Programmierung und Modellierung

Terme, Suchbäume und Pattern Matching

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

SS 2009

Inhalt

- Kap. 7 Benutzerdefinierte Datentypen
  7. Binärer Suchbaum
  8. Anwendung: Repräsentation und Auswertung von Termen
  9. Pattern Matching

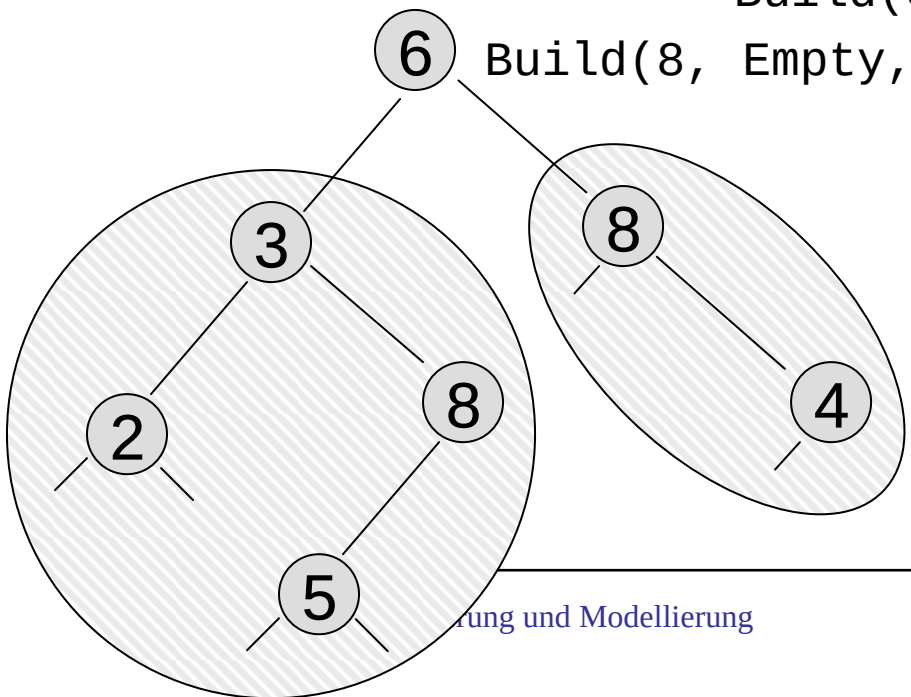
Binärbäume (Wiederholung)

- **Binärbäume in SML**

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

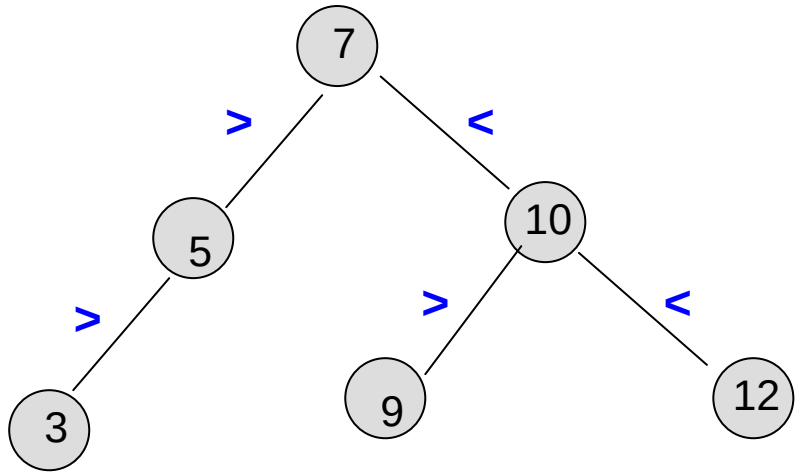
- Beispiel:

