Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion
CAU Kiel, Informatik und Prakt. Mathematik
Prof. Dr. M. Hanus, Dr. Frank Huch

24118 Kiel
Christian-Albrechts-Platz 4
Tel.: 0431 / 880-7277

9. Übung „Prinzipien von Programmiersprachen“

Bearbeitung bis zum 18. Januar 2004

Aufgabe 1

In der Vorlesung wurden lazy funktionale Sprachen mit Hilfe der Launchbury-Semantik formalisiert.

a) Berechnen Sie die lazy Auswertung des folgenden Programms:

```
main = let ones = 1:ones in
         (head ones) + (head (tail ones))

head (x:xs) = x
tail (x:xs) = xs
```

b) Was liefert die Launchbury-Semantik für

```
main = let ones = 1:ones in
         (head ones, head (tail ones))
```

c) Wie verändert sich die Auswertung, wenn `ones` als Funktion definiert wird:

```
main = let o = ones in
         (head ones) + (head (tail ones))

ones = 1: ones
```

Hinweis: Beachten Sie, dass die Programme nicht normalisiert sind! Ersetzen Sie zunächst das Patternmatching durch `case`-Ausdrücke und definieren dann in einem `let` Variablen für alle nicht-variablen Argumente.

Aufgabe 2

In dieser Aufgabe sollen Sie Turingmaschinen (vgl. 6. Übung, Aufgabe 2) in Haskell implementieren.

a) In Haskell können Stacks sehr einfach als Listen implementiert werden.

```
data Stack a = Stack [a]
```

Definieren Sie folgende Stack-Operationen:

- `push :: a -> Stack a -> Stack a`
- `pop :: Stack a -> (a, Stack a)`
- `isEmptyStack :: Stack a -> Bool`
- `stackToList :: Stack a -> [a]`

b) Definieren Sie eine Datenstruktur `TM a`, welche die Syntax einer Turingmaschine definiert. Das Alphabet entspricht hierbei der Typvariablen `a`.

c) Definieren Sie eine Funktion `runTM :: TM a -> [a] -> [a]`, welche eine Turingmaschine ausführt und ihre Ausgabe als Ergebnis hat.

d) Führen Sie die Turingmaschine zur Multiplikation von Unärzahlen (siehe 6. Übung) aus. Am einfachsten verwenden Sie das Alphabet `Char`, da dann der Typ `String` (=`[Char]`) als Ein- und Ausgabe verwendet werden kann.

Aufgabe 3
In der Vorlesung wurde die Funktion `twice` definiert:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Hierbei ist ein Typ für `twice` vorgegeben. Zeigen Sie, daß unter geeigneten Typannahmen für `f` und `x` die Regel für `twice` wohlgetypt ist, falls der obige Typ für `twice` in den Typannahmen enthalten ist.