = fn : 'a tree -> bool
```

Rekursion auf Bäumen

- Höhe eines Baumes:

```
fun height (Empty)          = 0
  | height (Node(l,_,r)) = 1 + Int.max (height l, height r)
```

- Eine Funktion `f` auf alle Beschriftungen anwenden:

```
fun mapTree f (Empty)          = Empty
  | mapTree f (Node(l,x,r)) =
    let val l' = mapTree f l
        val x' = f x
        val r' = mapTree f r
    in  Node(l',x',r')
    end
```

- *Beachte:* `mapTree f t` erzeugt einen komplett **neuen** Baum!
- `mapTree (fn x => x) t` liefert eine Kopie von `t`.

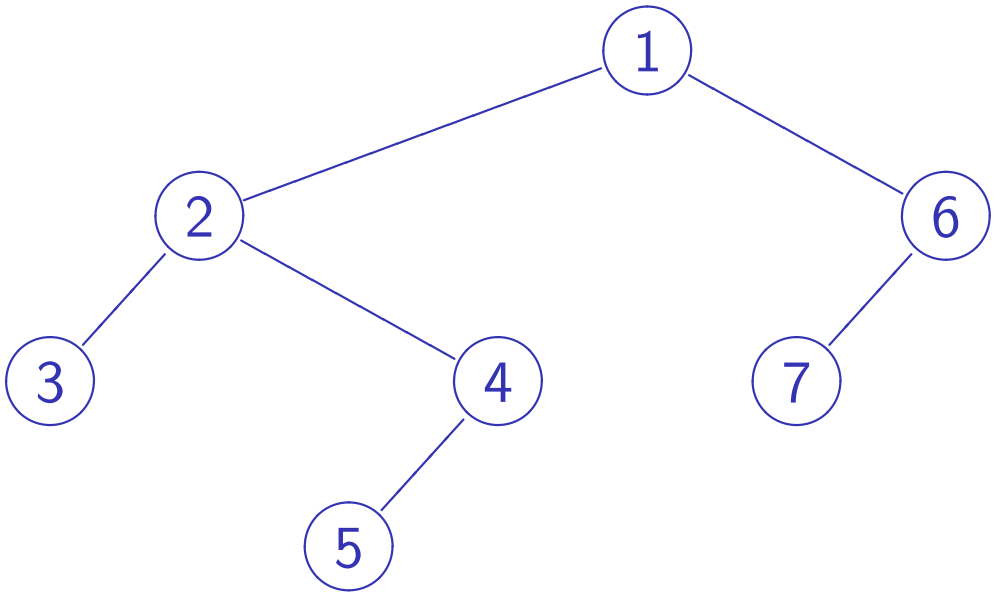
Linearisierung

- Mit Hilfe eines Baum**durchlaufs** (engl. *tree traversal*) können wir alle Markierungen in einer Liste aufsammeln.
- Die Reihenfolge der Listenelemente hängt von der Art des Durchlaufs ab:
  1. Vorordnung (engl. *preorder*)
  2. Symmetrische Ordnung (engl. *inorder*)
  3. Nachordnung (engl. *postorder*)

Vorordnung

Zuerst Markierung, dann linker, dann rechter Teilbaum.

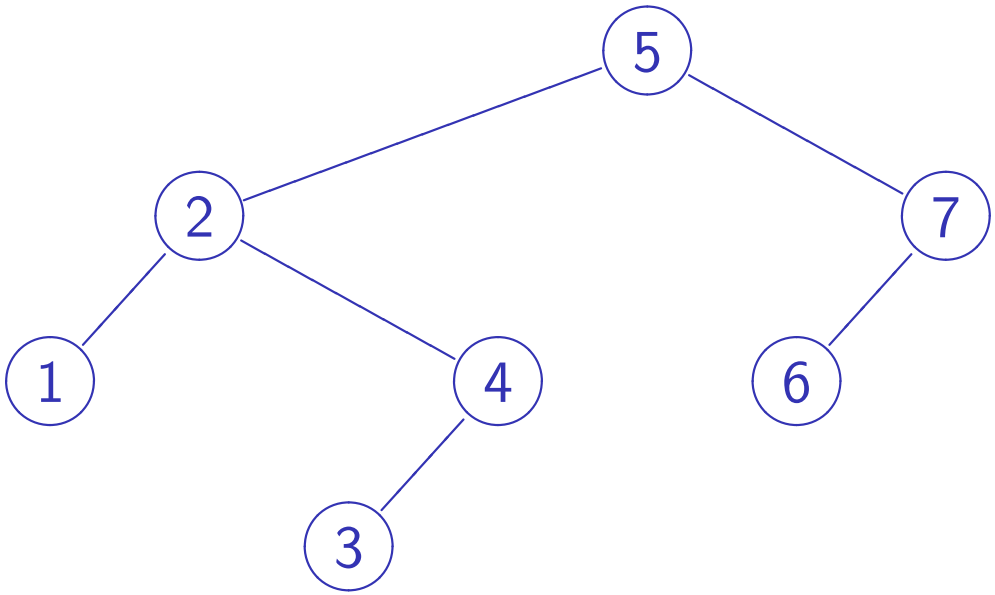
```
fun preorder (Empty)        = []
  | preorder (Node(l,x,r)) =
      [x] @ preorder l @ preorder r
```



Symmetrische Ordnung

Zuerst linker Teilbaum, dann Markierung, dann rechter Teilbaum.

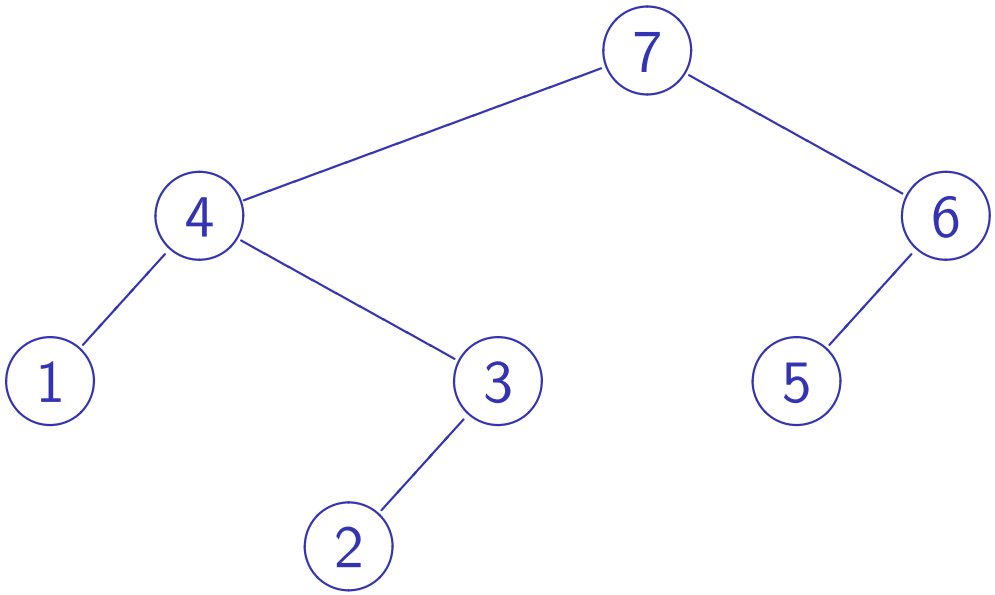
```
fun inorder (Empty)        = []
  | inorder (Node(l,x,r)) =
      inorder l @ [x] @ inorder r
```



Nachordnung

Zuerst linker, dann rechter Teilbaum, dann Markierung.