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



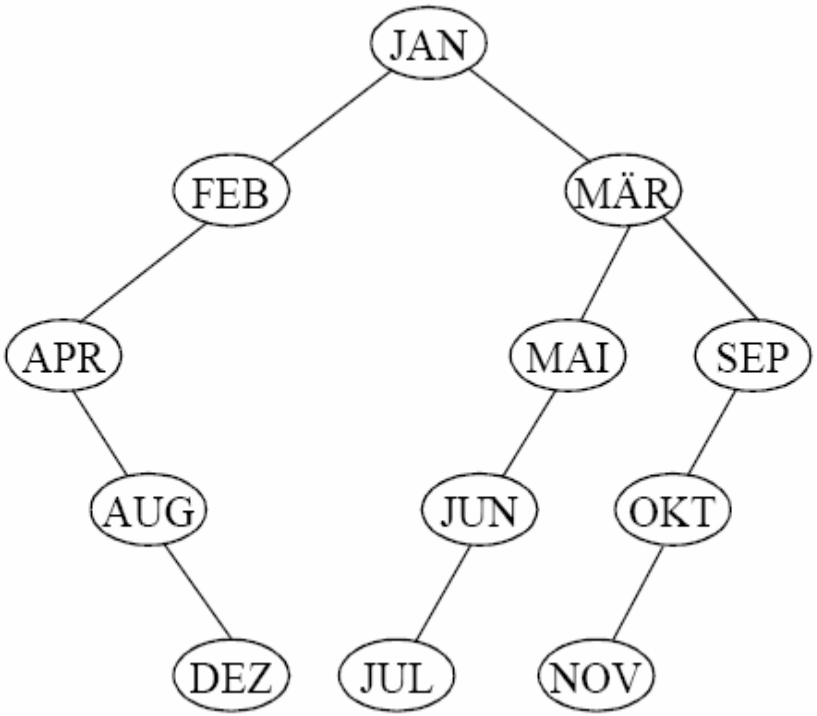
7.7 Binärer Suchbaum

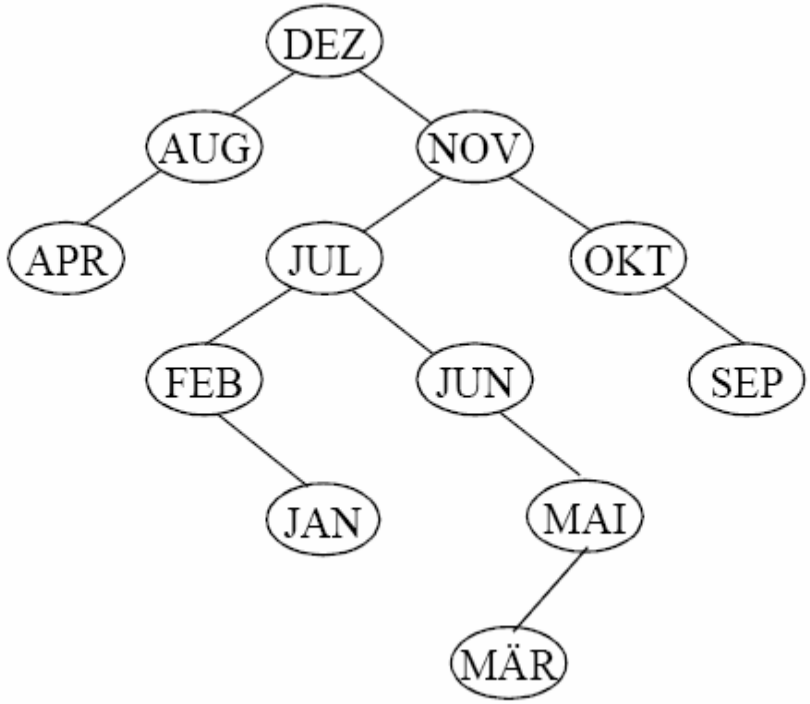
- Ein Binärbaum b heißt **geordnet (**oder auch **Suchbaum)**, wenn folgendes für alle nichtleeren Teilbäume t von b gilt:
  - Der Schlüssel von t ist
  - größer (oder gleich) als alle Schlüssel des linken Teilbaums von t und
  - kleiner (oder gleich) als alle Schlüssel des rechten Teilbaums von t
- **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.



