Name: Vorname: Matr.–Nr.:

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen WS 2007/08
Prof. Dr. Helmut Seidl 23. Juli 2007

Klausur zu Einführung in die Informatik II

Hinweis: In dieser Klausur können Sie insgesamt 50 Punkte erreichen. Zum Bestehen benötigen Sie mindestens 20 Punkte.

Wichtig: Bei der Lösung der Programmieraufgaben müssen Sie alle verwendeten Funktionen selbst implementieren. Sie dürfen also nur die Konstrukte der Programmiersprache benutzen und **nicht** auf Bibliotheksfunktionen zurückgreifen.

Wichtig: Bei der Lösung der Programmieraufgaben dürfen **ausschließlich funktionale Konstrukte** verwendet werden. Imperative Konstrukte (z.B. Referenzen und Arrays) sind, sofern nicht explizit verlangt, **nicht erlaubt**.

Aufgabe 1 Verifikation funktionaler Programme (10 Punkte)

Gegeben sei folgende MiniOcaml-Definition:

```
let rec f =
    fun l -> match l with
        [] -> []
      | (x,y)::r -> x::y::(f r)
```

Zeigen Sie mit Hilfe der Big-Step operationellen Semantik, dass der Aufruf `f l` für jedes `l`, für das

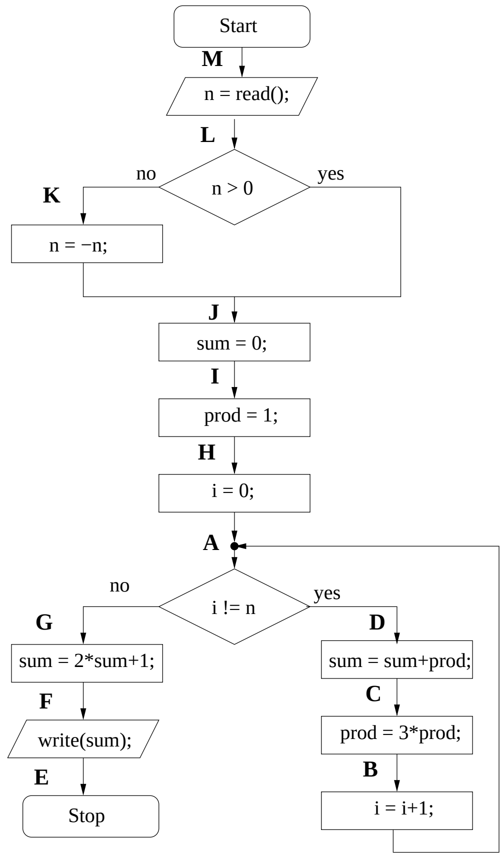
$$\mathtt{l} \Rightarrow [\mathtt{v_1}, \dots, \mathtt{v_n}]$$

für ein $n \geq 0$ gilt, terminiert. Dabei seien $\mathtt{v_1} = (\mathtt{v}'_1, \mathtt{v}''_1), \dots, \mathtt{v_n} = (\mathtt{v}'_n, \mathtt{v}''_n)$ Werte.

Hinweis: Eine Auflistung der Axiome und Regeln der Big-Step operationellen Semantik befindet sich im Anhang.

Aufgabe 2 Verifikation imperativer Programme (10 Punkte)

Gegeben sei folgendes Kontroll-Fluß-Diagramm:



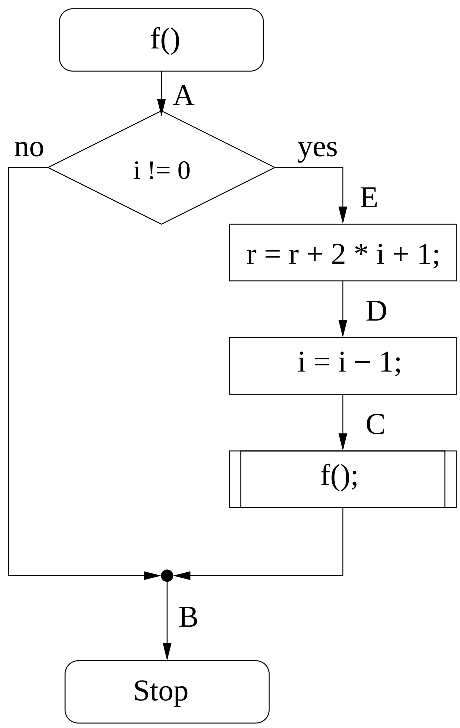
Zeigen Sie, dass am Ende des Programms die Zusicherung $\texttt{sum} = 3^{\texttt{n}}$ stets erfüllt ist.

Hinweis: Als Hilfestellung sei Ihnen folgende Rechenregel gegeben:

$$\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1}, \qquad \text{für } n \in \mathbb{N}, n \geq 0 \text{ und } q \neq 1$$

Aufgabe 3 Verfikation rekursiver Prozeduren (10 Punkte)

Gegeben sei folgendes Kontroll-Fluß-Diagramm der rekursiven Prozedur `f`:



Zeigen Sie, dass das Tripel

$$\{\mathtt{r} = l_{\mathtt{r}} \wedge \mathtt{i} = l_{\mathtt{i}}\}\ \mathtt{f}();\ \{\mathtt{r} = l_{\mathtt{r}} + l_{\mathtt{i}}^2 + 2l_{\mathtt{i}}\}$$

gültig ist, wobei $l_{\mathtt{i}}$ und $l_{\mathtt{r}}$ **logische Variablen** sind.

