Kapitel 6

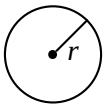
Konstruktortypen und Ausnahmen

Wir lernen jetzt, wie man mithilfe von Konstruktoren neue Typen deklariert. Mit Konstruktortypen lassen sich Mengen von geordneten Bäumen beschreiben. Außerdem lernen wir, wie man neue Ausnahmen deklariert und geworfene Ausnahmen fängt.

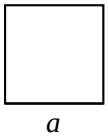
Mit Konstruktortypen können wir eine Vielzahl von interessanten Programmieraufgaben angehen. Wir können jetzt Ausdrücke darstellen und beispielsweise eine Prozedur schreiben, die Ausdrücke symbolisch differenziert.

6.1 Varianten und Konstruktoren

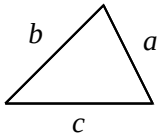
Wir wollen drei Arten von geometrischen Objekten als Standard ML Werte darstellen:



Kreis



Quadrat



Dreieck

Dies tun wir durch Paare, deren erste Komponente sagt, um welche Art von Objekt es sich handelt:

- Einen Kreis mit dem Radius $r \in \mathbb{R}^+$ repräsentieren wir durch das Paar $\langle 1,\ r\rangle$.
- Ein Quadrat mit der Seitenlänge $a \in \mathbb{R}^+$ repräsentieren wir durch das Paar $\langle 2,\ a\rangle$.

- Ein Dreieck mit den Seitenlängen $a, b, c \in \mathbb{R}^+$ repräsentieren wir durch das Paar $\langle 3,\ \langle a,\ b,\ c\rangle\rangle$.

Die erste Komponente der Paare bezeichnen wir als *Variantennummer*. Die Variantennummer eine Paares legt fest, um welche der drei betrachteten *Varianten* von geometrischen Objekten es sich handelt.[1]

Als Nächstes wollen wir eine Prozedur

```
area : shape -> real
```

schreiben, die den Flächeninhalt eines geometrischen Objektes bestimmt. Den dafür erforderlichen Typ `shape` können wir mathematisch durch die Summe

$$\mathbb{R}^+ \uplus \mathbb{R}^+ \uplus \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+$$

beschreiben. In Standard ML können wir einen entsprechenden Typ wie folgt deklarieren:[2]

```
datatype shape =
    Circle   of real
  | Square   of real
  | Triangle of real * real * real
```

Diese Typdeklaration führt für jede Variante einen sogenannten *Konstruktor* ein, mit dem die Werte der Variante *konstruiert* werden können. Die Konstruktoren können wie Prozeduren benutzt werden und haben die folgenden Typen:

```
con Circle   : real -> shape
con Square   : real -> shape
con Triangle : real * real * real -> shape
```

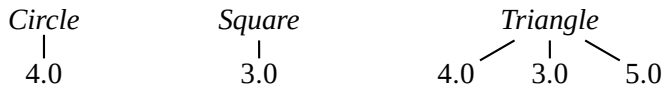
Hier sind einige Experimente mit dem Interpreter:

```
Circle 4.0
Circle 4.0 : shape

Square 3.0
Square 3.0 : shape

Triangle(4.0, 3.0, 5.0)
Triangle(4.0, 3.0, 5.0) : shape
```

Der Interpreter stellt die mit den Konstruktoren konstruierten Werte der besseren Lesbarkeit wegen nicht als Paare mit Variantennummern, sondern als Anwendungen des jeweiligen Konstruktors dar. Wir werden bei der grafischen Darstellung der konstruierten Werte entsprechend verfahren:

[1] Im Englischen bezeichnet man Variantennummern als „tags“.

[2] Da Standard ML (wie andere Programmiersprachen auch) keinen Typ für positive reelle Zahlen hat, approximieren wir $\mathbb{R}^+$ durch `real`.



Typen, die wie `shape` mit dem Schlüsselwort `datatype` deklariert wurden, heißen *Konstruktortypen*.

Eine Prozedur, die den Flächeninhalt eines geometrischen Objekts bestimmt, können wir jetzt wie folgt beschreiben:

```
fun area (Circle r)        = Math.pi*r*r
  | area (Square a)        = a*a
  | area (Triangle(a,b,c)) = let val s = (a+b+c)/2.0
                             in Math.sqrt(s*(s-a)*(s-b)*(s-c))
                             end
val area : shape -> real

area (Square 3.0)
9.0 : real

area (Triangle(6.0, 6.0, Math.sqrt 72.0))
18.0 : real
```

Die Prozedur `area` ist mit drei Regeln definiert, die eine Fallunterscheidung gemäß der drei Varianten von `shape` beschreiben. Die jeweilige Variante wird durch die Anwendung des entsprechenden Konstruktors im Muster der Regel angezeigt.

Wir können auch eine Prozedur schreiben, die die Variantennummer eines geometrischen Objekts liefert:

```
fun variant (Circle _)   = 1
  | variant (Square _)   = 2
  | variant (Triangle _) = 3

val variant : shape -> int

variant (Triangle(3.3, 6.6, 7.7))
3 : int
```

Konvention für die Groß- und Kleinschreibung von Bezeichnern

Für Konstruktoren verwenden wir stets Bezeichner, die mit einem Großbuchstaben beginnen (z.B. `Monday` und `True`). Dagegen verwenden wir für Typen und Werte stets Bezeichner, die mit einem Kleinbuchstaben beginnen. Diese Konvention verbessert die Lesbarkeit von Programmen.