Beispiele: Suchbäume

- **Beispiel:** Zwei verschiedene binäre Suchbäume über den Monatsnamen:



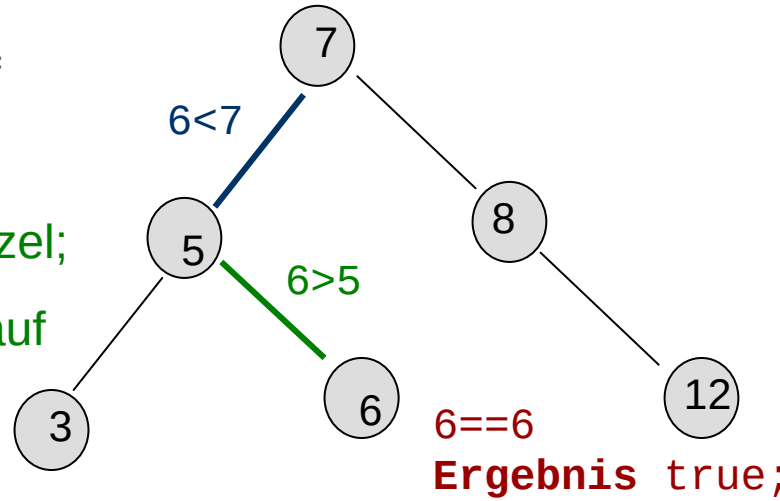


- **Nach welchen Kriterien (Vergleichsoperationen) sind diese Bäume geordnet?**

Effizientes Suchen in binären Suchbäumen

Prinzipieller Ablauf der Berechnung von enthaltenBST 6 t0

1. Vergleiche 6 mit dem Wert der Wurzel;
2. Da `6<7,` wende enthaltenBST rekursiv auf linken Teilbaum an;
    1. Vergleiche 6 mit dem Wert dessen Wurzel;
    2. Da 6>5, wende enthaltenBST rekursiv auf rechten Teilbaum an;
        1. Vergleiche 6 mit dem Wert dessen Wurzel
        2. Da 6==6, gebe Ergebnis true zurück.



Effizientes Suchen in binären Suchbäumen

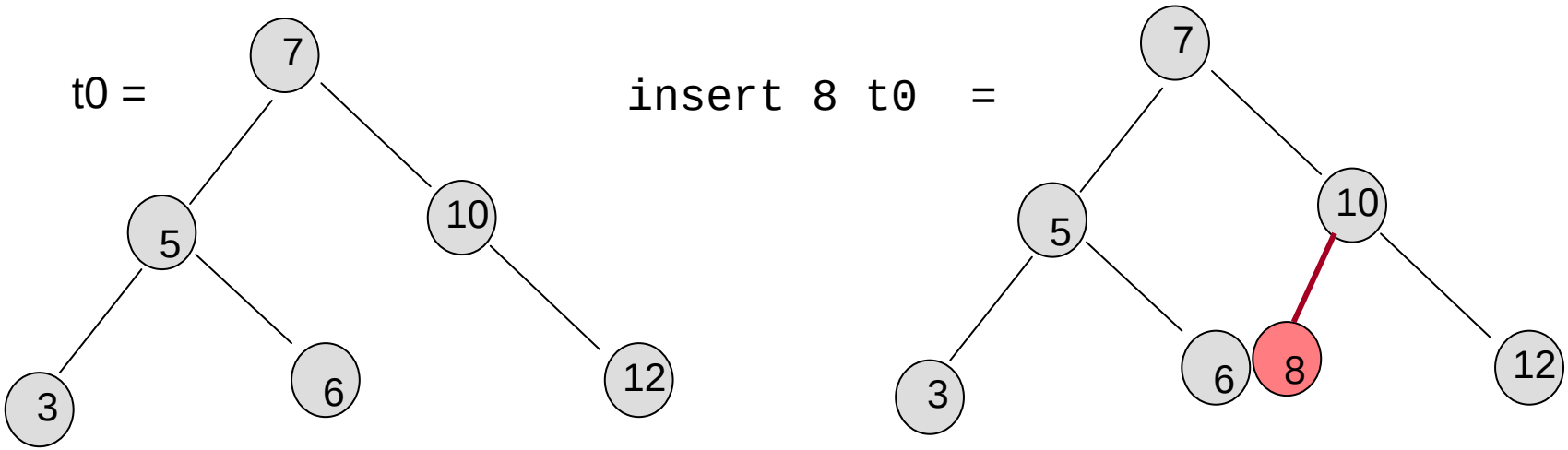
- Folgende Funktion stellt fest, ob ein Knoten in einem binären Suchbaum enthalten ist:

```
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.
- Die Komplexität ist O(h), wenn h die Höhe des Baums bezeichnet.
  - Im besten Fall ist h gleich log n (n = Anzahl der Knoten)
  - Im schlechtesten Fall ist h gleich n (entarteter Baum).

Einfügen in binäre Suchbäume

- Beim Einfügen in einen geordneten Binärbaum wird rekursiv die “richtige” Stelle gesucht, so dass wieder eine geordneter Binärbaum entsteht.
- Beispiel: `insert 8 t0` ergibt:



Einfügen in binäre Suchbäume

