Kapitel 5

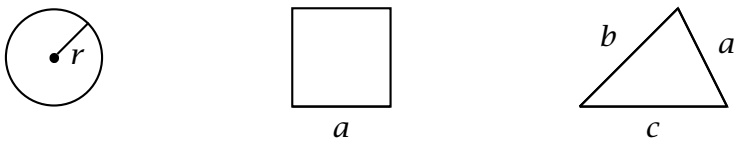
Konstruktortypen und Ausnahmen

In diesem Kapitel lernen Sie, wie man neue Typen deklariert. Durch die Deklaration neuer Typen können Sie die Programmiersprache um maßgeschneiderte Darstellungen für neue Objekte erweitern, die Sie im Rahmen einer Programmieraufgabe entwerfen. Die in diesem Kapitel eingeführten Sprachmittel versetzen Sie in die Lage, syntaktische Objekte darzustellen und zu verarbeiten. Beispielsweise werden Sie im Rahmen einer Übungsaufgabe eine Prozedur schreiben, die Ausdrücke symbolisch differenziert.

Zwei weitere Themen dieses Kapitels sind das Programmieren mit baumartigen Objekten und das Programmieren mit Ausnahmen.

5.1 Varianten und Konstruktoren

Wir wollen drei geometrische Formen als mathematische Objekte darstellen:



Kreis	Quadrat	Dreieck
$(1, r)$	$(2, a)$	$(3, (a, b, c))$

Wir tun das durch Paare, die in der ersten Komponente mit einer der Zahlen 1, 2, 3 anzeigen, um welche Form es sich handelt:

- Ein Paar $\langle 1, r\rangle$ stellt einen Kreis mit dem Radius r dar.
- Ein Paar $\langle 2, a\rangle$ stellt ein Quadrat mit der Seitenlänge a dar.
- Ein Paar $\langle 3, \langle a, b, c\rangle\rangle$ stellt ein Dreieck mit den Seitenlängen a, b, c dar.

Die erste Komponente dieser Paare bezeichnen wir als **Variantennummer**, und die zweite Komponente als **Datum**.[1]

Zusammenfassend stellen wir fest, dass wir unsere drei geometrischen Formen mit den Elementen der Summe

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

darstellen können. Diese Darstellung können wir mit der Deklaration eines neuen Typs `shape` nach Standard ML übertragen:

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

Da Standard ML keinen Typ für positive reelle Zahlen hat, approximieren wir $\mathbb{R}^+$ durch `real`. Für jede der drei geometrischen Formen führt die Deklaration einen sogenannten **Konstruktor** ein, mit dem die Darstellungen der Form **konstruiert** werden können. Die eingeführten Konstruktoren können wie Prozeduren benutzt werden und haben die folgenden Typen:

```
Circle   : real → shape
Square   : real → shape
Triangle : real * real * real → shape
```

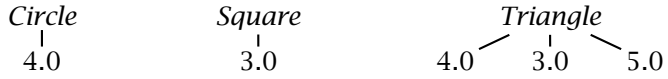
Hier sind einige Experimente mit dem Interpreter:

```
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 Werte des Typs `shape` der besseren Lesbarkeit wegen nicht als Paare aus Variantennummer und Datum dar, sondern als Anwendungen des jeweiligen Konstruktors. Diese Konvention machen wir uns auch bei der grafischen Darstellung dieser Werte zu Eigen:



Typen, die wie `shape` mit dem Schlüsselwort `datatype` deklariert sind, bezeichnen wir als **Konstruktortypen**. Die Werte eines Konstruktortyps bestehen aus

[1] Variantennummern werden beim maschinennahen Programmieren oft als *tags* bezeichnet.

einer **Variantennummer** und einem **Datum**. Werte, die mit dem *n*-ten Konstruktor des Typs beschrieben werden können, tragen die Variantennummer *n*. Wenn ein Konstruktortyp *n* Konstruktoren hat, sagen wir, dass er *n* **Varianten** hat. Zur *i*-ten Variante eines Konstruktortyps gehören alle Werte des Typs, die die Variantenummer *i* tragen.

Eine Prozedur, die den Flächeninhalt einer geometrischen Form bestimmt, können wir wie folgt deklarieren:

```
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 jeweils für eine der drei Varianten von `shape` zuständig sind. Die erste Regel ist für die Variante `Circle`, die zweite Regel für die Variante `Square` und die dritte Regel für die Variante `Triangle` zuständig. Die Muster der Regeln haben die Form einer Konstruktoranwendung. Wenn `area` beispielsweise mit einem Argument der Variante `Square` aufgerufen wird, kommt die zweite Regel zur Anwendung. Dabei wird die Variable `a` an das Datum des Arguments gebunden. Wenn die dritte Regel zur Anwendung kommt, werden die Variablen `a`, `b`, `c` an die drei Komponenten des Datums des Arguments gebunden.

