# Programmierung und Modellierung

## Benutzerdefinierte Datentypen

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

# Inhalt

- Kap. 7 Benutzerdefinierte Datentypen
  1. Aufzählungstypen
  2. Typen mit zusammengesetzten Werten
  3. Binärbäume
  4. Rekursive Datentypen in SML
  5. Linearisierung von Binärbäumen
  6. Tiefendurchlauf und Breitendurchlauf
  7. Anwendung: Repräsentation und Auswertung von Termen
  8. Binärer Suchbaum

# 7.1 Aufzählungstypen

- Ein Typ `Farbe` bestehend aus der Wertemenge {`Rot, Gelb, Blau`} kann in SML wie folgt definiert werden:

```
- datatype Farbe = Rot | Gelb | Blau;
datatype Farbe = Blau | Gelb | Rot
- Rot;
val it = Rot : Farbe
```

- Diese Deklaration legt das Folgende fest:
  - Mit dem Schlüsselwort `datatype` können in SML neue Datentypen deklariert werden.
  - Der Name `Farbe` ist eine **Typkonstante**.
  - Die Typkonstante Farbe wird an die Wertemenge {`Rot, Gelb, Blau`} gebunden.
  - Die Namen `Rot, Gelb` und `Blau` sind 0-stellige **(Wert-)Konstruktoren**.
- Ein Datentyp `dt`, der aus endlich vielen Konstanten besteht, wird **Aufzählungstyp** genannt und in SML folgendermaßen deklariert:

  `datatype dt = c`$_1$ `| ... |` `c`$_n$`;`

  Dadurch werden n Konstanten (0-stellige Konstruktoren) `c`$_i$`: dt` deklariert.

# Aufzählungstypen

- Operationen über Aufzählungstypen können durch Pattern Matching bzgl. der Wertkonstruktoren definiert werden:

```
- fun farbname(Rot) = "rot"
| farbname(Gelb) = "gelb"
| farbname(Blau) = "blau";
val farbname = fn : Farbe -> string
- farbname(Gelb);
val it = "gelb" : string
- [Rot, Blau];
val it = [Rot,Blau] : Farbe list
```

- Die Gleichheit wird implizit bei der datatype-Deklaration mitdefiniert:

```
- Blau = Blau;
val it = true : bool
- Blau = Rot;
val it = false : bool
```

# 7.2 Typen mit zusammengesetzten Werten

- Typen mit zusammengesetzten Werten können folgendermaßen definiert werden

```
- datatype Preis = DM of real | Euro of real;
datatype Preis = DM of real | Euro of real
```

- Diese Deklaration legt das Folgende fest:
  - Der Name `Preis` ist eine Typkonstante.
  - Die Typkonstante `Preis` wird an die folgende Wertemenge gebunden:

$$\{DM(x) \mid x \in R\} \cup \{Euro(x) \mid x \in R\}$$

  - Die Namen DM und Euro sind unäre (einstellige) (Wert-)Konstruktoren, beide vom Typ `real -> Preis`.
- Bemerkung
  - Man beachte die Syntax $\texttt{con}_j$ `of` $\texttt{typ}_j$ in der Definition eines Wertkonstruktors für einen benutzerdefinierten Datentyp `dt` :

$$\texttt{datatype dt = ... | con}_j \texttt{ of typ}_j \texttt{ | ...}$$

  - Dadurch werden Wertkonstruktoren $\texttt{c}_i\texttt{: typ}_j$ `-> dt` deklariert.
  - [Die Syntax unterstreicht, dass ein Wertkonstruktor eines Typs keine gewöhnliche Funktion ist.]

# Typen mit zusammengesetzten Werten

- Pattern Matching ist die bevorzugte Weise, Funktionen auf benutzerdefinierten Typen mit zusammengesetzten Werten zu definieren:

```
- fun wechseln(DM(x)) = Euro(0.51129 * x)
| wechseln(Euro(x)) = DM(1.95583 * x);
val wechseln = fn : Preis -> Preis
```

- Es ist empfehlenswert, die Fälle beim Pattern Matching in der Reihenfolge der Typdefinition aufzulisten.
- Unter Verwendung der vordefinierten Funktionen real und round kann `wechseln` verbessert werden, um auf die zweite Nachkommastelle zu runden:

```
fun wechseln(DM(x)) =
          Euro(real(round(0.51129 * x * 1e5)) * 1e~5)
| wechseln(Euro(x)) =
          DM(real(round(1.95583 * x * 1e5)) * 1e~5);
```

