8. Übungsblatt zu Programmierung WS 2002 / 03

Prof. Dr. Gert Smolka und Dipl.-Inform. Thorsten Brunklaus
www.ps.uni-sb.de/courses/prog-ws02/

Abgabe: Montag, 6. Januar 2003

Schicken Sie Ihre Lösungen per E-Mail an Ihren Übungsgruppenleiter. Geben Sie in der E-Mail zuerst ihren Name und die Nummer Ihrer Übungsgruppe an. Bitte reichen Sie alle Lösungen mit nur einer E-Mail ein.

Zusammengefügt sollen Ihre Lösungen ein semantisch zulässiges Programm ergeben. Prüfen Sie das, indem Sie ihre Lösungen in eine Datei `x.sml` schreiben und diese mittels der Eingabe `use "x.sml"` in einen neu gestarteten Interpreter einlesen. Wenn Sie eine Standardstruktur X benötigen, laden Sie diese am Anfang der Datei `x.sml` mit einer Zeile `; load "`X`" ;`. Trennen Sie ihre Lösungen durch Kommentare der Form *`(* Aufgabe 8.1 *)`*.

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 ($15 = 3 * 5$) Sie sollen ein Programm schreiben, dass Dreiecke wie folgt ausgibt:

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

(a) Schreiben Sie eine Prozedur *`for: int → (int → 'a) → unit`*, sodass die Ausführung von `for` *`n p`* denselben Effekt hat wie die Ausführung von

`(`*`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 ($15 = 10 + 5$) Schreiben Sie ein Programm, das die Zeilen einer als Argument angegebenen Datei mit Zeilennummer versehen 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 8.4: Taschenrechner ($60 = 10 + 10 + 5 + 10 + 10 + 5 + 5 + 5$) Sie sollen ein interaktives Programm schreiben, das arithmetische Ausdrücke auswertet:

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

Dieses Programm ist erheblich größer als die Programme, die Sie bisher schreiben sollten, die Musterlösung umfasst etwa 60 Zeilen. Aber keine Angst, wir haben die Aufgabe für Sie in eine Folge von kleinen Teilaufgaben zerlegt, die Sie Schritt für Schritt lösen können.

Für die Aufgabe ist es erforderlich, aus einer Zeichenreihe zunächst die Wortdarstellung und dann die Baumdarstellung eines Ausdrucks zu konstruieren (siehe Kapitel 1). Die dafür notwendigen Techniken zeigen wir Ihnen Schritt für Schritt bei der Formulierung der einzelnen Teilaufgaben. In Kapitel 12 werden wir diese und andere Techniken für die syntaktische Verarbeitung ausführlicher behandeln.

Viele der für die Aufgabe benötigten Prozeduren sollen mit iterativer Rekursion realisiert werden. Dabei lernen Sie wichtige Techniken für Akkumulatorargumente kennen.

Bitte gehen Sie genau nach der folgenden Bauanleitung vor. Verwenden Sie nur die angegebenen Hilfsprozeduren. 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 [#"1", #"2", #"1", #"3", #"+", #"3", #"7"]
(1213, [#"+", #"3", #"7"]) : int * char list

num 12 [#"1", #"3", #"+", #"3", #"7"]
(1213, [#"+", #"3", #"7"]) : int * char list
```

```
num 1213 [#"+", #"3", #"7"]
(1213, [#"+", #"3", #"7"]) : int * char list
```

Die Prozedur `num` soll solange Ziffern lesen, bis die Zeichenliste leer ist, oder ein Zeichen auftaucht, das keine Ziffer ist. Das erste Argument ist ein Akkumulatorargument, das die bisher gelesene Zahl darstellt. Verwenden Sie die vordeklarierten Prozeduren

```
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
```

Beim ersten Argument von `lex` handelt es sich um ein Akkumulatorargument, das die bisher gelesenen Wörter in reversierter Reihenfolge enthält. Die Reversierung ist bequem, da dann ein neues Wort jeweils mit Cons zur bisherigen Liste hinzugefügt werden kann. Wenn alle Zeichen von der Zeichenliste gelesen sind, soll die reversierte Akkumulatorliste als Ergebnis geliefert werden.

Schreiben Sie `lex` so, dass vor und nach Wörtern Leerzeichen und Zeilenwechsel stehen dürfen.

(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
```

Wenn das erste Wort keine Zahl darstellt, soll die Ausnahme `Error` geworfen werden.

(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 iterativen Prozedur

```
term' : int * token list → int * token list
```

die den Wert des bisher gelesenen Teilterms in einem Akkumulatorargument weiterreicht:

```
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) Unter einem **Ausdruck** wollen wir ein Folge $t_1 \pm \cdots \pm t_n$ von Termen t_i verstehen ($n \geq 1$), die jeweils durch einen der Operatoren $+$ oder $-$ verbunden sind. 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
```

Schreiben Sie `exp` mithilfe einer iterativen Prozedur

```
exp' : int * token list → int * token list
```

die den Wert des bisher gelesenen Teilausdrucks in einem Akkumulatorargument weiterreicht:

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

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

Addition und Subtraktion sollen wie in Standard ML nach links geklammert werden. Die Prozeduren `exp` und `exp'` können ähnlich wie die bereits geschriebenen Prozeduren `term` und `term'` realisiert werden, wobei `exp` und `exp'` die Prozedur `term` aufrufen (anstelle von `atom`).

(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 Ausdrücke einliest und auswertet:

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

Der Ausdruck muss dabei jeweils in einer Zeile dargestellt sein. Verwenden Sie das Zeichen # als Eingabeaufforderung. Ihr Programm soll solange Ausdrücke einlesen, bis es durch die Eingabe von `C-c` abgebrochen wird. 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 Wörter `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. Damit das geht, müssen die Prozeduren `atom`, `term'`, `term`, `exp'` und `exp` jetzt verschränkt rekursiv deklariert werden (`and` statt `fun`, siehe Abschnitt 5.10 im Skript).