Kapitel 1

Crashkurs

Dieses Kapitel gibt Ihnen einen ersten Eindruck von der Programmiersprache Standard ML und versetzt Sie in die Lage, mit einem Interpreter selbstständig Experimente durchzuführen.

Interpreter sind interaktive Werkzeuge, die dazu dienen, das Lernen einer Programmiersprache zu erleichtern. Ein Benutzer legt dem Interpreter schrittweise kleine Programmteile vor. Dieser überprüft zunächst, ob eine Eingabe gemäß den Regeln der Sprache zulässig sind. Wenn dies der Fall ist, führt der Interpreter das eingegebene Programmteil aus und zeigt dem Benutzer das Ergebnis. Wenn das eingegebene Programmteil fehlerhaft ist, zeigt der Interpreter dem Benutzer den Fehler mithilfe einer Fehlermeldung.

Experimentieren ist eine sehr nützliche Strategie beim Lernen einer Programmiersprache: Beim Durcharbeiten von Lehrtexten, die die Regeln und Prinzipien einer Sprache erklären, ergeben sich oft Fragen, die man mithilfe eines Interpreters schnell beantworten kann. Sie sollten alle Beispiele dieses Kapitels mit einem Interpreter ausprobieren. Nachdem Sie ein Beispiel wie angegeben ausprobiert und verstanden haben, werden Ihnen sofort interessante Varianten einfallen. Zunächst werden Sie nicht sicher sein, ob ihre Varianten zulässig sind und welche Ergebnisse sie liefern. Durch Experimente mit dem Interpreter können Sie diese Fragen schnell klären.

1.1 Ausdrücke und Deklarationen

Für unsere Beispiele verwenden wir das Programmiersystem Moscow ML.[1] Moscow ML wurde für die Ausbildung entworfen und ist für Windows und Unix kos-

[1] `www.dina.dk/~sestoft/mosml.html`

tenlos verfügbar. Neben einem Interpreter stellt Moscow ML auch einen Übersetzer zur Verfügung, den wir später verwenden werden.

Nachdem der Moscow ML Interpreter gestartet wurde, meldet er sich mit

```
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 Programmteils wartet. Wir beginnen mit der Eingabe eines arithmetischen Ausdrucks:

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
- 4+9;
```

Der Interpreter erwartet, dass Eingaben mit einem Semikolon ⟨`;`⟩ abgeschlossen und mit der Eingabetaste übergeben werden. Er wertet den eingegebenen Ausdruck aus und informiert den Benutzer über das Ergebnis:

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
- 4+9;
> val it = 13 : int
-
```

Mit der erneuten Ausgabe des Bindestrichs zeigt der Interpreter an, dass er auf die nächste Eingabe wartet. Die eigentliche Ausgabe

```
> val it = 13 : int
```

setzt sich wie folgt zusammen:

- Das Zeichen ⟨`>`⟩ am Zeilenanfang sagt, dass es sich bei dieser Zeile um eine Ausgabe handelt.
- Das Wort `val` zeigt an, dass das Ergebnis des eingegebenen Ausducks ein sogenannter *Wert* (englisch „value“) ist.
- Die Zeichenreihe `13 : int` sagt, dass das Ergebnis des Ausdrucks die ganze Zahl `13` ist. Jedem ausgegebenen Wert wird ein sogenannter *Typ* zugeordnet. Im Beispiel ist dies `int`, eine Abkürzung für das englische Wort integer (ganze Zahl).
- Die Zeichenreihe `it = 13` sagt, dass der *Ergebnisbezeichner* `it` an den Wert `13` *gebunden* wurde.

Hier sind Beispiele für die Verwendung des Ergebnisbezeichners:

```
- 3;
> val it = 3 : int

- it * it;
> val it = 9 : int

- it;
> val it = 9 : int

- it + it;
> val it = 18 : int
```

Im Folgenden stellen wir Interaktionen mit dem Interpreter in abgekürzter Form dar. Statt