# Typen mit zusammengesetzten Werten

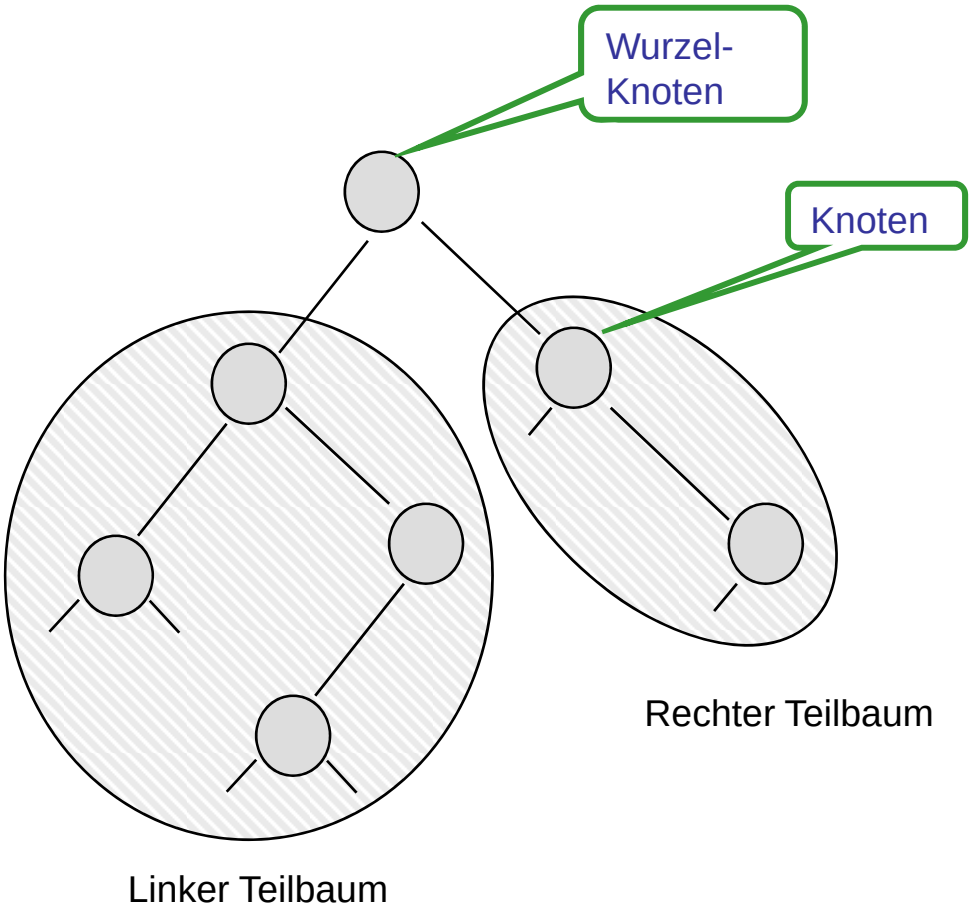
- Die Gleichheit für benutzerdefinierte Typen ist komponentenweise definiert:

```
- datatype zeitpunkt = Sek of int | Min of int | Std of int;
datatype zeitpunkt = Min of int | Sek of int | Std of int
- Sek(30) = Sek(20+10);
val it = true : bool
- Min(60) = Std(1);
val it = false : bool
Euro(2.0) = Euro(2.0);
Error: operator and operand don't agree [equality type
   required]
operator domain: ''Z * ''Z
operand: Preis * Preis
in expression: Euro(2.0) = Euror(2.0);
```

- Zwei Werte sind gleich, wenn
  - ihre (Wert-)Konstruktoren gleich sind (was für Min(60) und Std(1) nicht der Fall ist) und
  - der Argumenttyp des Wertkonstruktors eine Gleichheitsoperation besitzt u.
  - die Argumente der (Wert-)Konstruktoren gleich sind.

# 7.3 Bäume (abstrakt)

- Bäume sind **hierarchische Strukturen**.
- Bäume bestehen aus
  - **Knoten** und
  - **Teilbäumen**.
- Der oberste Knoten heißt **Wurzel**.
- Bei **Binärbäumen** hat jeder Knoten zwei Unterbäume:
  - den **linken Unterbaum**,
  - den **rechten Unterbaum**.
- In den Knoten kann Information gespeichert werden.