Die Deklaration von Konstruktoren versetzt uns in die Lage, die verschiedenen Varianten eines Konstruktortyps durch frei gewählte Namen zu bezeichnen. Das ist bequemer und lesbarer als die Verwendung von Variantenummern. Darüber hinaus sorgt die Verwendung von Konstruktoren dafür, dass man Ausdrücken ansehen kann, wie die durch sie beschriebenen Werte zu verstehen sind. Die richtige Lesart für den Wert eines Ausdrucks wird dabei durch den Typ des Ausdrucks beschrieben:

```
Circle 4.0
Circle 4.0 : shape

(1,4.0)
(1,4.0) : int * real
```

Wenn wir wollen, können wir eine Prozedur schreiben, die die Variantennummer einer geometrischen Form 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. `Circle`). Dagegen verwenden wir für Typen und Werte stets Bezeichner, die mit einem Kleinbuchstaben beginnen. Diese Konvention soll die Lesbarkeit von Programmen verbessern.

5.2 Enumerationstypen

Die Deklaration

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

führt einen Typ `day` ein, dessen Werte die verschiedenen Wochentage darstellen. Die Werte des Typs werden durch **nullstellige Konstruktoren** beschrieben, die wie Konstanten verwendet werden können. Hier ist ein Beispiel:

```
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
```

Nullstellige Konstruktoren beschreiben Werte, die nur aus einer Variantennummer bestehen. Wir können uns die Werte des Typs `day` als die Zahlen $1, \ldots, 7$ vorstellen. Typen, die wie `day` nur mithilfe von nullstelligen Konstruktoren definiert sind, werden als **Enumerationstypen** bezeichnet.

Der vordeklarierte Enumerationstyp

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

wird von einigen Standardstrukturen verwendet:

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

Real.compare(4.3,4.2)
GREATER : order

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
```

Bei `Listsort.sort` handelt es sich um eine polymorphe Sortierprozedur für Listen.

5.3 Typsynonyme

Punkte in der Ebene können wir durch Paare aus zwei reelen Zahlen darstellen. Wenn wir mit dieser Darstellung arbeiten, kann es sinnvoll sein, mithilfe der Deklaration

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

ein **Typsynonym** `point` einzuführen. Dabei handelt es sich um keinen neuen Typ, sondern lediglich um eine neue Bezeichnung für einen bereits existierenden Typ. Hier ist ein Beispiel für eine sinnvolle Verwendung des Typsynonyms `point`:

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

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

```
datatype point = P of real * real

P(2.0,3.0)
P(2.0,3.0) : point
```

Eine Prozedur, die einen Punkt an der x-Achse spiegelt, können wir mit dieser Darstellung wie folgt schreiben:

```
fun mirror (P(x,y)) = P(~x,y)
val mirror : point → point
```

5.4 Ausnahmen

Sie haben bereits gelernt, dass die Auswertung des Ausdrucks

```
raise Empty
! Uncaught exception: Empty
```

keinen Wert liefert, sondern die Ausnahme `Empty` wirft. Jetzt lernen Sie, wie man neue Ausnahmen deklariert und wie man geworfene Ausnahmen wieder einfängt.

Zunächst machen wir die überraschende Entdeckung, dass es sich bei Ausnahmen um Werte eines Typs `exn` handelt:

```
Empty
Empty : exn
```

Dabei ist der Typ `exn` als Konstruktortyp zu verstehen, und `Empty` als nullstelliger Konstruktor. Der Konstruktortyp `exn` hat die Besonderheit, dass man ihn durch die Deklaration neuer Konstruktoren erweitern kann. Beispielsweise führt die Deklaration

```
exception New
exn New = New : exn
```

einen neuen nullstelligen **Ausnahmekonstruktor** `New` ein. Man kann auch einstellige Ausnahmekonstruktoren deklarieren:

```
exception Newer of int
exn Newer : int → exn
```

Die folgenden Beispiele sollen zeigen, dass Ausnahmekonstruktoren wie normale Konstruktoren verwendet werden können:

```
(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
```

Mit Ausdrücken der Form

```
raise e
```

können Ausnahmen **geworfen** werden. Dabei muss der Ausdruck e den Typ `exn` haben. Hier ist ein Beispiel:

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

Da Raise-Ausdrücke keinen Wert liefern, können sie jeden Typ annehmen:

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

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)
```

stets die Ausnahme `Overflow`.[2]

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
```

Beim Programmieren mit Ausnahmen sind manchmal Ausdrücke der Form

$(e_1 \ ; \ \ldots \ ; \ e_n)$

[2] Die Auswertung von Ausdrücken wurde erstmals in Abschnitt 1.16 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.

hilfreich. Solche Ausdrücke werden als **Sequenzialisierungen** bezeichnet. Eine Sequenzialisierung $(e_1;\ldots;e_n)$ wird ausgewertet, indem zunächst die Teilausdrücke $e_1,\ldots,e_n$ in der angegebenen Reihenfolge ausgewertet werden. Wenn die Auswertung aller Teilausdrücke regulär terminiert, wird der Wert des letzten Teilausdrucks e_n geliefert. Hier sind Beispiele:

```
(5 ; 7)
7 : int

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

Sequenzialisierungen können auf Let-Ausdrücke zurückgeführt werden:

$$(e_1 \texttt{ ; ... ; } e_n) \implies \texttt{let val \_ = } e_1 \texttt{ ... val \_ = } e_{n-1} \texttt{ in } e_n \texttt{ end}$$

Wenn wir den Typ einer Prozedur spezifizieren, die für bestimmte Argumente eine Ausnahme wirft, geben wir 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
```

5.5 Arithmetische Ausdrücke

Mithilfe von Konstruktortypen können wir syntaktische Objekte darstellen. Als Beispiel betrachten wir Ausdrücke, die mit Konstanten, Variablen, Addition sowie Multiplikation gebildet sind (zum Beispiel $(2x+y)(x+3)$).

Zunächst drei englische Fachausdrücke: Ausdruck heißt expression, Umgebung environment, und Auswertung evaluation.

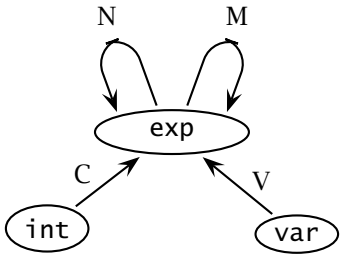
Variablen stellen wir durch ganze Zahlen dar:

```
type var = int
```

Ausdrücke stellen wir durch Werte des Konstruktortyps

```
datatype exp =
  C of int | V of var | A of exp * exp | M of exp * exp
```

dar. Jede der vier Ausdrucksformen (Konstante, Variable, Addition, Multiplikation) wird durch einen entsprechenden Konstruktor realisiert. Beachten Sie, dass die Definition des Typs `exp` rekursiv ist. Die Rekursion ist erforderlich, damit zwei Ausdrücke mithilfe von Addition oder Multiplikation zu einem größeren Ausdruck zusammengesetzt werden können. Grafisch können wir die Struktur des Typs `exp` wie folgt darstellen:



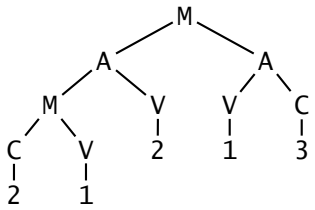
Als Beispiel betrachten wir die Darstellung des Ausdrucks

$$(2x + y)(x + 3)$$

Zunächst legen wir fest, dass wir die Variable x durch die Zahl 1 und die Variable y durch die Zahl 2 darstellen. Damit können wir den Ausdruck wie folgt darstellen:

```
val e = M(  A(M(C 2, V 1), V 2),  A(V 1, C 3)  )
```

Diese Darstellung wird lesbarer, wenn wir sie als Baum zeichnen:



Um den Ausdruck auswerten zu können, benötigen wir eine Umgebung, die den Variablen x und y Werte zuweist. Beispielsweise liefert $(2x + y)(x + 3)$ mit der Umgebung $\{x \mapsto 5, y \mapsto 3\}$ den Wert 104. Umgebungen stellen wir durch Prozeduren dar, die den Wert von Variablen liefern:

```
type env = var → int  (* Unbound *)
```

Wenn eine Umgebung den Wert einer Variablen nicht kennt, soll sie die Ausnahme `Unbound` werfen:

```
exception Unbound
```

Die Umgebung $\{x \mapsto 5, y \mapsto 3\}$ können wir jetzt wie folgt darstellen:

```
val env = fn 1 => 5 | 2 => 3 | _ => raise Unbound
```

Unser Ziel ist eine Prozedur

```
eval : exp → env → int
```

die zu einem Ausdruck und einer Umgebung den Wert des Audrucks liefert:

```
fun eval (C c)     _   = c
  | eval (V v)     env = env v
  | eval (A(e,e')) env = eval e env + eval e' env
  | eval (M(e,e')) env = eval e env * eval e' env

eval e env
104 : int
```

5.6 Optionen

Die Werte des vordeklarierten 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 vordeklarierte 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
```

Die vordeklarierte Prozedur

```
fun isSome NONE     = false
  | isSome (SOME _) = true

val isSome : 'a option → bool
```

testet, ob es sich bei einer Option um eine eingelöste Option handelt.

5.7 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 e of M1 => e1 | ··· | Mn => en
```

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

`(fn` M_1 `=>` e_1 `|` $\cdots$ `|` M_n `=>` e_n`)` e

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.

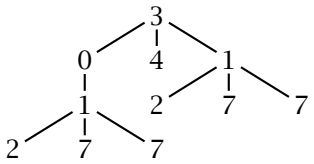
Übrigens sind aus Sicht der Standard ML Sprachdefinition auch Konditionale eine abgeleitete Form:

`if` e_1 `then` e_2 `else` e_3 $\quad\Longrightarrow\quad$ `(fn true =>` e_2 `| false =>` e_3`)` e_1

5.8 Bäume informell und formal

Baumartige Objekte sind in der Informatik allgegenwärtig. Bereits in Kapitel 1 haben wir die Baumdarstellung von Ausdrücken kennengelernt. Wir wollen das bisher vage Konzept eines baumartigen Objekts jetzt präziser fassen. Wir beginnen mit Bildern und Beispielen.

Wir beginnen mit der grafischen Darstellung eines Baums:[3]



Die **Knoten** des Baums entsprechen den Stellen der Darstellung, an denen Zahlen angegeben sind. Die zu einem Knoten gehörende Zahl heißt **Marke** des Knotens. Unser Baum hat 11 Knoten. Sein oberster Knoten ist mit 3 markiert. Der oberste Knoten eines Baums wird als **Wurzel** des Baums bezeichnet. Unter der **Größe eines Baums** versteht man die Anzahl seiner Knoten.

Die **Kanten** des Baums sind durch Striche dargestellt. Jede Kante verbindet zwei Knoten. Unser Baum hat 10 Kanten. Wenn zwei Knoten durch eine Kante verbunden sind, liegt der eine Knoten **höher** und der andere **tiefer**. Der höhere Knoten wird als **Vorgänger** des tieferen Knotens bezeichnet, und der tiefere Knoten als **Nachfolger** des höheren Knotens. Die Wurzel unseres Baums hat drei Nachfolger, die mit 0, 4 und 1 markiert sind. Wir sprechen vom **ersten, zweiten und dritten Nachfolger** (Nummerierung von links nach rechts, mit eins beginnend).

[3] Wenn wir die grafische Darstellung um 180 Grad drehen, bekommen wir ein Bild, das wir als einen sich verästelnden Baum interpretieren können.

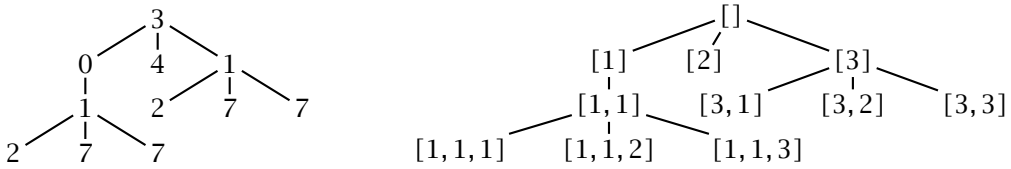