6.2 Enumerationstypen

Die Deklaration

```
datatype day = Monday | Tuesday | Wednesday
             | Thursday | Friday | Saturday | Sunday
```

führt einen Typ `day` ein, dessen Werte

$\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 3 \rangle$, $\langle 4 \rangle$, $\langle 5 \rangle$, $\langle 6 \rangle$, $\langle 7 \rangle$

die verschiedenen Wochentage repräsentieren. Die Werte des Typs werden durch *nullstellige Konstruktoren* dargestellt:

```
con Monday    : day
con Tuesday   : day
con Wednesday : day
con Thursday  : day
con Friday    : day
con Saturday  : day
con Sunday    : day
```

Hier sind Beispiele für die Verwendung von `day`:

```
fun weekend Saturday = true
  | weekend Sunday   = true
  | weekend _        = false

val weekend : day -> bool

weekend Saturday
true : bool

map weekend [Monday, Wednesday, Friday, Saturday, Sunday]
[false, false, false, true, true] : bool list
```

Typen, die wie `day` nur mithilfe von nullstelligen Konstruktoren definiert sind, werden als *Enumerationstypen* bezeichnet.

`order` Der vordefinierte Enumerationstyp

```
datatype order = LESS | EQUAL | GREATER
```

wird von einigen Standardstrukturen verwendet. Hier sind Beispiele:

```
Int.compare(2,4)
LESS : order

Real.compare(4.3,4.2)
GREATER : order
```

`Listsort` Mithilfe der Standardstruktur `Listsort` stellt Moscow ML eine polymorphe Sortierprozedur für Listen zur Verfügung:

```
Listsort.sort
fn : ('a * 'a -> order) -> 'a list -> 'a list

Listsort.sort Int.compare [4, 4, 4, 3, 2, 1]
[1, 2, 3, 4, 4, 4] : int list
```

6.3 Typsynonyme

Wenn wir Punkte in der Ebene durch Koordinatenpaare darstellen, kann es zweckmäßig sein, ein *Typsynonym*

```
type point = real * real
```

für den Typ der Koordinatenpaare einzuführen. Im Gegensatz zu einer Typdeklaration mit `datatype` führt eine Typdeklaration mit `type` keinen neuen Typ ein, sondern lediglich eine alternative Schreibweise für einen bereits existierenden Typ. Typsynonyme können beispielsweise in Typdeklarationen verwendet werden:

```
datatype object = Circle   of point * real
                | Triangle of point * point * point
```

Der Interpreter verzichtet bei der Ausgabe auf die Verwendung von Typsynonymen:

```
(2.0,3.0) : point
(2.0, 3.0) : real * real
```

Wenn wir wollen, können wir auch einen neuen Typ für Punkte einführen:

```
datatype point = P of real * real
```

Eine Prozedur, die den Abstand eines Punktes vom Ursprung berechnet, können wir damit wie folgt schreiben:

```
fun distance (P(x,y)) = Math.sqrt(x*x + y*y)

val distance : point -> real
```

6.4 Case-Ausdrücke und abgeleitete Formen

Im Zusammenhang mit Konstruktortypen ist es manchmal bequem, eine Fallunterscheidung mithilfe eines Case-Ausdrucks zu formulieren:

```
fun sign x =
    case Int.compare(x,0) of
        LESS => ~1
      | EQUAL => 0
      | GREATER => 1

val sign : int -> int

sign ~7
~1 : int
```

```
sign 55
1 : int
```

Die allgemeine Form von Case-Ausdrücken ist

`case` a `of` M_1 `=>` a_1 `|` $\cdots$ `|` M_n `=>` a_n

Die Standard ML Sprachdefinition führt Case-Ausdrücke auf Ausdrücke der Form

`(fn` M_1 `=>` a_1 `|` $\cdots$ `|` M_n `=>` a_n`)` a

zurück. Man sagt, dass Case-Ausdrücke eine *abgeleitete Form* sind. Bei abgeleiteten Formen handelt es sich um alternative Schreibweisen, die auf grundlegendere Schreibweisen zurückgeführt werden können. Abgeleitete Formen werden auch als syntaktischer Zucker bezeichnet.

In Kapitel 3 haben wir gelernt, dass Prozedurdeklarationen auf rekursive Wertdeklarationen zurückgeführt werden.

