2. Klausur Programmierung WS 2002/03

Prof. Dr. Gert Smolka, Dipl. Inf. Thorsten Brunklaus

25. Februar 2003

Vor- und Nachname Sitz-Nr.

Matrikelnummer Code

- Bitte öffnen Sie das Klausurheft erst dann, wenn Sie dazu aufgefordert werden (gleiche Bearbeitungszeit für alle).
- Sie können die Klausur nur auf dem für Sie vorgesehen Platz mitschreiben. Sie müssen das mit Ihrem Namen und Ihrer Matrikelnummer versehene Klausurheft verwenden.
- Hilfsmittel sind nicht zugelassen. Am Arbeitsplatz dürfen nur Schreibgeräte, Getränke, Speisen sowie Ausweise mitgeführt werden. Taschen, Jacken und Mäntel müssen an den Wänden des Klausursaals zurückgelassen werden.
- Verlassen des Saals ohne vorherige Abgabe des Klausurhefts gilt als Täuschungsversuch.
- Wenn Sie während der Bearbeitung auf die Toilette müssen, geben Sie bitte Ihr Klausurheft bei der Aufsicht ab. Es kann immer nur eine Person auf die Toilette.
- Alle Lösungen müssen auf den bedruckten rechten Seiten des Klausurhefts notiert werden. Die leeren linken Seiten dienen als Platz für Skizzen und werden **nicht korrigiert**. Notizpapier ist nicht zugelassen. Sie können mit Bleistift schreiben.
- Für die Bearbeitung der Klausur stehen 150 Minuten zur Verfügung. Insgesamt sind 150 Punkte erreichbar. Die für jede Aufgabe angebene Punktzahl gibt Ihnen also einen Anhaltspunkt, wieviel Zeit Sie auf die Bearbeitung der Aufgabe verwenden sollten. Zum Bestehen der Klausur genügen 75 Punkte.
- Bitte legen Sie Ihren Personalausweis oder Reisepass sowie Ihren Studierendenausweis zur Identifikation neben sich.
- Viel Erfolg!

1	2	3	4	5	6	7	$\sum$	Note
18	20	30	22	16	15	29	150	

Aufgabe 1: Statische Semantik (18 = 3 + 15) Wir betrachten Ausdrücke, die mit Bezeichnern, ganzzahligen Summen und Prozeduranwendungen gebildet sind, sowie Typen, die mit `int` und `->` gebildet sind. Ausdrücke, Typen und Typumgebungen implementieren wir mit den folgenden Deklarationen:

```
type id = string
datatype exp = Id of id | Sum of exp * exp | App of exp * exp
datatype ty = Int | Arrow of ty * ty
type tyenv = id -> ty
exception Error
```

Eine Typumgebung wirft die Ausnahme `Error`, wenn sie für einen Bezeichner nicht definiert ist.

(a) Geben Sie eine Deklaration an, die den Bezeichner `env` an eine Typumgebung bindet, die `x` den Typ `int` und `f` den Typ `int->int` gibt.

(b) Deklarieren Sie eine Prozedur

```
elab : tyenv -> exp -> ty  (* Error *)
```

die prüft, ob ein Ausdruck in einer Typumgebung zulässig ist. Wenn der Ausdruck zulässig ist, wird sein Typ geliefert, sonst die Ausnahme `Error`.

```
fun elab env e = case e of
```

Aufgabe 2: Dynamische Semantik (20 = 5 + 15) Wir betrachten Ausdrücke, die mit Bezeichnern, ganzzahligen Summen und Prozeduranwendungen gebildet sind. Als Werte betrachten wir ganze Zahlen und Prozeduren. Ausdrücke, Werte und Wertumgebungen implementieren wir mit den folgenden Deklarationen:

```
type id = string
datatype exp = Id of id | Sum of exp * exp | App of exp * exp
datatype value = IV of int | Proc of id * exp * valenv
withtype valenv = id -> value
exception Error
```

Eine Wertumgebung wirft die Ausnahme `Error`, wenn sie für einen Bezeichner nicht definiert ist.

(a) Geben Sie eine Deklaration an, die den Bezeichner `env` an eine Wertumgebung bindet, die `x` an den Wert 5 und `f` an die Prozedur `fn x => x+x` bindet.

(b) Deklarieren Sie eine Prozedur

```
eval : valenv -> exp -> value  (* Error *)
```

die einen Ausdruck in einer Wertumgebung auswertet.

```
fun eval env e = case e of
```

Aufgabe 3: Konkrete Syntax (30 = 10 + 3 + 17) Wir betrachten Ausdrücke, die mit Bezeichnern, ganzzahligen Summen und Prozeduranwendungen gebildet sind. Die Baum- und Wortdarstellung der Ausdrücke realisieren wir mit den Typdeklarationen

```
datatype exp = Id of string | Sum of exp * exp | App of exp * exp
datatype token = ID of string | ADD | LPAR | RPAR
```

gemäß der eindeutigen kontextfreien Grammatik

$$
\begin{aligned}
\mathit{exp} &= [\ \mathit{exp}\ \text{"+"}\ ]\ \mathit{apexp} \\
\mathit{apexp} &= [\ \mathit{apexp}\ ]\ \mathit{atexp} \\
\mathit{atexp} &= \mathit{identifier}\ \mid\ \texttt{"("}\ \mathit{exp}\ \texttt{")"}
\end{aligned}
$$

(a) Deklarieren Sie eine Prozedur

`exp : exp -> token list`

die Ausdrücke mit möglichst wenigen Klammern darstellt. Verwenden Sie zwei Hilfsprozeduren `apexp` und `atexp`.

(b) Geben Sie eine eindeutige Grammatik ohne Linksrekursionen an, die dieselben Sätze beschreibt wie die obige Grammatik.