Ein **Pfad** ist eine durch Kanten realisierte Verbindung zwischen zwei Knoten eines Baums. Die **Länge eines Pfads** ist die Anzahl der beteiligten Kanten. Pfade der Länge 0 verbinden einen Knoten mit sich selbst. Pfade der Länge 1 sind durch genau eine Kante gegeben.

Hier sind zwei charakteristische Eigenschaften von Bäume:

1. Zwischen zwei Knoten eines Baums existiert immer genau ein Pfad.
2. Die Wurzel eines Baums hat keinen Vorgänger, und alle anderen Knoten haben genau einen Vorgänger.

Die **Tiefe eines Knotens** ist die Länge des eindeutig bestimmten Pfades, der die Wurzel mit diesem Knoten verbindet. Die Tiefe der Wurzel ist immer 0. Der am weitesten rechts stehende Knoten unseres Baums hat die Tiefe 2. Die **Tiefe eines Baums** ist die maximale Tiefe seiner Knoten. Die Tiefe unseres Baums ist 3.

Die Knoten eines Baums können mit Listen **adressiert** werden, die den Pfad von der Wurzel zum Knoten beschreiben. Die folgende Darstellung zeigt rechts die **Adressen** der Knoten unseres Beispielbaums:



Die Wurzel eines Baums hat immer die Adresse $[]$. Die Adresse $[2]$ besagt, dass man zum zweiten Nachfolger der Wurzel gehen soll. Die Adresse $[3,2]$ besagt, dass man zuerst zum dritten Nachfolger der Wurzel und dann zu dessen zweitem Nachfolger gehen soll. Machen Sie sich klar, dass die Tiefe eines Knotens gerade die Länge seiner Adresse ist. Wenn u die Adresse eine Knotens ist, dann ist $u@[n]$ die Adresse seines n-ten Nachfolgers (wenn dieser existiert).

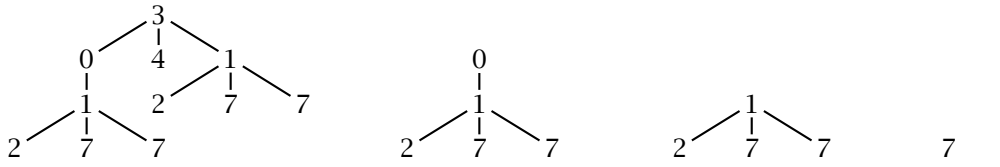
Jetzt kommt ein wichtiger Schritt in unserer Präzisierung des Begriffs Baum: Wir identifizieren die Knoten eines Baums mit ihren Adressen. Damit machen wir Knoten zu mathematischen Objekten. Bisher waren Knoten lediglich als „Stellen“ in einer grafischen Darstellung charakterisiert.

Jeder Baum hat mindestens einen Knoten. Bäume mit genau einem Knoten heißen **atomar**, und Bäume mit mindestens zwei Knoten heißen **zusammengesetzt**. Die **Blätter** eines Baums sind die Knoten des Baums, die keine Nachfolger haben. Hier sind die Blätter unseres Beispielbaums:

$$[1,1,1],\ [1,1,2],\ [1,1,3],\ [2],\ [3,1],\ [3,2],\ [3,3]$$

Jeder Baum hat genau eine Wurzel und mindestens ein Blatt. Ein Baum ist genau dann atomar, wenn seine Wurzel ein Blatt ist.

Bäume enthalten Bäume. In der Tat ist jedem Knoten eines Baums ein so genannter **Teilbaum** zugeordnet. Für unseren Beispielbaum haben wir unter anderem die folgenden Zuordnungen:



Beachten Sie, dass der Wurzel eines Baums der ganze Baum als Teilbaum zugeordnet ist. Beachten Sie weiter, dass den Blättern eines Baums atomare Teilbäume zugeordnet sind.

Unter den **Unterbäumen** eines Baums versteht man die Teilbäume der Nachfolger seiner Wurzel. Ensprechend der Nummerierung der Nachfolger der Wurzel spricht man vom *n*-**ten Unterbaum** eines Baums. Ein Baum ist atomar, wenn er keine Unterbäume hat, und zusammengesetzt, wenn er mindestens einen Unterbaum hat.

Der **Kopf** eines Baums ist die Marke seiner Wurzel. Wir können sagen, dass ein Baum aus seinem Kopf und seinen Unterbäumen zusammengesetzt ist.