Auch Konditionale sind aus Sicht der Standard ML Sprachdefinition eine abgeleitete Form:

`if` a_1 `then` a_2 `else` a_3 $\Longrightarrow$ `(fn true =>` a_2 `| false =>` a_3`)` a_1

6.5 Ein Typkonstruktor für Binärbäume

Die rekursive Typdeklaration

```
datatype 'a tree =
    L of 'a
  | N of 'a * 'a tree * 'a tree
```

definiert eine Repräsentation für nichtleere geordnete Binärbäume. Mit dem polymorphen Konstruktor

```
con L : 'a -> 'a tree
```

lassen sich alle Binärbäume beschreiben, die aus nur einem Knoten bestehen. Mit dem polymorphen Konstruktor

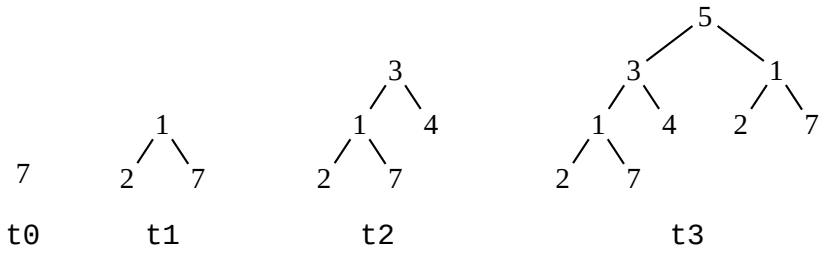
```
con N : 'a * 'a tree * 'a tree -> 'a tree
```

können zwei Binärbäume zu einem größeren Binärbaum zusammengesetzt werden. Für jeden Typ t beschreibt der Typ

t `tree`

also alle nichtleeren geordneten Binärbäume, deren Marken Werte von t sind.

Die Binärbäume



lassen sich mit den Konstruktoren `L` und `N` wie folgt beschreiben:

```
val t0 = L 7
val t1 = N(1, L 2, t0)
val t2 = N(3, t1, L 4)
val t3 = N(5, t2, t1)
```

Die Typdeklaration für nichtleere geordnete Bäume

```
datatype 'a tree =
    L of 'a
  | N of 'a * 'a tree * 'a tree
```

ist rekursiv und mit einem Typargument `'a` *parametrisiert*. Sie bindet den Bezeichner `tree` an einen *Typkonstruktor*, der zu einem Typ *t* einen neuen Typ *t* `tree` liefert. Die Werte des Typs *t* `tree` können mithilfe der polymorphen *Wertkonstruktoren* `L` und `N` konstruiert werden.

6.6 Prozeduren für Binärbäume

Die Prozedur *Größe*

```
fun size (L _)         = 1
  | size (N(_, t, t')) = 1 + size t + size t'

val size : 'a tree -> int
```

bestimmt die Größe von Binärbäumen. Die Prozedur *Tiefe*

```
fun depth (L _)         = 0
  | depth (N(_, t, t')) = 1 + Int.max(depth t, depth t')

val depth : 'a tree -> int
```

liefert die Tiefe von Binärbäumen.

Die *Grenze* eines geordneten Baums ist die Liste der Markierungen der Blätter in der Reihenfolge ihres Auftreten. Die Prozedur *Grenze*

```
fun frontier (L x)         = [x]
  | frontier (N(_, t, t')) = frontier t @ frontier t'

val frontier : 'a tree -> 'a list
```

berechnet die Grenze von Binärbäumen:

```
frontier t3
[2, 7, 4, 2, 7] : int list
```

Eine Liste $xs \in \mathcal{L}(X)$ heißt *Projektion* eines geordneten Baums $b \in \mathcal{B}(X)$ genau dann, wenn xs die Markierungen der Knoten von b mit der richtigen Anzahl von Mehrfachauftreten enthält. Zwei Projektionen desselben Baums unterscheiden sich nur in der Reihenfolge ihrer Elemente. Die Länge der Projektionen eines Baums ist gleich der Größe des Baums.

Präfixprojektion

Für Binärbäume definieren wir drei Projektionen. Die *Präfixprojektion*

```
fun prefixp (L x)       = [x]
  | prefixp (N(x,t,t')) = [x] @ prefixp t @ prefixp t'

val prefixp : 'a tree -> 'a list
```

Infixprojektion

setzt die Marke der Wurzel vor die Marken des linken Unterbaums gefolgt von den Marken des rechten Unterbaums. Die *Infixprojektion*

```
fun infixp (L x)       = [x]
  | infixp (N(x,t,t')) = infixp t @ [x] @ infixp t'

val infixp : 'a tree -> 'a list
```

Postfixprojektion

setzt die Marken des linken Unterbaums vor die Marke der Wurzel gefolgt von den Marken des rechten Unterbaums. Die *Postfixprojektion*

```
fun postfixp (L x)       = [x]
  | postfixp (N(x,t,t')) = postfixp t @ postfixp t' @ [x]

val postfixp : 'a tree -> 'a list
```

setzt die Marken des linken Unterbaums vor die Marken des rechten Unterbaums gefolgt von der Marke der Wurzel. Hier sind Beispiele:

```
prefixp t3
[5, 3, 1, 2, 7, 4, 1, 2, 7] : int list

infixp t3
[2, 1, 7, 3, 4, 5, 2, 1, 7] : int list

postfixp t3
[2, 7, 1, 4, 3, 2, 7, 1, 5] : int list
```

Die Knoten eines geordneten Binärbaums sind Listen über $\{1, 2\}$ (siehe Abschnitt 5.5). Die Prozedur

