Pseudoklausur SE-1, WS-2011/12

zu Übungs- und Vorbereitungszwecken
von Manuel Hoffmann

Diese Pseudoklausur soll lediglich eine zusätzliche Ansammlung an Übungsaufgaben sein, deren Stil auch in einer richtigen Klausur drankommen können. Die Auswahl der Aufgaben soll kein Hinweis darauf sein, welche Aufgaben in der richtigen Klausur drankommen.

0.1 Multiple Choice

Ja Nein

☐ ☐ Ein Programm, dessen Korrektheit bewiesen wurde, terminiert immer.

☐ ☐ Anforderungen an ein Softwaresystem können funktional und nicht-funktional sein.

☐ ☐ Eine Zeichenreihe ist eine endliche oder unendliche Folge von Zeichen.

☐ ☐ Eine Sprachdefinition mit Syntaxdiagramm kann ein Startsymbol enthalten.

☐ ☐ Die Mächtigkeit der Menge der Produktionen ist bei einer kontextfreien Grammatik immer kleiner als die Mächtigkeit der Terminalsymbolen.

☐ ☐ Ein Satz ist genau dann eindeitig, wenn er genau eine Rechtsableitung besitzt.

☐ ☐ Es gibt formale Sprachen ohne Semantik.

☐ ☐ Zwei Sprachen mit unterschiedlichen Produktionsregeln können nie gleich sein.

☐ ☐ Eine partielle Funktion ist auf mindestens einem Element ihres Argumentbereichs definiert.

☐ ☐ Bezeichner einer Programmiersprache können überladen sein.

☐ ☐ Deklarationen dienen dazu, einem in einem Programm verwendeten Element einen Wert zuzuweisen.

☐ ☐ Eine Funktionsdeklaration heißt direkt rekursiv, wenn der definierende Ausdruck eine Anwendung der definierten Funktion enthält.

☐ ☐ Eine repetitive Funktionsdeklaration ist linear rekursiv.

☐ ☐ Ein Typ mit Typvariablen ist immer spezieller als ein Typ ohne Typvariablen.

☐ ☐ Ein Programm, das richtige Ergebinsse liefert, wenn es terminiert heißt partiell korrekt.

☐ ☐ Mit Parameterinduktion kann man direkt zeigen, dass ein Programm terminiert.

☐ ☐ Bei einem Terminierungsbeweis muss man lediglich zeigen, dass man aus der Menge der zulässigen Parameter bijektiv in eine noethersche Ordnung abbilden kann.

☐ ☐ Ein Algorithmus, der für gleiche Eingabedaten immer gleich abläuft, heißt deterministisch.

☐ ☐ Ein Algorithmus, der für verschiedene Eingabedaten immer das gleiche Ergebnis ausliefert, heißt determiniert.

□ □ Die Schleifenanweisung einer for-Schleife kann ohne weitere Änderungen durch die einer while-Schleife ersetzt werden.

□ □ Die Schleifenanweisung einer while-Schleife kann ohne weitere Änderungen durch die einer for-Schleife ersetzt werden.

0.2 Funktionale Programmierung

(a) Ihnen sind die folgenden Funktionsdefinitionen gegeben, jedoch fehlen die Signaturen. Ergänzen Sie die Signaturen mit dem allgemeinstmöglichen Typ.

```
a ::

a x y z = (y x) && (y z)


b ::

b x y = b (x + head y) (tail y)


c ::

c x (y:z) = c [y] x ++ z


d ::

d x (y:z) = d (y x) z
```

(b) Implementieren Sie die Funktion *foldl*, die mit Hilfe einer Funktion und einem Startwert eine Liste von links ausgehend zu einem Ergebnis zusammenfaltet.

```
foldl ::
```

Implementieren Sie die Funktion *foldr*, die mit Hilfe einer Funktion und einem Startwert eine Liste von rechts ausgehend zu einem Ergebnis zusammenfaltet.

```
foldr ::
```

Implementieren Sie die Funktion *map*, die eine Funktion und eine Liste von Werten nimmt und als Ergebnis eine Liste mit den Funktionswerten liefert. Beispiele:

```
map ((+) 3) [0,1,2,3]             => [3,4,5,6]
map length [[], [2,3,4], [2,1]]   => [0,3,2]

map ::
```

0.3 Funktionale Programmierung

Gegeben sei der folgende Datentyp, der einen Binärbaum darstellt, dessen Knoten und Blätter mit Bruchzahlen markiert sind.

```
data FracTree = FracNode FracTree FracTree Integer Integer
              | FracLeaf Integer Integer
```