```
fun insert x Empty = Build(x,Empty,Empty)
| insert x (Build(a,l,r)) =
    if x<a then Build(a,insert x l,r)
   else   if a<x then Build(a,l,insert x r)
          else Build(a,l,r);
```

Einfügen in binäre Suchbäume

Bemerkung

- Da die Operation „<„ nur in wenigen Datentypen vorkommt, ist `insert` nicht polymorph. Das SML-System bevorzugt den `int`-Datentyp und liefert

```
val insert = fn : int -> int bintree -> int bintree
```

- Dagegen liefert das zusätzliches Constraint `(x:char)`

```
fun insert (x:char) Empty = Build(x,Empty,Empty)
| insert x (Build(a,l,r)) =
    if x<a then Build(a,insert x l,r)
   else   if a<x then Build(a,l,insert x r)
          else Build(a,l,r);
```

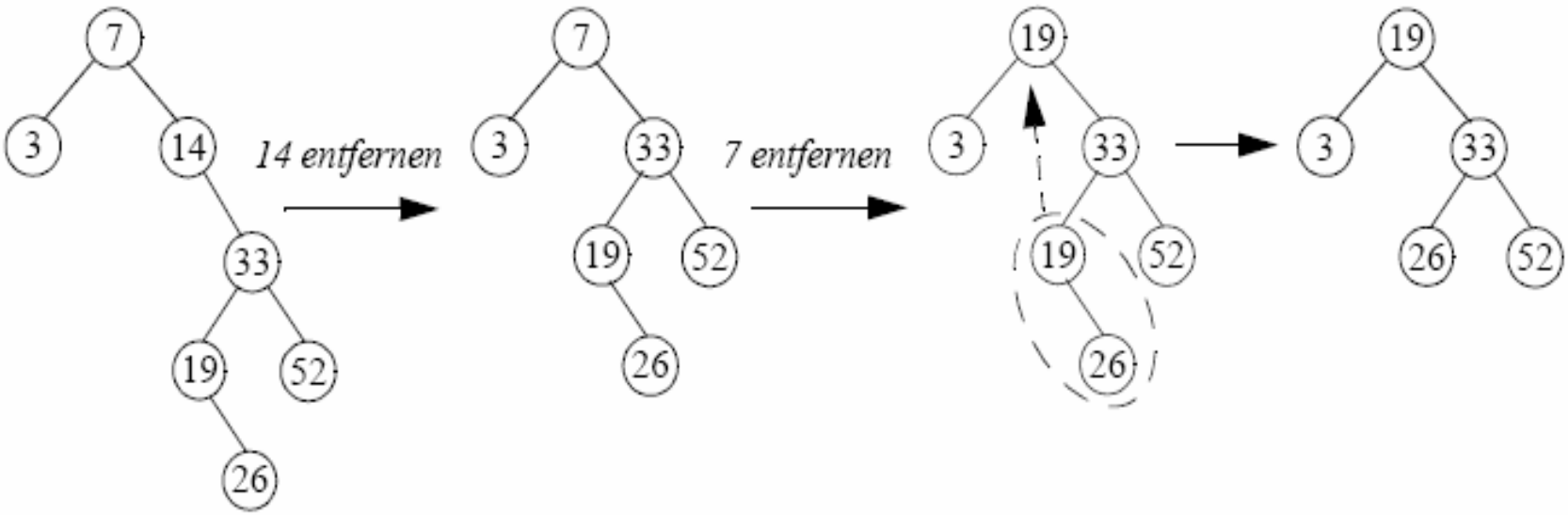
den Typ