Teilbäume

```
fun subtree t            nil    = t
  | subtree (N(_, t, _)) (1::u) = subtree t u
  | subtree (N(_, _, t)) (2::u) = subtree t u
  | subtree _            _      = raise Subscript

val subtree : 'a tree -> int list -> 'a tree
```

liefert zu einem Binärbaum und einem Knoten den von diesem Knoten aus erreichbaren Teilbaum:

```
subtree (N(3, N(1, L 2, L 7), L 4)) [1]
N(1, L 2, L 7) : int tree

subtree (N(3, N(1, L 2, L 7), L 4)) [1,2]
L 7 : int tree

subtree (N(3, N(1, L 2, L 7), L 4)) [2]
L 4 : int tree

subtree (N(3, N(1, L 2, L 7), L 4)) [2,1]
! Uncaught exception: Subscript
```

Die Ausnahme `Subscript` ist vordefiniert und wird zur Signalisierung von Grenzüberschreitungen verwendet.

Die Prozedur

Knoten

```
fun domain (L _)       = [nil]
  | domain (N(_,t,t')) = [nil] @
                         map (fn u => 1::u) (domain t) @
                         map (fn u => 2::u) (domain t')

val domain : 'a tree -> int list list
```

liefert zu einem Binärbaums eine Liste, die alle Knoten des Baums in Präfixordnung enthält (wie bei der Präfixprojektion).

```
domain (N(3, N(1, L 2, L 7), L 4))
[[], [1], [1, 1], [1, 2], [2]] : int list list
```

Die Prozedur

Test auf Balanciertheit

```
fun balanced (L _)         = true
  | balanced (N(_, t, t')) = balanced t andalso
                             balanced t' andalso
                             depth t = depth t'

val balanced : 'a tree -> bool
```

testet, ob ein Binärbaum balanciert ist. In Abschnitt 6.8 werden wir eine Prozedur angeben, die Balanciertheit von Binärbäumen schneller testet.

Erzeugung von balancierten Binärbäumen

Die Prozedur

```
fun tree n = if n<1 then L 0
             else let val t = tree(n-1) in N(n,t,t) end

val tree : int -> int tree
```

liefert zu $n \in \mathbb{N}$ einen balancierten Binärbaum der Tiefe n:

```
tree 2
N(2, N(1, L 0, L 0), N(1, L 0, L 0)) : int tree
```

6.7 Ein Typkonstruktor für Terme

Die Typdeklaration

```
datatype 'a term = T of 'a * 'a term list
```

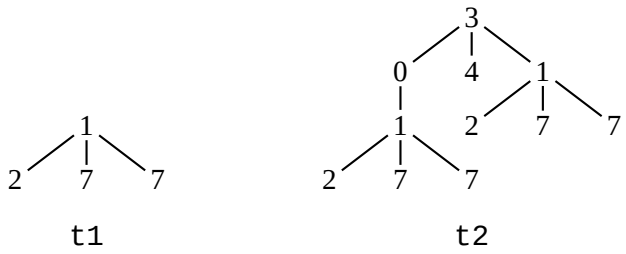
definiert eine Repräsentation für Terme. Diese besteht aus einem Typkonstruktor

```
datatype 'a term
```

und einem polymorphen Wertkonstruktor

```
con T : 'a * 'a term list -> 'a term
```

Damit lassen sich die Terme



wie folgt beschreiben:

```
val t1 = T(1, [T(2,[]), T(7,[]), T(7,[])])
val t2 = T(3, [T(0,[t1]), T(4,[]), t1])
```

Die Prozeduren

```
fun size (T(_, ts)) = foldl op+ 1 (map size ts)
val size : 'a term -> int

fun depth (T(_, ts)) = 1 + foldl Int.max ~1 (map depth ts)
val depth : 'a term -> int
```

bestimmen die Größe und die Tiefe von Termen.

6.8 Ausnahmen

Ausnahmen sind Werte

Ausnahmen sind Werte des Typs `exn`. Wir haben bereits drei Ausnahmen kennengelernt:

```
Overflow  : exn
Empty     : exn
Subscript : exn
```

Der Typ `exn` ist als ein Konstruktortyp zu verstehen. Die obigen Ausnahmen sind also nullstellige Konstruktoren für `exn`. Der Konstruktortyp `exn` hat die Besonderheit, dass man nach Belieben neue Konstruktoren hinzufügen kann. Wir sagen auch, dass `exn` im Gegensatz zu den mit `datatype` deklarierten *geschlossenen Konstruktortypen* ein *offener Konstruktortyp* ist. Neue Konstruktoren für `exn` können mit speziellen *Ausnahmedeklarationen* eingeführt werden. Hier sind Beispiele:

Deklarieren von Ausnahmen

```
exception New
exn New = New : exn

exception Newer of int
exn Newer : int -> exn

(Overflow, New, Newer)
(Overflow, New, fn) : exn * exn * (int -> exn)

fun test New       = 0
  | test (Newer x) = x
  | test _         = ~1

val test : exn -> int

test Overflow
~1 : int

test (Newer 13)
13 : int
```

