Kapitel 1

Schnellkurs

Dieses Kapitel macht Sie mit der Programmiersprache Standard ML bekannt. Sie lernen wie man kleine Programme schreibt und diese mit einem Interpreter ausführt. Dabei machen Sie Bekanntschaft mit einer grundlegenden Programmiertechnik, dem Programmieren mit rekursiven Prozeduren. Danach beginnen wir die Struktur von Programmen und Programmiersprachen zu erkunden. Sie lernen grundlegende Begriffe der programmiersprachlichen Fachsprache kennen und bekommen eine erste Einführung in die Themenkreise Syntax und Semantik.

Ein Interpreter stellt Ihnen ein virtuelles Labor zur Verfügung, in dem Sie mit Programmen experimentieren können. Nachdem Sie ein Programm eingegeben haben, sagt Ihnen der Interpreter zunächst, ob das Programm gemäß den Regeln der Programmiersprache zulässig ist. Wenn dies der Fall ist, führt der Interpreter das Programm aus und zeigt Ihnen das Ergebnis.

Um eine Programmiersprache zu erlernen, benötigen Sie einerseits ein Medium, das Ihnen die Regeln und Prinzipien der Sprache mithilfe von Beispielen erklärt (z. B. das vorliegende Buch). Andererseits benötigen Sie ein Werkzeug, das Sie in die Lage versetzt, mit der Sprache zu experimentieren (z. B. einen Interpreter).

Gezieltes Experimentieren ist ein nützliches Vorgehen beim Erlernen einer Programmiersprache, da damit Fragen zur Funktionsweise der Sprache schnell geklärt werden können. Sie sollten alle Beispiele dieses Buchs mit einem Interpreter erproben. Nachdem Sie ein Beispiel wie angegeben ausgeführt haben, werden Ihnen sofort interessante Varianten einfallen. Erproben Sie diese Varianten mit Ihrem Interpreter, Sie werden dabei in kurzer Zeit viel lernen.

1.1 Programme

Eine Programmiersprache beschreibt eine virtuelle Welt, in der viele verschiedene Arten von Objekten existieren. Einige davon lernen Sie jetzt kennen.

Werte sind Objekte, mit denen gerechnet werden kann. Typische Beispiele für Werte sind Zahlen.

Ausdrücke dienen dazu, Werte zu beschreiben. Der Ausdruck

```
4*7+3
```

hat den Wert 31. Er ist mit den ***Konstanten*** `4`, `7` und `3` sowie den ***Operatoren*** `*` und `+` gebildet.

Bezeichner dienen als Namen für programmiersprachliche Objekte. Mit ***Deklarationen*** können Bezeichner an Objekte gebunden werden. Die Deklaration

```
val x = 4*7+3
```

bindet den Bezeichner `x` an den Wert 31. Das die Deklaration einleitenden Wort `val` ist ein sogenanntes ***Schlüsselwort***, das aus dem englischen Wort value (Wert) abgeleitet ist. Wie andere Programmiersprachen auch orientiert sich Standard ML sprachlich am Englischen.

Bezeichner, die an Werte gebunden sind, können in Ausdrücken verwendet werden. Beispielsweise liefert der Ausdruck

```
(x-29)*x
```

den Wert 62, wenn der Bezeichner `x` an den Wert 31 gebunden ist.

Ein ***Programm*** ist eine Folge von Deklarationen. Einfache Programme bestehen nur aus einer Deklaration. Hier ist ein Programm, das aus 2 Deklarationen besteht:

```
val x = 4*7+3
val y = (x-29)*x
```

Die Ausführung dieses Programms berechnet die Werte (31 und 62) der durch das Programm deklarierten Bezeichner (`x` und `y`).

Es gibt verschiedene ***Typen*** von Werten. Vorerst beschränken wir uns auf Werte des Typs *int*. Dabei handelt es sich um ganze Zahlen. Das Wort *int* soll an das englische Wort integer (ganze Zahl) erinnern.

Schließlich erwähnen wir noch eine Besonderheit von Standard ML: Als Negationszeichen für Zahlen wird das Zeichen '~' verwendet. Sie müssen also `~7` schreiben, wenn Sie -7 meinen.

1.2 Interpreter

Ein Interpreter ist ein Software-Werkzeug, mit dem die Programme einer Programmiersprache schrittweise ausgeführt werden können. Es gibt mehrere kostenlos erhältliche Interpreter für Standard ML, zum Beispiel *Moscow ML* [1] und Standard ML of New Jersey.[2] Die Beispiele dieses Buchs wurden mit Moscow ML erprobt.[3]

Nachdem Sie den Moscow ML Interpreter gestartet haben,[4] meldet er sich mit dem Text

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
-
```

Mit dem Bindestrich '`-`' am Zeilenanfang sagt der Interpreter, dass er an dieser Stelle auf die Eingabe eines Programms wartet. Das Programm kann schrittweise Deklaration für Deklaration eingegeben werden. Wir beginnen mit der Deklaration

```
val x = 4*7+3
```

Um die Deklaration an den Interpreter zu übergeben, müssen wir die Eingabe mit einem Semikolon '`;`' abschliesen und die Eingabetaste (enter) drücken:

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
- val x = 4*7+3;
```

Der Interpreter prüft zunächst, ob es sich bei der Eingabe um eine zulässige Folge von Deklarationen handelt. Wenn dies wie in unserem Beispiel der Fall ist, führt er die eingebenen Deklarationen nacheinander aus. Danach informiert er den Benutzer über die berechneten Bezeichnerbindungen

[1] Moscow ML wurde für die Ausbildung entworfen und ist für Windows und Unix verfügbar. Neben einem Interpreter stellt Moscow ML auch einen Übersetzer zur Verfügung, den wir später verwenden werden. Die Homepage für Moscow ML ist `www.dina.dk/~sestoft/mosml.html`.

[2] `cm.bell-labs.com/cm/cs/what/smlnj/`

[3] Sie können auch einen anderen Standard ML Interpreter verwenden.

[4] Der Interpreter sollte unter Emacs gestartet werden, damit Sie Ihre Eingaben bequem editieren können (siehe die Homepage von Moscow ML). Alternativ können Sie Ihr Programm auch in eine Datei `x.sml` schreiben und diese durch die Eingabe `use "x.sml";` in den Interpreter laden.

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
- val x = 4*7+3;
> val x = 31 : int
-
```

und wartet auf die nächste Eingabe. Bei der Ausgabe der Bezeichnerbindungen gibt der Interpreter auch die für die deklarierten Bezeichner ermittelten Typen an (im Beispiel `int` für `x`).

Als Nächstes geben wir die Deklaration

```
val y = (x-29)*x
```

ein. Diese verwendet den bereits deklarierten Bezeichner `x`.

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
- val x = 4*7+3;
> val x = 31 : int
- val y = (x-29)*x;
> val y = 62 : int
-
```

Jetzt geben wir ein Programm ein, das aus zwei Deklarationen besteht:

```
- val a = x-y
  val b = x+y;
> val a = ~31 : int
  val b = 93 : int
-
```

Wenn Sie wollen, können Sie weitere Deklarationen eingeben. Wenn Sie genug haben, können Sie die Sitzung mit dem Interpreter durch die Eingabe

```
quit();
```

beenden.

Im Folgenden stellen wir Interaktionen mit dem Interpreter der Einfachheit halber ohne die Hifszeichen '`-`', '`>`' und '`;`' dar:

```
val x = 4*7+3
val x = 31 : int

val y = (x-29)*x
val y = 62 : int
```

Ausgabezeilen kann man daran erkennen, dass sie in *kursiver Proportionalschrift* gesetzt sind.

Bezeichner können mehrfach deklariert werden. Gültig ist die zuletzt ausgeführte Deklaration:

```
val x = 2
```
val x = 2 : int

```
val x = 3
```
val x = 3 : int

```
val y = x*x
```
val y = 9 : int

```
val x = x*x
```
val x = 9 : int

Wir betrachten die letzte Deklaration genauer. Zuerst wird der Ausdruck `x*x` mit der bereits existierenden Bindung $x = 3$ ausgewertet. Das liefert den Wert 9. Jetzt wird der Bezeichner `x` an den Wert 9 gebunden.