# Binärbäume

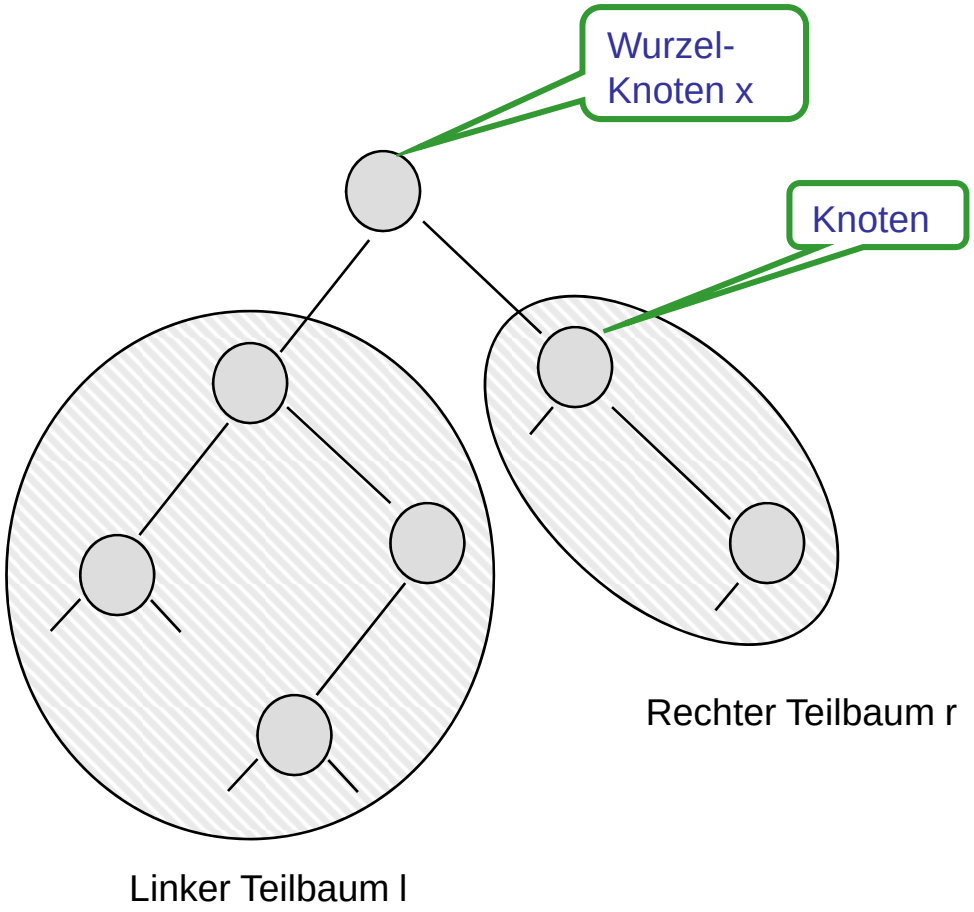
- **Definition:** Sei A eine Menge. Die Menge $A^\Delta$ der **Binärbäume über** A ist induktiv definiert wie folgt:
  - $A^\Delta$ enthält den **leeren Binärbaum** $\varepsilon$
  - Sind x in A und l, r $\in A^\Delta$, so ist das 3-Tupel (x, l, r) $\in A^\Delta$
- Man kann die induktive Definition wie folgt umgehen:

  Die Menge $A^\Delta_n$ der **Binärbäume der Höhe (Tiefe) höchstens n** ist rekursiv (über n) definiert durch
  - $A^\Delta_0 = \{\varepsilon\}$
  - $A^\Delta_{n+1} = A^\Delta_n \cup \{(x, l, r) \mid x \in A, l \in A^\Delta_n, r \in A^\Delta_n\}$.

  Es ist dann $A^\Delta = \bigcup\{A^\Delta_n \mid n \in N\}$
- Die **Höhe** (Tiefe) eines Binärbaumes t $\in A^\Delta$ ist das kleinste n, so dass t $\in A^\Delta_n$.