Werfen von Ausnahmen

Mit Ausdrücken der Form

```
raise a
```

können Ausnahmen *geworfen* werden. Dabei muss der Ausdruck *a* den Typ `exn` haben. Raise-Ausdrücke sind polymorph und können jeden Typ annehmen. Hier sind Beispiele:

```
raise Newer 6
!Uncaught exception: Newer 6
```

```
fn x => if x>0 then x else raise Newer 6
fn : int -> int
```

Wir sagen, dass die Auswertung eines Ausdrucks *regulär terminiert*, wenn sie mit einem Ergebnis terminiert (also nicht mit einer geworfenen Ausnahme).

strikte links-nach-rechts Auswertung

Durch das Werfen von Ausnahmen kann man feststellen, in welcher Reihenfolge Teilausdrücke ausgewertet werden. Standard ML schreibt eine strikte links-nach-rechts Auswertung vor. Beispielsweise wirft der Ausdruck

```
(raise Overflow, raise Subscript, 2)
```

stets die Ausnahme `Overflow`.[3]

Fangen von Ausnahmen

Geworfene Ausnahmen können mithilfe von Handle-Ausdrücken *gefangen* werden:

```
(raise New) handle New => ()
() : unit

(raise Newer 7) handle Newer x => 7
7 : int

fun test f = f() handle Newer x => x | Overflow => ~1
val test : (unit -> int) -> int

test (fn () => raise Newer 6)
6 : int

fun fac n = if n<1 then 1 else n*fac(n-1)
fac : int -> int

fac 15
! Uncaught exception: Overflow

test (fn () => fac 15)
~1 : int
```

Sequenzialisierung

Beim Programmieren mit Ausnahmen sind manchmal sequenzialisierende Ausdrücke der Form

```
(a1 ; ... ; an)
```

hilfreich. Diese werten ihre Teilausdrücke $a_1, \ldots, a_n$ in der angegebenen Reihenfolge aus. Wenn keiner der Unterausdrücke divergiert oder eine Ausnahme wirft, wird der Wert des letzten Unterausdrucks a_n geliefert. Hier sind Beispiele:

[3] Die Auswertung von Ausdrücken wurde erstmals in Abschnitt 1.3 beschrieben. Bereits dort haben wir eine strikte links-nach-rechts Auswertung vorgeschrieben. Für Ausdrücke, die keine Ausnahme werfen (auch nicht `Overflow` bei zu großen Zahlen), ist eine Abweichung von der strikten links-nach-rechts Auswertung nicht beobachtbar, da Terminierung und Ergebnis sich nicht ändern.

```
(5 ; 7)
7 : int

(raise New ; 7)
! Uncaught exception: New
```

Die Standard ML Sprachbeschreibung definiert Sequenzialisierungen als eine abgeleitete Form:

$$(a_1 \texttt{ ; } \ldots \texttt{ ; } a_n) \implies \texttt{let val \_ = } a_1 \ldots \texttt{ val \_ = } a_{n-1} \texttt{ in } a_n \texttt{ end}$$

Die Auswertung eines Ausdrucks kann auf zwei verschiedene Arten terminieren: mit einem Ergebnis oder mit einer geworfenen Ausnahme. Terminierung mit einem Ergebnis bezeichnen wir als *reguläre Terminierung*. Wenn die Auswertung eines Ausdrucks regulär terminiert, kann das Ergebnis durchaus eine Ausnahme sein (aber eben keine geworfene). Es kann auch sein, dass während der Auswertung Ausnahmen geworfen und gefangen wurden.

reguläre Terminierung

Spezifikation von möglichen Ausnahmen

Wenn wir den Typ einer Prozedur spezifizieren, die für bestimmte Argumente eine Ausnahme wirft, geben wir in Zukunft den entsprechenden Ausnahmekonstruktor oft wie folgt an:

```
hd       : 'a list -> 'a        (* Empty *)
List.nth : 'a list * int -> 'a  (* Subscript *)
```

Dabei handelt es sich um eine nützliche Konvention, die aber keine Entsprechung im Interpreter hat:

```
hd
hd : 'a list -> 'a
```

Schneller Test auf Balanciertheit

Ein Baum ist genau dann balanciert, wenn für jeden Knoten v alle Unterbäume von v die gleiche Tiefe haben. Wir entwickeln jetzt einen schnellen Test für die Balanciertheit von Binärbäumen.

Unser Ausgangspunkt ist die Prozedur

```
fun depth (L _)         = 0
  | depth (N(_, t, t')) = 1 + Int.max(depth t, depth t')

val depth : 'a tree -> int
```

die die Tiefe von Binärbäumen bestimmt. Wir ändern die zweite Regel so ab, dass sie zusätzlich überprüft, ob die beiden Unterbäume dieselbe Tiefe haben. Wenn dies nicht der Fall ist, ist der Baum nicht balanciert. Durch das Werfen

einer Ausnahme können wir diese Entdeckung kommunizieren, ohne den Rest von `depth` zu ändern. Die modifizierte Prozedur terminiert offensichtlich genau dann regulär, wenn der Baum balanciert ist. In diesem Fall bekommen wir die Tiefe des Baums als Ergebnis. Wir können unsere Idee wie folgt in die Tat umsetzen:

```
fun balanced t =
  let
    exception Unbalanced

    fun forward (x,y) = if x=y then x else raise Unbalanced

    fun depthb (L _)       = 0
      | depthb (N(_,t,t')) = 1 + forward(depthb t, depthb t')

  in
    (depthb t ; true) handle Unbalanced => false
  end

val balanced : 'a tree -> bool
```

Die obige Prozedur `balanced` läuft viel schneller als die in Abschnitt 6.6 angegebene Prozedur gleichen Namens. Vergleichen Sie dazu die beiden Prozeduren für den Baum `tree 23` (`tree` liefert einen balancierten Binärbaum der Tiefe 23, siehe Abschnitt 6.6). Die Prozedur `balanced` aus Abschnitt 6.6 rechnet langsamer, da sie die Tiefe der Unterbäume immer wieder neu berechnet.

