2. Klausur zu Programmierung WS 2000/2001

Prof. Gert Smolka

2. März 2001

Name und Vorname

Matrikelnummer

Hinweise:

- Bitte alle Lösungen direkt in das Klausurheft schreiben. Nur das Klausurheft kann abgegeben werden.
- Hilfsmittel sind nicht zugelassen.
- Für die Bearbeitung der Klausur stehen 150 Minuten zur Verfügung. Insgesamt sind 150 Punkte erreichbar. Die Punkteverteilung gibt Ihnen also einen Anhaltspunkt, wieviel Zeit Sie für jede Aufgabe verwenden sollten.
- Zum Bestehen der Klausur genügt die Hälfte (75) der maximal erreichbaren Punkte.

1	2	3	4	5	6	7	8	9	10	11	∑
12	14	12	12	9	9	20	8	12	24	18	150

Note:	

Sie können die folgenden vordefinierten Prozeduren benutzen:

```
rev : 'a list -> 'a list

length : 'a list -> int

hd : 'a list -> 'a

tl : 'a list -> 'a list

null : 'a list -> bool

foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

List.exists : ('a -> bool) -> 'a list -> bool
```

Die Befehle der virtuellen Maschine V sind wie folgt deklariert:

```
type index = int
type noi   = int               (* number of instructions      *)
type noa   = int               (* number of arguments         *)
type ca    = int               (* code address                *)

datatype instruction =
    con     of int
  | add                        (* addition                    *)
  | sub                        (* substraction                *)
  | mul                        (* multiplication              *)
  | leq                        (* less or equal test          *)
  | branch  of noi             (* unconditional branch        *)
  | cbranch of noi             (* conditional branch          *)
  | getS    of index           (* push value from stack       *)
  | putS    of index           (* update value in stack       *)
  | halt                       (* halt machine                *)
  | proc    of noa * noi       (* begin of procedure code     *)
  | getF    of index           (* push value from frame       *)
  | call    of ca              (* call procedure              *)
  | return                     (* return from procedure call  *)
  | callR   of ca              (* call procedure and return   *)
```

Aufgabe 1: Effiziente funktionale Schlangen (12) Schreiben Sie eine Struktur, die Schlangen gemäß der unten gezeigten Signatur implementiert. Die Struktur soll so realisiert werden, dass eine Folge von n `insert`- und `tail`-Operationen insgesamt die Laufzeit $O(n)$ hat (ausgehend von `empty`).

```
structure FQueue :>
  sig
    type 'a queue
    val empty  : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val head   : 'a queue -> 'a        (* Empty *)
    val tail   : 'a queue -> 'a queue  (* Empty *)
  end
=
struct
```

Aufgabe 2: Rot-Schwarz-Bäume (2+8+2+2)

(a) Ergänzen Sie die Typdeklarationen

```
datatype color =

datatype ctree =
```

so, dass die Werte von `ctree` Rot-Schwarz-Bäume mit Marken aus `int` darstellen können. Im Folgenden bezeichnen wir die Werte von `ctree` als *C-Bäume*.

(b) Schreiben Sie eine Prozedur

```
searchTree : ctree -> bool
```

die testet, ob ein C-Baum ein Suchbaum ist.

(c) Definieren Sie *Rot-Balanciertheit* von C-Bäumen.

(d) Definieren Sie *Schwarz-Balanciertheit* von C-Bäumen.

Aufgabe 3: Zahldarstellung (6+6) Die Darstellung einer Zahl $x \in \mathbb{N}$ zu einer Basis $B \in \mathbb{N}$, $B \geq 2$ ist eine Liste $[x_n, \ldots, x_0]$ wie folgt:

- $x = \sum_{i=0}^{n} x_i \cdot B^i$
- $\{x_n, \ldots, x_0\} \subseteq \{0, \ldots, B-1\}$
- $x_n = 0 \Rightarrow n = 0$.

(a) Schreiben Sie eine Prozedur

```
from10 : int -> int -> int list
```

die zu B und x die Darstellung von x zur Basis B liefert.

(b) Schreiben Sie eine Prozedur

```
to10 : int -> int list -> int
```

die zu B und einer Darstellung *xs* zur Basis B die dargestellte Zahl x liefert.

Aufgabe 4: Freie Bezeichner (12) Sei die abstrakte Syntax von F wie folgt deklariert:

```
datatype con = False | True | IC of int
type     id  = string
datatype ops = Add | Sub | Mul | Leq
datatype ty  = Bool | Int | Arrow of ty * ty
datatype exp = Con of con
             | Id  of id
             | Op  of exp * ops * exp
             | If  of exp * exp * exp
             | Abs of id  * ty  * exp
             | App of exp * exp
```

Schreiben Sie eine Prozedur

```
freeIds : exp -> id list
```

die zu einem Ausdruck eine Liste der frei auftretenden Bezeichner liefert. Die Liste darf denselben Bezeichner mehrfach enthalten. Verwenden Sie eine Hilfsprozedur

```
freeIds' : id list -> exp -> id list
```

die nur die frei auftretenden Bezeichner liefert, die nicht in einer Liste von gebundenen Bezeichnern enthalten sind.

