3. Benutzerdefinierte Datentypen

Im folgenden werden wir am Beispiel ML sehen, wie in Programmiersprachen neue Typen eingeführt werden können. Zusammen mit den Datenobjekten dieser Typen werden auch (nicht nullstellige) Konstruktoren zum Aufbau dieser Datenobjekte eingeführt. (Typen mit Konstruktoren werden auch "Datentypen" genannt.)

Wir betrachten die folgenden Arten von Typdeklarationen:

– Enumerationstypen,
– benutzerdefinierte Produkte,
– Typen mit Varianten (direkte Summe),
– Rechenstrukturen und ihre Signaturen,
– polymorphe Datentypen.

3.1 Enumerationstypen

Strukturen mit endlichen Trägermengen können durch einfache Aufzählung eingeführt werden:

```
– datatype direction = Down | Left | Right | Up;
> datatype direction = Down | Left | Right | Up
```

Damit wird der Typname `direction` eingeführt, sowie vier verschiedene Werte vom Typ `direction`, dargestellt durch die Konstanten (dh. null-stelligen Konstruktoren) `Up, Right, Down, Left` (vom Typ `direction`).

3.1.1. Allgemeine Form der Deklaration von Enumerationstypen in ML

`datatype T =` C_1 `|` ... `|` C_n

Diese Deklaration führt einen Typ T und n Konstanten

C_1: T, ..., C_n : T

ein. Der Typ T besitzt paarweise verschiedene Werte (bezeichnet durch C_1 | ... | C_n). Die C_i sind die Normalformen des Typs T :

$NF_T = \{C_1, ... ,C_n\}$

Die Konstanten von Aufzählungstypen können per Gleichheitstest (=, <>) verglichen werden.

```
–   Up = Right;
>   false : bool;
```

Funktionen über Aufzählungstypen werden mit Hilfe von Pattern Matching definiert.

```
– fun   turn Up = Right
=   |   turn Right = Down
=   |   turn Down = Left
=   |   turn Left = Up;
> val   turn = fn: direction –> direction
```

In ML vordefinierte Aufzählungstypen sind

```
–   datatype bool = true | false;
–   datatype unit = ();
```

Das leere Tupel `()` ist die einzige Konstante vom Typ `unit`.

Im nächsten Schritt betrachten wir Datentypen mit einem mehrstelligen Konstruktor.

3.2 Benutzerdefinierte Produkttypen

Dies sind Datentypen mit einem mehrstelligen vom **Benutzer** gewählten Konstruktor.

```
–   datatype rat = Rat of int * int;
>   datatype rat = Rat of int * int
```

`Rat` hat den Typ `int * int -> rat` und ist der Konstruktor des Produkttypen `rat`.
Ausdrücke vom Typ `rat` sind beispielsweise :

```
Rat (2,3)
Rat (2,0)
Rat (2,3+3)
```

aber **nicht** `(2,3)` `(* (2, 3)` ist vom Typ `int*int)`

`Rat (2,3)` und `Rat (2,0)` sind Normalformen vom Typ `rat`.
`Rat (2,3+3)` ist keine Normalform.

Man kann leicht Funktionen deklarieren, um auf die einzelnen Komponenten des Produkttyps zuzugreifen.

```
–   fun num (Rat (n,d)) = n;
>   val num = fn : rat –> int;
```

```
–   fun den (Rat (n,d)) = d;
>   val den = fn: Rat –> int;
```

Die Klammern um `Rat (n,d)` sind hier wichtig.
Bei Eingabe von

```
–   val den Rat (n,d) = d;
```

erhält man

```
>   Error: data constructor Rat used without argument pat-
    tern
```

Ein Produkttyp kann auch aus verschiedenen Typen gebildet werden.

```
–   datatype move = Move of direction * int;
```

Mit `move` können z.B. Züge auf einem Spielbrett repräsentiert werden.
Ausdrücke vom Typ `move` sind

```
Move (Left,3)
Move (Up,7)
Move (Up,4+3)
```

Benutzerdefinierte Konstruktoren können auch dazu dienen, vordefinierte Typen (z.B. `int`) zur Repräsentation verschiedenartiger Werte zu verwenden. In diesem Fall sind einstellige Konstruktoren sinnvoll.

```
–   datatype height = Ht of int;
>   datatype height = Ht of int
–   datatype age = Age of int;
>   datatype age = Age of int
```

Die Eingabe

```
–   Ht 180 + Age 20;
```

ergibt einen doppelten Fehler: Die Addition „+“ ist nicht definiert für den Typ `height`; außerdem müssen bei der Addition die Typen beider Argumente übereinstimmen, was hier nicht der Fall ist.