6.9 Optionen

Die Werte des vordefinierten Konstruktortyps

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

werden als *Optionen* bezeichnet. Dabei sind die mit `SOME` konstruierten Werte als *eingelöste Optionen* und der Wert `NONE` als die *uneingelöste Option* aufzufassen.

Als Beispiel für die Verwendung von Optionen betrachten wir die Prozedur

```
fun get xs n = SOME(List.nth(xs,n)) handle Subscript => NONE

val get : 'a list -> int -> 'a option
```

die das n-te Element einer Liste als eingelöste Option liefert, wenn es existiert:

```
get [3,4,5] 2
SOME 5 : int option
```

```
get [3,4,5] 3
NONE : int option
```

Das Feld

```
Int.minInt
SOME ~1073741824 : int option
```

der Standardstruktur `Int` ist an eine Option gebunden, die Auskunft über die kleinste darstellbare ganze Zahl gibt. Wenn `Int.minInt` den Wert `NONE` hat, bedeutet das, dass beliebig kleine Zahlen dargestellt werden können (was bei Moscow ML nicht der Fall ist). Analog gibt das Feld

```
Int.maxInt
SOME 1073741823 : int option
```

Auskunft über die größte darstellbare ganze Zahl.

Die vordefinierte Prozedur

```
fun valOf (SOME x) = x
  | valOf NONE     = raise Option.Option

val valOf : 'a option -> 'a
```

erlaubt den bequemen Zugriff auf *eingelöste* Optionen:

```
valOf Int.minInt
~1073741824 : int

valOf Int.minInt + valOf Int.maxInt
~1 : int
```

Übungsaufgaben

Aufgabe 6.1 (Binärbäume) Gegeben sei der Typkonstruktor `tree` für Binärbäume aus Abschnitt 6.5.

1. Schreiben Sie eine polymorphe Prozedur

   ```
   root: 'a tree -> 'a
   ```

   die die Marke der Wurzel eines Binärbaums liefert.

2. Schreiben Sie eine Prozedur

   ```
   inner : 'a tree -> 'a list
   ```

die die Marken der inneren Knoten eines Binärbaums liefert. Dabei sollen die Marken in der Liste in derselben Reihenfolge wie bei der Präfixprojektion auftreten. Hinweis: Schreiben Sie zuerst eine Hilfsprozedur `inner'`, die die Marken aller Knoten liefert, die keine Blätter sind.

3. Schreiben Sie eine Prozedur

   ```
   double: int tree -> int tree
   ```

   die die Marken eines Binärbaums verdoppelt. Bespielsweise soll

   ```
   double(N(5, L 3, L ~3)) = N(10, L 6, L ~6)
   ```

   gelten.

4. Schreiben Sie eine Prozedur

   ```
   mapTree : ('a -> 'b) -> 'a tree -> 'b tree
   ```

   die eine Prozedur auf jede Marke eines Binärbaums anwendet (analog zu `map` für Listen).

Aufgabe 6.2 (Symbolisches Differenzieren) Sie sollen eine Prozedur schreiben, die die Ableitung von Ausdrücken berechnet. Hier ist ein Beispiel für die Ableitung eines Ausdrucks:

$$(x^3 + 3x^2 + x + 2)' = 3x^2 + 6x + 1$$

Die zu betrachtenden Ausdrücke sollen gemäß dem folgenden Typ dargestellt werden:

```
datatype exp = Con of int                    c
             | X                             x
             | Add of exp * exp            u + v
             | Mul of exp * exp            u · v
             | Pow of exp * int             u^n
```

1. Schreiben Sie eine Deklaration, die den Bezeichner `u` an eine Darstellung des Ausdrucks $x^3 + 3x^2 + x + 2$ bindet.

2. Schreiben Sie eine Prozedur

   ```
   derive : exp -> exp
   ```

die die Ableitung eines Ausdrucks gemäß den folgenden Regeln berechnet:

$$\begin{aligned} c' &= 0 \\ x' &= 1 \\ (u+v)' &= u' + v' \\ (u \cdot v)' &= u' \cdot v + u \cdot v' \\ (u^n)' &= n \cdot u^{n-1} \cdot u' \end{aligned}$$

Die Ableitung darf vereinfachbare Teilausdrücke enthalten (z.B. $0 + u$ oder $0 \cdot u$).