```
fun postorder (Empty)        = []
  | postorder (Node(l,x,r)) =
      postorder l @ postorder r @ [x]
```



Generischer Tiefendurchlauf

- Alle Durchläufe bearbeiten die Teilbäume unabhängig voneinander.
- Das zugrundeliegende Schema heißt **Tiefendurchlauf** (engl. *depth-first*)

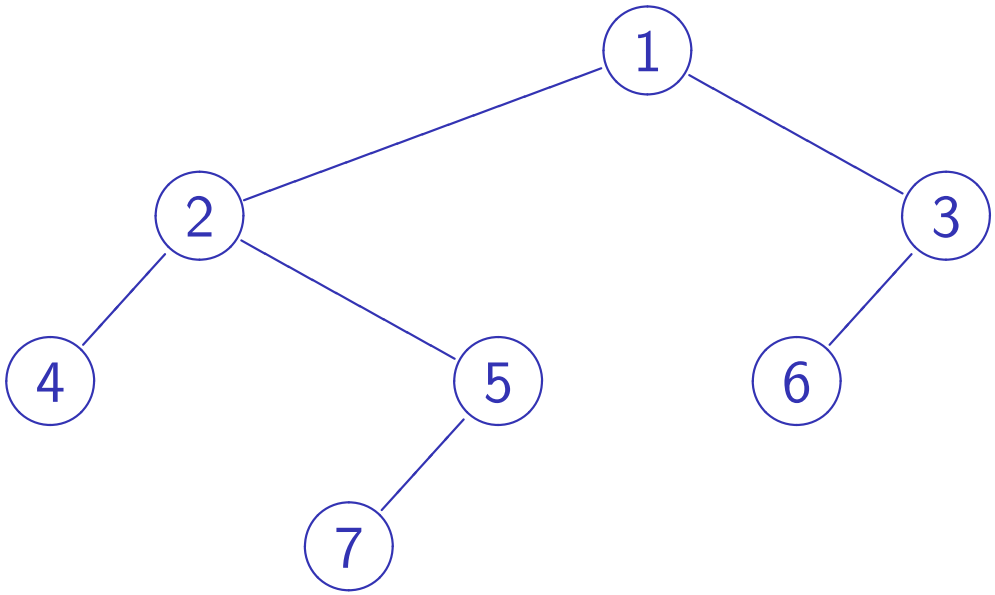
```
fun foldTree (n : 'b * 'a * 'b -> 'b)(e : 'b) : 'a tree -> 'b =
  let fun loop (Empty)        = e
        | loop (Node(l,x,r))  = n (loop l, x, loop r)
  in  loop
  end
```

- Die drei Baumlinearisierungen sind Instanzen davon:

```
fun preorder  t = foldTree (fn (l,x,r) => [x] @ l @ r) [] t
fun inorder   t = foldTree (fn (l,x,r) => l @ [x] @ r) [] t
fun postorder t = foldTree (fn (l,x,r) => l @ r @ [x]) [] t
```

Breitendurchlauf

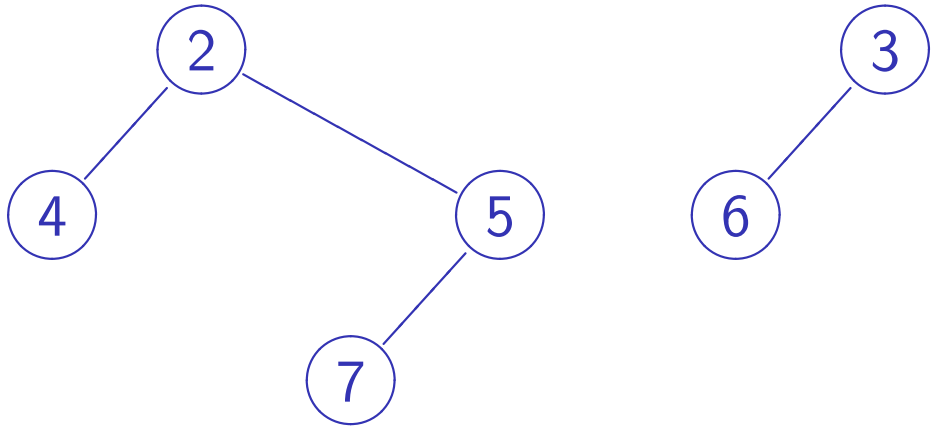
- Beim **Breitendurchlauf** (engl. *breadth-first traversal*) werden die Teilbäume nicht unabhängig behandelt.



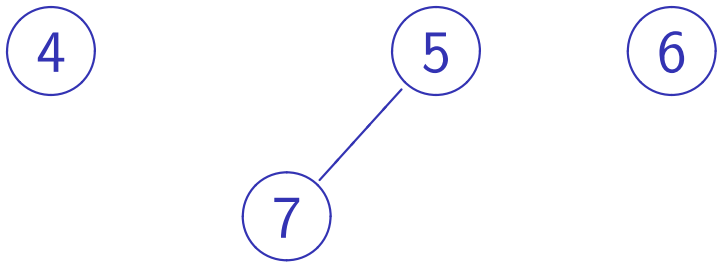
- Der Baum wird schichtenweise abgearbeitet.

Breitendurchlauf

- Nach Entfernen der 1 bleiben zwei Teilbäume:



- Nach Entfernen von 2 und 3 drei Teilbäume:



Breitendurchlauf

- Verallgemeinerung: Breitendurchlauf eines **Waldes** (engl. *forest*), einer Liste von Bäumen.