3.2.1 Allgemeine Form von benutzerdefinierten Produkttypen

```
datatype T = C of (T1 * ... * Tn)
```

wobei `T` und `C` Namen sind und $T_1,...,T_n$ Typen (ohne Typvariablen).
Durch diese Deklaration wird der Typ `T` und der Konstruktor

$C : (T_1 * \ ... \ * T_n) \rightarrow T$

definiert. Die Werte vom Typ `T` werden bezeichnet durch die Normalformen

$NF_T =_{def} \{C(e_1,...,e_n) \mid e_i \in NF_{Ti} \text{ für } i = 1, ..., n\}$

Hierbei können im Fall n=1 die extra Klammern weggelassen werden, wenn das Argument eine Konstante oder ein Tupel ist (z.B. `Ht 180`).

3.2.2 Gleichheit auf Produkttypen

Falls die T_i keine Funktionstypen beinhalten, so ist die Gleichheit auf dem Produkttyp

```
datatype T = C of (T1 * ... * Tn)
```

die komponentenweise Gleichheit, d.h.

$C(x_1,...,x_n) = C(y_1,...,y_n) \quad \Leftrightarrow \quad$ Für alle i = 1, ..., n gilt $x_i = y_i$

Beispiel:

```
-  Rat (2,2) = Rat (1,1);
>  val it = false : bool
```

Bemerkung :
Falls in der Typdeklaration

```
datatype T = C of (T1 * ... * Tn)
```

der "neue" Name `T` schon in $T_1,...,T_n$ auftritt, so hat der neue Typ keine Normalform (keine definierten Werte).

Beispiel :

```
-  datatype nothing = Make of nothing;   (zirkuläre Definition)
>  datatype nothing = Make of nothing;
```

Mit dem Prinzip der Fallunterscheidung kann man derartige leere Datentypen vermeiden und Datentypen rekursiv definieren.

3.3 Datentypen mit Varianten

Hier betrachten wir den allgemeineren Fall von mehreren Konstruktoren für einen Typ.

Beispiel:

```
-   datatype currency = DM of real | ATS of real;
```

```
>   datatype currency = DM of real | ATS of real;
```

`ATS 3.0` und `DM 3.0` bezeichnen verschiedene Werte vom Typ `currency`.

```
–   ATS 3.0 = DM 3.0;
>   val it = false : bool
```

Die Normalformen von `currency` sind von der Form

`ATS` z_1 oder `DM` z_2

wobei z_1, z_2 Normalformen von Typ `real` sind.

Funktionen über Typen mit Variablen werden über Fallunterscheidung definiert.

```
–   fun   wechsle (DM x) = ATS (7.0 * x)
=     |   wechsle (ATS x) = DM (x / 7.0);
```

Die folgende Funktion `equal` liefert **genau dasselbe** Ergebnis wie die Gleichheitsoperation :

```
–   fun   equal (ATS x, ATS y)= (x = y)
=     |   equal (DM x, DM y)   = (x = y)
=     |   equal (ATS x, DM y)  = false
=     |   equal (DM x, ATS y)  = false;
```

Mit Varianten kann auch die disjunkte Vereinigung verschiedener Typen gebildet werden:

```
–   datatype mix = Set1 of int | Set2 of bool;
>   datatype mix = Set1 of int | Set2 of bool
```

Durch diese Datentypdeklaration werden der Typ `mix` und die Konstruktoren

```
Set1: int -> mix
Set2: bool -> mix
```

eingeführt. Es gilt

$$NF_{mix} = \{Set1\ z \mid z \in NF_{int}\} \cup \{Set2\ true, Set2\ false\}$$

Beachte, daß sich in der Typvereinigung `mix` die beitragende Menge eines Wertes (hier `int` oder `bool`) durch einen zusätzlichen Namen (tag) eindeutig festlegen läßt.