```
- it + it;
> val it = 18 : int
```

schreiben wir kürzer

```
it + it
18 : int
```

Ausgabezeilen kann man daran erkennen, dass sie in *Kursivschrift* gesetzt sind. Insbesondere verzichten wir auf die Angabe des die Eingabe abschließenden Semikolons. Außerdem verzichten wir bei der Ausgabe meistens auf den Vorspann `val it =`.

Klammerung und einstelliges Minus

Als Nächstes probieren wir einen etwas größeren Ausdruck:

```
3*4 + 5 - 7
10 : int
```

Dieser Ausdruck wird vom Interpreter gemäß den Regeln von Standard ML automatisch wie folgt geklammert:

```
((3*4) + 5) - 7
```

Insbesondere gilt die übliche „Punkt-vor-Strich“-Regel. Durch explizite Klammerung kann man jede gewünschte Gruppierung vorgeben:

```
3*((4 + 5) - 7)
6 : int
```

Als einstelliges Minuszeichen verwendet Standard ML die Tilde ⟨~⟩:

```
5 + ~7
~2 : int
```

Wertdeklarationen

Mithilfe von *Wertdeklarationen* kann man *Bezeichner* an Werte *binden*:

```
val x = 3 * 4
val x = 12 : int

val y = 5
val y = 5 : int
```

Gebundene Bezeichner kann man anstelle der an sie gebundenen Werte in Ausdrücken verwenden:

```
x + y
17 : int
```

Es besteht die Möglichkeit, einen bereits gebundenen Bezeichner mithilfe einer Wertdeklaration neu zu binden:

```
val x = 7
val x = 7 : int

val y = 2 + x
val y = 9 : int

val x = 3 + y
val x = 12 : int

val x = x + y
val x = 21 : int
```

Man spricht auch von der *Redeklaration* eines Bezeichners. Redeklaration eines Bezeichners hat keinen Effekt auf die vor der Redeklaration erfolgten Eingaben.

Gleitkommazahlen

Neben ganzen Zahlen kennt Standard ML auch rationale Zahlen:

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

Der englischen Schreibweise folgend werden rationale Zahlen mit Punkten statt Kommata geschrieben.[2] „Real“ ist das englische Wort für reelle Zahl, verspricht also mehr als geboten wird. Ganze und rationale Zahlen sind in Standard ML disjunkte Mengen und können nicht gemischt werden. Wenn man es trotzdem versucht, lehnt der Interpreter die Eingabe mit einer Fehlermeldung ab:

[2] Die Sprache der Informatik ist Englisch. Dementsprechend folgen Programmiersprachen generell englischen Konventionen.

```
4.5 + 2 * 5.5
! 4.5 + 2 * 5.5
!       ^
! Type clash: expression of type int cannot have type real
```

Ähnlich wie Taschenrechner kennt Moscow ML nur solche rationalen Zahlen, deren Dezimaldarstellung eine bestimmte Anzahl von Stellen nicht überschreitet. Die programmiersprachlichen Operationen für rationale Zahlen unterscheiden sich von den mathematischen Operationen dadurch, dass sie beim Verlassen der maximalen Darstellungsbreite sogenannte *Rundungsfehler* machen:

```
1.0 / 3.0
0.333333333333 : real

2.0 * 5.00000000001
10.0 : real
```

Die gerade gezeigten Unterschiede zwischen mathematischen und programmiersprachlichen rationalen Zahlen haben prinzipielle Gründe und finden sich in allen Programmiersprachen. Um den Unterschied zu den rationalen Zahlen der Mathematik hervorzuheben, spricht man auch von *Gleitkommazahlen* und *Gleitkommaarithmetik*. Wenn mehrere Gleitkommaoperationen hintereinander ausgeführt werden, können sich deren Rundungsfehler so akkumulieren, dass das Gleitkommaergebnis keine Ähnlichkeit mehr mit dem Ergebnis bei rationaler Arithmetik hat.

Gleitkommaarithmetik spielt bei vielen Anwendungen eine wichtige Rolle. Die Numerik ist eine eigene Forschungsrichtung, die Algorithmen[3] für Gleitkommaarithmetik entwickelt und analysiert. Dabei interessiert man sich besonders für Algorithmen, für deren Ergebnisse man eine Mindestgenauigkeit garantieren kann.

Um auch sehr große und sehr kleine Gleitkommazahlen zu haben, wird eine Gleitkommazahl x durch eine *Mantisse* m und einen *Exponenten* e dargestellt:

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

Diese Darstellung wird durch spezielle Gleitkommakonstanten unterstützt. Zum Beispiel steht die Konstante

