# 9. Übungsblatt zu Programmierung (WS 1999/2000)

Andreas Rossberg, Prof. Gert Smolka

**Abgabe:** Freitag, 7. Januar 2000 vor der Vorlesung

## Aufgabe 9.1: Permutationen (10 Punkte)

(a) Schreiben Sie eine Prozedur

```
intersperse: 'a -> 'a list -> 'a list list
```

die eine Liste *xs* und ein Element *y* nimmt und daraus alle möglichen Listen erzeugt, bei denen *y* an einer beliebigen Position in *xs* eingefügt wurde. Zum Beispiel sei:

```
intersperse 0 [1,2] = [[0,1,2], [1,0,2], [1,2,0]]
```

Tipp: Verwenden Sie im Rekursionsfall `map`.

(b) Schreiben Sie nun eine Prozedur

```
permutations: 'a list -> 'a list list
```

die alle Permutationen einer Liste berechnet. Die leere Liste hat nur eine mögliche Permutation, nämlich die leere Liste selbst. Für eine nichtleere Liste lässt sich eine Permutation finden, indem man eine Permutation des Listenrumpfes erzeugt und dann den Listenkopf an beliebiger Stelle einfügt. Mit Hilfe der Prozeduren `map` und `intersperse` ist es auf Grundlage dieser Idee einfach, *alle* Permutationen zu erzeugen.

## Aufgabe 9.2: Stringreplikation (4 Punkte)

(a) Schreiben Sie eine Prozedur

```
replicate: string -> int -> string
```

die für eine nicht-negative Zahl einen String entsprechend oft repliziert. So sei beispielsweise:

```
replicate "palp" 3 = "palppalppalp"
```

(b) Schreiben Sie eine Prozedur

```
bar: int -> string
```

die für eine nicht-negative Zahl einen entsprechend langen 'Balken' aus Sternchen erzeugt. So sei beispielsweise:

```
bar 5 = "*****"
```

Benutzen Sie `replicate`.

**Aufgabe 9.3:** Konversion zwischen Strings und Listen (8 Punkte)

Schreiben Sie zwei Prozeduren

```
explode': string -> char list
implode': char list -> string
```

mit denen man einen String in die Liste der enthaltenen Zeichen und zurück konvertieren kann (die Prozeduren entsprechen also den Standardprozeduren `explode` und `implode`). Sie können dazu die Standardprozeduren

```
size:        string -> int
str:         char -> string
String.sub:  string * int -> char
```

benutzen, mit denen sich die Länge eines Strings ermitteln, ein Zeichen in einen String konvertieren, oder ein einzelnes Zeichen aus einem String extrahieren lässt.

**Aufgabe 9.4:** Stringvergleich (6 Punkte)

Schreiben Sie eine Prozedur

```
compareString: string * string -> order
```

die zwei Strings gemäß der üblichen Sortierung in Wörterbüchern (*lexikographische Ordnung*) vergleicht. So ist zum Beispiel

```
"bla" < "blab" < "bra"
```

Das Verhalten Ihrer Prozedur soll also mit `String.compare` identisch sein. Für den Vergleich einzelner Zeichen verwenden Sie `Char.compare`.

**Aufgabe 9.5:** Zeilennummern (7 Punkte)

Schreiben Sie ein direkt ausführbares Programm `num`, welches als Argumente zwei Dateinamen erwartet. Es soll die mit dem ersten Namen bezeichnete Datei einlesen und eine neue Datei unter dem an zweiter Stelle angegebenen Namen erzeugen. Die Zeilen der Quelldatei werden in die Zieldatei kopiert, dabei wird aber jeder Zeile ihre Zeilennummer vorangestellt. Zum Beispiel soll aus der Datei `foo.txt`:

```
Ursenfuns Pillepall
Gnalfo Punzenreuter
Gismalda Herklinde Schrunzengnulp
Pazo der Armpirat
```

durch den Aufruf

```
% num foo.txt bar.txt
```

eine Datei `bar.txt`:

```
1: Ursenfuns Pillepall
2: Gnalfo Punzenreuter
3: Gismalda Herklinde Schrunzengnulp
4: Pazo der Armpirat
```

erzeugt werden.

**Aufgabe 9.6:** Dateigröße (6 Punkte)

Schreiben Sie ein direkt ausfürhrbares Programm `size`, das als Argumente eine beliebige Anzahl von Dateinamen erwartet und die Gesamtgröße (also die Anzahl der Zeichen) aller dieser Dateien berechnet und ausgibt.

**Aufgabe 9.7:** Umgekehrtes Echo (5 Punkte)

Schreiben Sie ein direkt ausführbares, interaktives Programm, das zeilenweise Eingaben vom Benutzer einliest, und diese in zeichenweise umgekehrter Reihenfolge wieder ausgibt. Dazu soll jeweils ein einfache Eingabeaufforderung ausgegeben werden. Das Programm wird beendet, indem der Benutzer mit Ctrl-D (unter Windows Ctrl-Z) ein Dateiende-Zeichen eingibt.

**Aufgabe 9.8:** Taschenrechner (Milleniums-Special, keine Punkte)

Entwickeln Sie ein direkt ausführbares Programm, das interaktiv einfache Ausdrücke vom Benutzer einliest, und deren Ergebnis ausgibt. Als Eingaben sollen Ausdrücke der folgenden Form akzeptiert werden:

- $(a_1+a_2)$
- $(a_1-a_2)$
- $(a_1*a_2)$
- Zahlen (eine Folge von Ziffern)

Dabei sind $a_1$ und $a_2$ jeweils wieder Ausdrücke in derselben Form. Beachten Sie, dass Ausdrücke hier immer vollständig geklammert sind. Sie brauchen zunächst keine Leerzeichen innerhalb von Ausdrücken zu erlauben.

(a) Schreiben Sie zunächst eine Prozedur

```
splitDigits: char list -> char list * char list
```

welche alle führenden Ziffern aus einer Zeichenliste abtrennt und ein Paar aus den Ziffern und den Restzeichen liefert. Tipp: Die Prozedur `Char.isDigit` ist hier hilfreich.

(b) Eine Prozedur

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

soll den ersten vollständigen Ausdruck am Anfang der Zeichenliste finden und auswerten. Unterausdrücke werden rekursiv ausgewertet. Zurückgeliefert werden sollen der Ergebniswert und die Restliste. Im Falle einer ungültigen Eingabe soll eine geeignete Ausnahme geworfen werden.

(c) Die Prozedur

```
evalLine: char list -> int option
```

soll eine komplette Eingabe auswerten, indem sie `eval` anwendet und überprüft, ob die Restliste aus dem Zeilenwechsel besteht. Falls das nicht der Fall ist, oder die Eingabe fehlerhaft war, soll die Prozedur `NONE` liefern.

(d) Mit Hilfe der obigen Prozeduren sollten Sie ein interaktives Programm schreiben können, dass solange Zeilen einliest und Ausdrücke auswertet, bis der Benutzer abbricht.

(e) Schreiben Sie jetzt eine Prozedur

```
skipSpaces: char list -> char list
```

welches aus einer Liste von Zeichen alle führenden Leerzeichen entfernt. Indem Sie diese Prozedur an den geeigneten Stellen anwenden, können Sie ihr Programm so erweitern, dass es auch Leerzeichen innerhalb von Ausdrücken erlaubt.