Die folgende Funktion liefert aus einer Liste von `mix`–Elementen die ganzen Zahlen:

```
–   fun intfilter []             = []
=     | intfilter (Set1 x :: xl) = x :: intfilter xl
      | intfilter (Set2 x :: xl) = intfilter xl;
>   val intfilter = fn : mix list –> int list
```

```
-   intfilter [Set1 3, Set2 true, Set1 10];
>   val it = [3,10] : int list
```

Exkurs: Terminierungsbeweise für Funktionsdeklarationen durch Pattern Matching

Sei f: $T_1 \rightarrow T_2$ eine Funktion, die durch Pattern Matching definiert ist:

```
fun f p1    = H1
  | f p2    = H2
      .
      .
      .
  | f pk    = Hk
```

wobei die Ausdrücke H_i (i = 1, ..., k) rekursive Aufrufe der Form $f(E_i)$ enthalten können.
Zum Terminierungsbeweis sucht man nach einer Funktion

h: $NF_{T1} \rightarrow NF_{int}$

mit folgenden Eigenschaften:

(1) Für alle rekursiven Aufrufe E_i in H_i, $i \in \{1, ..., k\}$, gilt

$h(p_i) > h(E_i)$

(2) $h(x) \geq 0$ für alle $x \in T_1$.

Beispiel:
Zum Nachweis der Terminierung von `intfilter` definieren wir folgende Funktion

$h: NF_{mix\ list} \rightarrow NF_{int}$

$h(x) =_{def} length(x)$

Dann gilt für alle `y: mix, xl: mix list`

(1) $h(y::xl) > h(xl)$ und

(2) $h(xl) \geq 0$

d.h. `intfilter xl` terminiert für alle `xl:mix list`.

Bemerkung:
Falls der rekursive Aufruf E_i in H_i unter einer Bedingung C_i erfolgt ($i \in \{1, ..., k\}$) genügt es anstelle von (1) die folgende Eigenschaft zu beweisen:

(1') $C_i \Rightarrow h(p_i) > h(E_i)$

endExkurs

In Typdeklarationen mit Varianten können auch Konstanten auftreten.

Beispiel: Lifting in ML.

```
-   datatype int_lift = Ok of int | Bottom;
```

Die Werte von `int_lift` werden durch die Konstante `Bottom` sowie die Ausdrücke `Ok(z)` mit einer `int`–Zahl z bezeichnet.

Allgemeine Form von Datentypen mit Varianten

Seien $T_1,...,T_n$ Typen ohne Typvariablen. Die Typdeklaration

`datatype T =` C_1 `of` T_1 `|` ... `|` C_n `of` T_n

(wobei die T_i auch Produkttypen sein können) definiert einen Typ `T` zusammen mit n Konstruktoren

$C_i : T_i \rightarrow T \qquad (0 < i \leq n)$.

Ist ein Argumenttyp `of` T_i nicht vorhanden, so ist C_i eine Konstante vom Typ T.

Im Fall einer **nicht-rekursiven Definition** kommt der Name T nicht in $T_1,...,T_n$ vor. Die Normalformen NF_T zur Darstellung der Werte von T sind durch die disjunkte Vereinigung der Normalformen der T_i gegeben.

$$NF_T =_{def} \bigcup_{1 \leq i \leq n} \{C_i(e) \mid e \in NF_{Ti}\}$$

Falls C_i eine Konstante ist, entfällt e. Jede Konstante C_i ist eine Normalform von `T`.

"Lifting":
Jeder Konstruktor C_i $(0 < i \leq n)$ definiert eine strikte Funktion

$$C^{\perp} : (NF_{Ti})^{\perp} \rightarrow (NF_T)$$

$$C^{\perp}(x_i) = \begin{cases} C_i(x_i) & \text{falls } x_i \neq \perp \\ \perp & \text{falls } x_i = \perp \end{cases}$$

Falls auf allen T_i die Gleichheit definiert ist, so ist die Gleichheit "=" auf NF_T definiert durch:

$$(\forall\, i,e,e' : (C_i(e) = C_i(e')) \Leftrightarrow (e = e')) \wedge (\forall\, i,j,e,e' : i \neq j \Rightarrow (C_i(e) \neq C_j(e'))$$

Beachten Sie, daß auf `real` und Funktionstypen keine Gleichheit definiert ist.

3.4 Polymorphe Datentypen

Lifting geht auch polymorph!

```
-   datatype 'a LIFT = Ok of 'a | Bottom;
-   type int_lift = int LIFT;
```

D.h. `datatype`–Deklarationen dürfen auch Typvariablen enthalten. Ein anderes Beispiel ist die (binäre) Typvereinigung.

```
–   datatype ('a,'b) MIX = Set1 of 'a | Set2 of 'b;
–   type mix = (int,bool) MIX;
```

Bei polymorphen Datentypen ist die Beschreibung der Normalformen abhängig von den aktuellen Typen, die für die Typvariablen eingesetzt werden.