```
1.7878E45
1.7878E45 : real
```

für die rationale Zahl $1{,}7878 \cdot 10^{45}$. Für die Mantisse und den Exponenten gelten Größenbeschränkungen.

[3] Algorithmen sind Berechnungsverfahren.

Die Standardstruktur Math

Es gibt viele nützliche Operationen für Gleitkommazahlen. Standard ML stellt diese mithilfe der *Standardstruktur* `Math` zur Verfügung. Der Moscow ML Interpreter besteht darauf, dass man Standardstrukturen zuerst lädt bevor man sie benutzt:

```
load "Math"
() : unit
```

Nachdem `Math` geladen ist, stellt der *zusammengesetzte Bezeichner*

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

eine Approximation der Kreiskonstante π zur Verfügung. Neben anderen Operationen stellt `Math` Approximationen für die trigonometrischen Funktionen bereit:

```
Math.sin(Math.pi/2.0)
1.0 : real

Math.sin(Math.pi)
1.22460635382E~16 : real
```

Die Eingabe

```
help "Math"
val pi    : real
val e     : real
val sqrt  : real -> real
val sin   : real -> real
...
```

liefert eine Übersicht über die von `Math` bereitgestellten Konstanten und Operationen. In ausführlichere Form bekommt man diese Übersicht auch online unter `www.cs.bell-labs.com/who/jhr/sml/basis/`.

Größenbeschränkungen für ganze Zahlen

Moscow ML kann ganze Zahlen im Intervall

$$\{-2^{30}, \ldots, 2^{30} - 1\}$$

darstellen. Im Gegensatz zur Gleitkommaarithmetik ist die ganzzahlige Arithmetik innerhalb dieses Intervalls exakt. Wenn ein Zwischenergebnis nicht mehr innerhalb des zulässigen Intervalls liegt, wird die Berechnung mit einer Fehlermeldung abgebrochen.

Die kleinste und größte durch den Interpreter darstellbare ganze Zahl erhält man mit der Standardstruktur `Int`:

```
load "Int"
() : unit

Int.minInt
SOME ~1073741824 : int option

Int.maxInt
SOME 1073741823 : int option
```

Die Erklärung des Typs `int option` verschieben wir auf später. Hier sind Beispiele, die zeigen, wie der Interpreter auf ganze Zahlen außerhalb des zulässigen Bereichs reagiert:

```
1073741823
1073741823 : int

1073741824
! Lexical error: integer constant is too large

2*1073741823
! Uncaught exception: Overflow
```

Boolesche Werte und Konditionale

Die arithmetischen Vergleiche liefern Werte des zweielementigen Typs `bool` (`true` und `false`):

```
3<3
false : bool

3<=3
true : bool

3=3
true : bool

3<>3
false : bool
```

Die Zeichenkombination `<=` steht für den Vergleichsoperator $\leq$, `<>` für den Vergleichsoperator $\neq$.

Vergleiche werden besonders oft zusammen mit Konditionalen benutzt. Ein *Konditional* realisiert eine Wenn-Dann-Sonst-Entscheidung, die gemäß einer Bedingung getroffen wird:

```
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` a_1 `then` a_2 `else` a_3

Die drei Teilausdrücke a_1, a_2 und a_3 bezeichnet man als *Bedingung*, *Konsequenz* und *Alternative*.

