Kapitel 8

Direkt ausführbare Programme

Wir lernen jetzt, wie man Programme schreibt, die über das Betriebssystem eines Computers gestartet werden können. Solche *direkt ausführbaren Programme* werden mithilfe eines *Übersetzers* aus Standard ML Programmen erzeugt. Damit sie etwas sinnvolles tun können, müssen sie in der Lage sein, Daten einzulesen und Daten auszugeben. Daten werden dabei durch *Zeichenreihen* dargestellt.

8.1 Zeichen und Zeichenreihen

Eine Zeichenreihe ist eine möglicherweise leere Folge von Zeichen. In Standard ML sind *Zeichenreihen* (englisch „strings“) Werte des Typs `string`. Jede Zeichenreihe kann durch eine Konstante dargestellt werden:

```
"Saarbrücken"
"Saarbrücken" : string

"4567899999999 is a large number"
"4567899999999 is a large number" : string

"true ist ein Boolescher Wert"
"true ist ein Boolescher Wert" : string

""
"" : string
```

Die Konstante `""` stellt die *leere Zeichenreihe* dar.

Zeichen (englisch „characters“) sind Werte des Typs `char`. Jedes Zeichen kann durch eine Konstante dargestellt werden:

```
[#"S", #"M", #"L"]
[#"S", #"M", #"L"] : char list
```

Die vordefinierten Prozeduren

```
val explode : string -> char list

val implode : char list -> string
```

konvertieren zwischen Zeichenreihen und Zeichenlisten:

```
explode("Hallo")
[#"H", #"a", #"l", #"l", #"o"] : char list

implode [#"H", #"a", #"l", #"l", #"o"]
"Hallo" : string
```

Die vordefinierten Prozeduren

```
val ord : char -> int
val chr : int -> char
```

konvertieren zwischen Zeichen und natürlichen Zahlen:

```
ord #"1"
49 : int

chr 49
#"1" : char

map ord (explode "019abzü")
[48, 49, 57, 97, 98, 122, 252] : int list

chr 255
#"\255" : char

chr 256
! Uncaught exception: Chr
```

Insgesamt realisiert Moscow ML 256 Zeichen, denen eineindeutig die Zahlen 0 bis 255 zugeordnet sind.

Ein *Zeichenstandard* ist eine Festlegung, die sagt, wieviele Zeichen es gibt (man nummeriert die Zeichen immer mit 0 beginnend durch) und wie die Zeichen grafisch darzustellen sind. ASCII (2^7 Zeichen), Latin-1 (2^8 Zeichen) und Unicode (2^{16} Zeichen) sind Beispiele für gebräuchliche und international festgelegte Zeichenstandards. Die Standard ML Sprachdefinition legt sich auf keinen Zeichenstandard fest. Moscow ML benutzt Latin-1.

Wir können jetzt genau sagen, durch welche mathematischen Objekte wir Zeichen und Zeichenreichen darstellen: Zeichen werden durch die natürlichen Zahlen 0 bis 255 dargestellt, und Zeichenreihen durch Listen über diesen Zahlen. Die speziellen Konstanten für Zeichen und Zeichenreihen sorgen dafür, dass wir diese Werte entsprechend ihrer Interpretation mit `char` und `string` typisieren können.

Es gibt einige Zeichen, die in den Konstanten für Zeichen und Zeichenreihen nicht in der offensichtlichen Weise geschrieben werden können. Beispiele sind das Anführungszeichen, das Tabulatorzeichen und das Zeichen für Zeilenwechsel. Diese *Sonderzeichen* werden mithilfe des sogenannten *Fluchtsymbols* \ geschrieben:

`\"`	`"`	Anführungszeichen
`\\`	`\`	Backslash
`\t`		Tabulator
`\n`		Zeilenwechsel

Hier sind Beispiele:

```
"\"\\\t\n\aa"
"\"\\\t\n\aa" : string
```

```
explode "\"\\\t\n\aa"
[#"\"", #"\\", #"\t", #"\n", #"\a", #"a"] : char list
```

Die Operation ^ konkateniert zwei Zeichenreihen:

```
"Programmier" ^ "sprache"
"Programmiersprache" : string
```

```
"Aller " ^ "Anfang " ^ "ist schwer."
"Aller Anfang ist schwer." : string
```

```
op^
fn : string * string -> string
```