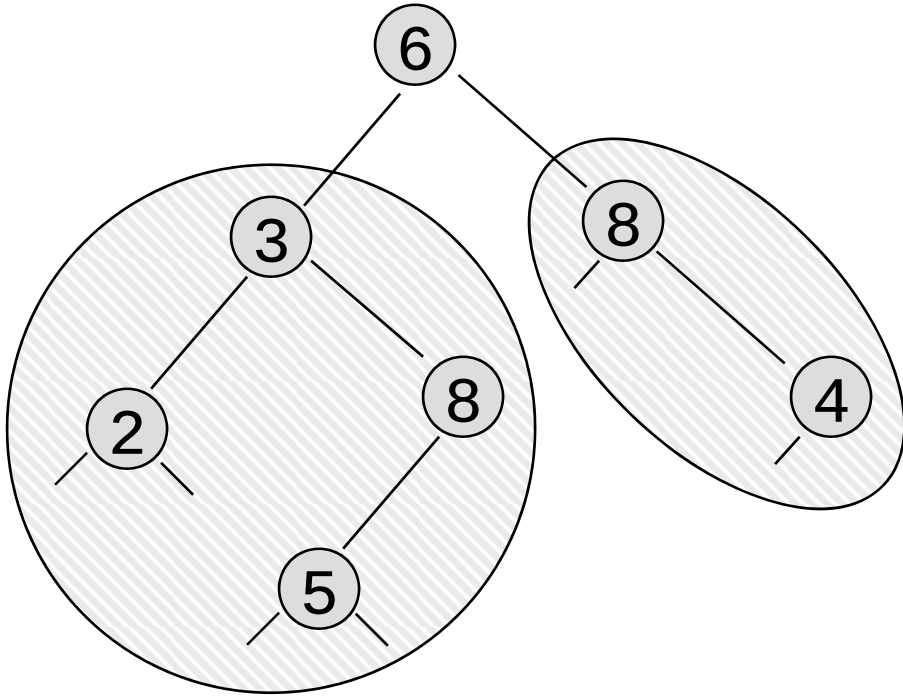
# Binärbäume Terminologie

- x heißt **Wurzel** oder Wurzelbeschriftung von (x, l, r).
- l, r heißen **linker, bzw. rechter Unterbaum** von (x, l, r). Ein Binärbaum der Form (x, ε, ε) heißt **Blatt**. Ein von ε verschiedener Binärbaum heißt **nichtleer**.
- Die **Knoten** und **Teilbäume** eines Binärbaums sind rekursiv definiert wie folgt:
  - ε hat keine Knoten und nur ε als Teilbaum.
  - Die Knoten von (x, l, r) sind x und die Knoten von l und die Knoten von r. Die Teilbäume von (x, l, r) sind (x, l, r) und die Teilbäume von l und von r.



# Binärbäume Beispiel

- t0 = (6,
  (3, (2, ε, ε), (8, (5, ε, ε,), ε)),
  (8, ε, (4, ε, ε))).
- Knoten von t0:
  {6, 3, 2, 8, 5, 4}.
- Teilbäume von t0:
  {t0, (3, (2, ε, ε), (8, (5, ε, ε,), ε)),
  (2 ε, ε ),
  (8, (5, ε, ε,), ε),
  (5, ε, ε ),
  (8, ε, (4, ε, ε)),
  (4, ε, ε), ε}



# 7.4 Rekursive Datentypen in SML

- Ein **benutzdefinierter Datentyp** in SML hat die Form

  `datatype dt = ...| c`$_i$`|...|con`$_j$` of typ`$_j$`|...`

  wodurch die (0-stelligen) Wertkonstruktoren `c`$_i$ `: dt` und die Wertkonstruktoren `con`$_j$ `: typ`$_j$ `-> dt` deklariert werden.
- Man nennt den deklarierten Typ `dt` einen **rekursiven Datentyp, wenn** `dt` auf der rechten Seite der Deklaration in mindestens einem der Typausdrücke `typ`$_j$ auftritt.
- **Beispiel Binärbäume**

```
datatype 'a bintree =
    Empty | Build of 'a * 'a bintree * 'a bintree;
```

- Nun können wir Binärbäume bilden und Funktionen darauf definieren:

```
val t0 =
    Build(6,Build(3,Build(2,Empty,Empty),
                    Build(8, Build(5,Empty,Empty), Empty)),
              Build(8, Empty, Build(4,Empty,Empty))
     );
```

- **Bemerkung**

  Polymorphe Datentypen haben die Form `'a dt` (oder `('a, 'b) dt` oder …)

# Funktionen für Binärbäume in SML

```
fun root (Build(x,_,_)) = x; (* unvollst
  Mustervergleich *)
fun left (Build(_,l,_)) = l;
fun right (Build(_,_,r)) = r;

fun isempty t = (t = Empty);

fun anzahl_knoten Empty = 0
|  anzahl_knoten (Build(_,l,r)) =
          1 + anzahl_knoten l + anzahl_knoten r;

fun max x y = if x>y then x else y;
fun tiefe Empty = 0
|  tiefe (Build(_,l,r)) = 1 + max (tiefe l) (tiefe r);
```

# Binärbäume als Rechenstruktur

- **Bemerkung**
  - Der Datentyp `'a bintree` und
  - die Operationen `Empty, Build, isempty, root, left, right`

  bilden die **Rechenstruktur der Binärbäume**.

# Grundalgorithmen für Binärbäume als Übung

- Gleichheit zweier Binärbäume:

```
bbeq : ''a bintree -> ''a bintree -> bool
```

- Suchen eines Datenelements in einem Binärbaum:

```
enthalten : ''a -> ''a bintree -> bool
```

# 7.5 Linearisierung von Binärbäumen

- Im folgenden definieren wir drei Funktionen

  ```
  linvor, linsym, linnach
  ```