Tupel

Bisher haben wir drei verschiedene Typen von Werten kennengelernt: ganze Zahlen, rationale Zahlen und Boolesche Werte. Diese sogenannten *skalaren Werte* lassen sich mithilfe von *Tupeln* zu komplexeren Werten zusammensetzen:

```
(3, 2.0, true)
(3, 2.0, true) : int * real * bool
```

Das obige Tupel hat drei *Komponenten*: die ganze Zahl `3`, die Gleitkommazahl `2.0` und den Booleschen Wert `true`. Man spricht von der ersten, zweiten und dritten Komponente des Tupels. Der Typ eines Tupels ergibt sich als das Produkt der Typen der Komponenten. Auf die Komponenten eines Tupels kann man mit Hilfe der sogenannten *Projektionen* zugreifen:

```
#2(3, 2.0, true)
2.0 : real

#3(3, 2.0, true)
true : bool
```

Die Komponenten eines Tupels können auch Tupel sein:

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

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

Es gibt auch das *leere Tupel*:

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

Dieses hat den einelementigen Typ `unit`.

Der Ausdruck

```
(4)
4 : int
```

liefert eine Zahl und kein einstelliges Tupel. In Standard ML kann man Ausdrücke mit *überflüssigen Klammern* schreiben, wenn man will. Hier ist ein Beispiel, das davon reichlich Gebrauch macht:

```
((((6), (7+9))))
(6,16) : int * int
```

1.2 Prozeduren

Die Deklaration

```
fun kreisflaeche(r:real) = Math.pi*r*r

val kreisflaeche : real -> real
```

bindet den Bezeichner `kreisflaeche` an eine *Prozedur*, die zu einem Radius die Fläche des entsprechenden Kreises berechnet:

```
kreisflaeche 2.0
12.5663706144 : real
```

Die Prozedur `kreisflaeche` berechnet eine mathematische Funktion. Im Gegensatz zu einer mathematischen Funktion wird eine programmiersprachliche Prozedur jedoch immer mit einem Algorithmus definiert, der festlegt, wie das Ergebnis aus dem Argument berechnet wird.

Sie werden sich wahrscheinlich wundern, warum wir die obige Prozeduranwendung ohne Klammern als

```
kreisflaeche 2.0
```

und nicht mit Klammern als

```
kreisflaeche(2.0)
```

geschrieben haben. Die Antwort ist einfach: Diese Klammern sind in Standard ML überflüssig.

Die Prozedur

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

val betrag : int -> int
```

berechnet den Absolutbetrag einer ganzen Zahl:

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

Die Prozedur

```
fun hmsNachS(h:int,m:int,s:int) = 3600*h + 60*m + s

val hmsNachS : int * int * int -> int
```

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

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

Die Prozedur[4]

```
fun sNachHms(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 sNachHms : int -> int * int * int
```

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

```
sNachHms 3666
(1, 1, 6) : int * int * int
```

Die Deklaration der Prozedur `sNachHms` führt der Übersichtlichkeit halber drei Hilfsbezeichner `h`, `m` und `s'` ein, deren Werte mit Hilfe der Operationen für ganzzahlige Division (`div`) und ganzzahlige Restbildung (`mod`) berechnet werden.

Die Prozedur `hmsNachS` wird auf Tupel bestehend aus drei ganzen Zahlen angewendet. Da die Prozedur `sNachHms` gerade solche Tupel liefert, können wir die beiden Prozeduren wie folgt kombinieren:

```
hmsNachS(sNachHms 3666)
3666 : int
```

Technisch gesehen haben Prozeduren in Standard ML immer genau ein Argument. Wenn eine Prozedur wie `hmsNachS` auf Tupel angewendet wird, sprechen wir trotzdem vom ersten, zweiten und letzten Argument. Diese Sprechweise ist ungenau aber bequem.

[4] Per Konvention lässt man Prozedurnamen in Standard ML immer mit einem Kleinbuchstaben beginnen. Um die Lesbarkeit zu verbessern, kann man danach auch Großbuchstaben benutzen.

1.3 Rekursive Prozeduren und Auswertungsprotokolle

Wir wollen jetzt eine Prozedur schreiben, die zu einer ganzen Zahl x und einer natürlichen Zahl n die Potenz x^n berechnet. Dabei helfen uns die Gleichungen

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

Damit deklarieren wir eine Prozedur

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

val potenz : int * int -> int
```

mit der wir beispielsweise die Potenz 3^4 berechnen können:

```
potenz(3,4)
81 : int
```