Eine äquivalente Prozedur (bis auf Effizienz) kann man auch selber programmieren:

```
fn (s,s') => implode(explode s @ explode s')
```

Die Vergleichsoperationen `<`, `<=`, `>` und `>=` sind auch für Zeichen und Zeichenreihen definiert.[1] Die Vergleiche für Zeichen entsprechen den Vergleichen für die zugeordneten Zahlen. Die Vergleiche für Zeichenreihen sind *lexikografisch* gemäß der Ordnung für Zeichen definiert. Hier sind Beispiele:

```
"Adam" < "Eva"
true : bool
```

```
"Adam" < "Adamo"
true : bool
```

Das Ergebnis des Vergleichs

```
"rufen" < "Zustand"
false : bool
```

[1] Insgesamt sind die Vergleichsoperationen vierfach überladen, da sie auch für die Typen `int` und `real` definiert sind.

ist zunächst überraschend, da „rufen“ im Lexikon vor „Zukunft“ steht. Die Erklärung ist einfach: Große Buchstaben werden mit kleineren Zahlen dargestellt als kleine Buchstaben:

```
(ord #"A", ord #"B", ord #"a", ord #"b")
(65, 66, 97, 98) : int * int * int * int
```

Das Ergebnis des Vergleichs

```
"Überlegung" < "Zukunft"
false : bool
```

erklärt sich damit, dass Umlaute mit größeren Zahlen dargestellt werden als normale Buchstaben:

```
(ord #"Ü", ord #"Z")
(220, 90) : int * int
```

Die vordefinierte Prozedur

```
val size : string -> int
```

liefert die Länge einer Zeichenreihe:

```
size "Mozart"
6 : int
```

Die Prozeduren

```
val Int.toString   : int -> string
val Int.fromString : string -> int option
```

konvertieren zwischen ganzen Zahlen und Zeichenreihen:

```
Int.toString ~34
"~34" : string

Int.fromString "   -7898   "
SOME ~7898 : int option

Int.fromString "  72test"
SOME 72 : int option

Int.fromString "test"
NONE : int option
```

Die Standardstrukturen `Char` und `String` stellen viele weitere nützlichen Prozeduren für Zeichen und Zeichenreihen zur Verfügung.

8.2 Übersetzung

Direkt ausführbare Programme werden mithilfe eines Übersetzers (englisch „compiler“) erzeugt und können ohne einen Interpreter ausgeführt werden. Von außen kann man einem direkt ausführbaren Programm nicht ansehen, in welcher Programmiersprache es geschrieben wurde. Web-Browser, Editoren, Interpreter und Übersetzer sind Beispiele für Progamme, die typischerweise in direkt ausführbarer Form vorliegen.

Der Moscow ML Übersetzer übersetzt SML Programme in direkt ausführbare Programme. Das zu übersetzende Programm wird als *Quellprogramm* bezeichnet, und das durch die Übersetzung erhaltene Programm als *Objektprogramm*. Das Quellprogramm muss in einer Datei stehen, und das Objektprogramm wird vom Übersetzer in eine Datei geschrieben. Der Übersetzer überprüft das Quellprogramm zunächst auf Zulässigkeit und meldet eventuelle Fehler. Nur wenn das Quellprogramm fehlerfrei ist, wird ein Objektprogramm erzeugt.

Ein *Standard ML Programm* ist eine Folge von Deklarationen.

Hello World

Unser erstes direkt ausführbares Programm soll

```
Hello world!
```

in die Benutzeroberfläche schreiben und dann sofort terminieren. Für die Ausgabe der Zeile `Hello world!` verwenden wir die vordefinierte Prozedur

```
print : string -> unit
```

Das Programm besteht aus der Deklaration

```
val _ = print "Hello world!\n"
```

Das Zeilenwechselzeichen `\n` sorgt dafür, dass eine komplette Zeile ausgegeben wird. Wir schreiben das Programm in eine Datei `hello.sml` und übersetzen es durch die Eingabe von

```
mosmlc hello.sml
```

in einen Kommando-Interpreter.[2] Unter Unix erzeugt der Übersetzer `mosmlc` eine Datei `a.out`, die das direkt ausführbare Programm enthält. Dieses kann durch die Eingabe von

```
./a.out
```

[2] Bei Windows hat der Kommando-Interpreter den schönen Namen „MS-DOS-Eingabeaufforderung“. In der Unix-Welt spricht man von sogenannten Shells.

ausgeführt werden und liefert das gewünschte Ergebnis. Unter Windows schreibt der Übersetzer das direkt ausführbare Programm in eine Datei `mosmlout.exe`. Dieses kann dann durch die Eingabe von

```
mosmlout
```

ausgeführt werden.