(c) Deklarieren Sie eine Prozedur

```
test : token list -> bool
```

die testet, ob eine Liste von Wörtern einen Ausdruck darstellt. Verwenden Sie für jedes Nonterminal der transformierten Grammatik eine Hilfsprozedur des Typs

```
token list -> token list  (* Error *)
```

Aufgabe 4: Imperative Listen $(22 = 7 + 7 + 8)$ Wir betrachten imperative Listen, die gemäß den folgenden Typdeklarationen realisiert sind:

```
datatype 'a state = Nil | N of 'a * 'a ilist
withtype 'a ilist = 'a state ref
```

(a) Deklarieren Sie eine Prozedur

gen : int -> int ilist

die zu $n \in \mathbb{N}$ eine imperative Liste mit den Marken $1, \ldots, n$ liefert (in aufsteigender Ordnung). Verwenden Sie eine iterative Hilfsprozedur `gen'`.

(b) Deklarieren Sie eine Prozedur

reverse : 'a ilist -> unit

die eine imperative Liste reversiert ohne neue Referenzen zu allozieren. Verwenden Sie eine iterative Hilfsprozedur `reverse'`.

(c) Deklarieren Sie eine Prozedur

cyclic : 'a ilist -> bool

die in linearer Zeit testet, ob eine imperative Liste zirkulär ist. Verwenden Sie den Hase-Igel-Algorithmus. Verwenden Sie zwei Hilfsprozeduren `cyclic'` und `tail`. Die Prozedur `tail` soll die Ausnahme `Empty` werfen, wenn sie auf eine leere Liste angewendet wird.

Aufgabe 5: Graphen mit Merkfeld (16 = 8 + 8) Wir betrachten Graphen, deren Knoten gemäß den folgenden Deklarationen mit einem Booleschen Merkfeld und einer ganzzahligen Marke realisiert sind:

```
datatype state = G of bool * int * node list
withtype node  = state ref
```

(a) Deklarieren Sie eine Prozedur

reset : node list -> unit

die das Merkfeld aller von einer Liste von Knoten aus erreichbaren Knoten auf `false` setzt. Knoten, deren Merkfeld bereits `false` ist, sollen von `reset` ignoriert werden.

(b) Deklarieren Sie eine Prozedur

sum : node list -> int

die die Summe der Marken aller von einer Liste von Knoten aus erreichbaren Knoten liefert. Die Prozedur soll davon ausgehen, dass die Merkfelder aller erreichbaren Knoten auf `false` stehen, und sie soll die Merkfelder auch in diesem Zustand hinterlassen. Verwenden Sie eine iterative Hilfsprozedur `sum'`.

Aufgabe 6: Übersetzer für V (15) Sie sollen eine Prozedur

```
compile : env -> exp -> code
```

schreiben, die Ausdrücke mit Konstanten, Bezeichnern, Summen und Konditionalen in Befehlssequenzen für die virtuelle Maschine V übersetzt. Die Ausdrücke seien gemäß

```
type id = string
datatype exp = Con of int | Id of id
  | Sum of exp * exp | If of exp * exp * exp
```

realisiert. Die Bedingung des Konditionals liefert eine ganze Zahl, wobei 0 die Alternative und alle anderen Zahlen die Konsequenz auswählen sollen. Die Werte der in den Ausdrücken vorkommenden Bezeichner sollen in Stapelzellen stehen, deren Adressen durch eine Umgebung

```
type env = id -> int
```

kommuniziert werden. Die Werte der für die Bezeichner verwendeten Stapelzellen können mit dem Befehl `getS` auf den Stapel gelegt werden.

```
fun compile env e = case e of
```

Aufgabe 7: VH Programmierung $(29 = 3 * 3 + 2 * 10)$ Sie sollen arithmetische Ausdrücke, die aus Konstante, Summen und Produkten gebildet sind, in der Halde der virtuellen Maschine VH darstellen. In Standard ML seien die zu betrachtenden Ausdrücke wie folgt dargestellt:

```
datatype exp = Con of int | Sum of exp * exp | Pro of exp * exp
```

(a) Geben Sie eine Befehlssequenz für eine Prozedur an, deren Anwendung der Anwendung des Konstruktors `Con` entspricht.

(b) Geben Sie eine Befehlssequenz für eine Prozedur an, deren Anwendung der Anwendung des Konstruktors `Sum` entspricht.

(c) Geben Sie eine Befehlssequenz für eine Prozedur an, deren Anwendung der Anwendung des Konstruktors `Pro` entspricht.

(d) Geben Sie eine Befehlssequenz an, die die Prozedur

```
fun gen n = if n<=0 then Con n
            else let val e = gen (n-1) in Pro(e,e) end
```

realisiert. Nehmen Sie dabei an, dass die Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt wird.

(e) Geben Sie eine Befehlssequenz an, die die Prozedur

```
fun size (Con n) = 1
  | size (Sum(e,e')) = size e + size e' + 1
  | size (Pro(e,e')) = size e + size e' + 1
```

realisiert. Nehmen Sie dabei an, dass die Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt wird.

Befehle der virtuellen Maschine VH

```
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           *)
  | eq                        (* equality 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        *)
  | proc    of noa * noi      (* begin of procedure code      *)
  | arg     of index          (* push procedure argument      *)
  | getF    of index          (* push value from frame        *)
  | return                    (* return from procedure call   *)
  | call    of ca             (* call procedure               *)
  | callR   of ca             (* call procedure and return    *)
  | new     of noa            (* new heap block               *)
  | getH    of index          (* push value from heap block   *)
  | putH    of index          (* put value into heap block    *)
  | len                       (* push length of heap block    *)
  | halt                      (* halt machine                 *)

type code = instruction list
```