Um dem Benutzer Schreibarbeit zu ersparen, bietet der Interpreter die Möglichkeit, Deklarationen des so genannten ***Ergebnisbezeichner*** `it` ohne den Vorspann `val it =` einzugeben. Statt

```
val it = 4*7+3
```
val it = 31 : int

kann man also auch

```
4*7+3
```
31 : int

eingeben. Das ist praktisch, wenn man zunächst nur den Wert eines Ausdrucks sehen will. Wenn man den Wert bei der nächsten Eingabe weiterverwenden will, kann man dies mithilfe des Ergebnisbezeichners tun:

```
4*7+3
```
31 : int

```
it+it
```
62 : int

```
it-60
```
2 : int

Der Interpreter prüft für jede Eingabe, ob sie gemäß den Regeln der Sprache zulässig ist. Unzulässige Eingaben werden mit einer Fehlermeldung beantwortet. Hier ist ein Beispiel:

```
vall x = 4;
```
! Toplevel input:
! vall x = 4;<EOF>
*! ^^^^^^^^*
! Unbound value identifier: vall

Das Wort `vall` wird vom Interpreter also als ungebundener Bezeichner erkannt. Zunächst werden die Fehlermeldungen des Interpreters für Sie weitgehend unver-

ständlich sein. Das liegt daran, dass die Ausgaben des Interpreters auf erfahrene Benutzer ausgerichtet sind, die sich mit Standard ML gut auskennen.

1.3 Vergleiche und Konditionale

Die arithmetischen Vergleiche liefern Werte des zweielementigen Typs *bool*:

```
3<3
```
false : bool

```
3<=3
```
true : bool

```
3=3
```
true : bool

```
3<>3
```
false : bool

Dabei bezeichnet `<=` den Vergleich $\leq$, und `<>` den Vergleich $\neq$. Bei den obigen Eingaben handelt es sich um Ausdrücke des Typs *bool*. Sind sind jeweils mit der Konstante `3` und einer Vergleichsoperation gebildet. Die zwei Werte des Typs *bool* heißen ***Boolesche Werte*** und werden durch die Konstanten `false` und `true` bezeichnet.

Ein ***Konditional*** ist ein Ausdruck, der eine Wenn-Dann-Sonst-Entscheidung gemäß eines Booleschen Werts realisiert:

```
if false then 3*5 else 7*1
```
7 : int

```
if true then 3*5 else 7*1
```
15 : int

Da Vergleiche Boolesche Werte liefern, können sie mit Konditionalen kombiniert werden:

```
if 4<2 then 3*5 else 7*1
```
7 : int

```
if 4=2*2 then 3*5 else 7*1
```
15 : int

Die allgemeine Form eines Konditionals ist:

`if` e_1 `then` e_2 `else` e_3

Die Teilausdrücke e_1, e_2 und e_3 werden als ***Bedingung***, ***Konsequenz*** und ***Alternative*** des Konditionals bezeichnet. Bei `if`, `then` und `else` handelt es sich um Schlüsselwörter.

1.4 Prozeduren

Eine Prozedur ist ein programmiersprachliches Objekt, das eine Funktion berechnet. Die Deklaration

```
fun quadrat (x:int) = x*x
```
val quadrat : int → int

bindet den Bezeichner `quadrat` an eine Prozedur des Typs *int → int*, die das Quadrat einer Zahl berechnet:

```
quadrat 4
```
16 : int

```
quadrat ~7
```
49 : int

Der Ausdruck `quadrat 4` beschreibt das Ergebnis, das die Prozedur `quadrat` für das Argument 4 liefert. Ausdrücke dieser Bauart heißen ***Prozeduranwendungen***.

Wir betrachten die ***Prozedurdeklaration***

```
fun quadrat (x:int) = x*x
```

jetzt genauer. Sie beginnt mit dem Schlüsselwort `fun`. Danach folgt der Bezeichner `quadrat`, an den die Prozedur gebunden wird. Das so genannte ***Argumentmuster*** `(x:int)` führt die ***Argumentvariable der Prozedur*** zusammen mit ihrem Typ ein. Nach dem Schlüsselwort `=` folgt der Ausdruck `x*x`, der das Ergebnis der Prozedur beschreibt. Man spricht vom ***Rumpf der Prozedur***.

Die obigen Beispiele zeigen, dass der Argumentausdruck einer Prozeduranwendung nicht unbedingt geklammert werden muss (`quadrat 4` statt `quadrat(4)`). Klammerung ist nur dann erforderlich, wenn der Argumentausdruck aus mehreren Teilausdrücken zusammengesetzt ist:

```
quadrat (2+3)
```
25 : int

Wenn man hier die Klammern weglässt, wird die Prozeduranwendung der Addition untergeordnet:

```
quadrat 2 + 3
```
7 : int

```
(quadrat 2) + 3
```
7 : int

Hier sind Beispiele für Ausdrücke mit zwei Prozeduranwendungen:

```
quadrat 2 + quadrat 3
```
13 : int

```
quadrat (quadrat 3)
```
81 : int

Hier ist eine Prozedur, die den Absolutbetrag einer ganzen Zahl berechnet:

```
fun betrag (x:int) = if x<0 then ~x else x
```
val betrag : int → int

```
betrag ~3
```
3 : int

Prozeduren werden oft auch als Funktionen bezeichnet. Wir verzichten auf diese Sprechweise, da es sich bei Prozeduren und mathematischen Funktionen (siehe nächstes Kapitel) um verschiedene Konzepte handelt. Der Zusammenhang ist dadurch gegeben, dass Prozeduren Funktionen berechnen.

1.5 Lokale Deklarationen und Hilfsprozeduren

Oft ist es sinnvoll, im Rumpf einer Prozedur lokale Bezeichner für Hilfswerte einzuführen, die für die Berechnung des Endergebnisses benötigt werden. Als Beispiel betrachten wir eine Prozedur, die zu einer Zahl x die Zahl x^8 berechnet. Mithilfe von lokalen Deklarationen können wir x^8 mit nur 3 Multiplikationen berechnen:

```
fun hoch8 (x:int) =
    let
        val a = x*x
        val b = a*a
    in
        b*b
    end
```
val hoch8 : int → int

Alternativ können wir die Prozedur `hoch8` auch mit einer Hilfsprozedur deklarieren:

```
fun hoch8 (x:int) =
    let
        fun q (y:int) = y*y
    in
        q (q (q x))
    end
```
val hoch8 : int → int

Wenn wir wollen, können wir die Hilfsprozedur `q` auch unabhängig von der Prozedur `hoch8` deklarieren:

```
fun q (y:int) = y*y
fun hoch8 (x:int) = q (q (q x))
```

1.6 Tupel

Tupel sind Werte, die mehrere Werte zusammenfassen. Wenn wir n Werte $v_1, \ldots, v_n$ haben, können wir sie zu einem ***n-stelligen Tupel***

$(v_1, \ldots, v_n)$

zusammenfassen. Das i-te Teilobjekt v_i eines Tupels bezeichnen wir als seine ***i-te Komponente***. Zweistellige Tupel bezeichnen wir als ***Paare***, und dreistellige als ***Tripel***.

Der Ausdruck

```
(5-2, 1<2, 2*2)
```
*(3, true, 4) : int * bool * int*

beschreibt ein dreistelliges Tupel. Der Typ des Tupels ist $int * bool * int$ und ergibt sich aus den Typen der die Komponenten beschreibenden Teilausdrücke. Auf die Komponenten eines Tupels kann man mit Hilfe von sogenannten ***Projektionen*** zugreifen:

```
#3 it
```
4 : int

```
#2 (3, true, 7)
```
true : bool

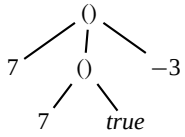
```
#1 (3, true, 7)
```
3 : int

Die Komponenten eines Tupels können wieder Tupel sein:

```
(7, (2,true), ~3)
```
*(7, (2,true), ~3) : int * (int * bool) * int*

```
#1 (#2 it)
```
2 : int

Grafisch können wir das obige Tupel wie folgt darstellen:



Man spricht von der ***Baumdarstellung*** des Tupels.

Es gibt auch das ***leere Tupel***:

```
()
() : unit
```

Dieses hat den einelementigen Typ *unit*.

Die Komponenten eines Tupels können wir wie folgt an Bezeichner binden:

```
val (x,y) = (1,true)
val x = 1 : int
val y = true : bool
```

Diese Deklaration bindet `x` an die erste und `y` an die zweite Komponente des Paares (1, *true*). Die Deklaration

```
val (x,y) = (y,x)
val x = true : bool
val y = 1 : int
```

vertauscht die Bindungen der Bezeichner `x` und `y`.