4. Rekursive Datentypen

4.1 Allgemeiner Ansatz

Falls in einer Datentypdeklaration (mit Varianten) der deklarierte Typ T auch im Ausdruck auf der rechten Seite vorkommt, bezeichnen wir T als **rekursiven Datentyp**.

Beispiel 1: "Natürliche Zahlen"
Eine natürliche Zahl ist entweder `Zero` (das einzige Grundelement) oder der Nachfolger `Succ` (einstelliger Konstruktor) einer natürlichen Zahl, d.h. in ML:

```
–   datatype nat = Zero | Succ of nat;
```

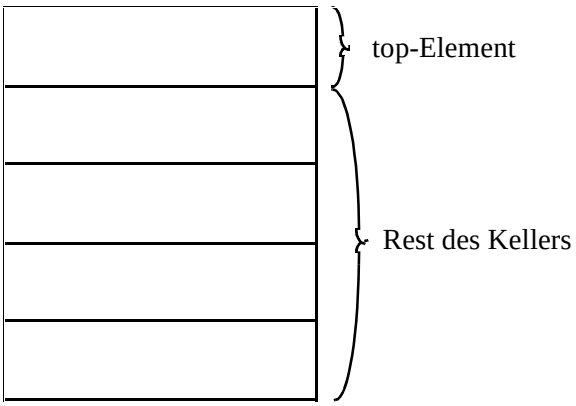
Die **Normalformen** NF_{nat} von `nat` sind induktiv definiert :

1) `Zero` ist in NF_{nat}
2) Falls n $\in$ NF_{nat}, so auch `Succ` n $\in$ NF_{nat},
   d.h. NF_{nat} = {`Zero`, `Succ Zero`, `Succ (Succ Zero)`, ...}.

Beachte: `nat` ist ein neuer ML Typ, über dem noch keine Addition, Multiplikation, etc. definiert ist. Funktionen über `nat` werden durch Fallunterscheidung definiert:

```
–   fun plus (n,Zero)   = n
=     | plus (n,Succ k) = Succ (plus (n,k));
```

Beispiel 2: "Keller" (stack, "Stapel")
Kellerstrukturen sind wichtige Datenstrukturen bei der Implementierung von Programmiersprachen. Man kann sie sich als vertikale Listen vorstellen:



Operationen auf Kellern:
- Leerer Keller: `EmptyStack` (Konstruktor)
- Aufsetzen eines Elementes: `Push` (Konstruktor)
- Wegnehmen eines Elementes: `pop` (Destruktor)
- oberstes Element: `top` (Selektor)
- Frage nach leerem Keller: `isempty` (Selektor)

```
-   datatype 'a STACK = EmptyStack | Push of 'a * 'a STACK
-   exception pop_error and top_error;
-   fun pop EmptyStack    = raise pop_error
=     | pop (Push (x,s))  = s;
-   fun top EmptyStack    = raise top_error
=     | top (Push (x,s))  = x;
-   fun isempty EmptyStack   = true
      | isempty (Push (x,s)) = false;
```

Jede Instanz T STACK (für einen Typ T) hat folgende Normalformen:

1) `EmptyStack` $\in NF_{STACK}$
2) Ist $n \in NF_T$ und $s \in NF_{STACK}$ (d.h. $(n,s) \in NF_{T*STACK}$)
   so ist `Push (n,s)` $\in NF_{STACK}$.

Semantik rekursiver Typendeklarationen:

Eine Typdeklaration

`datatype` T = d_1 | ... | d_k | C_1 `of` T_1 | ... | C_n `of` T_n

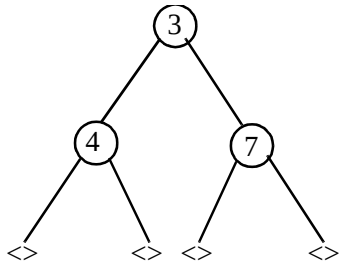
definiert induktiv folgende Menge NT_F von Normalformen:

1) $d_1, ..., d_k \in NF_T$
2) $(\forall\, i : 0 \le i \le n \wedge e_i \in NF_{Ti} \Rightarrow C_i(e_i) \in NF_{Ti})$

Bemerkung:
Bei rekursiven Typdeklarationen enthält mindestens eines der Konstruktorargumente T_i den Typ T.