Der Moscow ML Übersetzer bietet die Möglichkeit, das Objektrogramm in eine beliebige Datei zu schreiben. Unter Unix schreibt der Aufruf

```
mosmlc hello.sml -o hello
```

das Objektprogramm in die Datei `hello`. Es kann dann durch den Aufruf

```
./hello
```

ausgeführt werden. Unter Windows schreibt der Aufruf

```
mosmlc hello.sml -o hello.exe
```

das übersetzte Programm in die Datei `hello.exe`. Es kann dann durch den Aufruf

```
hello
```

ausgeführt werden.

Potenzen

Als Nächstes schreiben wir ein Programm `power`, das zu zwei *Startargumenten* $x \in \mathbb{Z}$ und $n \in \mathbb{N}$ die Potenz x^n ausgibt. Beispielsweise soll der Aufruf (unter Unix)

```
./power 2 10
```

beziehungsweise (unter Windows)

```
power 2 10
```

die Zahl 1024 liefern. Dafür benötigen wir die Prozedur

```
CommandLine.arguments : unit -> string list
```

die die Startargumente als Zeichenreihen liefert. Abbildung 8.1 zeigt das für `power` erforderliche Quellprogramm. Wenn wir dieses in eine Datei `power.sml` schreiben, können wir `power` mit (unter Unix)

```
mosmlc power.sml -o power
```

beziehungsweise (unter Windows)

```
mosmlc power.sml -o power.exe
```

```
fun power (x,n) =
    if n<1 then 1 else x*power(x,n-1)

fun main ss =
    let
        val x = valOf(Int.fromString(hd ss))
        val n = valOf(Int.fromString(hd(tl ss)))
    in
        Int.toString(power(x,n))
    end
    handle
        Overflow => "Overflow"
      | Empty    => "Need 2 integers"
      | Option   => "Need 2 integers"

val _ =
    print(main(CommandLine.arguments()) ^ "\n")
```

Abbildung 8.1: Quelle für ein Programm, das Potenzen berechnet

erzeugen.

Oft ist es hilfreich, den Übersetzer mit der Option `-i` aufzurufen, zum Beispiel

Übersetzeroption `-i`

```
mosmlc -i power.sml -o power
```

Er gibt dann die Typen der deklarierten Bezeichner aus:

```
val power : int * int -> int
val main  : string list -> string
```

Das beim Arbeiten mit dem Interpreter erforderliche Laden der benutzten Standardstrukturen (`power` benutzt `Int` und `CommandLine`) ist beim Arbeiten mit dem Übersetzer nicht erforderlich.

Kein Laden von Standardstrukturen

App

Die vordefinierte Prozedur

```
app : ('a -> unit) -> 'a list -> unit
```

wendet eine Prozedur auf jedes Element einer Liste an:

app p [$x_1, \ldots, x_n$] = ($p\ x_1$; ... ; $p\ x_n$)

Damit können wir die Quelle für ein Programm, dass seine Startargumente zeilenweise ausgibt, wie folgt schreiben:

```
val _  = app
         (fn s => (print s ; print "\n"))
         (CommandLine.arguments())
```

8.3 Dateien und Prozesse

Wenn Sie mit einem Computer arbeiten, sehen Sie zunächst zwei Dinge: Die Hardware und das Betriebssystem. Das Betriebssystem ist ein nichttriviales Stück Software, ohne das ein Computer nicht benutzbar wäre. Es stellt zwei grundlegende Dienste zur Verfügung: Ein Dateisystem und ein Prozesssystem.

Das *Dateisystem* realisiert einen Speicher, dessen Einträge (die Dateien) jeweils aus einem Namen (dem Dateinamen) und einer Zeichenreihe (dem Inhalt der Datei) bestehen. Unter einem Namen kann immer nur eine Zeichenreihe abgelegt werden. Dateinamen werden als Zeichenreihen und Zeichen als Zahlen (typischerweise von 0 bis 255) realisiert. Der Dateispeicher ist persistent[3]. Das bedeutet, dass Dateien beim Ausschalten des Computers nicht verloren gehen.