```
fun upRoot (ts : 'a tree list) : 'a list * 'a tree list =
  case ts
    of ([])                    => ([],[])
     | (Empty         :: ts)   => upRoot ts
     | (Node(l,x,r)   :: ts)   =>
        let val (xs, ts') = upRoot ts
        in  (x :: xs, l :: r :: ts')
        end
```

- upRoot (dt. *Entwurzeln*) zerlegt eine Liste von Bäumen in eine Liste der Wurzeln und eine Liste der Teilbäume.

Breitendurchlauf

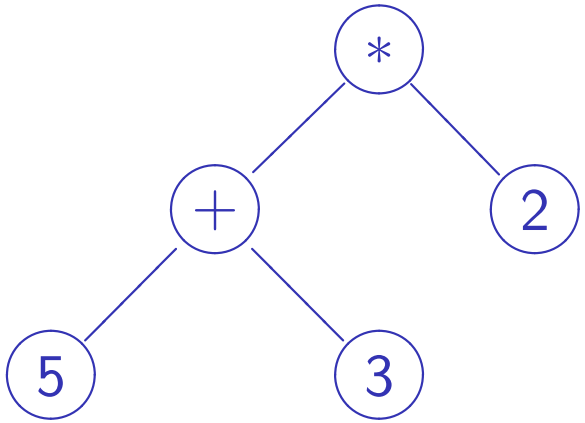
- Beim Breitendurchlauf durch einen Wald sammeln wir erst die Wurzeln und dann durchlaufen wir rekursiv die Teilbäume.

```
fun breadthForest ([]) = []
  | breadthForest (ts) =
      let val (xs, ts') = upRoot ts
      in  xs @ breadthForest ts'
      end

fun breadthFirst t = breadthForest [t]
```

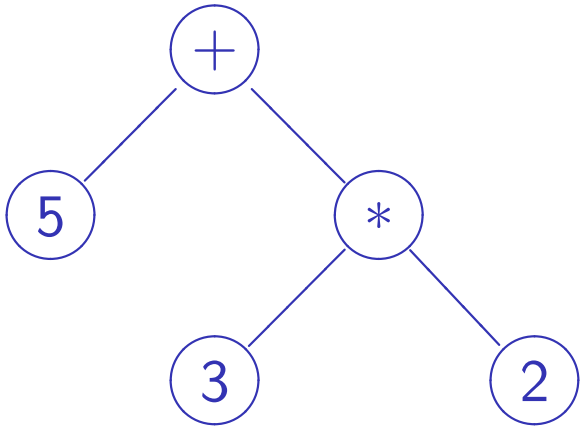
Arithmetische Ausdrücke als Bäume

- Beispiel: Ein Mathe-Nachhilfe-Programm für Grundschüler. Es druckt zufällig einen **arithmetischen Ausdruck** wie $(5 + 3) * 2$, und vergleicht den Lösungsvorschlag mit dem **Wert** des Ausdrucks (hier: 16). Die Zahlen sollen einstellig sein und die Operationen $+$ und $*$.
- Den Ausdruck können wir intern als Binärbaum repräsentieren.

Richtig:



Falsch:



Arithmetische Ausdrücke als Binärbäume

- Es gibt drei Typen von Knotenmarkierungen: Zahl, “+”, “*”.