```
val insert = fn : char -> char bintree -> char bintree
```

Löschen in binären Suchbäumen

Beim Entfernen eines Schlüssels muss die Suchbaumstruktur aufrecht erhalten werden. Zur Entfernung der Wurzel sucht man das kleinste Element im rechten Teilbaum.

Beispiel:



Löschen in binären Suchbäumen

```
fun delete x Empty = Empty
|   delete x (Build(a,l,r)) =
      if x<a then Build(a,delete x l,r)
      else if x>a then Build(a,l,delete x r)
           else if l=Empty then r
                else if r=Empty then l
                     else let val (m,r1) = remove_min r
                           in Build(m,l,r1)
                          end;
```

wobei `remove_min` das kleinste Element entfernt:

```
fun remove_min (Build(a,Empty,r)) = (a,r)
|   remove_min (Build(a,l,r)) =
          let val (m,l1) = remove_min l
          in (m,Build(a,l1,r))
          end;
```

Knotenanzahl und Höhe

Sei n die Knotenanzahl eines Baumes t und h seine Höhe.

- Im besten Fall gilt $n = 2^h-1$, also $h = O(\log n)$.
- Im schlechtesten Fall gilt $n = h$, also $h = O(n)$.
- Suchen, Einfügen, Löschen haben Komplexität O(h), also $O(\log n)$ bzw. $O(n)$ je nach Art von t.

Zwei extreme Beispiele

- ```
  val b1 =
    insert 6 (insert 5 (insert 4 (insert 3
      (insert 2 (insert 1 (insert 0 Empty))))));
  ```
- ```
  val b2=
    insert 6 (insert 4 (insert 1 (insert 0
      (insert 5 (insert 1 (insert 3 Empty))))));
  ```
- t1 hat maximale Höhe (7) und t2 hat minimale Höhe (3)

- **Bemerkung**
  - Eine Menge von Bäumen B heißt **balanciert**, wenn

    max {Höhe(t) | knotanz(t) <= n; t $\in$ B} = O(log(n))
  - Für balancierte BST mit n Knoten erfordern Einfügen, Löschen, Suchen die Zeit O(log(n)).(siehe Vorlesung effiziente Algorithmen); dazu sind lokale Umordnungen (sogenannte Rotationen) des Baums erforderlich.

7.8 Darstellung von Termen

- Siehe Tafel

Terme als Bäume

```
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 Vorordnung der Präfixnotation und die symmetrische Ordnung der Infixnotation.
- **Nachteile dieser Repräsentation:**
  - Sehr viel “Empty”,
  - Auch syntaktisch falsche Terme haben Repräsentationen:
    `Z.B. Build("x",Build(...),Build(...)`