Mithilfe des Betriebssystems können (direkt ausführbare) Programme ausgeführt werden. Programme sind als Zeichenreihen realisiert, die in Dateien stehen. Ein Programmaufruf ist eine Folge

$$p \;\; a_1 \;\; \ldots \;\; a_n$$

von Zeichenreihen. Dabei ist p der Name der Datei, in der das Programm steht. Die Zeichenreihen $a_1, \ldots, a_n$ sind die sogenannten *Startargumente*. Wenn das Betriebsystem einen Programmaufruf ausführt, erzeugt es einen *Prozess*, in dem das Programm ausgeführt wird. Dem Prozess werden die Startargumente des Aufrufs übergeben. Nachdem die Ausführung des Programms beendet ist, verschwindet der dafür geschaffene Prozess. Prozesse laufen in zeitlich diskreten Schritten ab, gesteuert durch das Programm. Programme beschreiben also den zeitlichen Verlauf von Prozessen. Prozesse haben Zugriff auf die Dienste des Betriebssystems. Sie sind also beispielsweise in der Lage, Dateien zu lesen und zu schreiben. Wenn der Computer ausgeschaltet wird, werden alle Prozesse endgültig beendet. Man sagt, dass Prozesse *volatil*[4] sind, da sie das Auschalten des Computers nicht überleben.

8.4 Lesen und Schreiben von Dateien

Mithilfe der Standardstruktur `TextIO` können Programme auf das Dateisystem des Betriebssystems zugreifen und Dateien lesen und schreiben. Abbildung 8.2 zeigt einige der durch `TextIO` bereitgestellten Objekte. Um eine Datei zu lesen, muss man sie zuerst mithilfe der Prozedur

[3] *persistent*: lateinisch für anhaltend, dauernd, hartnäckig
[4] *volatil*: lateinisch für flüchtig, verdunstend

```
type instream

val openIn       : string -> instream
val inputLine    : instream -> string
val closeIn      : instream -> unit

val stdIn        : instream

type outstream

val openOut      : string -> outstream
val output       : outstream * string -> unit
val closeOut     : outstream -> unit

val stdOut        : outstream
```

Abbildung 8.2: Einige Objekte der Standardstruktur `TextIO`

`TextIO.openIn : string -> instream`

öffnen. Dabei bekommt man einen *Eingabestrom*. Die Prozedur

`TextIO.inputLine : instream -> string`

liest bei jedem Aufruf eine Zeile aus einem Eingabestrom. Wenn alle Zeilen eines Eingabestroms gelesen sind, liefert `TextIO.inputLine` die leere Zeichenreihe . Spätestens dann sollte man die Datei mit der Prozedur

`TextIO.closeIn : instream -> unit`

schließen.

Abbildung 8.3 zeigt die Quelle für ein Programm `show`, dass eine als Argument angegebene Datei in die Benutzeroberfläche schreibt.

Eine Datei enthält eine Folge von Zeichen. Das Zeichen `\n` markiert die Zeilen einer Datei. Wenn man mit `TextIO.inputLine` eine Zeile liest, bekommt man eine nichtleere Zeichenreihe, die mit dem Zeichen `\n` endet. Das Zeichen `\n` sorgt bei der Ausgabe in die Benutzeroberfläche dafür, dass eine neue Zeile begonnen wird.

Das Schreiben von Dateien erfolgt analog zum Lesen. Mit der Prozedur

`TextIO.openOut : string -> outstream`

kann man eine Datei zum Schreiben *öffnen*. Wenn die Datei bereits existiert, wird ihr bisheriger Inhalt gelöscht. Wenn die Datei noch nicht existiert, wird sie neu angelegt. Mit der Prozedur

`TextIO.output : outstream * string -> unit`

```
fun show instream =
    case TextIO.inputLine instream of
        "" => ()
      | s  => (print s ; show instream)

fun main file =
    let
        val instream  = TextIO.openIn file
    in
        show instream ;
        TextIO.closeIn instream
    end

val _ =
    case CommandLine.arguments() of
        [s] => main s
      | _   => print ("Need one file name\n")
```

Abbildung 8.3: Quelle für ein Programm, das Dateien ausgibt

kann man wiederholt Zeichen in eine Datei schreiben, die jeweils am Ende hinzugefügt werden.