Die Tatsache, dass die Prozedur `potenz` durch Rückgriff auf sich selbst deklariert ist, bezeichnet man als *Rekursion*. Rekursion ist ein grundlegendes Prinzip, das in der Programmierung in vielen Variationen vorkommt. Das folgende *Auswertungsprotokoll* zeigt, wie ein Interpreter mit rekursiven Prozeduren rechnet:

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

Die Essenz dieses Auswertungsprotokolls kann wie folgt dargestellt werden:

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

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

Auswertung von Ausdrücken

Auswertungsprotokolle erklären, wie ein Interpreter die Ergebnisse von Ausdrücken berechnet. Die Ergebnisse von Ausdrücken bezeichnet man als *Werte*. Die Berechnung des Ergebnisses eines Ausdrucks bezeichnet man als *Auswertung* des Ausdrucks. Die Auswertung eines Ausdrucks überführt den Ausdruck schrittweise in einfachere Ausdrücke, bis ein Wert erreicht wird. Ein Auswertungsprotokoll zeichnet die Auswertungsschritte des *Auswertungsprozesses* auf. Es beginnt mit dem auszuwertenden Ausdruck und endet mit dem Ergebnis. Wir beschreiben den *Auswertungsprozess* wie fogt:[5]

1. Ganze Zahlen, Gleitkommazahlen, Boolesche Werte und das leere Tupel werten zu sich selbst aus.

2. Bezeichner werten zu den Werten aus, an die sie gebunden sind.

3. Ein Operationsausdruck (zum Beispiel $a_1 + a_2$) wird wie folgt ausgewertet:

   a) Zuerst werden alle Unterausdrücke (im Beispiel a_1 und a_2) in der angegebenen Ordnung ausgewertet (im Beispiel a_1 vor a_2).

   b) Auf die so erhaltenen Werte wird die Operation angewendet. Der so erhaltene Wert ist das Ergebnis des Operationsausdrucks.

4. Ein Konditional `if` a_1 `then` a_2 `else` a_3 wird wie folgt ausgewertet:

   a) Zuerst wird die Bedingung a_1 ausgewertet.

   b) Wenn die Auswertung der Bedingung das Ergebnis `true` liefert, wird die Konsequenz a_2 ausgewertet. Das Ergebnis der Konsequenz ist dann das Ergebnis des Konditionals.

   c) Wenn die Auswertung der Bedingung das Ergebnis `false` liefert, wird die Alternative a_3 ausgewertet. Das Ergebnis der Alternative ist dann das Ergebnis des Konditionals.

5. Ein Tupelausdruck (zum Beispiel (a_1, a_2)) wird wie folgt ausgewertet:

   a) Zuerst werden alle Unterausdrücke (im Beispiel a_1 und a_2) in der angegebenen Ordnung ausgewertet (im Beispiel a_1 vor a_2).

   b) Das Ergebnis des Tupelausdrucks ist das Tupel, das aus den Ergebnissen der Unterausdrücke besteht.

   Hier ist ein Beispiel für die Auswertung eines Tupelausdrucks:

[5] Da wir bisher nur über bescheidene Beschreibungstechniken verfügen, verzichten wir vorerst auf die Beschreibung der Auswertung von Let-Ausdrücken.

```
(4+8, 3.0*2.5)
(12, 7.5) : int * real
```

6. Eine Prozeduranwendung $p\ a$ wird wie folgt ausgewertet:

   a) Zuerst wird der Argumentausdruck a ausgewertet. Das Ergebnis bezeichnen wir mit w.

   b) Sei `fun` $p(x : t) = a'$ die Prozedurdeklaration, die den Bezeichner p an eine Prozedur bindet. Der Ausdruck a' wird als *Rumpf* der Prozedur bezeichnet, und der Bezeichner x als die *Argumentvariable* der Prozedur.

   c) Sei a'' der Ausdruck, den man aus a' erhält, indem man die Auftreten von x in a' durch w ersetzt. Der Ausdruck a'' wird als *substituierter Rumpf* bezeichnet.

   d) Als Nächstes wird der substituierte Rumpf a'' ausgewertet. Das Ergebnis von a'' ist dann das Ergebnis der Prozeduranwendung $p\ a$.

   e) Bei Prozeduren mit mehr als einer Argumentvariable wird sinngemäß verfahren. Dabei muss der Argumentausdruck zu einem Tupel auswerten, das für jede Argumentvariable eine Komponente hat.