Wir sind jetzt in der Lage, Bäume durch mathematische Objekte zu beschreiben. Dazu stellen wir einen Baum durch ein Paar da, das aus seinem Kopf und der Liste seiner Unterbäume besteht. Ein atomarer Baum mit dem Kopf x wird also durch das Paar $(x, [\,])$ beschrieben. Ein zusammengesetzter Baum mit dem Kopf x und den Unterbäumen $t_1, \ldots, t_n$ wird durch das Paar $(x, [t_1, \ldots, t_n])$ beschrieben. Hier sind Beispiele:

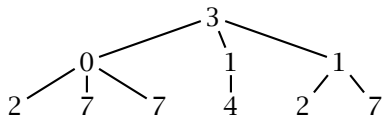
Wir machen jetzt einen radikalen Schritt. Unter einem Baum wollen wir in Zukunft das mathematische Objekt verstehen, das ihn wie gerade erklärt beschreibt. Dieser Schritt ist radikal, weil es sich bei einem Baum ab jetzt nicht mehr um ein vage beschriebenes Objekt handelt, sondern um eine konkretes,

vollständig beschriebenes, mathematisches Objekt. Die Überführung von gedanklichen Konzepten in konkrete mathematische Objekte bezeichnet man übrigens als **Formalisierung**.

Damit die gerade beschriebene Formalisierung von Bäumen wirklich mathematischen Standards genügt, müssen die oben eingeführten Begriffe alle auf der Grundlage der mathematischen Realisierung von Bäumen definiert werden. Bisher haben wir diese Begriffe nämlich informell (Gegenteil von formal) mithilfe von Bildern, Beispielen und Analogien eingeführt. Wir werden uns die mathematische Definition dieser Begriffe hier sparen. Stattdessen werden wir diese Begriffe im nächsten Abschnitt mit Prozeduren definieren.

Die Formalisierung von Bäumen erlaubt sofort eine wichtige Verallgemeinerung: Für die Marken eines Baums können wir beliebige mathematische Objekte verwenden, statt wie bisher nur Zahlen. Sei X eine Menge. Unter einem **Baum über** X verstehen wir einen Baum, dessen Marken alle Elemente von X sind.

Wir führen noch drei wichtige Begriffe ein. Ein Baum heißt **linear** genau dann, wenn jeder seiner Knoten höchstens einen Nachfolger hat. Offensichtlich ist ein Baum genau dann linear, wenn er genau ein Blatt hat. Ein Baum heißt **binär** genau dann, wenn jeder seiner Knoten, der kein Blatt ist, genau zwei Nachfolger hat. Ein Baum heißt **balanciert** genau dann, wenn alle sein Blätter die gleiche Tiefe haben. Jeder atomare Baum ist linear, binär und balanciert. Unser Beispielbaum ist nicht balanciert. Sein dritter Unterbaum ist balanciert. Hier ist ein balancierter Baum der Tiefe 3:



5.9 Ein ausführbares Modell für Bäume

Wir realisieren das gerade eingeführte mathematische Modell für Bäume jetzt durch ein Programm. Damit erhalten wir ein ausführbares Modell für Bäume, dass dem mathematischen Modell genau entspricht. Der Vorteil des ausführbaren Modells ist, dass man damit nach Belieben experimentieren kann. Wenn Sie zum Beispiel nicht sicher sind, wie der Teilbaum eines bestimmten Knotens eines bestimmten Baums aussieht, können Sie sich den Teilbaum mithilfe des ausführbaren Modells sofort zeigen lassen.

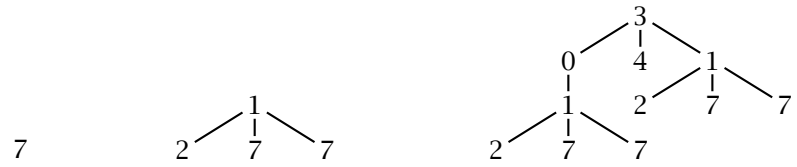
Zunächst definieren wir einen Typkonstruktor, mit dem wir Bäume darstellen können:

```
datatype 'a tree = T of 'a * ('a tree list)
```

Dieser Typkonstruktur liefert uns zu jedem Typ `'a` einen Typ `'a tree`, dessen Werte Bäume darstellen, deren Marken Werte des Typs `'a` sind. Dabei wird ein Baum durch seinen Kopf und die Liste seiner Unterbäume dargestellt. Die Deklarationen

```
val t0 = T(7,[])
val t1 = T(1, [T(2,[]), t0, t0])
val t2 = T(3, [T(0,[t1]), T(4,[]), t1])
```

liefern die folgenden Bäume:



Die Prozedur

```
fun head (T(x,_)) = x
val head : 'a tree → 'a
```

liefert den Kopf eines Baums. Die Prozedur

```
fun dst (T(_,ts)) n = List.nth(ts,n-1)
val dst : 'a tree → int → 'a tree
```

liefert zu einem Baum t und einer positiven Zahl n den n-ten Unterbaum von t. Die Unterbäume werden dabei mit 1 beginnend nummeriert. Wenn der Baum keinen n-ten Unterbaum hat, wird die Ausnahme `Subscript` geworfen. Der Bezeichner `dst` steht für direct subtree, der englischen Bezeichnung für Unterbäume. Die Prozedur

```
fun atomic (T(_,ts)) = null ts
val atomic : 'a tree → bool
```

testet, ob ein Baum atomar ist. Die Prozedur

```
fun subtree t nil = t
  | subtree t (n::ns) = subtree (dst t n) ns
val subtree : 'a tree → int list → 'a tree
```

liefert zu einem Baum und einer Liste den zugeordneten Teilbaum:

```
subtree t2 [1,1,3]
T(7, []) : int tree
```

```
subtree t2 [1,1]
T(1, [T(2,[]), T(7,[]), T(7,[])]) : int tree
```

Wenn die Liste kein Knoten des Baums ist, wird die Ausnahme `Subscript` geworfen:

```
subtree t2 [1,2]
! Uncaught exception: Subscript
```

Die Prozedur

```
fun node t ns = (subtree t ns ; true) handle Subscript => false
val node : 'a tree → int list → bool
```

testet für einen Baum, ob es sich bei einer Liste um einen Knoten des Baums handelt:

```
node t2 [1,2]
false : bool

node t2 [1,1,3]
true : bool
```

Die Prozedur

```
fun leaf t ns = atomic (subtree t ns) handle Subscript => false
val leaf : 'a tree → int list → bool
```

testet für einen Baum, ob es sich bei einer Liste um ein Blatt des Baums handelt:

```
leaf t2 [1,1]
false : bool

leaf t2 [1,1,3]
true : bool
```

Die Prozedur

```
fun label t ns = head (subtree t ns)
val label : 'a tree → int list → 'a
```

liefert die Marke eines Knotens:

```
label t2 [1,1]
1 : int

label t2 [1,1,3]
7 : int
```

Die Prozedur

```
fun pred t nil = raise Subscript
  | pred t ns  = (subtree t ns ; rev(tl(rev ns)))
val pred : 'a tree → int list → int list
```

liefert zu einem Baum und einem Knoten den Vorgänger des Knotens:

```
pred t2 [1,1,3]
[1,1] : int list
```

Wenn es sich bei der Liste um die Wurzel des Baums oder um keinen Knoten des Baums handelt, wird die Ausnahme `Subscript` geworfen. Die Prozedur

```
fun succ t ns n = let val ns' = ns@[n] in subtree t ns' ; ns' end
val succ : 'a tree → int list → int → int list
```

liefert zu einem Baum, einem Knoten und einer Zahl n den n-ten Nachfolger des Knotens:

```
pred t2 [1,1] 2
[1,1,2] : int list
```

Wenn der Nachfolger nicht existiert, wird die Ausnahme `Subscript` geworfen.

Die Größe eines Baum ist die Anzahl seiner Knoten. Um die Größe eines Baums zu berechnen, ist die folgende Eigenschaft nützlich:

- Die Größe eines Baums ist eins plus die Summe der Größen seiner Unterbäume.

Damit bekommen wir die folgende Prozedur:

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

size t2
11 : int
```

Die Tiefe eine Baums ist die maximale Tiefe seiner Knoten. Um die Tiefe eines Baums zu berechnen, ist die folgende Eigenschaft nützlich:

- Die Tiefe eines zusammengesetzten Baums ist eins plus die maximale Tiefe seiner Unterbäume.