Abbildung 8.4 zeigt die Quelle für ein Programm, dass eine Kopie einer Datei erstellt. Die Namen der Quell- und Zieldatei werden als Startargumente übergeben.

8.5 Interaktive Programme

Es ist nicht schwer, interaktive Programme zu schreiben, die mit dem Benutzer so wie ein Interpreter interagieren. Dafür gibt es den immer offenen Eingabestrom

```
TextIO.stdIn
```

auf dem die vom Benutzer eingegebenen Zeilen landen. Wenn der Benutzer noch keine Zeile eingegeben hat, wartet die Ausführung des Ausdrucks

```
TextIO.inputLine TextIO.stdIn
```

solange, bis der Benutzer eine Zeile eingibt. Statt warten spricht man auch von *blockieren.* Der Benutzer kann auch Zeilen eingeben, ohne dass diese sofort von `TextIO.stdIn` gelesen werden. Sie werden dann einfach auf `TextIO.stdIn` geschrieben und können später gelesen werden.

Abbildung 8.5 zeigt die Quelle eines direkt ausführbaren Programms, das eine *Eingabeaufforderung* (englisch „prompt“)

```
echo-
```

```
fun copy instream outstream =
    case TextIO.inputLine instream of
        "" => ()
      | s  => (TextIO.output(outstream, s) ;
               copy instream outstream )

fun main inFile outFile =
    let
        val instream  = TextIO.openIn inFile
        val outstream = TextIO.openOut outFile
    in
        copy instream outstream ;
        TextIO.closeIn instream ;
        TextIO.closeOut outstream
    end

val _ =
    case CommandLine.arguments() of
        [s,s'] => main s s'
      |   _    => print ("Need two file names\n")
```

Abbildung 8.4: Quelle für ein Program, das Dateien kopiert

ausgibt und vom Benutzer eingegebene Zeilen jeweils wieder ausgibt. Das Interaktionsprinzip dieses Echo-Programms entspricht dem des Moscow ML Interpreters. Unter Unix kann der Benutzer das Echo-Programm durch die Eingabe von `C-d` (Steuertaste und `d`) beenden, unter Windows gibt er stattdessen `C-z` (Steuertaste und `z`) ein. Diese Eingaben sorgen dafür, dass

```
TextIO.inputLine TextIO.stdIn
```

die leere Zeichenreihe liefert.

Abbruch von Programmen mit `C-c`

Die Ausführung eines direkt ausführbaren Programms kann übrigens immer mit Gewalt durch die Eingabe von `C-c` (Steuertaste und `c`) abgebrochen werden. Das ist nützlich, wenn ein Programm zu lange rechnet.

8.6 Imperative Prozeduren

In diesem Kapitel haben wir erstmals Prozeduren gesehen, deren Semantik nicht durch eine Funktion beschrieben werden kann. Ein Beispiel ist die Prozedur

```
print : string -> unit
```

die eine Zeichenreihe in die Benutzeroberfläche schreibt. Ein weiteres Beispiel ist die Prozedur

```
fun echo () =
    (print "echo- " ;
     case TextIO.inputLine(TextIO.stdIn) of
         "" => print "Good bye!\n"
       | s  => (print("echo> " ^ s) ; echo()) )

val _ =  echo()
```

Abbildung 8.5: Quelle für ein interaktives Echo-Programm

```
TextIO.inputLine : TextIO.instream -> string
```

die für einen Eingabestrom unter Umständen solange verschiedene Ergebnisse liefert, bis alle Zeilen gelesen sind. Die Prozedur

```
fun date() = Date.toString(Date.fromTimeLocal(Time.now()))
val date : unit -> string
```

liefert das aktuelle Datum mit Uhrzeit:

```
date()
"Thu Dec 16 16:45:07 1999" : string

date()
"Thu Dec 16 16:45:11 1999" : string
```

Es ist also durchaus möglich, dass der Ausdruck

```
date() = date()
```

zu `false` auswertet.

Prozeduren, deren Semantik vollständig durch die zugeordnete Funktion beschrieben wird, werden als funktionale Prozeduren bezeichnet (siehe Kapitel 7). Alle anderen Prozeduren werden als *imperative Prozeduren bezeichnet.*

Die imperativen Prozeduren, die wir in diesem Kapitel kennengelernt haben, sind nicht funktional, da sie auf Dienste des Betriebssystems zugreifen. In Kapitel 13 werden wir Sprachkonstrukte einführen, mit denen man imperative Prozeduren schreiben kann, ohne auf das Betriebssystem zuzugreifen.