```
datatype Label
  = Num of int
  | Plus
  | Times

type Expr = Label tree
```

- Beispiel:

```
val a1 = Node(Node(leaf(Num 5), Plus, leaf(Num 3)),
              Times,
              leaf(Num 2))
```

Auswertung

- Den **Wert** des Ausdrucks berechnen wir rekursiv:

```
fun eval (Empty)                 = 0
  | eval (Node(_, Num n, _))     = n
  | eval (Node(l, Plus, r))      = eval l + eval r
  | eval (Node(l, Times, r))     = eval l * eval r
```

- Wir sagen, `eval` (dt. *auswerten*) **interpretiert** den Ausdruck.

```
val a1v = eval a1
```

```
val a1v = 16 : int
```

- **Übung:** Implementieren Sie `eval` als Instanz von `foldTree`!

Ausdrücke ausdrucken

- Eine andere Interpretation des Ausdrucks ist seine Repräsentation as Zeichenkette.

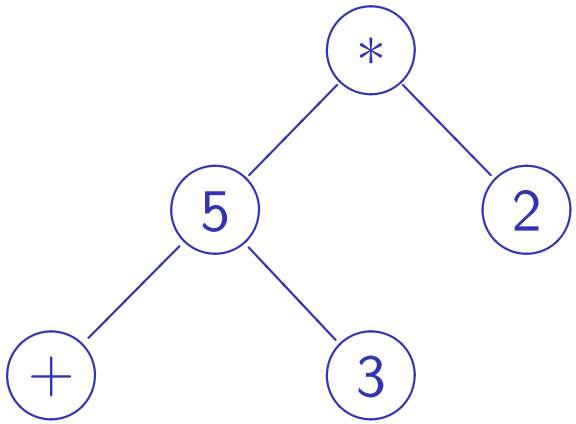
```
fun exprToString (Empty)          = ""
  | exprToString (Node(_, Num n, _)) = Int.toString n
  | exprToString (Node(l, Plus, r)) =
    "(" ^ exprToString l ^ " + " ^ exprToString r ^ ")"
  | exprToString (Node(l, Times, r)) =
    "(" ^ exprToString l ^ " * " ^ exprToString r ^ ")"
```

```
val a1s = "((5 + 3) * 2)" : string
```

- **Übung:** Implementieren Sie `exprToString` als Instanz von `foldTree`!
- **Übung:** Schreiben Sie eine bessere Ausgabefunktion, die Klammern nur verwendet wenn nötig!

Ungültige Ausdrücke

- Die Binärbaum-Repräsentation erlaubt ungültige Ausdrücke (engl. *malformed expressions*).




```
val bad = Node(Node(leaf Plus, Num 5, leaf(Num 3)),
               Times,
               leaf(Num 2))
```

- Zahlen sollen nur als Blattknoten auftreten!
- Operationen nur als innere Knoten!

Arithmetische Ausdrücke als Datentyp

- Wir definieren einen speziellen Baumtyp `expr`:

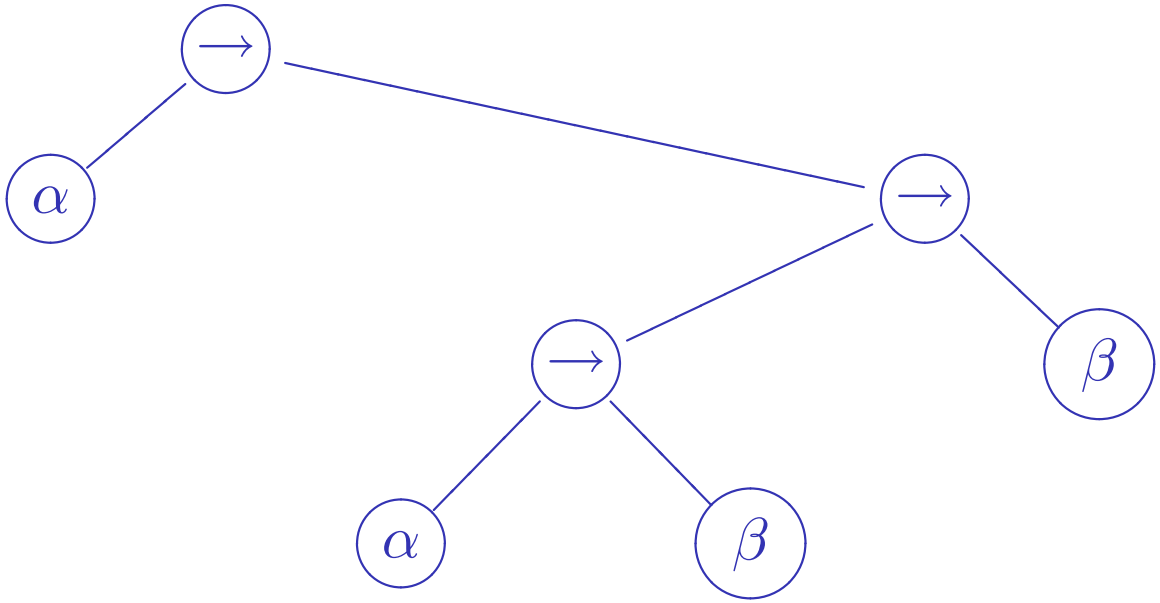
```
datatype expr
  = Num   of int
  | Plus  of expr * expr
  | Times of expr * expr