Damit bekommen wir die folgende Prozedur:

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

depth t2
3 : int
```

5.10 Test auf Balanciertheit

Ein Baum ist genau dann balanciert, wenn seine Blätter alle die gleiche Tiefe haben. Um die Balanciertheit eines Baums zu testen, ist die folgende Charakterisierung nützlich:

- Ein Baum ist genau dann balanciert, wenn für jeden seiner Teilbäume t gilt, dass alle seine Unterbäume die gleiche Höhe haben.

Unser Ausgangspunkt ist die bereits angegebene Prozedur `depth` zur Berechnung der Tiefe eines Baums. Diese formulieren wir geringfügig um, indem wir eine eigentlich unnötige Fallunterscheidung einführen:

```
fun depth' (T(_, nil  )) = 0
  | depth' (T(_, t::ts)) = 1 + foldl Int.max (depth t) (map depth ts)
val depth : 'a tree → int
```

Wir ändern die zweite Regel so ab, dass sie zusätzlich prüft, ob alle 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:

```
exception Unbalanced

fun depthb (T(_, nil  )) = 0
  | depthb (T(_, t::tr)) = 1 + foldl forward (depth t) tr

and forward (t,n) = if depthb t = n then n else raise Unbalanced

exn Unbalanced : exn
val depthb : 'a tree → int
val forward : 'a tree * int → int
```

Neu ist die Tatsache, dass die Hilfsprozedur `forward` statt mit `fun` mit `and` deklariert ist. Das liegt daran, dass `depthb` und `forward` **verschränkt rekursiv** sind und daher eine Deklaration mit `fun` unzulässig wäre (auch, wenn wir `forward` vor `depthb` deklarieren).

Den gewünschten Test auf Balanciertheit erhalten wir jetzt wie folgt:

```
fun balanced t =
  let exception Unbalanced
      fun depthb ···
      and forward ···
  in
      (depthb t ; true) handle Unbalanced => false
  end
val balanced : 'a tree → bool
```

5.11 Linearisierungen und Projektionen

Eine Liste x heißt **Segment** einer Liste y, wenn es Listen u und v gibt mit u@x@v=y. Eine Liste heißt **Linearisierung** eines Baums, wenn sie genau die Knoten des Baums enthält (ohne Doppelauftreten), und wenn sie die Knoten jedes Teilbaums des Baums als Segment enthält. Die **Präfixlinearisierung** eines Baums ist die folgende Linearisierung:

1. Für jeden Teilbaum gilt: Die Wurzel erscheint vor den Knoten der Unterbäume.
2. Für jeden Teilbaum mit mindestens zwei Unterbäumen gilt: Die Knoten der Unterbäume erscheinen in der Reihenfolge der Unterbäume (also Knoten des n-ten Unterbaums vor Knoten des $n+1$-ten Unterbaums).

Unter der **Präfixordnung** versteht man die durch die Präfixlinearisierung gegebene Ordnung für die Knoten eines Baums.

Die **Projektion einer Liste** von Knoten erhält man, indem man jeden Knoten durch seine Marke ersetzt:

```
fun project t nss = map (label t) nss
val project : 'a tree → int list list → 'a list

project t2 [[1], [2], [3]]
[0, 4, 1] : int list
```

Die **Präfixprojektion** eines Baums ist die Projektion seiner Präfixlinearisierung. Sie kann wie folgt berechnet werden:

```
fun pre (T(x,ts)) = x :: List.concat (map pre ts)
val pre : 'a tree → 'a list

pre t2
[3, 0, 1, 2, 7, 7, 4, 1, 2, 7, 7] : int list
```

Die **Postfixlinearisierung** eines Baums ist analog zur Präfixlinearisierung definiert, nur mit dem Unterschied, dass die Wurzel eines Teilbaums diesmal nach den Knoten seiner Unterbäume erscheint. Unter der **Postfixordnung** versteht man die durch die Postfixlinearisierung gegebene Ordnung für die Knoten eines Baums. Die **Postfixprojektion** eines Baums ist die Projektion seiner Postfixlinearisierung. Sie kann wie folgt berechnet werden:

```
fun post (T(x,ts)) = List.concat (map pre ts) @ [t]
val post : 'a tree → 'a list

post t2
[2, 7, 7, 1, 0, 4, 2, 7, 7, 1, 3] : int list
```