  jeweils vom Typ

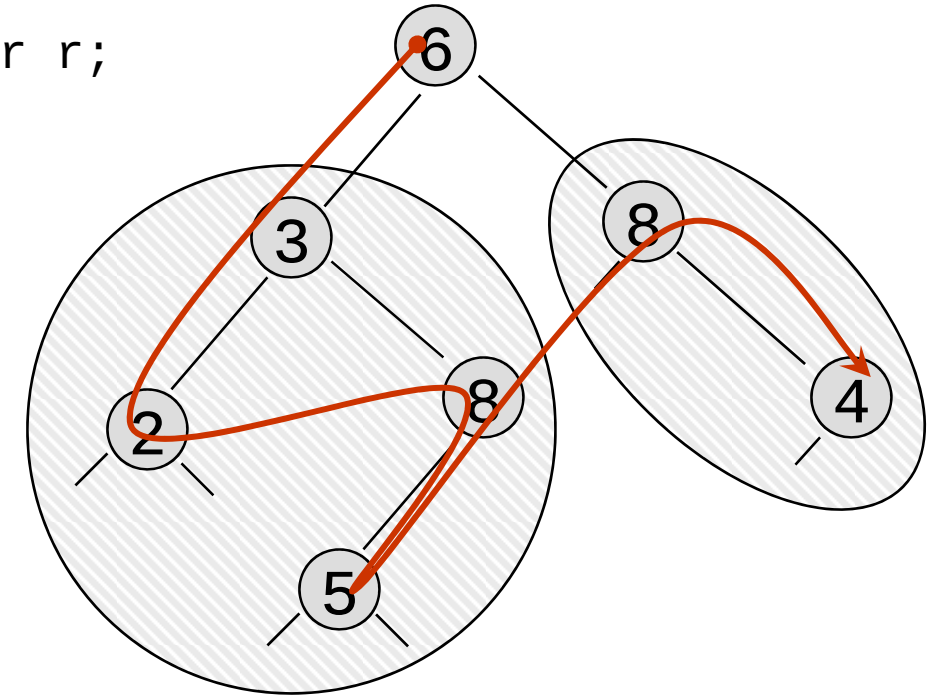
  ```
  'a bintree -> 'a list
  ```

  welche die Knoten eines Baums in einer bestimmten Reihenfolge als Liste berechnen.

# Linearisierung: Vorordnung (Präfix-Schreibweise)

```
fun linvor Empty = []
| linvor (Build(a,l,r)) =
    a :: linvor l @ linvor r;
```

```
- linvor t0;
val it =
[6, 3, 2, 8, 5, 8, 4]
  : int list
```