1.7 Prozeduren und Tupel

Tupel können als Argumente und Ergebnisse von Prozeduren verwendet werden. Die Prozedur

```
fun paar (x:int) = (x, x*x)
val paar : int → int * int
```

liefert zu einer ganzen Zahl x das Paar aus x und x^2:

```
paar 5
(5, 25) : int * int
```

Die Prozedur

```
fun min (x:int, y:int) = if x<y then x else y
val min : int * int → int
```

liefert zu einem Paar von ganzen Zahlen die kleinere Zahl:

```
min (7,5)
5 : int
```

Beachten Sie, dass wir die Prozedur `min` mithilfe von zwei Argumentvariablen `x` und `y` deklariert haben. Wenn man will, kann man `min` als eine Prozedur mit zwei Argumenten auffassen. Das ändert aber nichts an der Tatsache, dass `min`

immer auf ein Paar angewendet wird. Dieser Sachverhalt wird durch das folgende Beispiel bestätigt:

```
min (paar 4)
```
4 : int

Wenn Sie wollen, können sie die Prozedur `min` mithilfe von Projektionen mit nur einer Argumentvariablen formulieren:

```
fun min (p: int*int) = if #1p < #2p then #1p else #2p
```
*val min : int * int → int*

Die Prozedur

```
fun swap (x:int, y:bool) = (y, x)
```
*val swap : int * bool → bool * int*

vertauscht die Komponenten eines Paars:

```
swap (4,false)
```
*(false, 4) : bool * int*

```
swap (swap (4,false))
```
*(4, false) : int * bool*

Die Prozedur

```
fun sec (h:int,m:int,s:int) = 3600*h + 60*m + s
```
*val sec : int * int * int → int*

rechnet eine mit Stunden, Minuten und Sekunden angegebene Zeitdauer in Sekunden um:

```
sec(1,1,6)
```
3666 : int

Für eine Umkehrprozedur

`hms : int → int * int * int`

die eine in Sekunden gegebene Zeitdauer in eine mit Stunden, Minuten und Sekunden gegebene Zeitdauer übersetzt, benötigen wir die Operatoren `div` und `mod`. Mit `div` bekommen wir den ganzzahligen Quotient von zwei ganzen Zahlen, und mit `mod` den dazu passenden Rest:

```
17 div 6
```
2 : int

```
17 mod 6
```
5 : int

Jetzt können wir die gewünschte Prozedur wie folgt deklarieren:

```
fun hms (s:int) =
    let
        val h  = s div 3600
        val m  = (s mod 3600) div 60
        val s' = s mod 60
    in
        (h,m,s')
    end
```
*val hms : int → int * int * int*

Der Übersichtlichkeit halber führen wir drei Hilfsbezeichner `h`, `m` und `s'` ein. Den Bezeichner `s'` spricht man „s Strich“ aus. Wir testen unsere Prozedur mit zwei Beispielen:

```
hms(sec(2,15,33))
```
*(2, 15, 33) : int * int * int*

```
sec(hms 4500)
```
4500 : int

1.8 Rekursive Prozeduren

Wir wollen jetzt eine Prozedur

*potenz : int * int → int*

schreiben, die zu einer ganzen Zahl x und einer natürlichen Zahl n die Potenz x^n berechnet. Die Prozedur soll die Potenz durch wiederholte Multiplikation mit x berechnen.

Unsere Berechnungsidee für Potenzen können wir mit zwei Gleichungen formulieren. Mit der Gleichung

$$x^n = x \cdot x^{n-1} \qquad \text{falls } n > 0$$

können wir größere Potenzen auf kleinere zurückführen. Die Gleichung

$$x^0 = 1$$

liefert uns den Wert trivialer Potenzen. Das genügt, um eine Potenz x^n Schritt für Schritt auf Multiplikationen zurückzuführen. Wir zeigen das am Beispiel von x^3:

$$x^3 \;\to\; x \cdot x^2 \;\to\; x \cdot (x \cdot x^1) \;\to\; x \cdot (x \cdot (x \cdot x^0)) \;\to\; x \cdot (x \cdot (x \cdot 1))$$

Die ersten drei Schritte verkleinern den Exponenten der Potenz mithilfe der ersten Gleichung. Der vierte und letzte Schritt eliminiert die triviale Potenz gemäß der zweiten Gleichung.

```
potenz(4,2) → if 2>0 then 4 * potenz(4, 2-1) else 1
            → if true then 4 * potenz(4, 2-1) else 1
            → 4 * potenz(4, 2-1)
            → 4 * potenz(4,1)
            → 4 * (if 1>0 then 4 * potenz(4, 1-1) else 1)
            → 4 * (if true then 4 * potenz(4, 1-1) else 1)
            → 4 * (4 * potenz(4, 1-1))
            → 4 * (4 * potenz(4,0))
            → 4 * (4 * (if 0>0 then 4 * potenz(4, 0-1) else 1))
            → 4 * (4 * (if false then 4 * potenz(4, 0-1) else 1))
            → 4 * (4 * 1)
            → 4 * 4
            → 16
```

Abbildung 1.1: Ein Auswertungsprotokoll

Damit können wir jede Potenz rein mechanisch berechnen. Wir zeigen das am Beispiel von 2^3:

$$2^3 \rightarrow 2 \cdot 2^2 \rightarrow 2 \cdot (2 \cdot 2^1) \rightarrow 2 \cdot (2 \cdot (2 \cdot 2^0)) \rightarrow 2 \cdot (2 \cdot (2 \cdot 1))$$
$$\rightarrow 2 \cdot (2 \cdot 2) \rightarrow 2 \cdot 4 \rightarrow 8$$

Wir realisieren unsere Berechnungsidee mit der folgenden Prozedurdeklaration:

```
fun potenz (x:int, n:int) : int =
    if n>0 then x*potenz(x,n-1) else 1
```

Die Wahl der richtigen Gleichung erfolgt mithilfe eines Konditionals. Die Tatsache, dass die Prozedur `potenz` mithilfe einer ***Selbstapplikation*** definiert ist, wird als ***Rekursion*** bezeichnet. Entsprechend sagt man, dass `potenz` eine ***rekursive Prozedur*** ist. Bei rekursiven Prozeduren deklarieren wir den ***Ergebnistyp*** nach dem Argumentmuster, damit der Typ des Prozedurbezeichners unabhängig vom Prozedurrumpf bestimmt ist (in dem er ja vorkommt).

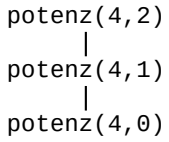
Das in Abbildung 1.1 gezeigte ***Auswertungsprotokoll*** zeigt Schritt für Schritt, wie der Wert der Prozeduranwendung `potenz(4,2)` mechanisch berechnet werden kann. Die Essenz des Auswertungsprotokolls kann wie folgt dargestellt werden:

```
potenz(4,2)  →+  4 * potenz(4,1)
             →+  4 * (4 * potenz(4,0))
             →+  4 * (4 * 1)
             →+  16
```

Das Zeichen $\rightarrow^+$ stellt dabei einen oder mehrere Auswertungsschritte dar.

Unter einem ***Prozeduraufruf*** verstehen wir ein Paar aus einer Prozedur und einem als Argument dienenden Wert. Für die Auswertung eines Aufrufs einer re-

kursiven Prozedur müssen in der Regel weitere Aufrufe der Prozedur ausgewertet werden. Die daraus resultierende Abfolge von Aufrufen kann durch einen ***Rekursionsbaum*** dargestellt werden:



Damit die Aufruffolgen einer rekursiven Prozedur nach endlich vielen Schritten abbrechen, muss die Selbstapplikation der Prozedur durch ein Konditional gesteuert werden. Betrachten Sie dazu nochmals die Deklaration

```
fun potenz (x:int, n:int) : int =
    if n>0 then x*potenz(x,n-1) else 1
```

Die Auswertung eines Konditionals hat die wichtige Eigenschaft, das seine Konsequenz nur dann ausgewertet wird, wenn die Auswertung der Bedingung *true* liefert. Entsprechend wird die Alternative des Konditionals nur dann ausgewertet, wenn die Auswertung der Bedingung *false* liefert.