4.2 Baumstrukturen [Broy, Kap.8.2]

4.2.1) Binärbäume mit Knotenmarkierungen



<> bezeichnet den leeren Baum

```
–   datatype ´a BTREE = Notree
                      | Node of ´a BTREE * ´a * ´a BTREE;
```

Beispiel : Der oben gezeichnete Baum ist wie folgt repräsentiert:

```
–   Node (Node (Notree,4,Notree),
=         3,
=         Node (Notree,7,Notree));
```

Selektor–Funktionen auf Bäumen:

a) **linker Sohn**

```
–   exception left_error;
–   fun left Notree             = raise left_error
=     | left (Node (l,x,r))   = l;
```

b) **rechter Sohn**

```
–   exception right_error;
–   fun right Notree            = raise right_error
=     | right (Node (l,x,r))  = r;
```

c) **Wurzel**

```
–   exception label_error;
–   fun label Notree            = raise label_error
=     | label (Node (l,x,r))  = x;
```

d) **Prüfen, ob ein Baum leer ist**

```
–   fun isempty Notree              = true
=     | isempty (Node (l,x,r))  = false;
```

e) **Maximale Tiefe eines Baumes**

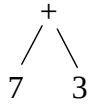
```
-   fun depth Notree            = 0
=     | depth (Node (l,x,r))  = 1 + max (depth l,depth r);
```

f) **Anzahl der Knoten eines Baumes**

```
-   fun nrnode Notree            = 0
=     | nrnode (Node (l,x,r))  = nrnode l + 1 + nrnode r;
```

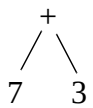
Baumdurchlauf (Linearisierung von Bäumen):
Drei verschiedene typische Arten Bäume zu durchlaufen entsprechen den Möglichkeiten binäre Ausdrücke zu notieren:

a) Infix



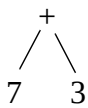
7+3

b) Präfix



+73

c) Postfix



73+

Notation:

- [L] durchlaufe zuerst den linken Teilbaum,
- [K] nimm den Knoten,
- [R] durchlaufe den rechten Teilbaum.

a) Inordnung (inorder):[L]; [K]; [R]

```
–   fun inorder Notree        = nil
=     | inorder (Node(l,x,r)) = inorder l @ [x] @ inorder r;
```

b) Präordnung (preorder): [K]; [L]; [R]

```
–   fun preorder Notree         = nil
=     | preorder (Node(l,x,r)) =
=       x :: preorder l @ preorder r;
```

c) Postordnung (postorder):[L]; [R]; [K]

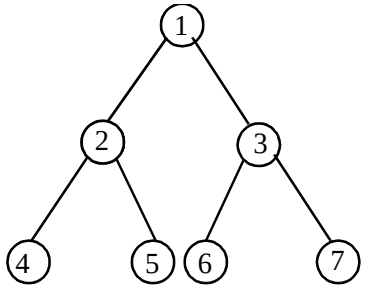
```
–   fun postorder Notree          = nil
=     | postorder (Node(l,x,r)) =
```

```
=       postorder l @ postorder r @ [x];
```

d) Breitenordnung (breadth–first):
Diese Ordnung listet höhere vor tieferen Knoten und auf der selben Ebene linke vor rechten Knoten. Sie läßt sich nicht mit der Notation [L], [K], [R] beschreiben.

```
–   fun breadthfirst tree =
=     let fun doForest []                    = []
=           | doForest (Notree :: t)         = doForest t
=           | doForest (Node (l,x,r) :: t)   =
=                           x :: doForest (t @ [l,r])
=         in doForest [tree]
=     end;
```

Beispiel :

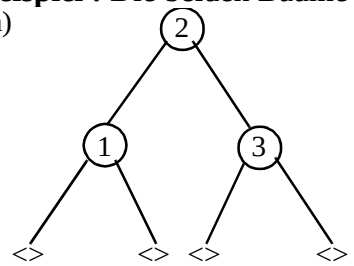


Inordung:	[4,2,5,1,6,3,7]
Präordnung:	[1,2,4,5,3,6,7]
Postordnung:	[4,5,2,6,7,3,1]
Breitenordnung:	[1,2,3,4,5,6,7]