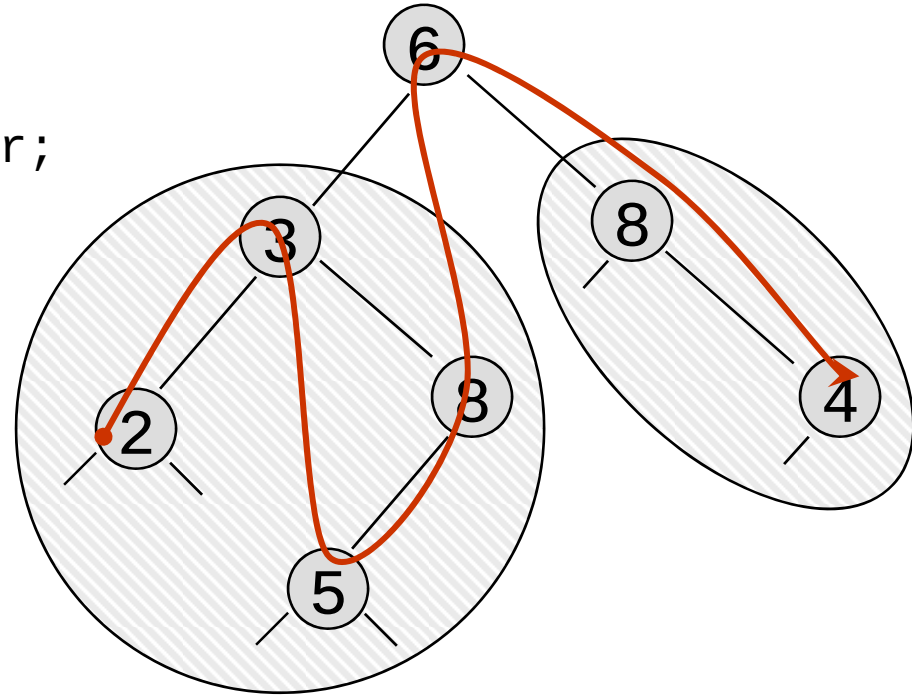


# Linearisierung: Symmetrische Ordnung

```
fun linsym Empty = []
| linsym (Build(a,l,r)) =
  linsym l @ [a] @ linsym r;

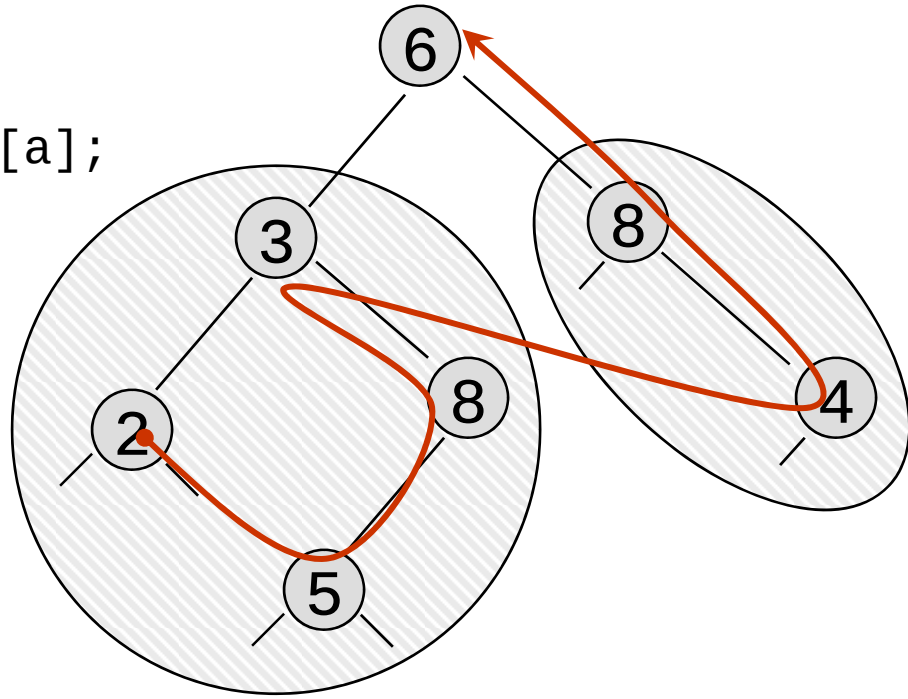


- linsym t0;
val it =
[2, 3, 5, 8, 6, 8, 4]
: int list
```



# Linearisierung: Nachordnung

```
fun linnach Empty = []
| linnach (Build(a,l,r)) =
  linnach l @ linnach r @ [a];
```




```
- linnach t0;
val it =
[2, 5, 8, 3, 4, 8, 6]
: int list
```

# 7.6 Tiefendurchlauf und Breitendurchlauf

- Vorordnung, Nachordnung und symmetrische Ordnung haben gemeinsam, dass die ***Teilbäume*** des Baums unabhängig voneinander ***durchlaufen werden***, d.h. dass alle “Äste” des Baums systematisch durchlaufen werden.
- Der Unterschied der Funktionen liegt nur darin, wie das Ergebnis jeweils zusammengesetzt wird.
- Das gemeinsame Prinzip ist als **Tiefendurchlauf (Depth-First-Durchlauf)** bekannt und kann durch Funktionen höherer Ordnung als gemeinsame Abstraktion definiert werden:

```
fun depth_first c f Empty = c
|   depth_first  c f (Build(a, l ,r)) =
          f(a, depth_first c f l,
               depth_first c f r);
```

# Tiefendurchlauf

- ```
  fun linsym t =
      depth_first nil
                 (fn (a,lv,rv) => lv @ [a] @ rv)
                 t;
  ```
- ```
  fun linvor t =
      depth_first nil
                 (fn (a,lv,rv) => a :: lv @ rv)
                 t;
  ```
- ```
  fun linnach t =
      depth_first nil
                 (fn (a,lv,rv) => lv @ rv @ [a])
                 t;
  ```
  ```

# Tiefendurchlauf

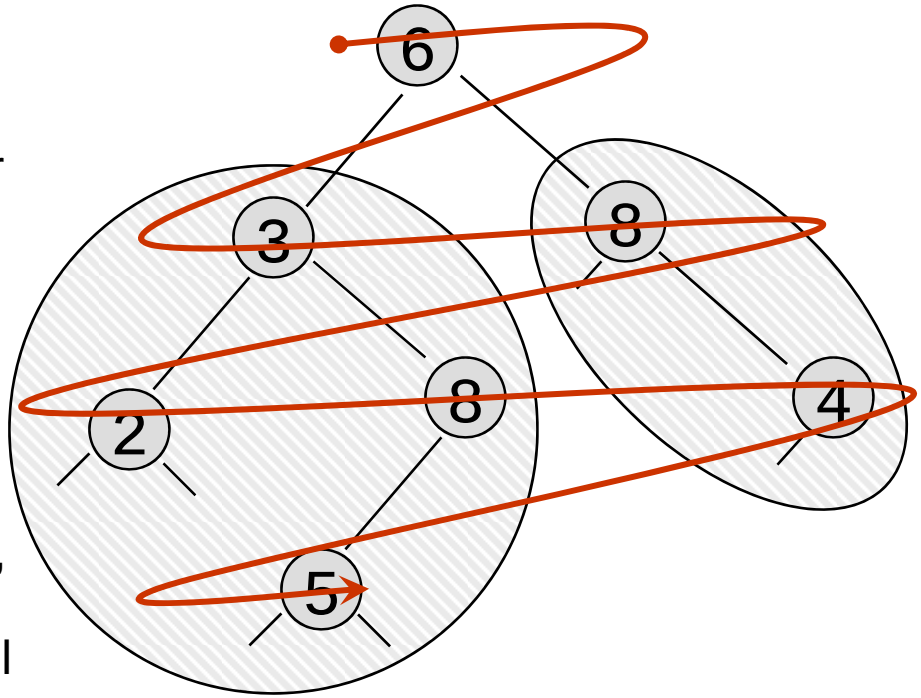
- Aber auch andere nützliche Funktionen auf Binärbäumen lassen sich mit Hilfe des Tiefendurchlaufs leicht implementieren, z.B.