Wenn Sie den Ausdruck `potenz(4,2)` gemäß dieser Beschreibung auswerten, erhalten Sie genau das oben angebene Auswertungsprotokoll. Stellen Sie sicher, dass Sie jeden einzelnen Auswertungsschritt verstehen. Dazu werden Sie die Beschreibung des Auswertungsprozesses mehr als einmal lesen müssen.

Machen Sie sich klar, dass das Konditional im Rumpf der Prozedur `potenz` dafür sorgt, dass bei der Auswertung des Ausdrucks `potenz(3,4)` nur zwei rekursive Anwendungen der Prozedur `potenz` ausgewertet werden müssen. Das Konditional wirkt also als „Rekursionsbremse“.

Statt Auswertung und auswerten sagt man auch *Evaluation* und *evaluieren*.

Divergenz

Mit Rekursion kann man Prozeduren schreiben, die für bestimmte Argumente unendlich lange Auswertungsprotokolle erzeugen. Dies ist beispielsweise für die Prozedur `potenz` und das Argument `(4, ~1)` der Fall:

```
potenz(4, ~1) →+ 4 * potenz(4, ~2)
              →+ 4 * (4 * potenz(4, ~3))
              →+ 4 * (4 * (4 * potenz(4, ~4)))
              →+  ...
```

Wir sagen, dass die Prozedur `potenz` für das Argument `(4, ~1)` *divergiert*.

Was macht ein Interpreter, wenn eine Prozedur divergiert? Es gibt drei Fälle. Im ersten Fall braucht die Berechnung im Interpreter zunehmend mehr Speicherplatz und wird durch den Interpreter automatisch abgebrochen, sobald der verfügbare Speicherplatz erschöpft ist:

```
potenz(4, ~1)
! Uncaught exception: Out_of_memory
```

Im zweiten Fall benötigt die Berechnung immer größere Zahlen und wird vom Interpreter mit einer Fehlermeldung abgebrochen, sobald eine zu große Zahl erreicht wird:

```
fun f(x:int) = 1+f(x*x)
val f : int -> int

f 5
! Uncaught exception: Overflow
```

Im dritten Fall braucht die Berechnung im Interpreter nach einer gewissen Zeit keinen zusätzlichen Speicherplatz mehr und benötigt auch keine zu großen Zahlen. Dann rechnet der Interpreter so lange, bis die Berechnung durch den Benutzer abgebrochen wird:[6]

```
fun f(x:int) = if x<>0 then f x else x
val f : int -> int

f 2
Interrupted
```

Wir definieren Divergenz als ein theoretisches Konzept. Eine Prozedur *p divergiert* für ein Argument *w* genau dann, wenn wir für die Anwendung *p w* ein unendliches Auswertungsprotokoll angeben können. Dabei vereinbaren wir, das Auswertungsprotokolle mit beliebig großen Zahlen rechnen können. Wenn wir eine divergierende Anwendung *p w* mit einem Interpreter ausführen, kann es also durchaus sein, dass der Interpreter nach endlich vielen Schritten mit einer der Fehlermeldungen `Out_of_memory` oder `Overflow` abbricht.

Wir sagen, dass eine Prozedur *p* für ein Argument *w terminiert* genau dann, wenn wir für die Anwendung *p w* ein endliches Auswertungsprotokoll angeben können. Statt terminieren sagen wir auch *konvergieren*.

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

1.4 Syntax und Semantik

Für jede Eingabe prüft der Interpreter zunächst, ob diese nach den Regeln der Sprache Standard ML zulässig ist. Da diese Regeln selbst für die bisher gezeigten Beispiele nicht ganz einfach sind, begnügen wir uns hier mit vereinfachten Regeln.