Der erste Integer entspricht dem Zähler, der zweite dem Nenner; d.h.

```
(FracLeaf 5 7)
```

entspricht dem Bruch: $\frac{5}{7}$

(a) Implementieren Sie eine Funktion *countN*, die zählt, wie viele Knoten eines *FracTree* natürliche Zahlen sind. Geben Sie auch eine sinnvolle Signatur an. Beispiele:

```
countN (FracNode (FracLeaf 2 5) (FracLeaf 2 1) 3 8) == 1
countN (FracNode (FracLeaf 12 3) (FracLeaf 14 7) 2 4) == 2
```

(b) Implementieren Sie eine Funktion *infty*, die testet, ob ein FracTree einen Bruch mit Nenner 0 enthält. Geben Sie auch eine sinnvolle Signatur an. Beispiele:

```
infty (FracNode (FracLeaf 2 5) (FracLeaf 2 1) 3 8) == False
infty (FracNode (FracLeaf 2 5) (FracLeaf 2 0) 3 8) == True
```

0.4 Funktionale Programmierung

Gegeben ist eine Datenstruktur *ListTree*, die einen Binärbaum darstellt, dessen Knoten und Blätter mit Listen von Integern markiert sind. Diese Datenstruktur verfügt über die Konstruktoren *ListNode ListTree ListTree [Integer]* und *ListLeaf [Integer]*. Außerdem ist die Datenstruktur *IntTree* mit Integer-Markierungen gegeben. Diese hat die Konstruktoren *IntNode IntTree IntTree Integer* und *IntLeaf Integer*.

(a) Schreiben Sie eine Funktion *sumUpToIntTree*, die einen ListTree bekommt und einen IntTree zurückgibt. Dabei soll an jeder entsprechenden Markierung des zurückgegebenen Baumes die Summe der Werte des Eingabebaums stehen. Vergessen Sie die Signatur nicht. Beispiel:


```
sumUpToIntTree ::
```

0.5 Sprachen

(a) Betrachten Sie folgende (unvollständige) Sprachdefinition mit Syntaxdiagrammen:

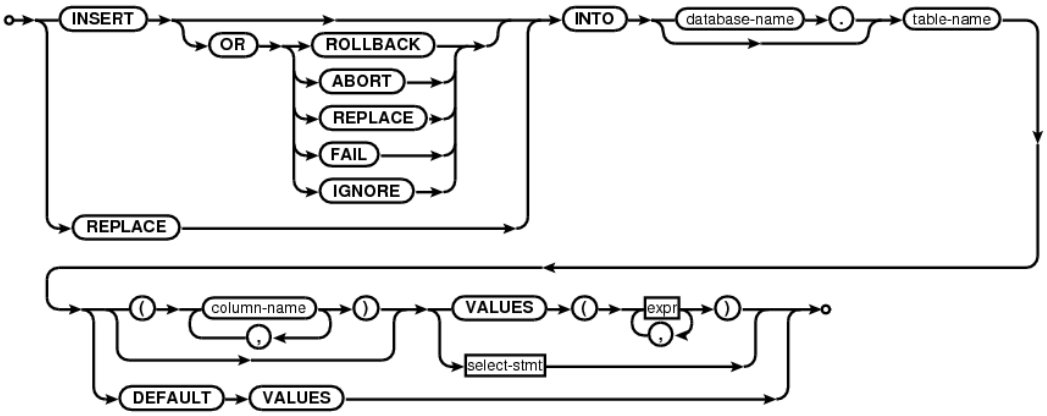

Abbildung 0.1: Insert-Statement von SQLite

database-name, *table-name* und *column-name* sollen dabei Konkatenationen von Groß- und Kleinbuchstaben sein, d.h. insbesondere keine Leer- und Sonderzeichen enthalten, ebenso für die Ausdrücke des nicht dargestellten Diagramms *expr*. Das weiterhin nicht dargestellte Diagramm *select-stmt* können Sie vernachlässigen.

Kreuzen Sie diejenigen Worte an, die mit dieser Sprachdefinition gebildet werden können.

- ☐ REPLACE OR ABORT INTO mytable DEFAULT VALUES
- ☐ INSERT INTO mydatabase DEFAULT VALUES
- ☐ INSERT OR FAIL INTO mydb.mytbl (INSERT, OR, FAIL) VALUES (mydb, mytbl)
- ☐ insert into MYDATABASE default values

(b) Definieren Sie eine Grammatik, deren erzeugte Sprache nur das leere Wort enthält.

