8. Übungsblatt zu Programmierung

Prof. Gert Smolka, Thorsten Brunklaus
www.ps.uni-sb.de/courses/prog-ws00/

Abgabe: 12. Januar 2001 vor der Vorlesung (per Email)

Allgemeine Hinweise: Die Übungsblätter sollen in Zweiergruppen bearbeitet werden. Die Lösungen schicken Sie bitte bis Freitag vor der Vorlesung per Email an Ihren Übungsgruppenleiter. Jede Gruppe soll nur eine Lösung einreichen, versehen mit den Namen und den Matrikelnummern der Gruppenmitglieder.

Aufgabe 8.1: Lexikographische Ordnung (10) Schreiben Sie eine Prozedur

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

die zwei Zeichenlisten lexikographisch 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: Dreiecke (5+5+5) Sie sollen ein Programm schreiben, dass Dreiecke wie folgt ausgibt:

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

(a) Schreiben Sie eine Prozedur *for : int -> (int -> 'a) -> unit* wie folgt:

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

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

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

Aufgabe 8.3: Zeilennummern (10+5) 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, werden die nummerierten Zeilen in die Datei y geschrieben (statt auf den Bildschirm).

Aufgabe 8.4: Taschenrechner (5+10+5+10+10+5+5+10) 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. Für die Entwicklung größerer Programme ist schrittweises Testen unumgänglich.

(a) 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
```

(b) 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.

(c) 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
```

(d) 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
```

(e) 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
```

(f) 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 "8888888888888888888"
"! Overflow" : string
```

Verwenden Sie die Prozedur

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

(g) 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.

(h) 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
```