Eine Eingabe liegt zunächst als eine Folge von *Zeichen* vor. Der erste Verarbeitungsschritt überführt die Zeichenfolge in eine Folge von *Wörtern*. Wir unterscheiden drei Arten von Wörtern:

- *Konstanten*, zum Beispiel `4`, `~5.0`, `true` und `()`.
- *Schlüsselwörter*, zum Beispiel `val`, `fun`, `+`, `=`, `(`, `,` und `)`.
- *Bezeichner*, zum Beispiel `x`, `x'`, `Math.pi` und `sNachHms`.

Der zweite Verarbeitungsschritt überführt die Wortfolge in eine *Phrase*. Wir unterscheiden vier Arten von Phrasen: *Ausdrücke*, *Typen*, *Deklarationen* und *Programme*. Ein Programm ist eine Folge von Deklarationen. Die Ausführung einer Deklaration bindet einen Bezeichner an einen Wert. Bisher haben wir Wertdeklarationen und Prozedurdeklarationen gesehen. Wertdeklarationen werden durch das Schlüsselwort `val`, Prozedurdeklarationen durch das Schlüsselwort `fun` eingeleitet. Die Ausdrücke, die wir bisher gesehen haben, waren entweder Konstanten, Bezeichner, Operationsausdrücke (zum Beispiel `4<8`), Konditionale, Tupelausdrücke (zum Beispiel `(4<8,3)`), Prozeduranwendungen oder Let-Ausdrücke.

Die sogenannte *lexikalische Syntax* regelt, wie aus Zeichen Wörter gebildet werden. Die *Phrasensyntax* regelt, wie aus Wörtern Phrasen gebildet werden. Die in diesem Kapitel angegebenen Beispiele geben Ihnen einen ersten Eindruck der Syntax von Standard ML. Diesen Eindruck können Sie durch Experimente mit dem Interpreter vertiefen. Es ist keineswegs notwendig, dass Sie alle syntaktischen Regeln von Standard ML verstehen. Wenn Sie sicher sein wollen, dass Sie eine Phrase richtig geschrieben haben, fragen Sie einfach den Interpreter.

Neben den syntaktischen Regeln müssen Sie zudem die Regeln der *statischen Semantik* einhalten. Auch diese werden durch den Interpreter vor der Ausführung einer Phrase geprüft. Die statischen Semantik besteht aus *Bindungs-* und *Typregeln*. Beispielsweise können Sie Bezeichner in Ausdrücken nur dann verwenden, wenn diese gebunden sind. Die Ausdrücke `1+2.0` und `()+2` sind Beispiele für unzulässige Ausdrücke, die die Typregeln von Standard ML verletzen.

Schließlich gibt es noch die sogenannte *dynamische Semantik*. Diese legt fest, welche Ergebnisse die Ausführung von zulässigen Ausdrücken und Deklarationen liefern soll.

Später werden wir lernen, wie man die Syntax und Semantik von Programmiersprachen präzise beschreibt. Man kann eine Programmiersprache erfolgreich benutzen, ohne ihre Syntax und Semantik vollständig zu kennen. Das sollte Sie nicht zu sehr überraschen, da Sie ja auch die Syntax und Semantik ihrer Muttersprache nicht vollständig kennen. Im Gegensatz zu natürlichen Sprachen sind Programmiersprachen jedoch so entworfen, dass ihre Syntax und Semantik vollständig definiert werden kann.

Übungsaufgaben

Aufgabe 1.1 (Prozedurdeklarationen) Deklarieren Sie Prozeduren, die die Funktionen

1. $g(x) = 2x + 1.4$
2. $h(x) = \sin x + \cos(2\pi x)$

mit Gleitkommazahlen berechnen. Verwenden Sie dazu die Standardstruktur `Math`. Testen Sie Ihre Prozeduren, indem Sie einige Ausdrücke mit Moscow ML auswerten.