```
fun anzahl_knoten t =
    depth_first 0
            (fn (a,lv,rv) => 1 + lv + rv)
             t;

fun tiefe t =
    depth_first 0
            (fn (a,lv,rv) => 1 + Int.max(lv,rv))
             t;
```

# Breitendurchlauf

- Bei einem **Breitendurchlauf (Breadth-First-Durchlauf)** werden die Knoten nach wachsender Tiefe besucht.
- Zuerst wird die Wurzel aufgesammelt, dann die Wurzeln der Nachfolger, dann die Wurzeln von deren Nachfolgern usw.
- Das Ergebnis des Breitendurchlaufs ist also für Baum `t0` die Liste [6,3,8,2,8,4,5].
- Zur Implementierung bedienen wir uns einer Hilfsfunktion `entwurzeln`, die angewandt auf eine Liste von Bäumen von jedem Baum die Wurzel aufsammelt und die Nachfolger der Wurzel am Ende der Liste für die spätere Weiterverarbeitung einfügt.



# Breitendurchlauf

```
- fun breadth_first(t) =
  let
  fun entwurzeln nil = nil
    | entwurzeln(Empty :: tl) = entwurzeln(tl)
    | entwurzeln(Build(a,l,r) :: tl)=
            a :: entwurzeln(tl @ [l,r])
  in
    entwurzeln(t :: nil)
  end;
val breadth_first =
            fn : 'a bintree -> 'a list
- breadth_first t0;
val it = [6,3,8,2,8,4,5] : int list
```

# 7.7. Anwendung: Repräsentation und Auswertung von Termen

```
val t =
  Build("-",
       Build("+",
             Build("10",Empty,Empty),
             Build("*",Build("x",Empty,Empty),
                       Build("85",Empty,Empty))),
       Build("+",
            Build("124",Empty,Empty),
            Build("y1",Empty,Empty)))
- linnach t;
val it =
["10", "x", "85", "*", "+", "124", "y1", "+", "-"] : string
   list
```

- Terme lassen sich als Bäume repräsentieren. Die Nachordnung entspricht der Postfixnotation. Die Nachordnung entspricht der Postfixnotation, die Vorordnung der Präfixnotation und die symmetrische Ordnung der Infixnotation.

# Bessere Repräsentation von Termen

- Nachteile der vorigen Repräsentation:
  - Sehr viel “Empty”,
  - Auch syntaktisch falsche Terme haben Repräsentation:
    `Z.B. Build("x",Build(...),Build(...)`
- Besser ist die Verwendung einer speziellen rekursiven Variante:

```
datatype binop = Plus | Minus | Mal | Geteilt;
datatype expr =
   Var of string | Num of int |
   Op of binop * expr * expr;

val t =
Op(Minus,
   Op(Plus, Num 10, Op(Mal, Var "x", Num 85)),
   Op(Plus, Num 124, Var "y1"));
```

- D.h. führt man ein:
  - einstellige Wertkonstruktoren für die Variablen und Konstanten und
  - einen n-stelligen Wertkonstruktor für die n-stelligen Operationen .

# Auswertung von Termen

- Repräsentiere **Umgebung** als Liste von Paaren der Form **Variable-Zahl**, z.B.:
  - Umgebung {<"x", 9>, <"y1", 0>, <"z", 3>}
  - repräsentiert als `[("x", 9), ("y1", 0), ("z", 3)]`
- Einfügen mit `::`
- Auslesen mit