Übungen

Aufgabe 8.1 (Lexikographische Ordnung) Schreiben Sie eine Prozedur

```
compare : char list * char list -> order
```

die zwei Zeichenlisten lexikographisch gemäß der Ordnung für Zeichen vergleicht. Verwenden Sie dabei die Prozedur

```
Char.compare : char * char -> order
```

Ihre Prozedur soll dieselben Ergebnisse wie

```
fn (xs,ys) => String.compare(implode xs, implode ys)
```

liefern.

Aufgabe 8.2 (Sortierte Ausgabe der Startargumente) Schreiben Sie ein Programm, das seine Startargumente lexikografisch sortiert und zeilenweise ausgibt. Benutzen Sie die folgenden vordefinierten Prozeduren:

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

Aufgabe 8.3 (Verschlüsselung) Schreiben Sie ein Programm, das eine Datei verschlüsselt, indem es auf jedes Zeichen die Prozedur

```
fn c => chr((ord c + 3) mod 256)
```

anwendet. Die Quell- und Zieldatei sollen durch zwei Startargumente benannt werden. Verwenden Sie die Prozeduren

```
TextIO.input1  : instream -> char option
TextIO.output1 : outstream * char -> unit
```

um zeichenweise zu lesen und zu schreiben. Mit der Prozedur

```
fn c => chr((ord c - 3) mod 256)
```

können Sie verschlüsselte Dateien übrigens wieder entschlüsseln.

Aufgabe 8.4 (Dreiecke) Sie sollen ein Programm schreiben, dass Dreiecke wie folgt ausgibt:

```
*
**
***
****
```

1. Schreiben Sie eine Prozedur

   ```
   for : int -> (int -> 'a) -> unit
   ```

   wie folgt:

   ```
   for n p = (p 1 ; ... ; p n ; ())
   ```

2. Schreiben Sie ein Programm, das für ein Argument $n \in \mathbb{N}$ ein n-zeiliges Dreieck ausgibt.

3. Erweitern Sie Ihr Programm so, dass es mit mehreren Argumenten arbeitet und für jedes Argument ein passendes Dreieck ausgibt.

Aufgabe 8.5 (Zeilennummern) Schreiben Sie ein Programm, das die Zeilen einer als Argument angegebenen Datei mit Zeilennummer ausgibt. Beispielsweise soll eine Datei mit den Zeilen

```
Ich bestehe aus
zwei Zeilen!
```

wie folgt ausgegeben werden:

```
1 : Ich bestehe aus
2 : zwei Zeilen!
```

Falls ein zweites Argument *y* angegeben ist, sollen die nummerierten Zeilen in die Datei *y* geschrieben werden (statt auf den Bildschirm).

Aufgabe 8.6 (Taschenrechner) Sie sollen ein interaktives Programm schreiben, dass einfache arithmetische Ausdrücke auswertet:

```
# 5*7 - 2*3*5 + 15
20
```

Die Aufgabe ist nicht ganz einfach. Sie lernen dabei wichtige Techniken, die man beispielsweise für Interpreter benötigt. Bitte gehen Sie genau nach der folgenden Bauanleitung vor. Überzeugen Sie sich nach jedem Teilschritt durch Tests davon, dass Ihre Prozeduren korrekt arbeiten (mit dem Interpreter). Für die Entwicklung größerer Programme ist schrittweises Testen unumgänglich.

1. Schreiben Sie eine iterative Prozedur

   ```
   num : int -> char list -> int * char list
   ```

   die Ziffern von einer Zeichenliste liest und die entsprechende Zahl und den Rest der Liste liefert:

   ```
   num 0  (explode "12+13")
   (12, [#"+", #"1", #"3"]) : int * char list

   num 73 (explode "12+13")
   (7312, [#"+", #"1", #"3"]) : int * char list
   ```