Rekursion ist eine grundlegende Berechnungstechnik. Mit rekursiven Prozeduren können wir viele interessante Funktionen berechnen, zum Beispiel die Quersumme einer Zahl oder der größte gemeinsame Teiler zweier Zahlen (siehe Übungsaufgaben).

Die Formulierung rekursiver Prozeduren erfordert Übung. Ausgangspunkt ist die zu berechnende Funktion. Benötigt werden Gleichungen, mithilfe derer Anwendungen der Funktion von größeren zu kleineren Argumenten reduziert werden können. Für die Potenzfunktion leistet dies die Gleichung

$$x^n = x \cdot x^{n-1} \qquad \text{falls } n > 0$$

Außerdem benötigt man Gleichungen, die die Werte der Funktion für nicht weiter reduzierbare Argumente liefern. Für unser Beispiel erfüllt die Gleichung

$$x^0 = 1$$

diesen Zweck. Nachdem die passenden Gleichungen gefunden sind, ist die Formulierung einer entsprechenden Prozedurdeklaration einfach. Abhängig vom Prozedurargument entscheidet man mithilfe von Konditionalen, welche der Gleichungen zur Anwendung kommen soll.

1.9 Prozeduren und Umgebungen

Wir betrachten das folgende Programm:

```
fun p (x:int) = x
fun q (x:int) = 3+(p x)
fun p (x:int) = 2*x
val y = q 5
```

An welchen Wert wird y gebunden? An 8 oder an 13?

Der Rumpf `3+(p x)` der an den Bezeichner `q` gebundenen Prozedur appliziert die an den Bezeichner `p` gebundene Prozedur. Die Frage ist: Welche der zwei Bindungen für `p` ist maßgeblich?

Hier ist die Antwort: Maßgeblich ist die Bindung von `p`, die bei der Ausführung der Deklaration von `q` gültig war. Das ist die durch die erste Deklaration bewirkte Bindung. Also wird `y` an 8 gebunden.

Wir merken uns, dass eine Prozedur ein unveränderliches und in sich abgeschlossenes Objekt ist, so wie wir das von Zahlen her kennen. Für sich alleine betrachtet ist die Deklaration

```
fun q (x:int) = 3+(p x)
```

keine vollständige Beschreibung einer Prozedur, da sie von dem „externen“ Bezeichner `p` abhängt. Erst nachdem wir eine passende Bindung für `p` vorgeben, beschreibt die Deklaration eine Prozedur. Wir sagen, dass `p` ein ***freier Bezeichner*** der Deklaration ist.

Generell versteht man unter den ***freien Bezeichnern einer Deklaration*** diejenigen Bezeichner, die in der Deklaration ein Auftreten haben, das nicht im Rahmen der Deklaration gebunden ist. Deklarationen ohne freie Bezeichner bezeichnen man als ***geschlossen***, und Deklarationen mit freien Bezeichnern als ***offen***. Die obigen Deklarationen für `p` sind geschlossen, die Deklaration für `q` ist offen.

Hier ist ein interessantes Beispiel für eine offene Deklaration:

```
val x = x
```

Diese Deklaration hat den freien Bezeichner `x`, da die durch die Deklaration eingeführte Bindung nicht innerhalb der Deklaration gilt. Dagegen ist die Deklaration

```
fun f (x:int) : int = if x<1 then 1 else x*f(x-1)
```

geschlossen, da die durch die Deklaration eingeführte Bindungen für `f` auch innerhalb der Deklaration gilt. Dieses Bindungsverhalten ermöglicht die Definition rekursiver Prozeduren.

Eine ***Umgebung*** ist eine Sammlung von Bezeichnerbindungen, sodass keinem Bezeichner mehr als ein Wert zugeordnet wird. Beispielsweise ***bindet*** die Umgebung

$\{\mathsf{x} \mapsto 5,\ \mathsf{y} \mapsto 7\}$

den Bezeichner `x` an den Wert 5 und den Bezeichner `y` an den Wert 7. Mit diesen Bezeichnerbindungen beschreibt der Ausdruck `x+y` den Wert 12. Wir reden vom Wert eines Ausdrucks ***in einer Umgebung***.

Prozeduren können mithilfe einer Prozedurdeklaration und einer Umgebung, die die freien Bezeichner der Deklaration bindet, vollständig beschrieben werden. Beispielsweise können wir die durch die Deklarationen

```
fun p (x:int) = x
fun q (x:int) = 3+(p x)
```

deklarierten Prozeduren wie folgt beschreiben:

(*fun p (x:int) = x*, {})

(*fun q (x:int) = 3+(p x)*, { *p* ↦ (*fun p (x:int) = x*, {})})

Die Auswertung eines Programms liefert Bezeichnerbindungen, die durch eine Umgebung dargestellt werden können. Beispielsweise berechnet das Programm

```
fun p (x:int) = x
fun q (x:int) = 3+(p x)
fun p (x:int) = 2*x
val y = q 5
```

die Umgebung

{ *p* ↦ (*fun p (x:int) = 2*x*, {}) ,

q ↦ (*fun q (x:int) = 3+(p x)*, { *p* ↦ (*fun p (x:int) = x*, {})}) ,

y ↦ 8 }

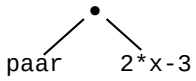
1.10 Syntax

Nach welchen Regeln werden Ausdrücke und Deklarationen gebildet? Wie werden Ausdrücke und Deklarationen mit Wörtern beschrieben? Welche Wörter gibt es? Wie werden Wörter mit Zeichen beschrieben? Diese Fragen gehören zu einem Themenkreis, den man als *Syntax* bezeichnet.

Zunächst wollen wir uns genauer ansehen, wie Ausdrücke aufbebaut sind. Als Beispiel betrachten wir den Ausdruck

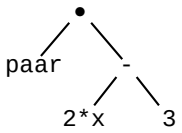
```
paar(2*x-3)
```

Es handelt sich um eine Prozeduranwendung, die aus dem Bezeichner `paar` und dem Ausdruck `2*x-3` gebildet ist. Grafisch können wir die Prozeduranwendung wie folgt darstellen:

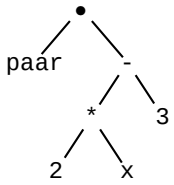


Die zwei Objekte, aus denen die Prozeduranwendung gebildet ist, werden als ihre ***Konstituenten*** bezeichnet.

Die rechte Konstituente der Prozeduranwendung ist wiederum ein zusammengesetzter Ausdruck. Es handelt sich um eine Differenz, die aus dem Ausdruck `2*x`, dem Operator `'-'` und der Konstante `3` gebildet ist:



Die linke Konstituente der Differenz ist ein Produkt, das aus der Konstante `2`, dem Operator `*` und dem Bezeichner `x` gebildet ist:



Wir haben jetzt die ***Baumdarstellung*** des Ausdrucks `paar(2*x-3)` vorliegen. Diese zeigt, wie der Ausdruck Schritt für Schritt aus einfacheren Ausdrücken konstruiert ist. Am Anfang der Konstruktion stehen ***atomare Ausdrücke*** (die Bezeichner `paar` und `x` sowie die Konstanten `2` und `3`). Aus diesen werden mithilfe der ***Formen*** für Operator- und Prozeduranwendungen ***zusammengesetzte Ausdrücke*** gebildet.

Ausdrücke sind abstrakte Objekte, die auf verschiedene Weise dargestellt werden können. Die übliche Darstellung durch Zeichen wird als ***Zeichendarstellung*** bezeichnet. Sie wird für die Interaktion mit dem Interpreter verwendet. Der Interpreter übersetzt die eingegebenen Zeichendarstellungen in Baumdarstellungen und benutzt diese als Grundlage für die nachfolgenden Verarbeitungsschritte.

```
if x*y<15then(x,2+y)else paar(2*x-3)

if x * y < 15 then ( x , 2 + y ) else paar ( 2 * x - 3 )
```

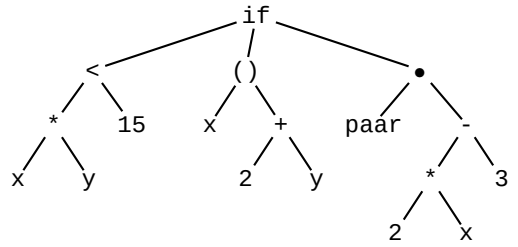


Abbildung 1.2: Zeichen-, Wort- und Baumdarstellung eines Ausdrucks

Als Zwischenstufe für den Übergang von der Zeichen- zur Baumdarstellung gibt es die so genannte ***Wortdarstellung***. Diese beschreibt einen Ausdruck durch einer Folge von Wörtern. Die Wortdarstellung unseres Beispielausdrucks `paar(2*x-3)` besteht aus 8 Wörtern:

```
paar  (  2  *  x  -  3  )
```

Die Zeichendarstellung kann die dargestellten Wörter durch Leerzeichen trennen. Das obige Beispiel zeigt, dass in vielen Fällen keine Trennung durch Leerzeichen erforderlich ist. Hier sind zwei Beispiele, bei denen Leerzeichen erforderlich sind:

```
paar 2
4+ ~5
```

So wie es dasteht, beschreibt das erste Beispiel eine Prozeduranwendung. Wenn wir das Leerzeichen weglassen, bekommen wir den Bezeichner `paar2` (nur ein Wort). Das zweite Beispiel beschreibt eine Summe. Wenn wir hier das Leerzeichen weglassen, bekommen wir die Wortfolge

```
4  +~  5
```

Diese kann nur dann als Darstellung eines Ausdrucks interpretiert werden, wenn das Wort `+~` zuvor als Operator eingeführt wurde.