0.6 Terminierung

(a) Geben Sie für folgende Funktionen je die größtmögliche Parametermenge an, sodass sie terminieren

```
f :: Int -> String -> Int
f 0 _ = 77
f x y = f (x - 1 + length y) (tail y)

g :: Int -> String -> Int
g 0 _ = 77
g _ "" = 88
g x y = g (x - 1 + length y) (tail y)

h :: Int -> String -> Int
h _ "" = 88
h 0 _ = 77
h x y = h (x - 1 + length y) (tail y ++ tail y)
```

(b) Geben Sie für die folgende Funktion den größtmöglichen Parameterbereich an und beweisen sie, dass sie terminiert:

```
t :: Int -> Int -> Int
t 0 0 = 0
t 0 x = t x + (x `div` 2)
t x y = y * (t (x - (x `div` 2)) (x + x `div` 2))
```

0.7 Prozedurale Programmierung

Gegeben sind folgende Datenstrukturen:

```
class Vektor {
  double[] values;
}

class Matrix {
  double[][] values;
}
```

Schreiben Sie eine Prozedur *skalarMat(Vektor x, Vektor y, Matrix A)*, die das wie folgt definierte Skalarprodukt berechnet: $\langle x, y \rangle_A = x^T A y$.

Dabei bezeichnet x^T den transponierten Vektor; die Matrixmultiplikation ist rechtsassoziativ.

```
  double skalarMat(Vektor x, Vektor y, Matrix A) {
```

0.8 Prozedurale Programmierung

Eine logische Variable hat einen Namen und kann mit *true* oder *false* belegt sein kann. Ein Literal ist eine logische Variable oder deren Negation (mit $\neg$). Eine disjuktive Verknüpfung (mit $\vee$) von Literalen heißt *Klausel*, eine konjunktive Verknüpfung von Klauseln (mit $\wedge$) heißt *Konjunktive Normalform (KNF)*.

Die Verknüpfungen sind wie folgt definiert:

a	b	a $\wedge$ b	a $\vee$ b
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Das Ziel der Aufgabe ist es, Prozeduren auf Formeln in konjunktver Normalform anzuwenden. Beispiele für KNF:

$$(a \vee b \vee \neg c) \wedge (\neg b \vee c)$$
$$(\neg a \vee \neg b \vee c) \wedge c \wedge (a \vee b \vee c)$$

(a) Modellieren Sie Datenstrukturen für:

1. Variable
2. Literal
3. Klausel mit max. 10 Literalten
4. KNF mit max. 10 Klauseln

(b) Schreiben sie eine Prozedur *simplify*, die als Parameter eine KNF nimmt und dabei Klauseln streicht, in denen ein Literal und seine Negation vorkommen, und Literale streicht, die doppelt vorkommen. Beispiel:

$$(a \vee \neg a \vee b) \wedge (a \vee \neg b \vee a) \overset{\text{simplify}}{\Rightarrow} (a \vee \neg b)$$

(c) Schreiben Sie eine Prozedur *eval*, die als Parameter eine KNF nimmt, und logisch auswertet. Das heißt, je nachdem wie die Variablen belegt sind, soll ein anderer Wahrheitswert zurückgegeben werden. Beispiel:

$$a, b := \text{True}, c := \text{False}$$
$$(a \vee b \vee \neg c) \wedge (\neg b \vee c) = True \wedge False = False$$
$$(\neg a \vee \neg b \vee c) \wedge c \wedge (a \vee b \vee c) = False \wedge False \wedge True = False$$

0.9 Objektorientierte Modellierung

(a) Modellieren Sie folgendes Szenario als UML-Klassendiagramm mit Attributen und Methoden. Vergessern Sie die Multiplizitäten dabei nicht. Treffen Sie für nicht angegebene Informationen vernünftige Annahmen.

In einem Handballverein sind Mitglieder entweder Spieler oder Trainer. Alle Mitglieder haben einen Namen und eine Mitgliedsnummer. Spieler haben zusätzlich eine Spielernummer. Der Verein hat mehrere Mannschaften, die alle einen Trainer und einen Co-Trainer haben.

(b) Ergänzen Sie ihr UML-Diagramm um weitere Klassen gemäß der folgenden Beschreibung. Dabei brauchen Sie Attribute und Methoden nicht einzeichnen.

Wir betrachten jetzt den Handballverband. In ihm gibt es mehrere Ligen, in denen jeweils die Mannschaften mehrerer Vereine spielen. Dies tun sie in Hallen. Jedem Verein werden Hallen zur Verfügung gestellt, die dann in einem Spiel die Heimhalle sind.

0.10 Objektorientierte Programmierung - Implementierung

Sie haben in der Übung bereits gebräuchliche Datenstrukturen implementiert. Dabei fehlt bisher noch die *Menge*. Eine Menge hat folgende Eigenschaften:

1. Eine Menge besteht aus Elementen eines Typs T
2. Eine Menge kann leer sein
3. Eine Menge ist duplikatfrei
4. Mit der Methode *add(T t)* wird ein neues Element der Menge hinzugefügt
5. Mit der Methode *remove(T t)* wird das Element t - falls vorhanden - aus der Menge entfernt
6. Damit man die Daten einfach auch mit anderen Algorithmen benutzen kann, gibt die Methode *toArray()* alle (noch nicht gelöschten) Elemente der Menge zurück.