Abstrakter Syntaxbaum

- Eine bessere Darstellung erhält man durch Verwendung von "**abstrakter Syntax**": Man beschreibt die Syntax einer Sprache durch eine Menge rekursiver Datentypen, so dass **jedes (essentielle) syntaktische Konstrukt durch einen Wertkonstruktor spezifiziert** wird.
- **Beispiel**

```
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.
- Bemerkung:

  Die abstrakte Syntax entspricht einer **kontextfreien Grammatik (bzwder Backus-Naur-Form BNF)**

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);
```

Konvertierung von Ausdrücken in Strings

- Mit Hilfe der symmetrischen Ordnung (und Einführung von Klammern) lässt sich ein abstrakter Syntaxbaum in einen arithmetischen Ausdruck umwandeln:

```
  show(Num x)   = Int.toString x
| show (Var x) = x
| show (Op(Plus,t1,t2)) =
            "(" ^(show t1) ^ "+" ^(show t2) ^ ")"
| show (Op(Minus,t1,t2))  =
            "(" ^(show t1) ^ "-" ^(show t2) ^ ")"
| show (Op(Mal,t1,t2)) =
            "(" ^(show t1) ^ "*" ^(show t2) ^ ")"
| show (Op(Geteilt,t1,t2)) =
            "(" ^(show t1) ^ "div" ^(show t2) ^ ")";
```

7.9 Pattern Matching

- **Muster kommen in SML in vielen Formen vor**
  - In Wertdeklarationen, z.B.

```
val (c1, c2) = (1.37, 0.66);
```

  - In Funktionsdeklarationen, z.B.
    -
```
fun laenge nil = 0
|   laenge (x :: l) = 1+ length l;
```
    -
```
fun is_ordered nil = true
| is_ordered (x::nil) = true
| is_ordered(x :: y :: l) = (x<y) andalso is_ordered (y :: l)
```
    -
```
fun size Empty = 0
|     size Build(x, l, r) = 1 + size l +size r;
```

- **Pattern (Muster)**

  Ein **Muster** ist ein Ausdruck M, der aus Wertkonstruktoren und freien Variablen gebildet wird und folgende **Linearitätsbedingung** erfüllt:
  - Jede freie Variable darf in einem Muster höchstens einmal auftreten.

Linearitätsbedingung für Muster

- **Beispiel**

  Die folgende polymorphe Funktion

```
- fun gleich(x, y) = (x = y);
val gleich = fn : ''a * ''a -> bool
```

  kann also **nicht** wie folgt deklariert werden:

```
- val gleich = fn (x, x) => true
| _ => false;
Error: duplicate variable in pattern(s): x
```

- Diese Einschränkung erlaubt es, den Pattern-Matching Algorithmus einfach und effizient zu halten. Ohne diese Einschränkung müssten bei Mehrfachvorkommen von Variablen in Mustern die entsprechenden Werte zu Gleichheitstypen gehören.

Informelle Spezifikation des Pattern-Matching Algorithmus

- Der Pattern-Matching Algorithmus ist ein „Angleich“ eines Musters M an einen Wert W
  - Gelingt der "Angleich“, so werden etwaige Namen, die im Muster vorkommen, gebunden. Anderfalls der Algorithmus fest, dass der Angleich von Muster M und Wert W unmöglich ist, d.h. gescheitert ist.
  - Beim Angleich werden die Strukturen von Muster und Wert gleichzeitig durchlaufen (bzw. zerlegt), indem die Teilausdrücke komponentenweise und rekursiv angeglichen werden.
  - Basisfälle dieses rekursiven Algorithmus sind Variablen und Konstanten, die im Muster vorkommen.
  - Der Pattern-Matching Algorithmus soll auch feststellen, dass der Angleich von Muster M und Wert W unmöglich ist, d.h. gescheitert ist.

Informelle Spezifikation des Pattern-Matching Algorithmus

- Pattern Matching erfolgt zwischen einem Muster M und einem Wert W; in einem Programm sind aber jeweils ein Muster M und ein Ausdruck A gegeben, z.B. in der Form

  `val M = A;` oder `fun f M = … ; … f(A) …`

- Bei der Auswertung wird zunächst A in der aktuellen Umgebung ausgewertet zu einem Wert W.
- Danach wird der Pattern-Matching Algorithmus auf das Muster M und den Wert W angewandt.