Abbildung 1.2 zeigt die Zeichen-, Wort- und Baumdarstellung eines größeren Ausdrucks. Die Zeichendarstellung verwendet nur da Leerzeichen, wo sie erforderlich sind.

Offensichtlich ist die Baumdarstellung eines Ausdrucks einfacher zu verstehen als die Zeichen- oder Wortdarstellung. Das liegt daran, dass die Baumdarstellung den Aufbau eines Ausdrucks grafisch darstellt. Aus der Zeichendarstellung muss der Aufbau des Ausdrucks dagegen erst gemäß den Regeln der Sprache erschlossen

```
fun min (x:int, y:int) = if x<y then x else y
```

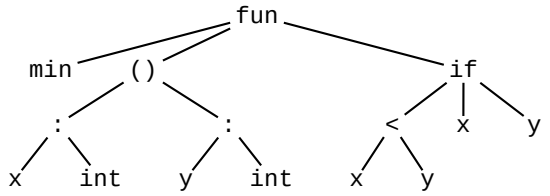


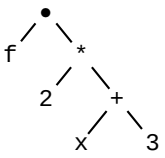
Abbildung 1.3: Zeichen- und Baumdarstellung einer Prozedurdeklaration

werden. Bei dieser nicht immer leichten Aufgabe profitieren Sie von Ihrer Erfahrung im Umgang mit den Zeichendarstellungen mathematischer Formeln.

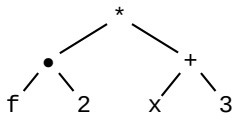
Syntaktische Objekte wie Ausdrücke, Deklarationen, Programme und Typen bezeichnen wir übergreifend als ***Phrasen***. Unsere Bemerkungen zum Aufbau und zur Darstellung von Ausdrücken gelten sinngemäß auch für andere Phrasen. Abbildung 1.3 zeigt die Zeichen- und Baumdarstellung einer Prozedurdeklaration.

1.11 Klammern

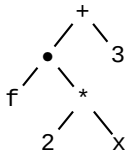
Eine Folge von Wörtern, die einen Ausdruck oder einen Typ beschreibt, kann stets in Klammern eingeschlossen werden. Erst die dadurch gegebene Gruppierungsmöglichkeit versetzt uns in die Lage, jede Phrase durch eine Wortdarstellung zu beschreiben. Hier sind drei Beispiele:




```
f(2*(x+3))
```




```
(f 2)*(x+3)
```




```
f(2*x)+3
```

Wenn Sie wollen, können Sie überflüssige Klammern setzen. Beispielsweise beschreiben die Zeichendarstellungen

```
2*f 2+3    2*(f 2)+3    (2*(f 2))+3    ((2*(f 2))+3)
```

alle den gleichen Ausdruck.

Standard ML ist mit diversen Klammersparregeln ausgestattet. Hier sind Beispiele

für die wichtigsten:

`3*4+5`	$\rightsquigarrow$	`(3*4)+5`	Punkt vor Strich
`2+3+4`	$\rightsquigarrow$	`2+(3+4)`	Operatoranwendung gruppiert nach rechts
`f 3+4`	$\rightsquigarrow$	`(f 3)+4`	Prozeduranwendung vor Operatoranwendung
`f g 3`	$\rightsquigarrow$	`(f g)3`	Prozeduranwendung gruppiert nach links

Die letzte Regel wird Ihnen erst dann einleuchten, wenn Sie mehr von Standard ML gesehen haben (siehe kaskadierte Prozeduren in Kapitel 3). Wir erwähnen sie aber bereits hier, damit Sie nicht zu viele Klammern weglassen. Wenn Sie es versehentlich doch tun, wird der Interpreter dies mit vorerst undurchschaubare Fehlermeldungen quittieren.

Merken Sie sich die folgenden Tatsachen zur Darstellung von Phrasen:

1. Jede Phrase hat genau eine Baumdarstellung.
2. Eine Phrase kann mehrere Wortdarstellungen haben, die sich durch das Weglassen oder Hinzufügen von Klammerpaaren unterscheiden.
3. Die Wortdarstellungen von verschiedenen Phrasen der gleichen Art (z. B. Ausdrücke) sind verschieden.

Seien e_1 und e_2 zwei Ausdrücke. Aussage (3) besagt, dass keine Wortdarstellung von e_1 eine Wortdarstellung von e_2 ist, und dass keine Wortdarstellung von e_2 eine Wortdarstellung von e_1 ist.

1.12 Syntaxübersicht

Hier ist eine Übersicht über die syntaktischen Objekte der bisher vorgestellten Teilsprache von Standard ML.

1.12.1 Wörter

Wir haben die folgenden Wortarten kennen gelernt:

- **Konstanten.** Beispiele für Konstanten sind `3`, `~56`, `false`, `true` und `()`.
- **Operatoren.** Beispiele für zweistellige Operatoren sind `+`, `-`, `*`, `<`, `<=`, `=`, `<>`, `div` (ganzzahlige Division) und `mod` (ganzzahliger Rest). Der Negationsoperator `~` ist einstellig.

- **Schlüsselwörter.** Beispiele sind `val`, `fun`, `if`, `then`, `else`, `let`, `in`, `end`, `int`, `bool`, `unit`, `=`, `(`, `)`, `,`, `:`, `->`, `*` und #.
- **Bezeichner.** Bezeichner sind Wörter, die aus Buchstaben, Ziffern sowie den Zeichen '_' und `''` gebildet sind. Sie müssen mit einem Buchstaben beginnen. Außerdem gelten nur solche Wörter als Bezeichner, die nicht zu einer der bereits erwähnten Wortarten gehören.

Die Wörter `=` und `*` spielen Doppelrollen, da sie als zweistellige Operatoren oder als Schlüsselwörter auftreten können. Betrachten Sie dazu die Deklaration

```
fun f (t : int*int*int) = if #1t=0 then #2t else 2*(#3t)
```

Vor dem Schlüsselwort `if` treten `=` und `*` als Schlüsselwörter auf, danach als Operatoren.

1.12.2 Phrasen

Wir haben die folgenden Phrasenarten kennen gelernt: Ausdrücke, Deklarationen, Programme, Typen und Argumentmuster.

1.12.3 Ausdrücke

Wir haben die folgenden Ausdrucksarten kennen gelernt:

- **Atomare Ausdrücke.** Ausdrücke, die nur aus einer Konstanten oder nur aus einem Bezeichner bestehen.
- **Applikationen**

  ⟨*einstelliger Operator*⟩ ⟨*Ausdruck*⟩

  ⟨*Ausdruck*⟩ ⟨*zweistelliger Operator*⟩ ⟨*Ausdruck*⟩

  # ⟨*Konstante für positive ganze Zahl*⟩ ⟨*Ausdruck*⟩

  ⟨*Ausdruck*⟩ ⟨*Ausdruck*⟩

  Die mit # beginnenden Applikationen heißen *Projektionen*, und die gemäß der letzten Form gebildeten Applikationen heißen *Prozeduranwendungen.*
- **Konditionale**

  `if` ⟨*Ausdruck*⟩ `then` ⟨*Ausdruck*⟩ `else` ⟨*Ausdruck*⟩

  Der erste Teilausdruck eines Konditionals wird als *Bedingung*, der zweite als *Konsequenz*, und der dritte als *Alternative* bezeichnet.