```
- fun lookup x ((y,z):: l) =
            if x=y then z else lookup x l;
val it = fn : 'a -> ('a * 'b) list -> 'b
```

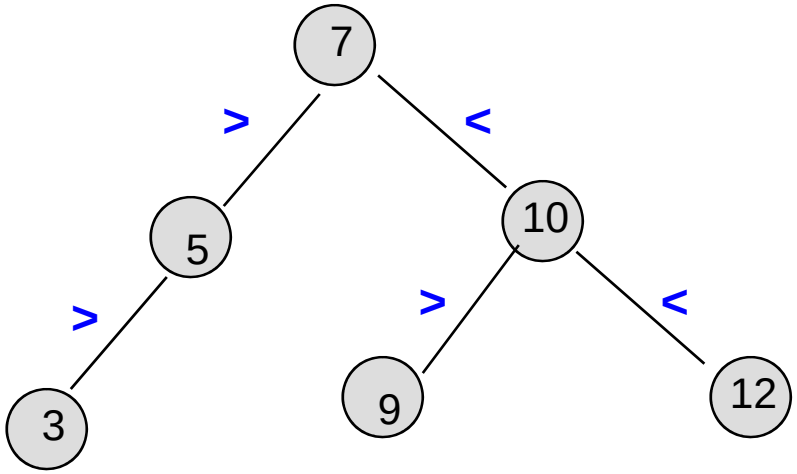
- Eine so verwendete Liste von Paaren heißt **Assoziationsliste.**

# Rekursive Auswertung von Termen

```
fun eval (Num x) env = x
| eval (Var x) env = lookup x env
| eval (Op(Plus,t1,t2)) env = (eval t1 env) + (eval t2 env)
| eval (Op(Minus,t1,t2))env = (eval t1 env) - (eval t2 env)
| eval (Op(Mal,t1,t2)) env  = (eval t1 env) * (eval t2 env)
| eval (Op(Geteilt,t1,t2)) env =
              (eval t1 env) div (eval t2 env);
```

# 7.8 Binärer Suchbaum

- Ein Binärbaum b heißt **geordnet (**oder auch **Suchbaum)**, wenn folgendes für alle nichtleeren Teilbäume t von b gilt:
  - Der Schlüssel von t ist
  - größer (oder gleich) als alle Schlüssel des linken Teilbaums von t und
  - kleiner (oder gleich) als alle Schlüssel des rechten Teilbaums von t
- **Genauer:** Ein binärer Baum `t` heißt **binärer Suchbaum** (binary search tree, BST), wenn
  - `t` = ε oder `t = (a, l, r)` und
    - Für jeden Knoten x von `l` gilt `x < a`.
    - Für jeden Knoten x von `r` gilt `a < x`
    - `l` und `r` sind selbst wiederum binäre Suchbäume.



# Effizientes Suchen in binären Suchbäumen

- Folgende Funktion stellt fest, ob ein Knoten in einem binären Suchbaum enthalten ist:

```
fun enthaltenBST x Empty = false
| enthaltenBST x (Build(y,l,r)) =
    x=y orelse
          if x<y then enthaltenBST x l
          else enthaltenBST x r;
```

- Es werden nur die Knoten auf dem Pfad von der Wurzel zum gesuchten Element, bzw. bis zu einem Blatt angesehen.
- Dagegen werden bei `enthalten` *alle* Knoten angesehen.
- Dafür ist `enthaltenBST` aber nur korrekt für binäre Suchbäume.

# Zusammenfassung (I)

- Ein Datentyp, der aus endlich vielen Konstanten besteht, wird **Aufzählungstyp** genannt.
- Ein **benutzdefinierter Datentyp** in SML hat die Form

  `datatype dt = ...| c`$_i$`|...|con`$_j$` of typ`$_j$`|...`

  wodurch die Konstanten (0-stelligen Wertkonstruktoren) `c`$_i$ `: dt` und die Wertkonstruktoren `con`$_j$ `: typ`$_j$ `-> dt` deklariert werden.
- Man nennt den deklarierten Typ `dt`
  - **Aufzählungstyp**, wenn die rechte Seite nur aus Konstanten besteht;
  - **rekursiven Datentyp,** wenn `dt` auf der rechten Seite der Deklaration in mindestens einem der Typausdrücke `typ`$_j$ auftritt.
- Ein Baum besteht aus **Knoten** und **Teilbäumen**.
  - Der oberste Knoten heißt **Wurzel**.
  - Bei einem **Binärbaum** hat jeder Knoten zwei Unterbäume:
    - den **linken Unterbaum**,
    - den **rechten Unterbaum**.

# Zusammenfassung (II)

- **Terme** lassen sich als Bäume repräsentieren.
  - Die Nachordnung entspricht der Postfixnotation, die Vorordnung der Präfixnotation und die symmetrische Ordnung der Infixnotation.
  - Typischerweise führt man ein:
    - einstellige Wertkonstruktoren für die Variablen und Konstanten und
    - einen n-stelligen Wertkonstruktor für die n-stelligen Operationen .
  - Zur Auswertung der Terme benötigt man eine Umgebung (Assoziationsliste), die die Werte der (freien) Variablen verwaltet.
- Ein Binärbaum b heißt **geordnet (**oder auch **Suchbaum)**, wenn Folgendes für alle nichtleeren Teilbäume t von b gilt:
  - Der Schlüssel von t ist
  - größer (oder gleich) als alle Schlüssel des linken Teilbaums von t und
  - kleiner (oder gleich) als alle Schlüssel des rechten Teilbaums von t