Realisierung in OCAML

Binärbäume sind nicht fest eingebaut, können aber leicht als *rekursive Variante* definiert werden.

```
type 'a bintree = Empty | Build of 'a * 'a bintree * 'a bintree
```

Nun können wir Binärbäume bilden und Funktionen darauf definieren:

```
let t = Build(6,Build(3,Build(2,Empty,Empty),
                        Build(8,Empty,Build(5,Empty,Empty))),
                Build(8,Build(4,Empty,Empty),Empty)
              )
let root (Build(x,_,_)) = x (* unvollst Mustervergleich *)
let left (Build(_,l,_)) = l
let right (Build(_,_,r)) = r
let isempty t = (t = Empty)
let rec knotanz t = if isempty t then 0 else
         1 + knotanz (left t) + knotanz (right t)
let rec hoehe t = if isempty t then 0 else
         1 + max (hoehe (left t))  (hoehe (right t))
```

Binärbäume als Rechenstruktur

Der Datentyp `'a bintree` und die Operationen `Empty, Build, isempty, left, right` bilden die *Rechenstruktur der Binärbäume*.

In [Kröger, S62] wird diese zusätzlich auf der Pseudocodeebene eingeführt.

Weitere Grundalgorithmen als Übung

Gleichheit zweier Binärbäume:

```
bbeq : ’ a bintree * ’a bintree -> bool
```

Suchen eines Datenelements in einem Binärbaum:

```
enthalten2 : ’ a * ’a bintree -> bool
```

Linearisierungen

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.

Vorordnung

```
let rec linvor = function
    Empty -> []
  | Build(a,l,r) -> a :: linvor l @ linvor r

let t = Build(6,Build(3,Build(2,Empty,Empty),
                        Build(8,Empty,Build(5,Empty,Empty))),
                Build(8,Build(4,Empty,Empty),Empty)
              )
linvor t;;
- : int list = [6; 3; 2; 8; 5; 8; 4]
```

Symmetrische Ordnung

```
let rec linsym = function
    Empty -> []
  | Build(a,l,r) -> linsym l @ [a] @  linsym r

let t = Build(6,Build(3,Build(2,Empty,Empty),
                        Build(8,Empty,Build(5,Empty,Empty))),
                Build(8,Build(4,Empty,Empty),Empty)
              )
linsym t;;
- : int list = [2; 3; 8; 5; 6; 4; 8]
```

Nachordnung

```
let rec linnach = function
    Empty -> []
  | Build(a,l,r) -> linnach l @ linnach r @ [a]

let t = Build(6,Build(3,Build(2,Empty,Empty),
                        Build(8,Empty,Build(5,Empty,Empty))),
                Build(8,Build(4,Empty,Empty),Empty)
              )
linnach t;;
- : int list = [2; 5; 8; 3; 4; 8; 6]
```

Anwendung: Repräsentation von Termen

```
let 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;;
- : string list =
    ["10"; "x"; "85"; "*"; "+"; "124"; "y1"; "+"; "-"]
```

Terme lassen sich als Bäume repräsentieren. Die Nachordnung entspricht der Postfixnotation (vgl.: HP Taschenrechner)

Bessere Repräsentation von Termen

Nachteile der vorigen Repräsentation: Sehr viel “`Empty`”, auch syntaktisch falsche Terme haben Repräsentation:
“`Build("x",Build(...),Build(...)`”

Besser ist die Verwendung einer speziellen rekursiven Variante:

```
type op = Plus | Minus | Mal | Geteilt
type expr = Var of string | Num of int | Op of op * expr * expr

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

Auswertung solcher Terme

Repräsentiere Umgebung als *Liste* von Paaren Variable-Zahl, z.B.:

$$\{<\texttt{"x"}, 9>, <\texttt{"y"}, 0>, <\texttt{"z"}, 3>\} \mapsto [(\texttt{"x"}, 9); (\texttt{"y"}, 0); (\texttt{"z"}, 3)]$$

Einfügen mit `::`

Auslesen mit

```
List.assoc : 'a -> ('a * 'b) list -> 'b
```

Eine so verwendete Liste von Paaren heißt *Assoziationsliste*.

Rekursive Auswertung

```
let rec eval t env = match t with
    Num x -> x
|   Var x -> List.assoc x env
|   Op(op,t1,t2) -> let v1 = eval t1 env in
                    let v2 = eval t2 env in
                    (match op with
                        Plus -> v1 + v2
                    |   Minus -> v1 - v2
                    |   Mal -> v1 * v2
                    |   Geteilt -> v1 / v2)
```

Parsing

```
type token = TokNum of int | TokL | TokR |
             TokOp of op | TokVar of string
```

Übung: Man definiere eine (rekursive) Funktion

```
parse : token list -> token list * expr
```

derart, dass aus `parse l = (rest,e)` folgt, dass `l = anfang @ rest` und `anfang` den Ausdruck `e` bezeichnet. Außerdem soll `rest` so kurz wie möglich gewählt werden.

Beginnt `l` nicht mit der Darstellung eines Ausdrucks, so wird die (selbstdefinierte) Ausnahme `ParseError` ausgelöst.

Beispiel:

```
parse [TokNum 13;TokNum 13] = ([TokNum 13], Num 13)
parse [TokNum 2;TokOp Plus;TokL;TokNum 3;TokOp Mal;TokNum 5;TokR]=
      ([], Op (Plus, Num 2, Op (Mal, Num 3, Num 5)))
parse [TokR] --> ParseError
```

Binärer Suchbaum

Definition: Sei t binärer Baum mit Knoten aus `int`. Der Baum t heißt *binärer Suchbaum* (*binary search tree, BST*), wenn $t = \tau$ oder $t = (a, l, r)$ und

- Für jeden Knoten x von l gilt $x \leq a$.
- Für jeden Knoten x von r gilt $a \leq r$.
- l und r sind selbst wiederum binäre Suchbäume.

Effizienteres Suchen in binären Suchbäumen

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

```
let rec enthaltenBST(x,t) = match t with
        Empty -> false
|       Build(a,l,r) -> x=a ||
                  (if x<=a 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 angeschaut.

Dagegen werden bei `enthalten2` *alle* Knoten angeschaut.

Dafür funktioniert aber `enthaltenBST` nur für binäre Suchbäume.