- **Tupelausdrücke**

  (⟨*Ausdruck*⟩ , ... , ⟨*Ausdruck*⟩)

- **Let-Ausdrücke**

  let ⟨*Deklaration*⟩ ... ⟨*Deklaration*⟩ in ⟨*Ausdruck*⟩ end

1.12.4 Deklarationen

Wir haben die folgenden Deklarationsformen kennen gelernt:

val ⟨*Bezeichner*⟩ = ⟨*Ausdruck*⟩

val (⟨*Bezeichner*⟩ , ... , ⟨*Bezeichner*⟩) = ⟨*Ausdruck*⟩

fun ⟨*Bezeichner*⟩ ⟨*Argumentmuster*⟩ = ⟨*Ausdruck*⟩

fun ⟨*Bezeichner*⟩ ⟨*Argumentmuster*⟩ : ⟨*Typ*⟩ = ⟨*Ausdruck*⟩

Die mit fun beginnenden Deklarationen heißen *Prozedurdeklarationen.* Der Ausdruck einer Prozedurdeklaration wird als *Prozedurrumpf* bezeichnet, und die im Argumentmuster vorkommenden Bezeichner als *Argumentvariablen.* Der bei der Deklaration von rekursive Prozeduren nach dem Argumentmuster erscheinende Typ wird als *Ergebnistyp* bezeichnet.

1.12.5 Programme

Eine Programm ist eine Folge von Deklarationen.

1.12.6 Argumentmuster

Argumentmuster haben die folgende Form:

(⟨*Bezeichner*⟩ : ⟨*Typ*⟩ , ... , ⟨*Bezeichner*⟩ : ⟨*Typ*⟩)

1.12.7 Typen

Wir haben die folgenden Typen kennen gelernt:

- **Atomare Typen.** int, bool, unit

- **Prozedurtypen.** ⟨*Typ*⟩ `->` ⟨*Typ*⟩
- **Tupeltypen.** ⟨*Typ*⟩ `*` `...` `*` ⟨*Typ*⟩

1.13 Semantische Zulässigkeit

Bevor ein Interpreter ein Programm ausführt, prüft er zunächst, ob das Programm ***semantisch zulässig*** ist. Die damit verbundenen Bedingungen betreffen die Bindung von Bezeichnern und den typgerechten Aufbau von Ausdrücken.

Zunächst muss ein semantisch zulässiges Programm die Bedingung erfüllen, dass jedes in einem Ausdruck des Programms vorkommende Auftreten eines Bezeichners durch eine Deklaration gebunden ist. Beispielsweise ist das Programm

```
fun f (x:int) : int = if x<y then 1 else x*f(x-1)
```

für sich alleine betrachtet unzulässig, da das Auftreten des Bezeichners `y` in der Bedingung des Konditionals ungebunden ist. Dagegen sind die Auftreten der Bezeichner `f` und `x` durch die umfassende Deklaration gebunden. Wenn wir eine Deklaration für `y` hinzufügen

```
val y = 1
fun f (x:int) : int = if x<y then 1 else x*f(x-1)
```

bekommen wir ein Programm, in dem alle Bezeichnerauftreten ordnungsgemäß gebunden sind.

Die Regeln für den typgerechten Aufbau von Ausdrücken stellen sicher, dass Operationen bei der Ausführung nur auf typgerechte Argumente angewendet werden. Damit wird beispielsweise ausgeschlossen, dass eine Addition auf ein Tupel oder eine Projektion auf eine Zahl angewendet wird.

Typgerecht aufgebaute Phrasen bezeichnen wir auch als ***wohlgetypt***. Ein Ausdruck ist genau dann wohlgetypt, wenn ihm ein Typ zugeordnet werden kann. Dabei müssen für die im Ausdruck vorkommenden Bezeichner Typen vorgegeben werden. Hier ist die ***Typregel*** für die Anwendung von zweistelligen Operatoren:

$$\frac{e_1 : t_1 \qquad o : t_1 * t_2 \to t \qquad e_2 : t_2}{e_1\ o\ e_2\ :\ t}$$

Diese Regel besagt, dass eine Applikation $e_1\ o\ e_2$ wohlgetypt ist und den Typ t hat, wenn die folgenden Bedingungen erfüllt sind:

1. Der linke Teilausdruck e_1 hat den Typ t_1.
2. der Operator o hat den Typ $t_1 * t_2 \to t$.

3. Der rechte Teilausdruck e_2 hat den Typ t_2.

Als Beispiel betrachten wir den Ausdruck `x+5` und nehmen an, dass der Bezeichner `x` den Typ `int` hat. Da der Operator `+` den Typ $\texttt{int} * \texttt{int} \to \texttt{int}$ und die Konstante `5` den Typ `int` hat, folgt mit der obigen Typregel, dass der Ausdruck wohlgetypt ist und den Typ `int` hat.

Wir werfen noch einen Blick auf die Typregel für Konditionale:

$$\frac{e_1 : \texttt{bool} \quad e_2 : t \quad e_3 : t}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ :\ t}$$

Die Regel verlangt, dass die Bedingung des Konditionals den Typ `bool` hat. Weiter verlangt die Regel, dass die Konsequenz und die Alternative des Konditionals den gleichen Typ haben. Wenn diese Bedingungen erfüllt sind, ordnet die Regel dem Konditional den gemeinsamen Typ der Konsequenz und Alternative zu.

1.14 Verabeitungsphasen

Wir wollen uns jetzt genauer ansehen, wie ein Interpreter die Eingaben eines Benutzers verarbeitet. Dabei werden sich die bisher diskutierten Teilaspekte zu einem sinnvollen Ganzen zusammenfügen.

Als Eingabe bekommt ein Interpreter eine Folge von Zeichen. Zunächst prüft der Interpreter, ob es sich dabei um die Darstellung eines semantisch zulässigen Programms handelt. Wenn dies der Fall ist, führt der Interpreter das Programm aus und teilt dem Benutzer die dadurch berechneten Ergebnisse mit.

Bei genauerer Betrachtung unterscheiden wir vier aufeinanderfolgende Verarbeitungsphasen des Interpreters:

1. ***Lexikalische Analyse.*** Prüfe, ob die eingegebene Zeichenfolge als eine Folge von Wörtern aufgefasst werden kann.

2. ***Syntaktische Analyse.*** Prüfe, ob die von der lexikalischen Analyse gelieferte Wortfolge ein Programm beschreibt. Statt von syntaktischer Analyse spricht man auch von *Parsing.*

3. ***Semantische Analyse.*** Prüfe, ob das von der syntaktischen Analyse gelieferte Programm semantisch zulässig ist. Statt von semantischer Analyse spricht man auch von *Elaboration.*

4. ***Ausführung.*** Führe das Programm aus. Statt von Ausführung spricht man auch von *Auswertung* oder von *Evaluation.*

In jeder Phase könnenn Fehler entdeckt werden. Eine Phase wird nur dann ausgeführt, wenn bis dahin keine Fehler entdeckt wurden.

Man unterscheidet zwischen statischen und dynamischen Aspekte von Programmiersprachen. Die die Analysephasen betreffenden Aspekte bezeichnet man als ***statisch***, und die die Ausführungsphase betreffenden als ***dynamisch***.

Die ***Syntax einer Programmiersprache*** legt die Regeln für die lexikalische und syntaktische Analyse fest. Entsprechend unterscheidet man zwischen der lexikalischen und der phrasalen Syntax einer Programmiersprache. Die ***lexikalische Syntax*** regelt, wie Wörter aus Zeichen gebildet werden, und die ***phrasale Syntax*** regelt, wie Phrasen aus Wörtern gebildet werden.

Die ***Semantik einer Programmiersprache*** legt die Regeln für die semantische Analyse und die Ausführung fest. Entsprechend unterscheidet man zwischen der statischen und der dynamischen Semantik einer Programmiersprache. Die ***statische Semantik*** formuliert die Bedingungen, die semantisch zulässige Programme erfüllen müssen. Die ***dynamische Semantik*** beschreibt, welche Ergebnisse die Ausführung von Programmen liefern soll.

Man unterscheidet zwischen ***syntaktischen und semantischen Objekten.*** Wörtern und Phrasen sind syntaktische Objekte, Werte und Umgebungen sind semantische Objekte.

