Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Institut für Wissens- und Sprachverarbeitung
Prof. Dr. Dietmar Rösner

Magdeburg, 20.03.07

Klausur

Programmierkonzepte und Modellierung (PKM)

WS 2006/2007

Name:	Matrikelnummer:
Vorname:	Studiengang:
Blattanzahl:	
Unterschrift Student/in:	Unterschrift Aufsicht:

Aufgabe 1	Aufgabe 2	Aufgabe 3	Aufgabe 4	Summe
(von 10)	(von 10)	(von 10)	(von 10)	(von 40)

Allgemeine Hinweise

- Schreiben Sie auf jedes Blatt Ihren Namen, Ihre Matrikelnummer und die Seitennummer.
- Benutzen Sie für jede Aufgabe ein eigenes Blatt.
- Die Programme sind gut zu kommentieren.
- Zugelassene Hilfsmittel: Fünf handgeschriebene DIN-A4-Zettel und Schreibmaterialien
- Die Klausur besteht aus 4 Aufgaben, die Bearbeitungszeit beträgt 120 Minuten.

Klausur zur Vorlesung „Programmierkonzepte und Modellierung (PKM)“ (WS 2006/2007)

20. März 2007, 10:00 – 12:00

Aufgabe 1 [Gesamtpunktzahl: 10]

In dieser Aufgabe geht es darum, auf unterschiedliche Weise zu einer Liste `list` die Liste ihrer endlichen „Listenanfänge“ zu bestimmen, d.h. eine Liste, welche die leere Liste und alle – jeweils mit dem ersten Element beginnend – aus direkt aufeinanderfolgenden Elementen von `list` gebildeten Teillisten enthält.

Beispiel: Eine polymorphe **Haskell-Funktion** `sublists` sollte als Wert von
```
sublists [1,2,3,4]
```
liefern
```
[[],[1],[1,2],[1,2,3],[1,2,3,4]]
```

a) Definieren Sie die **Haskell-Funktion** `sublists` *durch eine Listenkomprehension.* [Punkte: 2]

b) Definieren Sie die **Haskell-Funktion** `sublists` *direkt durch Rekursion.* [Punkte: 3,5]

c) Könnte man Ihre Lösungen für `sublists` aus a) bzw. aus b) auch auf *unendliche* Listen anwenden? Begründen Sie Ihre Antworten. [Punkte: 1]

d) Durch welche Query in **Prolog** mit dem eingebauten Prädikat `append` könnten Sie sich nacheinander die Listenanfänge zu einer gegebenen *endlichen* Liste liefern lassen? [Punkte: 1]

e) Nutzen Sie d), um ein **Prolog-Prädikat** `sublists(List,Sublists)` zu definieren, das für beliebige *endliche* Listen zutrifft, wenn `Sublists` die Liste der „Listenanfänge“ von `List` ist. [Punkte: 2,5]

Beispiel:
Die Query
```
sublists( [1,2,3], Sublists).
```
soll ergeben
```
Sublists = [[],[1],[1,2],[1,2,3]]
```

Aufgabe 2 [Gesamtpunktzahl: 10]

Aus Haskell kennen Sie die Funktion `concat`, die einer Liste von Listen die Liste mit den Elementen dieser Listen zuordnet.

Die Funktionalität von `concat` soll nun auch in **Scheme** bzw. **Prolog** bereitgestellt werden.

a) Definieren Sie **in Scheme** eine Funktion `concat`, die einer Liste von Listen die Liste mit den Elementen dieser Listen in ihrer gegebenen Reihenfolge zuordnet. [Punkte: 3]

b) Definieren Sie **in Scheme** eine Funktion `concat-stream`, die einem (ggf. unendlichen) *Strom* mit (endlichen) *Listen* als Elementen einen *Strom* gebildet aus den Elementen dieser Listen in ihrer gegebenen Reihenfolge zuordnet.
(Sie können die aus der Vorlesung bekannten Basis-Funktionen für Ströme wie `cons-stream`, `head`, `tail`, `empty-stream?`, sowie die Konstante `the-empty-stream` verwenden). [Punkte: 4]

Beispiel:
Dem Strom beginnend mit den Elementen
`(1 2), (a b c), (() (4 5 6)), ...`
soll durch `concat-stream` der Strom beginnend mit den Elementen
`1, 2, a, b, c, (), (4 5 6), ...`
zugeordnet werden.

c) Definieren Sie ein **Prolog-Prädikat** `concat(ListOfLists, List)`, das zutrifft, wenn `ListOfLists` eine Liste mit Listen als Elemente ist und `List` eine Liste ist, welche die Elemente der Elemente von `ListOfLists` in ihrer gegebenen Reihenfolge enthält. [Punkte: 3]

