PHILIPPS-UNIVERSITÄT MARBURG
Fachbereich Mathematik und Informatik
Prof. Dr. R. Loogen

D-35032 Marburg
Hans-Meerwein-Straße
Lahnberge

Klausur zur „Praktischen Informatik III“, WS 2001/02

31. Januar 2002

Hinweise:

- **Bearbeitungszeit:** 2 Stunden
  Gesamtpunktzahl: 80 Punkte
  Zum Bestehen der Klausur sind **32 Punkte** erforderlich.
- Hilfsmittel sind nicht erlaubt.
- Bis auf das Konzeptpapier sind alle ausgehändigten Blätter zurückzugeben.

Viel Erfolg!

Name: .....

Mat.-Nr.: ..... Studienfach: .....

Aufgabe	max. Punktzahl	erreichte Punktzahl
1	5	
2	8	
3	8	
4	8	
5	10	
6	9	
7	8	
8	8	
9	16	
Summe	80	
Note:		

1. Fragebogen — 5 Punkte

(a) Kann die Listenspiegelungsfunktion `reverse :: [a] -> [a]` als Instanz von `foldr` definiert werden? ☐ ja ☐ nein

(b) Besitzt der Ausdruck `\ x y -> y (x x)` im Hindley-Milner Typsystem einen allgemeinsten Typ? ☐ ja ☐ nein

(c) Bildet der monadische Typ `IO ()` eine Instanz der Typklasse `Eq`? ☐ ja ☐ nein

(d) Versteht man unter referentieller Transparenz die Zeigerfreiheit in funktionalen Sprachen? ☐ ja ☐ nein

(e) Definiert die Abseitsregel (offside rule) in Haskell eine formatgebundene Syntax? ☐ ja ☐ nein

(f) Hat die Funktion `foldr` den allgemeinsten Typ `(a -> b -> c) -> a -> [b] -> c`? ☐ ja ☐ nein

(g) Stellt Haskell mit dem Datentyp `Integer` ganze Zahlen beliebiger Größe bereit? ☐ ja ☐ nein

(h) Bilden algebraische Datenstrukturen in Haskell die einzige Möglichkeit, Daten zu strukturieren? ☐ ja ☐ nein

(i) Ist der Ausdruck `(2*3):([4*5]++[6*7])` in Kopfnormalform? ☐ ja ☐ nein

(j) Ist der Ausdruck `([2*3,4*5]++[6*7])` in Kopfnormalform? ☐ ja ☐ nein

2. Funktionsdefinitionen — 8 Punkte

Die Funktion `removeNth :: Int -> [a] -> [a]` entfernt das `n`-te Element einer Liste, wenn `n` als erster Parameter gegeben ist. Definieren Sie diese Funktion

(a) rekursiv

(b) mit Funktionen höherer Ordnung.

3. Nachweis von Programmeigenschaften **8 Punkte**

Gegeben seien die folgenden Funktionsdefinitionen:

```
(++) :: [a] -> [a] -> [a]
[]     ++  ys =  ys             (app1)
(x:xs) ++  ys =  x : (xs ++ ys) (app2)
```

```
(||) :: Bool -> Bool -> Bool
True  || b =  True     (||1)
False || b =  b        (||2)
```

```
elem ::  a  -> [a] -> Bool
elem x []     =  False              (elem1)
elem x (y:ys) =  x == y || elem x ys (elem2)
```

Zeigen Sie ausführlich, dass für beliebige endliche Listen `xs, ys :: [a]` und Listenelemente `z :: a` gilt:

```
elem z (xs ++ ys) = elem z xs || elem z ys.
```

Die Assoziativität von `(||)` können Sie ohne Beweis voraussetzen.

4. Folgen von IO-Aktionen 8 Punkte

(a) Definieren Sie eine Funktion

```
accumulate :: [IO a] -> IO [a]
```

die eine Folge von IO-Aktionen ausführt und die Ergebnisse in einer Liste akkumuliert.

(b) Definieren Sie eine Funktion

```
seqList :: [a -> IO a] -> a -> IO a
```

die eine Folge von IO-Aktionen mit Wertübergabe ausführt und die Ergebnisse von einer Aktion zur nächsten weiterreicht.