val a2 = Times(Plus(Num 5, Num 3)), Num 2)
```

- Die Ausdrucksrepräsentation ist ökonomischer.
- Die Funktionsdefinitionen sind auch klarer!

```
fun eval (Num   n)       = n
  | eval (Plus  (l,r))   = eval l + eval r
  | eval (Times (l,r))   = eval l * eval r
```

Typausdrücke als Bäume

- Auch Typausdrücke repräsentiert man als Bäume:
- Beispiel: $\alpha \to ((\alpha \to \beta) \to \beta)$



Typausdrücke als Bäume

- Wie würden wir Typinferenz selbst implementieren? Zuerst müssen wir Typausdrücke reprasentieren:
- Variablen `'a,'b,'c,...` zählen wir einfach durch als `0,1,2,...`:

```
type tyVar = int
```

- Typkonstanten wie `int,string,...` repräsentieren wir durch ihren Namen:

```
type tyBas = string (* "int", "string", "real", "bool" *)
```

- Typausdrücke sind Binärbaume mit speziellen Blättern:

```
datatype tyExp
  = TyVar of tyVar            (* 'a, 'b, ...        *)
  | TyBas of tyBas            (* int, string, ...   *)
  | TyArr of tyExp * tyExp    (* A -> B             *)
```

Drucken von Typausdrücken

- Wir drucken die ersten 26 Typvariablen als 'a,'b,...,'z und die restlichen als 'a0,'a1,'a2,....

```
fun tyVarToString (n) = "'" ^
  (if n < 26 then Char.toString(chr(n + 97))
   else "a" ^ Int.toString(n - 26))
```

- Funktionstypen müssen wir Klammern:

```
fun tyToString (TyVar a)     = tyVarToString a
  | tyToString (TyBas b)     = b
  | tyToString (TyArr(A,B)) =
      "(" ^ tyToString A ^ " -> " ^ tyToString B ^ ")"

val ty1 = TyArr (TyVar 0,
           TyArr (TyArr (TyVar 0, TyVar 1), TyVar 1))
val ty1s = tyToString ty1
```

```
val ty1s = "('a -> (('a -> 'b) -> 'b))" : string
```

Substitution in Typausdrücken

- Eine Substitution $\sigma = [A/\alpha, B/\beta]$ repräsentieren wir als Liste von Paaren `[(0,A),(1,B)]`.

```
type tySubst = (tyVar * tyExp) list
```

- Substitution $A\sigma$ ersetzt in A alle Typvariablen α durch $\sigma(\alpha)$, falls jenes definiert ist.

```
fun substTy (A : tyExp, sigma : tySubst) : tyExp =

  let fun loop (TyVar a) = lookupWithDefault(TyVar a, a, sigma)
        | loop (TyBas b) = TyBas b
        | loop (TyArr(B,C) = TyArr (loop B, loop C)
  in
      loop A
  end
```

Zusammenfassung

- Benutzerdefinierte Datentypen: Aufzählungen, `option`, Listen, **Bäume**.
- Binärbäume.
- Tiefen- und Breitendurchlauf.
- Ausdrücke und Interpretation.