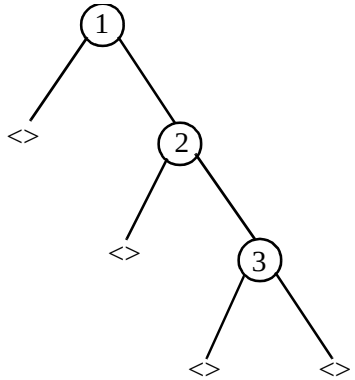
Beachte: Im Falle der Prä- und Postordnung läßt sich der Baum eindeutig aus der erzeugten Sequenz wieder generieren (sofern die Anzahl der Söhne bekannt ist).
Im Falle der Inordnung gilt das nicht.

Beispiel : Die beiden Bäume

(a)



(b)



erzeugen die gleiche Inordnung [1,2,3]
(Präordnung: (a) [2,1,3], (b) [1,2,3]).

Allgemein kann man eine Baumoperation formulieren, die das folgende Schema benutzt:

```
–   fun btreeop c f Notree         = c
=     | btreeop c f (Node (l,x,r)) =
=                   f (btreeop c f l, x, btreeop c f r);
```

Die Konstante `c` steht dabei für das Bild des leeren Baumes unter `f` und die Operation `f` wird rekursiv auf die Wurzel sowie auf die Ergebnisse der beiden Teilbäume angewendet. Dann gilt:

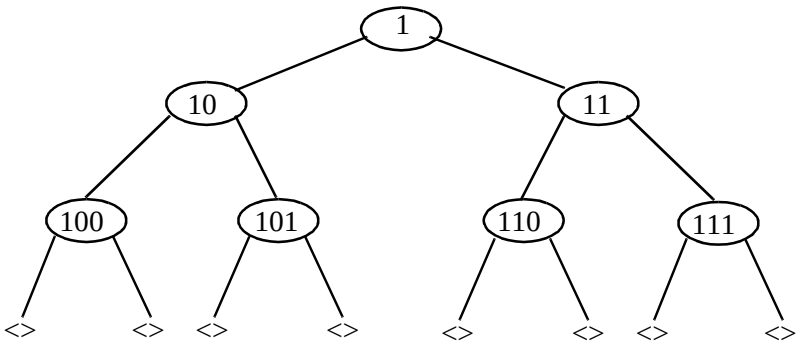
```
depth  x   = btreeop 0  (fn (a,b,c) => 1 + max(a,c))   x;
nrnode x   = btreeop 0  (fn (a,b,c) => a + 1 + c)      x;
preorder  x = btreeop [] (fn (a,b,c) => b :: a @ c)  x;
inorder   x = btreeop [] (fn (a,b,c) => a @ [b] @ c) x;
postorder x = btreeop [] (fn (a,b,c) => a @ c @ [b]) x;
```

Der Durchlauf `breadth–first` läßt sich nicht mit `btreeop` ausdrücken.

Die Knoten binärer Bäume können systematisch mit Binärsequenzen folgendermaßen markiert werden:

(1) Die Wurzel hat die Bezeichnung 1.
(2) Hat ein Knoten die Bezeichnung x, so hat sein linker Sohn die Bezeichnung x0 und sein rechter Sohn die Bezeichnung x1.

Beispiel :



Präordnung: [1,10,100,101,11,110,111]
Inordnung : [100,10,101,1,110,11,111]
Postordnung: [100,101,10,110,111,11,1]
Breitenordnung: [1,10,11,100,101,110,111]

Die folgende Funktion `gen` generiert für $n \geq 0$ den vollständigen Baum der Tiefe n mit obigen Markierungen:

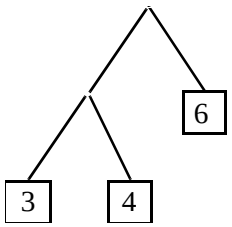
```
–   fun h (n,s) =
              if n=0 then Notree
=             else Node (h (n–1,s@[0]), s, h (n–1,s@[1]));
–   fun gen n = h (n,[1]);
```

4.2.2 Beblätterte Binärbäume

Beblätterte Binärbäume haben keine Markierungen an den Verzweigungen, sondern nur an den Blättern.

```
–     datatype ´a BBTREE =
=             BBLeaf of ´a
=           | BBNode of ´a BBTREE * ´a BBTREE;
```