Pattern Matching Algorithmus

Zur Illustration definieren wir in SML einen Pattern Matching Algorithmus für Ausdrücke über `int list`

- **Abstrakte Syntax der Pattern**

```
datatype pat =
  Var of string | Num of int |
  Nil |
  Cons of pat * pat;
```

  - **Beispiele**

```
val pat0 = Cons(Var("x"), Var("l"));
val pat1 = Cons(Var("x"), Cons(Num 3, Nil));
```

- **Werte** werden nur mit `Num`, `Nil` und `Cons` konstruiert.
  - Beispiele

```
val w1 = Cons (Num 17, Cons(Num 3, Nil));
val w2 = Cons (Num 17, Cons(Num 3, Cons(Num 13, Nil)));
```

Der Datentyp `option`

- Der polymorphe Datentyp **`option`** wird oft zur Darstellung partiell definierter Funktionen verwandt. Er liefert eine Unterscheidung zwischen einem Wert und “keinem Wert”:

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

- **Bemerkung:** Man kann den Wert `NONE` verstehen als äquivalent zu den Null-Werten in Datenbanken.

Pattern Matching Algorithmus

Der Pattern Matching Algorithmus ist rekursiv definiert:

```
- fun match Nil Nil = SOME []
  | match (Cons (parg1, parg2))  (Cons (arg1, arg2))  =
          (case (match parg1 arg1, match parg2 arg2)
          of (SOME e1, SOME e2) => SOME (e1 @ e2)
          | _ => NONE )
  | match (Num i) (Num  j) =
          if (i = j) then SOME [] else NONE
  | match (Var(x)) arg = SOME [(x, arg)]
  | match _ _ = NONE;
val match = fn : pat -> pat -> (string * pat) list option

- match pat1 w2;
val it = NONE : (string * pat) list option
- match pat0 w1;
val it = SOME [("x",Num 17),("l",Cons (#,#))]
    : (string * pat) list option
```

Zusammenfassung (I)

- **Suchbäume**
  - 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
  - Einfügen, Löschen und Suchen erfordert lineare Zeit in der Höhe desSuchbaums; d.h. ist im schlechtesten Fall linear in der Anzahl der Knoten.
  - In balancierten Bäumen haben Einfügen, Löschen und Suchen jeweils logarithmische Zeitkomplexität (in der Anzahl der Knoten).
- **Terme** lassen sich als Bäume repräsentieren.
  - Die Nachordnung entspricht der Postfixnotation, die Vorordnung der Präfixnotation und die symmetrische Ordnung der Infixnotation.
  - Man beschreibt die **abstrakte Syntax** von Ausdrücken durch rekursive Datentypen, wobei stehen:
    - einstellige Wertkonstruktoren für die Variablen und Konstanten und
    - n-stellige Wertkonstruktoren für die n-stelligen Operationen .
  - Zur Auswertung der Terme benötigt man eine **Umgebung (Assoziationsliste),** die die Werte der (freien) Variablen verwaltet.

Zusammenfassung (II)

- **Pattern (Muster)**

  Ein **Muster** ist ein Ausdruck, der aus Wertkonstruktoren und freien Variablen gebildet wird und folgende **Linearitätsbedingung** erfüllt:
  - Jede freie Variable darf in einem Muster höchstens einmal auftreten.
- Der Pattern-Matching Algorithmus realisiert ein „Angleich“ eines Musters an einen Wert.
  - Gelingt der "Angleich“, so werden etwaige Namen, die im Muster vorkommen, gebunden. Anderfalls stellt der Algorithmus fest, dass der Angleich von Muster und Wert unmöglich ist, d.h. gescheitert ist.
  - Zur Unterscheidung dieser Ergebnisse wird der polymorphe Datentyp `option` verwandt, der häufig zur Darstellung partieller Funktionen dient.