5. Funktionen höherer Ordnung 10 Punkte

(a) Gegeben seien folgende Funktionsdefinitionen:

```
ones :: Num a => a -> [a]
ones 0 = []
ones n = 1 : ones (n-1)

length :: [a] -> Int
length []    = 0
length (_:t) = 1 + length t

sum :: Num a => [a] -> a
sum []    = 0
sum (h:t) = h + sum t
```

Entwickeln Sie zu den folgenden Funktionen effizientere Versionen, die mit einem Listendurchlauf auskommen und Funktionen höherer Ordnung verwenden:

i.
```
list2ones :: [a] -> [Int]
list2ones =  ones . length
```

ii.
```
average    :: Fractional a => [a] -> a
average xs =  sum xs / length xs
```

(b) Verwenden Sie Funktionen höherer Ordnung zur Definition von Funktionen `f1` und `f2`, mit denen die Reduktion des folgenden Ausdrucks das angegebene Ergebnis liefert:

```
f1 ( f2 (*) [1,2,3,4] ) 5  =>*  [5,10,15,20]
```

Hinweis: Folgende Funktionen sind in Haskell vordefiniert:

```
-- Funktionsapplikation
($) :: (a -> b) -> a -> b
f $ x = f x

-- Vertauschen von Funktionsargumenten
flip :: (a -> b -> c) -> b -> a -> c
flip f y x = f x y
```

6. Typen

9 Punkte

Die Funktionen `curry` und `uncurry` sind in Haskell wie folgt vordefiniert:

```
curry f x y = f (x,y)          uncurry f (x,y) = f x y
```

(a) Geben Sie den allgemeinsten Typ dieser Funktionen im Hindley-Milner Typsystem an.

(b) Bestimmen Sie zu den folgenden Ausdrücken jeweils den allgemeinsten Typ, falls dieser existiert. `id` bezeichnet die durch die Gleichung `id x = x` festgelegte Identitätsfunktion.

i. `curry id`

ii. `curry curry`

iii. `uncurry uncurry`

7. Mengen

8 Punkte

Gegeben sei folgender Beginn einer Moduldefinition

```
module Set
 (   Set,
     emptySet,            -- Set a
     singleton,           -- Eq a => a -> Set a
     memSet,              -- Set a -> a -> Bool
     union, intersection, -- Set a -> Set a -> Set a
     complement           -- Set a -> Set a
 )  where
```

(a) Geben Sie eine Implementierung dieses Moduls auf der Basis der folgenden Typfestlegung an:

```
type Set a = a -> Bool
```

(b) Welches Problem tritt auf, wenn man dem Modul eine Funktion

```
subset :: Eq a => Set a -> Set a -> Bool
```

hinzufügen möchte?

8. Polynomfunktionen

8 Punkte

Polynome in einer Variablen können in Haskell als unendliche Listen von Zahlen dargestellt werden: `type Poly = [Float]`.

Beispielsweise repräsentiert die Liste `(1:0:2:(-4):0:3:repeat 0)` das Polynom $1 + 2x^2 - 4x^3 + 3x^5$.

Definieren Sie die folgenden Operationen über Polynomen:

(a) `scale :: Float -> Poly -> Poly` zur Multiplikation mit einem Skalar:

$$a * \sum_{i=0}^{\infty} b_i x^i = \sum_{i=0}^{\infty} a b_i x^i$$

(b) `addPoly :: Poly -> Poly -> Poly` zur Addition von zwei Polynomen

$$\sum_{i=0}^{\infty} a_i x^i + \sum_{i=0}^{\infty} b_i x^i = \sum_{i=0}^{\infty} (a_i + b_i) x^i$$

(c) `mulPoly :: Poly -> Poly -> Poly` zur Multiplikation von Polynomen

$$\sum_{i=0}^{\infty} a_i x^i * \sum_{i=0}^{\infty} b_i x^i = a_0 * \sum_{i=0}^{\infty} b_i x^i + x * \sum_{i=0}^{\infty} a_{i+1} x^i * \sum_{i=0}^{\infty} b_i x^i$$

9. Suchprogramm

16 Punkte

Schreiben Sie eine Funktion `suche :: [Wort] -> Matrix -> [Wort]`, die zu einer Liste von Worten `ws` und einer Matrix von Zeichen `css` eine Liste der Worte in `ws` bestimmt, die in der Matrix enthalten sind. Dabei sei

```
type Wort   = String
type Matrix = [String]
```

Eine Matrix ist also eine Liste von Listen von Zeichen (`type String = [Char]`). Ein Wort ist in einer Matrix enthalten, wenn es vorwärts oder rückwärts gelesen als Teil einer Zeile oder Spalte auftritt.

Zum Beispiel soll der Aufruf

```
suche ["abc","aa","bb"] ["bcba",
                         "abbc",
                         "caba"]
```

das Ergebnis `["abc","bb"]` liefern.

Hinweis: Definieren Sie Hilfsfunktionen

- `transpose :: [[a]] -> [[a]]` zur Transposition einer Matrix, d.h. zur Überführung einer Matrixdarstellung als Liste von Zeilen in eine Darstellung als Liste von Spalten.
- `subList :: Eq a => [a] -> [a] -> Bool` zum Test, ob eine Liste Teilliste einer anderen Liste ist.