Aufgabe 5: Statische Semantik von F (9) Vervollständigen Sie die folgenden Inferenzregeln für die statische Semantik von F:

$$\frac{}{T \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow t}$$

$$\frac{}{T \vdash \texttt{fn}\ x : t\ \texttt{=>}\ e \Rightarrow t \to t'}$$

$$\frac{}{T \vdash e_1\ e_2 \Rightarrow t}$$

Aufgabe 6: Dynamische Semantik von F (9) Vervollständigen Sie die folgenden Inferenzregeln für die dynamische Semantik von F:

$$\frac{V \vdash e_1 \Rightarrow 0}{V \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v}$$

$$\frac{}{V \vdash \texttt{fn}\ x : t\ \texttt{=>}\ e \Rightarrow}$$

$$\frac{}{V \vdash e_1\ e_2 \Rightarrow v}$$

Aufgabe 7: Kontextfreie Syntax von arithmetischen Ausdrücken (4+8+8) Sei die folgende abstrakte Syntax für arithmetische Ausdrücke gegeben:

```
datatype exp =                (* expression          *)
    Id    of string             (* identifier        *)
  | Add   of exp * exp          (* addition          *)
  | Mul   of exp * exp          (* multiplication    *)
```

(a) Sie sollen eine eindeutige kontextfreie Grammatik für diese Ausdrücke angeben, die die Ausdrücke wie üblich darstellt:

(i) Addition steht auf höherer Rangstufe als Multiplikation.

(ii) Addition und Multiplikation werden linksassoziativ gruppiert.

(iii) Teilaudrücke können nach Belieben geklammert werden.

Nehmen Sie an, dass das Nonterminal *identifier* für Bezeichner bereits definiert ist (durch die lexikalische Syntax) und geben Sie die Grammatik durch die Definition der folgenden Nonterminale an:

$$exp \;=$$

$$mulexp \;=$$

$$atexp \;=$$

(b) Schreiben Sie eine Prozedur

```
exp : exp -> string
```

die Ausdrücke mit möglichst wenig Klammern darstellt. Verwenden Sie dabei zwei Hilfsprozeduren *mulexp* und *atexp* und verschränkte Rekursion.

(c) Modifizieren Sie die Grammatik so, dass sie sich für die Ableitung der Parsingprozeduren eignet:

$$exp =$$

$$exp' =$$

$$mulexp =$$

$$mulexp' =$$

$$atexp =$$

Aufgabe 8: Generatoren (8) Ein Generator für eine Folge $x_1, x_2, \ldots$ von Werten eines Typs t ist eine Prozedur `unit` $\rightarrow t$, die beim n-ten Aufruf x_n liefert.

Schreiben Sie eine Prozedur

```
newGenerator : (int -> 'a) -> unit -> 'a
```

die zu einer Prozedur f einen Generator für die Folge

$$f(0),\ f(1),\ f(2),\ \ldots$$

liefert.

Aufgabe 9: Imperative Listen (12) Nehmen Sie an, dass imperative Listen gemäß der Typdeklaration

```
datatype 'a ilist = Nil | Cons of 'a * 'a ilist ref
```

dargestellt werden. Schreiben Sie eine Prozedur

```
refs : 'a ilist -> 'a ilist ref list
```

die die Liste aller Referenzen einer imperativen Liste liefert (ohne Doppelauftreten). Die Prozedur soll **auch für zyklische Listen** funktionieren.

Aufgabe 10: Maschinennahe Programmierung (10+6+8) Seien die folgenden Prozeduren gegeben:

```
fun sqrt'(n,x) = if x<n*n then n-1 else sqrt'(n+1,x)

fun sqrt x = sqrt'(0,x)
```

Die Prozedur `sqrt` berechnet zu $x \in \mathbb{N}$ das größte $n \in \mathbb{N}$ mit $n^2 \leq x$.

(a) Übersetzen Sie die Prozeduren in eine Befehlssequenz *sqrt*, so dass das V-Programm

```
sqrt @ [con x, call 0, halt]
```

dasselbe Ergebnis liefert wie der Prozeduraufruf `sqrt` x (für $x \in \mathbb{N}$). Alle Endaufrufe sollen dabei mit `callR` übersetzt werden.

(b) Schreiben Sie die Prozedur `sqrt` mit einer Schleife und ohne Rekursion.

(c) Schreiben Sie eine Befehlssequenz *sqrt*, so dass das V-Programm

```
con x :: sqrt
```

dasselbe Ergebnis liefert wie der Prozeduraufruf `sqrt` x (für $x \in \mathbb{N}$). Verwenden Sie dabei keine Prozedurbefehle.

Hinweis: Die Befehle von V finden Sie auf Seite 2.

Aufgabe 11: Übersetzung (18) Sei die folgende abstrakte Syntax für arithmetische Ausdrücke mit Konditionalen gegeben:

```
type id  = string            (* identifier            *)
datatype exp =               (* expression            *)
  Con    of int                  (* constant           *)
| Var    of id                   (* variable           *)
| Add    of exp * exp            (* addition           *)
| Leq    of exp * exp            (* less or equal test *)
| If     of exp * exp * exp      (* conditional        *)
```

Schreiben Sie eine Prozedur

```
compile : (id -> int) -> exp -> instruction list
```

die Ausdrücke in Befehlssequenzen von V übersetzt. Das erste Argument von `compile` soll eine Prozedur sein, die zu jedem vorkommenden Bezeichner die Adresse liefert, an der die entsprechende imperative Variable im Stapel alloziert ist (Zugriff erfolgt mit `getS`).