Beispiel :




```
–   val abbtree =
=         BBNode (BBNode (BBLeaf 3, BBLeaf 4), BBLeaf 6);
```

Die Grunddatenstruktur der S-Ausdrücke (s–expressions) der Sprache LISP besteht aus beblätterten binären Bäumen über sogenannten Atomen.

```
–   type sexp = atom BBTREE;
```

wobei `atom` alle LISP–Grunddatenstrukturen (`bool`, `int`, `char`, etc.) umfaßt. LISP ist eine typenlose Sprache; deshalb werden die Unterscheidungen zwischen Atomen und S-Ausdrücken zur Laufzeit getroffen. Die Selektorfunktionen im Falle von LISP heißen:

`car` (steht für: contents of address register) für den linken Teilbaum,
`cdr` (steht für: contents of data register) für den rechten Teilbaum

Bemerkung: Die Namensgebung ist geschichtlich bedingt; dies waren Register, die bei der ersten Implementierung auf der IBM 704 die Referenzen der entsprechenden Teilbäume enthielten. Die Namen wurden beibehalten, da sich verschachtelte Ausdrücke gut abkürzen lassen, z.B. `(car (cdr (car x)))` zu `(cadar x)`.

Listen können als spezielle beblätterte Binärbäume verstanden werden.

Definition: (Legale Liste, proper list)

Eine **legale Liste (proper list)** ist

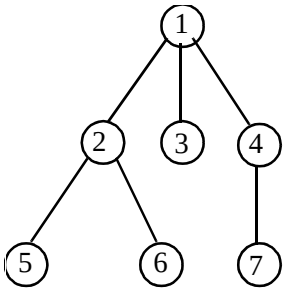
(1) der knotenlose Baum, der nur ein Blatt mit der leeren Liste enthält,
(2) ein Baum, in dem jeder linke Teilbaum aus einem Listeneintrag besteht, und jeder rechte Teilbaum aus einer legalen Liste.

```
–   datatype 'a LISTENTRY = Nolist | Entry of 'a
–   fun properlist (BBLeaf Nolist) = true
=     | properlist (BBNode (BBLeaf (Entry x), y)) =
=                                       properlist y
=     | properlist _ = false;
```

In diesem Modell, das in der Sprache LISP verwendet wird, entspricht der Konstruktor `BBNode` der Listenoperation `::`; in LISP heißt sie `cons`. Die beiden Listenselektoren (in ML: `hd` und `tl`) heißen in LISP `car` und `cdr`. Wegen der Typenlosigkeit entfällt in LISP die Notwendigkeit der Struktur `LISTENTRY`.

4.2.3 n-äre Bäume

Beispiel:




```
–   datatype ´a NTREE = NNode of ´a * ´a NTREE list;
–   type ´a FOREST = ´a NTREE list;
```

n-äre Bäume werden durch ihre Wurzel und eine Liste von Bäumen (einen "Wald") beschrieben. (Der leere Baum gehört nicht dazu.)

4.3 Suchen in Binärbäumen

Es gibt zwei Methoden, einen allgemeinen Binärbaum (mit Knotenmarkierungen) nach einem Element, das eine Eigenschaft `p` erfüllt, zu durchsuchen. Um bei der Suche Erfolg von Mißerfolg unterscheiden zu können, benötigen wir einen weiteren Datentyp:

```
–   datatype ´a SEARCH = NotFound | Found of ´a;
```

a) Breitensuche (breadth–first search):

```
–     fun breadthsearch p tree =
=         let fun searchForest []                  = NotFound
=               |  searchForest (Notree :: t)      =
=                                searchForest t
=               |  searchForest (Node (l,x,r) :: t)=
=                    if p x  then Found x
=                    else searchForest (t @ [l,r])
=         in searchForest [tree]
=   end;
```

b) Tiefensuche (depth–first search):

```
–   fun depthsearch p  Notree       = NotFound
=    |  depthsearch p (Node (l,x,r))=
=          if p x then Found x
=          else let val result = depthsearch p l
=               in
=                 if result = NotFound then depthsearchp p r
=                 else result
                  end;
```

Eine Breitensuche findet garantiert ein geeignetes Element (falls eines existiert), eine Tiefensuche nur, wenn der Baum keine unendlichen Zweige hat. Trotzdem wird aus Effizienzgründen oft die Tiefensuche vorgezogen, wenn die Lösungen relativ tief im Geflecht des Baumes liegen.