1.15 Auswertung

Wir wollen jetzt genauer auf die dynamische Semantik der bisher vorgestellten Programmiersprache eingehen. Wir tun das, indem wir sagen, wie die folgenden Auswertungsprobleme gelöst werden können:

- *Auswertung von Ausdrücken.* Gegeben einen Ausdruck e und eine Umgebung V, bestimme den Wert, den e für V liefert.
- *Auswertung von Deklarationen.* Gegeben eine Deklaration d und eine Umgebung V, bestimme die Umgebung, die d für V liefert.
- *Auswertung von Prozeduraufrufen.* Gegeben eine Prozedur p und einen Wert v, der als Argument für p zulässig ist, bestimme den Wert, den p für v liefert.

Zwischen diesen Problemen besteht eine zirkuläre Abhängigkeit, da Ausdrücke Deklarationen und Prozeduranwendungen enthalten können.

Bisher haben wir die folgenden Werte kennengelernt:

- Ganze Zahlen.
- Die zwei Booleschen Werte *false* und *true*.
- Prozeduren.
- Tupel.

Auswertung von Ausdrücken

Die Auswertung eines Ausdrucks in einer Umgebung erfolgt je nach Form des Ausdrucks gemäß einer der folgenden Regeln:

1. Die Auswertung eines Ausdrucks, der nur aus einer Konstante besteht, liefert den durch die Konstante bezeichneten Wert.
2. Die Auswertung eines Ausdrucks, der nur aus einem Bezeichner besteht, liefert den von der vorliegenden Umgebung für den Bezeichner vorgegebenen Wert.
3. Die Auswertung einer Operatoranwendung $o\ e$ beginnt mit der Auswertung des Teilausdrucks e. Sie liefert den Wert, den die durch den einstelligen Operator o bezeichnete Operation für den Wert von e liefert.
4. Die Auswertung einer Operatoranwendung $e_1\ o\ e_2$ beginnt mit der Auswertung der Teilausdrücke e_1 und e_2. Sie liefert den Wert, den die durch den zweistelligen Operator o bezeichnete Operation für die Werte von e_1 und e_2 liefert.
5. Die Auswertung einer Projektion $\#k\ e$ beginnt mit der Auswertung des Teilausdrucks e. Wenn dieser ein Tupel mit mindestens k Komponenten liefert, liefert die Projektion die k-te Komponente dieses Tupels.
6. Die Auswertung einer Prozeduranwendung $e_1\ e_2$ beginnt mit der Auswertung der Teilausdrücke e_1 und e_2. Wenn diese die Prozedur p und den Wert v liefern, wird der Prozeduraufruf $p\ v$ ausgewertet. Die Applikation liefert den so erhaltenen Wert.
7. Die Auswertung eines Konditionals `if` e_1 `then` e_2 `else` e_3 beginnt mit der Auswertung des Teilausdrucks e_1. Wenn e_1 den Wert *true* liefert, wird der Teilausdruck e_2 ausgewertet und das Konditional liefert den so erhaltenen Wert. Wenn e_1 den Wert *false* liefert, wird der Teilausdruck e_3 ausgewertet und das Konditional liefert den so erhaltenen Wert.
8. Die Auswertung eines Tupelausdrucks $(e_1, \ldots, e_n)$ beginnt mit der Auswertung der Teilausdrücke $e_1, \ldots, e_n$. Wenn diese die Werte $v_1, \ldots, v_n$ liefern, liefert der Tupelausdruck das Tupel $(v_1, \ldots, v_n)$.

9. Die Auswertung eines Let-Ausdrucks `let` $d_1 \ldots d_n$ `in` e `end` beginnt mit der Auswertung der Deklarationen $d_1, \ldots, d_n$ ausgehend von der vorliegenden Umgebung. Wenn die Auswertung der Deklarationen die Umgebung V liefert, wird der Ausdruck e in V ausgewertet. Der Let-Ausdruck liefert den so erhaltenen Wert.

Auswertung von Deklarationen

Wir benötigen zunächst eine Funktion $V_1 + V_2$, die zwei Umgebungen miteinander kombiniert. Wenn ein Bezeichner von V_1 und V_2 gebunden wird, soll die kombinierte Umgebung $V_1 + V_2$ den Bezeichner so wie V_2 binden. Hier sind 2 Beispiele:

$$\{\mathsf{x} \mapsto 5,\ \mathsf{y} \mapsto 7\} + \{\mathsf{x} \mapsto 1,\ \mathsf{z} \mapsto 3\} = \{\mathsf{x} \mapsto 1,\ \mathsf{y} \mapsto 7,\ \mathsf{z} \mapsto 3\}$$

$$\{\mathsf{x} \mapsto 1,\ \mathsf{z} \mapsto 3\} + \{\mathsf{x} \mapsto 5,\ \mathsf{y} \mapsto 7\} = \{\mathsf{x} \mapsto 5,\ \mathsf{y} \mapsto 7,\ \mathsf{z} \mapsto 3\}$$

Sei jetzt eine Deklaration `val` $x = e$ und eine Umgebung V gegeben. Dann wird zunächst der Ausdruck e in V ausgewertet. Wenn das den Wert v liefert, liefert die Deklaration die Umgebung $V + \{x \mapsto v\}$.

Eine Deklaration `val` $(x_1, \ldots, x_n) = e$ wird nach demselben Schema ausgewertet. Wenn der Ausdruck e das Tupel $(v_1, \ldots, v_n)$ liefert, liefert die Deklaration die Umgebung $V + \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$.

Auswertung von Prozeduraufrufen

Sei eine Prozedur $p = (\mathit{fun}\ f\ M = e,\ V)$ und ein Wert v gegeben. Zunächst wird die Umgebung V' bestimmt, die die Argumentvariablen der Prozedur gemäß des Argmentmusters M und des Arguments v bindet. Dann wird der Prozedurrumpf e in der Umgebung $V + \{f \mapsto p\} + V'$ ausgewertet. Der Prozeduraufruf liefert den so erhaltenen Wert. Die Bindung $f \mapsto p$ ermöglicht rekursive Prozeduraufrufe.

1.16 Laufzeitfehler und Divergenz

Für den Verlauf der Ausführung eines Programms gibt es die folgenden prinzipiellen Möglichkeiten:

- ***Reguläre Terminierung.*** Die Ausführung endet nach endlich vielen Schritten erfolgreich. Das war für alle bisherigen Programme der Fall.
- ***Abbruch wegen Lauzeitfehler.*** Die Ausführung wird abgebrochen, da eine Operation einen Fehler signalisiert.

- ***Abbruch durch den Benutzer.*** Die Ausführung wird auf Verlangen des Benutzers abgebrochen.[5]
- ***Abbruch wegen Ressourcenüberschreitung.*** Die Ausführung muss abgebrochen werden, da der dem Interpreter zur Verfügung stehende Speicherplatz erschöpft ist.
- ***Divergenz.*** Die Ausführung endet nicht.

Ein typischer Laufzeitfehler ist Division durch null. Als Beispiel betrachten wir den Ausdruck

```
1 div 0
! Uncaught exception: Div
```

Beachten Sie, dass der Ausdruck semantisch zulässig ist, da `div` den Typ $\texttt{int} * \texttt{int} \to \texttt{int}$ hat.[6] Beachten Sie, dass die Auswertung des Konditionals

```
if false then 1 div 0 else 0
0 : int
```

regulär terminiert, da die Konsequenz wegen der Bedingung `false` nicht ausgewertet wird. Eine Prozedur, die auch durch null dividieren kann, kann mithilfe eines Konditionals wie folgt definiert werden:

```
fun f (x:int, y:int) = if y<>0 then x div y else 0
val f : int * int → int
```

Hier ist ein Beispiel für ein divergierendes Programm:

```
fun f (x:int) : int = f x
val x = f 0
```

Die Divergenz ergibt sich aus der Tatsache, dass die Auswertung des Prozeduraufrufs `f 0` wieder zur Auswertung des Prozeduraufrufs `f 0` führt:

$$\texttt{f 0} \to \texttt{f 0} \to \texttt{f 0} \to \texttt{f 0} \to \cdots$$

Da für die Ausführung dieses Programms nur wenig Speicherplatz erforderlich ist, wird ein Interpreter die Ausführung dieses Programms so lange fortsetzen, bis er durch ein externes Signal aufgefordert wird, die Ausführung abzubrechen.