Aufgabe 1.2 (Signum) Die Signumsfunktion

$$\mathit{signum}\colon \mathbb{Z} \to \{-1, 0, 1\}$$

liefert -1 für negative Argumente, 0 für 0 und 1 für positive Argumente. Schreiben Sie eine Prozedur

```
signum : int -> int
```

die die Signumsfunktion berechnet.

Aufgabe 1.3 (Let) Schreiben Sie eine Prozedur, die die Funktion

$$f(x, y) = (x - 3)(y + 5)^2$$

mit Gleitkommazahlen berechnet. Benutzen Sie dabei einen Let-Ausdruck, um zu erreichen, dass die Addition $y + 5$ nur einmal berechnet wird.

Aufgabe 1.4 (Fakultäten) Schreiben Sie eine rekursive Prozedur

```
fak : int -> int
```

die die Fakultätsfunktion berechnet:

$$fak\colon \mathbb{N} \to \mathbb{N}$$
$$fak(n) = 1 \cdot 2 \cdot \cdots \cdot n$$

Dabei soll $fak(0) = 1$ gelten. Für negative Argumente soll die Prozedur `fak` divergieren. Was ist das größte n, für das Sie die Fakultät mit Moscow ML noch berechnen können?

Aufgabe 1.5 (Binomialkoeffizienten) Schreiben Sie eine rekursive Prozedur

```
binom : int * int -> int
```

die für $n, k \in \mathbb{N}$ den Binomialkoeffizienten $\binom{n}{k}$ berechnet. Verwenden Sie dazu die folgenden Gleichungen:

$$\binom{n}{0} = 1 \qquad \binom{0}{k} = 0 \quad \text{für } k > 0 \qquad \binom{n}{k} = \frac{n \cdot \binom{n-1}{k-1}}{k} \quad \text{für } n, k > 0$$

Dabei soll n das erste und k das zweite Argument von `binom` sein. Verwenden Sie `x div y` für ganzzahlige Division.

Aufgabe 1.6 (Quersumme) Schreiben Sie eine rekursive Prozedur

```
quer : int -> int
```

die die Quersumme einer ganzen Zahl berechnet. Die Quersumme einer Zahl ist die Summe der Dezimalziffern dieser Zahl. Beispielsweise hat die Zahl -3754 die Quersumme 19. Verwenden Sie ganzzahlige Division (`div`) und ganzzahliger Restbildung (`mod`).

Aufgabe 1.7 (Natürliche Quadratwurzel) Schreiben Sie eine Prozedur

```
wurzel : int -> int
```

die zu einer natürlichen Zahl x die größte natürliche Zahl n berechnet mit $n^2 \leq x$. Schreiben Sie dazu eine Hilfsprozedur

```
wurzel' : int * int -> int
```

die zu zwei natürlichen Zahlen n und x das kleinste $m \geq n$ berechnet mit $m^2 > x$.

Aufgabe 1.8 (Primzahlen (schwer)) Eine natürliche Zahl $n \geq 2$ heißt Primzahl, wenn sie nur durch 1 und sich selbst teilbar ist. Schreiben Sie eine Prozedur

```
nthPrime : int -> int
```

die zu $n \in \mathbb{N}$ die n-te Primzahl berechnet. Die ersten 4 Primzahlen sind 2, 3, 5, und 7. Verwenden Sie mehrere Hilfsprozeduren.

Aufgabe 1.9 (Auswertungsprotokolle) Gegeben sei die Prozedurdeklaration

```
fun f(n:int, a:int) = if n=0 then a else f(n-1, a*n)
```

1. Geben Sie das Auswertungsprotokoll für den Ausdruck `f(3,1)` an. Halten Sie sich dabei genau an die angebene Beschreibung des Auswertungsprozesses. Sie bekommen dann 17 Auswertungsschritte.
2. Für welche Argumente divergiert die Prozedur `f`? Denken Sie daran, daß Auswertungsprotokolle mit beliebig großen Zahlen rechnen können.
3. Berechnen die Prozeduren

   ```
   fun g(n:int) = f(n,1)

   fun h(n:int) = if n<2 then 1 else n*h(n-1)
   ```

   für Argumente $n \geq 0$ dieselbe Funktion?