3. Schreiben Sie eine Prozedur

   ```
   simplify1 : exp -> exp
   ```

   die versucht, einen Ausdruck auf oberster Ebene durch die Anwendung einer der folgenden Regeln zu vereinfachen:

$$\begin{aligned} 0+u &\to u & \qquad u+0 &\to u \\ 0 \cdot u &\to 0 & \qquad u \cdot 0 &\to 0 \\ 1 \cdot u &\to u & \qquad u \cdot 1 &\to u \\ u^0 &\to 1 & \qquad u^1 &\to u \end{aligned}$$

   Wenn keine der Regeln auf oberster Ebene anwendbar ist, wird der Ausdruck unverändert zurückgeliefert.

4. Schreiben Sie eine Prozedur

   ```
   simplify : exp -> exp
   ```

   die einen Ausdruck gemäß der obigen Regeln solange vereinfacht, bis keine Regel mehr anwendbar ist. Gehen Sie bei zusammengesetzten Ausdrücken wie folgt vor:

   a) Vereinfachen Sie zuerst die Unterausdrücke.

   b) Vereinfachen Sie dann den zusammengesetzten Ausdruck mit den vereinfachten Unterausdrücken mithilfe von `simplify1`.

Aufgabe 6.3 (Dreifach-Partition) Sie sollen Prozeduren schreiben, die eine Liste *xs* in drei Listen *us*, *vs*, *ws* zerlegen, sodass *us@vs@ws* eine Permutation von *xs* ist.

1. Schreiben Sie mithilfe von `foldl` eine Prozedur

   ```
   partition : ('a -> order) ->
                  'a list -> 'a list * 'a list * 'a list
   ```

die für eine Prozedur f eine Liste wie folgt zerlegt:

a) *us* enthält genau die Elemente von *xs*, für die f den Wert `LESS` liefert.

b) *ws* enthält genau die Elemente von *xs*, für die f den Wert `GREATER` liefert.

2. Schreiben Sie mithilfe von `partition` eine Prozedur

```
myPartition : int * int ->
                int list -> int list * int list * int list
```

die für (m, n) mit $m \leq n$ eine Liste wie folgt zerlegt:

a) *us* enthält genau die Elemente von *xs*, die echt kleiner als *m* sind.

b) *ws* enthält genau die Elemente von *xs*, die echt größer als *n* sind.

Aufgabe 6.4 (Optionen) Schreiben Sie eine Prozedur

```
find : ('a -> bool) -> 'a tree -> 'a option
```

die zu einer Prozedur f und einem Binärbaum die erste Marke des Baums liefert (gemäß der Präfixordnung), für die f den Wert `true` liefert. Wenn der Baum keine solche Marke enthält, soll `NONE` geliefert werden.

Aufgabe 6.5 (Schneller Test auf Doppelauftreten) Sie sollen eine Prozedur

```
test : int list -> bool
```

schreiben, die testet, ob in einer Liste ein Element mehrfach auftritt. Dabei soll die Liste mit einer modifizierten Sortierprozedur sortiert werden, die eine Ausnahme wirft, sobald zwei Elemente der Liste miteinander verglichen werden, die gleich sind. Wenn man einen schnellen Sortieralgorithmus verwendet, bekommt man mit dieser Methode einen schnellen Test auf Doppelauftreten.

1. Deklarieren Sie eine Ausnahme `Double`.

2. Schreiben Sie eine Prozedur

```
order : int * int -> order  (* Double *)
```

die zu (m, n)

a) den Wert `LESS` liefert, wenn $m < n$.

b) den Wert `GREATER` liefert, wenn $m > n$.

c) die Ausnahme `Double` wirft, wenn $m = n$.

3. Schreiben Sie mithilfe der Sortierprozedur

```
Listsort.sort : ('a * 'a -> order) -> 'a list -> 'a list
```

aus der Standardstruktur `Listsort` eine modifizierte Sortierprozedur

```
dsort : int list -> int list  (* Double *)
```

die die Ausnahme `Double` wirft, sobald sie zwei gleiche Elemente miteinander vergleicht.

4. Schreiben Sie eine Prozedur

   ```
   test : int list -> bool
   ```

   die testet, ob in einer Liste ein Element mehrfach auftritt. Schreiben Sie `test` mit einem Let-Ausdruck, in dem Sie `Double`, `order` und `dsort` lokal deklarieren.

Aufgabe 6.6 (Terme) Sie sollen Konzepte und Prozeduren, die Sie für Binärbäume kennengelernt haben, auf Terme übertragen, die gemäß der Typdeklaration

```
datatype 'a term = T of 'a * 'a term list
```

dargestellt werden. Benutzen Sie die Deklarationen

```
val t1 = T(1, [T(2,[]), T(7,[]), T(7,[])])
val t2 = T(3, [T(0,[t1]), T(4,[]), t1])
```

zum Testen ihrer Prozeduren.