Beispiel:
Die Query
```
concat([[1,2],[a,b,c],[[],[4,5,6]]], List).
```
soll ergeben
```
List = [1,2,a,b,c,[],[4,5,6]]
```

Aufgabe 3 [Gesamtpunktzahl: 10]

Gegeben seien die folgenden Definitionen der **Haskell-Funktionen** `zip` und `unzip`.

```
zip :: [a] -> [b] -> [(a,b)]

zip (x:xs) (y:ys) = (x,y):zip xs ys                    (zip.1)

zip _ _ = []                                           (zip.2)

unzip :: [(a,b)] -> ([a],[b])

unzip [] = ([],[])                                     (unzip.1)

unzip ((x,y):ps) = (x:xs, y:ys)                        (unzip.2)
                   where
                   (xs,ys) = unzip ps                  (unzip.2')
```

a) **Beweisen Sie** durch strukturelle Induktion über die Listenlänge, dass für alle endlichen Listen `ps` von Paaren gilt:

```
zip (fst (unzip ps)) (snd (unzip ps)) = ps
```

[Punkte: 4]
b) Geben Sie ein (möglichst einfaches) Beispiel dafür an, dass die folgende Beziehung **nicht** immer (d.h. nicht für *beliebige* Listen `xs` und `ys`) gilt:

```
unzip (zip xs ys) = (xs,ys)
```

[Punkte: 2]
c) Unter welcher Bedingung an die Listen `xs` und `ys` gilt die folgende Beziehung?

```
unzip (zip xs ys) = (xs,ys)
```

Beweisen Sie dies durch strukturelle Induktion über die Listenlänge.
[Punkte: 4]

Wichtiger Hinweis: Geben Sie bitte bei allen Umformungen in Beweisschritten jeweils die verwendeten Gleichungen an.

Aufgabe 4 [Gesamtpunktzahl: 10]

Bei dem auf den Mathematiker Cantor (aus Halle/Saale) zurückgehenden sog. Cantorschen Diagonalverfahren arbeitet man mit einer (als Kreuzprodukt von N mit N gebildeten) unendlichen Matrix aus Paaren natürlicher Zahlen, wobei das erste Element jedes Paars der Zeilenindex und das zweite der Spaltenindex ist.
Die linke obere Ecke dieser Matrix sieht dann wie folgt aus:

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	...
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	...	...
(3,1)	(3,2)	(3,3)	(3,4)	...	...	...
(4,1)	(4,2)	(4,3)	...	...	...	...
(5,1)	(5,2)	...	...	...	...	...
(6,1)	...	...	...	...	...	...
...	...	...	...	...	...	...

a) Definieren Sie *mit einer Listenkomprehension* eine unendliche Liste `intpairs` **in Haskell**, welche die Elemente der obigen Matrix jeweils diagonalenweise mit einem Paar in der ersten Spalte beginnend und dann nach rechts oben aufsteigend enthält. [Punkte: 5]
Mit anderen Worten:
`intpairs` soll wie folgt aussehen:

```
[(1,1), (2,1), (1,2), (3,1), (2,2), (1,3), (4,1), (3,2), ... ]
```

b) Die Liste `intpairs` enthält nun „zuviele Elemente" in dem Sinne, dass zahlreiche Paare auftreten, welche dieselbe rationale Zahl – gebildet als Bruch mit dem ersten Element als Zähler und dem zweiten Element als Nenner – darstellen. So sind z.B. `(1,1)`, `(2,2)` und allgemein `(n,n)` Varianten der Zahl 1, `(1,2)`, `(2,4)`, `(3,6)`, .. Varianten der Zahl 1/2, usw.

Definieren Sie sich zunächst ein **Haskell-Prädikat** `pairEq p1 p2`, das genau dann zutrifft, wenn die Paare `p1` und `p2` dieselbe rationale Zahl darstellen. [Punkte: 2,5]

c) Wenden Sie damit die aus der Vorlesung bekannte Technik des „Siebens" so auf die unendliche Liste `intpairs` an, dass wiederholte Vorkommen von als rationale Zahlen gleichwertigen Paaren **nicht mehr enthalten** sind.
[Punkte: 2,5]
Zur Erinnerung: Die Berechnung der Liste aller Primzahlen erfolgte durch „Sieben" nach Eratosthenes mit folgendem Haskell-Code:

```
primes = sieve [2 ..]
sieve [] = []
sieve (x:xs) = x:(sieve (filter (\y -> not (divisiblep y x)) xs))
divisiblep x y = rem x y == 0
```