Hier ist ein Beispiel für ein divergierendes Programm, das mit fortschreitender Auswertung mehr und mehr Speicherplatz benötigt und daher vom Interpreter nach einer gewissen Zeit wegen Ressourcenüberschreitung abgebrochen wird:

[5] Das Abbrechen einer laufenden Programmausführung gelingt oft durch die Eingabe von `Strg+c` (gleichzeitiges Drücken der Tasten `Strg` und `c`).

[6] Hier wäre ein „besserer“ Typ für `div` wünschenswert, der 0 als zweites Argument ausschließt. Diese Idee erweist sich jedoch als nicht umsetzbar.

```
fun g (x:int) : int = 0 + g x
val x = g 0
! Uncaught exception: Out_of_memory
```

Die Ausführung dieses Programms können wir uns wie folgt veranschaulichen:

`g 0` $\rightarrow$ `0+(g 0)` $\rightarrow$ `0+(0+(g 0))` $\rightarrow$ `0+(0+(0+(g 0)))` $\rightarrow \cdots$

Da die durch die Ausdrücke dargestellten Berechnungszustände mit dem Fortschreiten der Auswertung immer größer werden, wird mehr und mehr Speicherplatz benötigt.

1.17 Festkomma- und Gleitkommazahlen

Die Hardware von Computern arbeitet mit zwei unterschiedlichen Darstellungssystemen für Zahlen, die als ***Festkomma-*** und ***Gleitkommadarstellung*** bezeichnet werden. Die Darstellung einer Zahl gemäß der Festkommadarstellung bezeichnet man als eine ***Festkommazahl***, und die Darstellung gemäß der Gleitkommadarstellung als ***Gleitkommazahl***. Festkommazahlen stellen ganze Zahlen dar, und Gleitkommazahlen reelle Zahlen mit endlicher Dezimalbruchdarstellung. Da die Hardware für jede der beiden Darstellungen eine fixe Größe vorgibt (zum Beispiel 32 und 64 Bit), können jeweils nur endlich viele Zahlen dargestellt werden.

Die Festkommazahlen heutiger PCs können alle ganzen Zahlen im Intervall $\{-2^{31}, \ldots, 2^{31} - 1\}$ darstellen. Wenn eine Operation wie Addition oder Multiplikation zu einer Zahl außerhalb des darstellbaren Intervalls führt, darf sie ein beliebiges Ergebnis im darstellbaren Bereich liefern. Man spricht dann von einem so genannten ***Überlauf***.

Fast alle Standard ML Interpreter realisieren die Werte des Typs `int` mit Festkommazahlen. Auf PCs schränkt Moscow ML den darstellbaren Bereich auf das Intervall $\{-2^{30}, \ldots, 2^{30} - 1\}$ ein. Wenn es bei der Ausführung eines Programms zu einem Überlauf kommt, wird der Laufzeitfehler *Overflow* signalisiert:

```
4*1073741823
! Uncaught exception: Overflow
```

Eine Gleitkommazahl stellt eine Zahl x gemäß der Gleichung

$$x = m \cdot 10^n$$

durch zwei ganze Zahlen m und n dar, die als ***Mantisse*** und ***Exponent*** bezeichnet werden. Beispielweise wird die Zahl 12, 65 durch das Paar $(1265, -2)$ dargestellt. Mantisse und Exponent müssen jeweils innerhalb eines vorgegebenen endlichen

Intervalls liegen. Die entsprechenden Gleitkommazahlen beschreiben endlich viele Stützpunkte innerhalb eines Intervalls, das aus unendlich vielen reellen Zahlen besteht.

Gleitkommazahlen sind in Standard ML als Werte des Typs `real` verfügbar. Hier ist ein Beispiel:

```
4.5 + 2.0 * 5.5
12.0 : real
```

Die Konstanten für `real` halten sich an die englische Schreibweise und verwenden statt des bei uns üblichen Kommas den Punkt. Für die Bezeichnung der Gleitkommaoperationen (z. B. Addition, Multiplikation) verwendet Standard ML dieselben Operatoren (z. B. `+`, `*`) wie für die Festkommaoperationen. Man sagt, dass diese Operatoren ***überladen*** sind. Sie können gemäß zweier Typen angewendet werden:

```
int * int -> int
real * real -> real
```

Bei einem semantisch zulässigen Programm ist für jedes Auftreten eines überladenen Operators klar, welche der dem Operator zugeordneten Operationen gemeint ist.

Die Typen der überladenen Operatoren lassen keine gemischten Argumente zu:

```
2 * 5.5
! Type clash: expression of type int cannot have type real
```

Eine solche Mischung wäre auch nicht sinnvoll, da Festkomma- und Gleitkommazahlen zu unterschiedlichen Darstellungssystemen gehören.

Die Beschreibung sehr großer und sehr kleiner Gleitkommazahlen wird durch spezielle Konstanten unterstützt:

$1{,}7878 \cdot 10^{45} \quad \rightsquigarrow \quad$ `1.7878E45`

$1{,}7878 \cdot 10^{-45} \quad \rightsquigarrow \quad$ `1.7878E~45`

Wenn eine Gleitkommaoperation zu einer Zahl außerhalb des darstellbaren Bereichs führt, liefert sie eine Gleitkommazahl, die vom wirklichen Ergebnis so wenig wie möglich abweicht:

```
1.0 / 3.0
0.333333333333 : real

2.0 * 5.00000000001
10.0 : real
```

Man sagt, dass die Gleitkommaoperationen ***Rundungsfehler*** machen. Wenn mehrere Gleitkommaoperationen hintereinander ausgeführt werden, können sich deren Rundungsfehler so akkumulieren, dass das Gleitkommaergebnis keine Ähnlichkeit mehr mit dem exakten Ergebnis hat.

Das Rechnen mit Gleitkommaoperationen bezeichnet man als ***Gleitkommaarithmetik***. Gleitkommaarithmetik spielt für vielen Anwendungen eine wichtige Rolle (z. B. Wettervorhersagen). Die ***Numerik*** ist eine eigenständige Forschungsrichtung, die sich mit der Entwicklung von Algorithmen[7] beschäftigt, für deren Ergebnisse man trotz Rundungsfehlern eine Mindestgenauigkeit garantieren kann.

1.18 Standardstrukturen

Im Zusammenhang mit Zahlen benötigen wir eine Vielzahl von Prozeduren, die Standardaufgaben wie die Berechnung von Quadratwurzeln erledigen. Standard ML stellt solche ***Standardprozeduren*** im Rahmen so genannter ***Standardstrukturen*** zur Verfügung. Abbildung 1.4 zeigt einige Prozeduren aus den Standardstrukturen `Math` und `Real`. Hier sind Anwendungsbeispiele:

```
Math.sqrt 4.0
2.0 : real

Real.fromInt 45
45.0 : real

Real.round 1.5
2 : int
```

Neben Prozeduren können Standardstrukturen auch andere Werte zur Verfügung stellen. Beispielsweise stellt die Standardstruktur `Math` unter dem Bezeichner `pi` eine Gleikommazahl zur Verfügung, die die reelle Zahl π so gut wie möglich approximiert:

```
Math.pi
3.14159265359 : real
```

Die Objekte einer Standardstruktur werden durch ***zusammengesetzte Bezeichner*** identifiziert (z. B. `Math.pi`), die aus dem Bezeichner der Struktur (z. B. `Math`) und aus dem lokalen Bezeichner des Objektes (z. B. `pi`) bestehen. Syntaktisch gesehen sind zusammengesetzte Bezeichner Wörter, die genau wie normale Bezeichner verwendet werden können.

Eine Übersicht über die Standardstrukturen für Standard ML finden Sie unter `www.cs.bell-labs.com/who/jhr/sml/basis/`.

[7] Algorithmen sind Berechnungsverfahren.

`Math.sqrt`	`: real -> real`	$\sqrt{x}$
`Math.sin`	`: real -> real`	
`Math.asin`	`: real -> real`	
`Math.exp`	`: real -> real`	e^x
`Math.pow`	`: real * real -> real`	x^y
`Math.ln`	`: real -> real`	
`Math.log10`	`: real -> real`	
`Real.fromInt`	`: int -> real`	
`Real.round`	`: real -> int`	
`Real.floor`	`: real -> int`	*Rundung nach unten*
`Real.ceil`	`: real -> int`	*Rundung nach oben*
`Math.pi`	`: real`	π
`Math.e`	`: real`	

Abbildung 1.4: Einige Standardprozeduren