1. Schreiben Sie eine Prozedur

   ```
   prefixp : 'a term -> 'a list
   ```

   die die Präfixprojektion eines Terms liefert. Verwenden Sie dazu `map` und `List.concat`. Es soll gelten:

   ```
   prefixp t2 = [3, 0, 1, 2, 7, 7, 4, 1, 2, 7, 7]
   ```

2. Schreiben Sie eine Prozedur

   ```
   postfixp : 'a term -> 'a list
   ```

   die die Postfixprojektion eines Terms liefert. Es soll gelten:

   ```
   postfixp t2 = [2, 7, 7, 1, 0, 4, 2, 7, 7, 1, 3]
   ```

3. Schreiben Sie eine Prozedur

   ```
   frontier : 'a term -> 'a list
   ```

   die die Grenze eines Terms liefert. Es soll gelten:

   ```
   frontier t2 = [2, 7, 7, 4, 2, 7, 7]
   ```

4. Schreiben Sie eine Prozedur

```
subterm : 'a term -> int list -> 'a term
```

die zu einem Term und einem Knoten den von diesem Knoten aus erreichbaren Teilterm liefert. Verwenden Sie `List.nth`. Es soll gelten:

```
subterm t2 [1,1] = t1
```

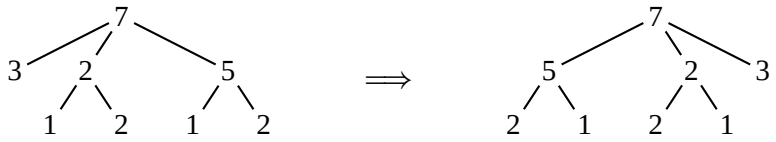
Wenn der Knoten kein Knoten des Terms ist, soll die Ausnahme `Subscript` geworfen werden.

5. Schreiben Sie eine Prozedur

   ```
   balanced : 'a term -> bool
   ```

   die testet, ob ein Term balanciert ist. Gehen Sie dabei wie beim schnellen Balanciertheitstest für Binärbäume vor (siehe Abschnitt 6.8). Verwenden Sie eine modifizierte Version der Prozedur `depth` in Abschnitt 6.7, die statt `Int.max` eine passend definierte Prozedur `forward` benutzt.

Aufgabe 6.7 (Spiegeln von geordneten Bäumen) Spiegeln invertiert die Anordnung der Unterbäume eines geordneten Baums:



1. Schreiben Sie eine Prozedur

   ```
   mirror : 'a tree -> 'a tree
   ```

   die Binärbäume spiegelt.

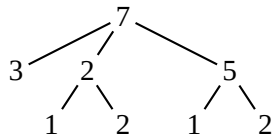
2. Schreiben Sie eine Prozedur

   ```
   mirror : 'a term  -> 'a term
   ```

   die Terme spiegelt.

Aufgabe 6.8 (Ebenen von geordneten Bäumen) Sei b ein geordneter Baum. Die *Tiefe* eines Knotens $u \in \mathit{Dom}\ b$ ist $|u|$ (also der Abstand zur Wurzel). Die *n-te Ebene von b* ($n \in \mathbb{N}$) ist die Liste der Marken aller Knoten der Tiefe n, angeordnet entsprechend der Anordnung der Knoten.

Als Beispiel betrachten wir den Baum



Die nullte Ebene ist [7], die erste Ebene ist $[3, 2, 5]$, die zweite Ebene ist $[1, 2, 1, 2]$, und die dritte und alle nachfolgenden Ebenen sind [].

1. Schreiben Sie eine Prozedur

   ```
   level : 'a tree * int -> 'a list
   ```

   die für $n \in \mathbb{N}$ die n-te Ebene eines Binärbaums liefert.

2. Schreiben Sie eine Prozedur

   ```
   level : 'a term * int -> 'a list
   ```

   die für $n \in \mathbb{N}$ die n-te Ebene eines Terms liefert.

Aufgabe 6.9 (Listen mit Konstruktortypen) Die Typdeklaration

```
datatype 'a mylist = Nil | Cons of 'a * 'a mylist
```

definiert einen Typkonstruktor `mylist`, mit dem sich Listen darstellen lassen.

1. Schreiben Sie zwei Prozeduren

   ```
   to   : 'a list -> 'a mylist
   from : 'a mylist -> 'a list
   ```

   sodass für alle Listen *xs* gilt:

   `from(to` *xs*`) =` *xs*

2. Schreiben Sie eine Prozedur

   ```
   mylength : 'a mylist -> int
   ```

   sodass für alle Listen *xs* gilt:

   `mylength(to` *xs*`) =` $|xs|$

Aufgabe 6.10 (Zahlen mit Konstruktortypen) Mit dem Typ

```
datatype nat = N | S of nat
```

kann man die natürlichen Zahlen darstellen.

1. Schreiben Sie zwei Prozeduren

   ```
   to   : int -> nat  (* Subscript *)
   from : nat -> int
   ```

   sodass für alle $n \in \mathbb{N}$ gilt:

   `from(to` n`) =` n

2. Schreiben Sie eine Prozedur

   ```
   add : nat * nat -> nat
   ```

sodass für alle $m, n \in \mathbb{N}$ gilt:

```
from(add(to m, to n)) = m + n
```

3. Deklarieren Sie mithilfe von `nat` einen Typ `myint`, mit dem die ganzen Zahlen dargestellt werden können. Schreiben Sie zwei Prozeduren

```
to'   : int -> myint
from' : myint -> int
```

sodass für alle $z \in \mathbb{Z}$ gilt:

```
from'(to' z) = z
```