Aufgabe 4 Schwach besetzte Vektoren (10 Punkte)

Vektoren von ganzen Zahlen lassen sich in Ocaml als Listen von `int` repräsentieren. Als **schwach besetzt** bezeichnet man Vektoren, die einen hohen Anteil von Nullelementen aufweisen. In solchen Fällen kann es sinnvoll sein, diese Vektoren als Listen von `(int * int)`-Tupeln zu speichern, wobei die erste Zahl die Position im Vektor und die zweite Zahl den eigentlichen Wert angibt. Es ist dann nicht mehr nötig, die Nullwerte zu speichern.

Beispiel:
Darstellung eines Vektors als Liste von `int`: [1;0;5;0;0;6]
Darstellung des Vektors als Liste von `(int * int)` ohne Nullelemente: [(0,1);(2,5);(5,6)]

a) Schreiben Sie eine Funktion `sb_vektor`, die einen Vektor, der als Liste von ganzen Zahlen repräsentiert ist, in die oben erläuterte Form für schwach besetzte Vektoren bringt.

b) Schreiben Sie eine Funktion `add_sb_vektor`, die schwach besetzte Vektoren in obiger Darstellung addiert.

c) Schreiben Sie eine Funktion `mult_sb_vektor`, die das Skalarprodukt schwach besetzter Vektoren in obiger Darstellung berechnet. Das Skalarprodukt zweier Vektoren ist folgendermaßen definiert:

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n$$

Aufgabe 5 Binäre Bäume (10 Punkte)

Ein binärer Baum für einen Datentyp `'a` sei induktiv definiert als

- ein Blatt, das genau einen Wert vom Typ `'a` speichert oder
- ein Knoten mit genau zwei binären Bäumen vom Typ `'a` als Kind-Knoten.

a) Definieren Sie einen geeigneten Datentyp `'a bintree` für binäre Bäume in Ocaml.

b) Definieren Sie eine Funktion `mapBintree`, die für einen gegebenen binären Baum `t` und eine Funktion `f` einen neuen binären Baum zurückliefert, bei dem `f` auf alle Blätter angewandt wurde.

c) Definieren Sie eine Funktion `sumUp`, die für eine gegebenen binären Baum über den ganzen Zahlen (Datentyp: `int bintree`) die Summe aller Werte in den Blättern berechnet.

d) Definieren Sie eine Funktion `halveList`, die eine gegebene Liste von Werten in zwei Hälften teilt. Sie können davon ausgehen, dass die Listenlänge ein **Vielfaches von 2** ist.

e) Definieren Sie eine Funktion `list2bintree`, die für eine gegebene Liste `t` von Werten einen vollständigen binären Baum zurückliefert, der genau diese Werte speichert. Wenn `t` die leere Liste ist, soll eine `EmptyList`-Exception geworfen werden. Sie können davon ausgehen, dass die Listenlänge eine **Potenz von 2** ist (d.h. es existiert ein $k \in \mathbb{N}_0$ so, dass die Listenlänge 2^k ist).

Anhang: Big-Step Operationelle Semantik

Axiome: $v \Rightarrow v$ für jeden Wert v

Tupel:
$$\frac{e_1 \;\Rightarrow\; v_1 \qquad \ldots \qquad e_k \;\Rightarrow\; v_k}{(e_1,\ldots,e_k) \;\Rightarrow\; (v_1,\ldots,v_k)}$$

Listen:
$$\frac{e_1 \;\Rightarrow\; v_1 \qquad e_2 \;\Rightarrow\; v_2}{e_1 \texttt{::} e_2 \;\Rightarrow\; v_1 \texttt{::} v_2}$$

Globale Definitionen:
$$\frac{f = e \qquad e \;\Rightarrow\; v}{f \;\Rightarrow\; v}$$

Lokale Definitionen:
$$\frac{e_1 \;\Rightarrow\; v_1 \qquad e_0[v_1/x] \;\Rightarrow\; v_0}{\texttt{let}\; x \texttt{=} e_1 \;\texttt{in}\; e_0 \;\Rightarrow\; v_0}$$

Funktionsaufrufe:
$$\frac{e_1 \;\Rightarrow\; \texttt{fun}\; x \texttt{->} e_0 \qquad e_2 \;\Rightarrow\; v_2 \qquad e_0[v_2/x] \;\Rightarrow\; v_0}{e_1\; e_2 \;\Rightarrow\; v_0}$$

Pattern Matching:
$$\frac{e_0 \;\Rightarrow\; v' \equiv p_i[v_1/x_1,\ldots,v_k/x_k] \qquad e_i[v_1/x_1,\ldots,v_k/x_k] \;\Rightarrow\; v}{\texttt{match}\; e_0 \;\texttt{with}\; p_1 \texttt{->} e_1 \;\texttt{|}\; \ldots \;\texttt{|}\; p_m \texttt{->} e_m \;\Rightarrow\; v}$$

— sofern v' auf keines der Muster $p_1,\ldots,p_{i-1}$ passt

Eingebaute Operatoren:
$$\frac{e_1 \;\Rightarrow\; v_1 \qquad e_2 \;\Rightarrow\; v_2 \qquad v_1 \;\texttt{op}\; v_2 = v}{e_1 \;\texttt{op}\; e_2 \;\Rightarrow\; v}$$
— Unäre Operatoren werden analog behandelt.