Bei beiden Suchmethoden muß man schlimmstenfalls alle Knoten einmal durchlaufen um einen geeigneten Kandidaten zu finden (lineare Komplexität). In Spezialfällen läßt sich die Suche beschleunigen. Dazu muß der Baum aber eine bestimmte Struktur haben.

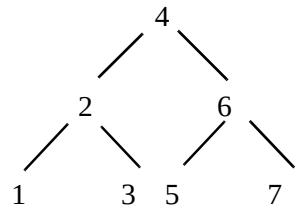
Definition: (Binärer Suchbaum)

Ein **binärer Suchbaum (binary search tree)** ist ein knotenmarkierter Binärbaum, dessen Markierungen eine total geordnete Menge bilden, und in dem jeder Knoten x die folgenden Eigenschaften hat:

- Alle Knoten im linken Teilbaum von x haben kleinere Markierungen als x.
- Alle Knoten im rechten Teilbaum von x haben größere Markierungen als x.

Beachte: Das bedeutet, daß kein Element mehrmals im Baum vorkommen darf.

Beispiel:



In der vorgegebenen ML Repräsentation eines binären Suchbaums treten die Markierungen in aufsteigender Ordnung auf:

```
–   val bintree = Node (Node (Node (Notree,1,Notree),
=                             2,
=                             Node (Notree,3,Notree)),
=                       4,
=                       Node (Node (Notree,5,Notree)
=                             6,
=                             Node (Notree,7,Notree)));
```

Der binäre Suchalgorithmus ist durch die folgende Funktion definiert:

```
–   fun binarysearch less item =
=     let fun lookAt Notree         = NotFound
=           | lookAt (Node (l,x,r)) =
=                 if less (x,item) then lookAt r
=                  else if less (item,x)then lookAt l
=                                       else Found x
=      in lookAt
=     end;
```

Beispiel:
Für den obigen Beispielbaum wird eine Suche nach Element `6` wie folgt spezifiziert:

```
–   binarysearch (op <) 6 bintree;
>   Found 6: int SEARCH
```

Für eine Suche gemäß der Ordnung `>` statt `<` müßte der Baum seitenverkehrt sortiert sein:

```
–   binarysearch (op >) 6 bintree;
>   Notfound: int SEARCH
```

Die Länge einer Binärsuche beschränkt sich schlimmstenfalls auf die Tiefe des Baumes, nicht auf die (im allgemeinen größere) Anzahl der Knoten. Wenn der Baum"balanciert" ist, d.h. eine minimale Tiefe hat, bedeutet das eine logarithmische Komplexität. Dafür

bezahlt man mit einem größeren Aufwand bei dem Einfügen (beim Sicherstellen, daß der Baum eine minimale Tiefe hat) und bei der Herausnahme. Binäre Suchbäume sind besonders für die Repräsentation von Datenmengen geeignet, die oft benutzt aber selten modifiziert werden, wie Wörterbücher und Telefonbücher – beides Mengen von Strings in der lexikographischen Ordnung (wobei ein String als eine Liste von Zeichen aufgefaßt wird).

Definition: Lexikographische Ordnung (siehe Übungen)

Sei M eine Menge von homogenen Listen, d.h. Listen, deren Elemente alle von derselben Menge A stammen. Sei < eine totale Ordnung auf der Menge A. Dann ist die **lexikographische Ordnung** << auf M wie folgt definiert:

$(\forall\ m : m \in M : nil << m) \wedge (\forall\ m1,m2 : m1,m2 \in M : m1 << m2 \Leftrightarrow$
$hd(m1) < hd(m2) \vee (hd(m1) = hd(m2) \wedge tl(m1) << tl(m2)))$

In ML kann die lexikographische Ordnung wie folgt definiert werden:

```
-   fun lex less (   []    ,   []    )  = false
=     | lex less ((x :: xl),   []    )  = false
=     | lex less (   []    ,(y :: yl))  = true
=     | lex less ((x :: xl),(y :: yl)) = less(x,y)
=         orelse (x=y andalso lex less (xl,yl));

-   infix 4 <<;       (stellt einen Infixoperator der Präzedenz 4 bereit)

-   fun (xl << yl) = lex (op <) ((xl:int list),
=                                (yl:int list));
          (definiert << als die lexikographische Ordnung auf Integerlisten)
```

Für Strings wird die lexikographische Ordnung in ML mit dem Zeichen < bereitgestellt.