   Verwenden Sie dazu

   ```
   Char.isDigit : char -> bool
   Char.ord     : char -> int
   ```

2. Die eingelesene Zeichenliste soll zunächst in eine Wortliste übersetzt werden. Wörter sollen gemäß

```
datatype  token = INT of int | ADD | SUB | MUL
```

dargestellt werden. Außerdem sei die Ausnahme

```
exception Error
```

deklariert. Schreiben Sie eine iterative Prozedur

```
lex : token list -> char list -> token list  (* Error *)
```

die eine Zeichenliste in eine Wortliste übersetzt:

```
lex nil (explode "12+13")
[INT 12, ADD, INT 13] : token list

lex [INT 12] (explode "+13")
[INT 12, ADD, INT 13] : token list

lex [ADD, INT 12] (explode "13")
[INT 12, ADD, INT 13] : token list
```

Vor und nach Wörtern dürfen Leerzeichen und Zeilenwechsel stehen.

3. Schreiben Sie eine Prozedur

   ```
   atom : token list -> int * token list  (* Error *)
   ```

   die eine Zahl von einer Wortliste liest:

   ```
   atom [INT 5, ADD, INT 7]
   (5, [ADD, INT 7]) : int * token list
   ```

4. Unter einem *Term* wollen wir ein Produkt $x_1 * \cdots * x_n$ von Zahlen verstehen ($n \geq 1$). Schreiben Sie eine Prozedur

   ```
   term : token list -> int * token list  (* Error *)
   ```

   die einen möglichst langen Term von einer Wortliste liest und seinen Wert liefert:

   ```
   term(lex nil (explode "4*3*7+12"))
   (84, [ADD, INT 12]) : int * token list
   ```

   Schreiben Sie `term` mithilfe einer Prozedur

   ```
   term' : int * token list -> int * token list
   ```

   wie folgt:

   ```
   term'(4, lex nil (explode "*3*7+12"))
   (84, [ADD, INT 12]) : int * token list

   term'(12, lex nil (explode "*7+12"))
   (84, [ADD, INT 12]) : int * token list
   ```

5. Schreiben Sie eine Prozedur

```
exp : token list -> int * token list  (* Error *)
```

die einen möglichst langen Ausdruck von einer Wortliste liest und seinen Wert liefert:

```
exp(lex nil (explode "4*3*7-3*4+3-25"))
(50, []) : int * token list
```

Beachten Sie, dass Additionen und Subtraktionen nach links geklammert werden müssen, und das Multiplikationen zuerst ausgeführt werden müssen (wie in Standard ML). Schreiben Sie `exp` mithilfe einer Prozedur

```
exp' : int * token list -> int * token list
```

wie folgt:

```
exp'(84, lex nil (explode "-3*4+3-25"))
(50, []) : int * token list

exp'(72, lex nil (explode "+3-25"))
(50, []) : int * token list
```

6. Schreiben Sie eine Prozedur

```
eval : string -> string
```

die einen als Zeichenreihe dargestellten Ausdruck auswertet:

```
eval "4*3*7-3*4+3-25"
"50" : string

eval ""
"! Syntax error" : string

eval "8888888888888888"
"! Overflow" : string
```

Verwenden Sie die Prozedur

```
Int.toString : int -> string
```

7. Schreiben Sie ein interaktives Programm, dass wiederholt in einer Zeile dargestellte Ausdrücke einliest und diese auswertet:

```
# 5*7 - 2*3*5 + 15
20
```

Verwenden Sie dabei das Zeichen # als Eingabeaufforderung. Mit

```
TextIO.inputLine TextIO.stdIn : string
```

können Sie eine Eingabezeile lesen und mit

```
fn s:string => print(s^"\n")
```

können Sie eine Ausgabezeile schreiben.

8. Erweitern Sie Ihr Programm so, dass auch geklammerte Ausdrücke möglich sind:

```
# 5*(7-2)*((3))*5 + 15
390
```

Führen Sie dazu zwei neue Worte `LPAR` und `RPAR` ein, die öffnende und schließende Klammern darstellen. Erweitern Sie die Prozedur `atom` so, dass sie mithilfe der Prozedur `exp` auch geklammerte Ausdrücke lesen kann. Die Prozeduren `atom`, `term'`, `term`, `exp'` und `exp` müssen jetzt *verschränkt rekursiv* deklariert werden. Wie das geht, können Sie mithilfe des folgenden Beispiels sehen:

```
fun even x = if x=0 then true else odd(x-1)
and odd  x = if x=0 then false else even(x-1)
```

```
val even : int -> bool
val odd : int -> bool
```