3. Klausur zu Programmierung WS 2000/2001

Prof. Gert Smolka

30. 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 180 Minuten zur Verfügung. Insgesamt sind 180 Punkte erreichbar. Die Punkteverteilung gibt Ihnen also einen Anhaltspunkt, wieviel Zeit Sie für jede Aufgabe verwenden sollten.
- Zum Bestehen der Klausur genügt die Hälfte (90) der maximal erreichbaren Punkte.

1	2	3	4	5	6	7	8	$\sum$
8	32	20	26	36	20	28	10	180

Note:	

Sie können die folgenden vordefinierten Prozeduren und Typen benutzen:

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

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

datatype order = LESS | EQUAL | GREATER
```

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: Summe von Termen (8) Seien Terme gemäß der Typdeklaration

```
datatype 'a term = T of 'a * 'a term list
```

dargestellt. Schreiben Sie eine Prozedur

```
sum : int term -> int
```

die zu einem Term über `int` die Summe seiner Marken liefert.

Aufgabe 2: Sortieren und Doppelauftreten (8+8+8+8) Sie sollen eine Prozedur schreiben, die Listen durch Mischen sortiert (aufsteigend).

(a) Schreiben Sie mithilfe von `foldl` eine Prozedur

```
split : 'a list -> 'a list * 'a list
```

die eine Liste in zwei etwa gleich große Teillisten zerlegt.

(b) Schreiben Sie eine Prozedur

```
merge : int list * int list -> int list
```

die zwei sortierte Listen in eine sortierte Liste verschmilzt.

(c) Schreiben Sie eine Prozedur

```
sort : int list -> int list
```

die Listen sortiert.

(d) Schreiben Sie eine Prozedur

```
elim : int list -> int list
```

die zu einer Liste *xs* eine Liste *ys* wie folgt liefert:

(i) *ys* enthält dieselben Zahlen wie *xs*.

(ii) In *ys* kommt kein Element mehrfach vor.

Die Prozedur `elim` soll für Listen der Länge n die **Laufzeit** $\Theta(n \log n)$ haben.

Aufgabe 3: Funktor für endliche Mengen mit Suchbäumen (3+1+8+8) Schreiben Sie einen Funktor

```
functor Set
  (type elem
   val  compare : elem * elem -> order)
  :>
  sig
    type set
    val empty  : set
    val insert : elem * set -> set
    val member : elem * set -> bool
  end
  =
  struct Deklarationen end
```

der endliche Mengen über einem geordneten Elementtyp implementiert. Realisieren Sie die Mengen mithilfe von Suchbäumen. Es genügt, wenn Sie die Deklarationen für den Rumpf der obigen Funktordeklaration angeben.

Aufgabe 4: Statische Semantik (10+6+10) Sei die abstrakte Syntax einer Teilsprache von F wie folgt deklariert:

```
datatype ty = Int
            | Arrow of ty * ty

type id = string

datatype exp = Id  of string          (* Identifier  *)
             | Abs of id * ty * exp   (* Abstraction *)
             | App of exp * exp       (* Application *)
```

(a) Schreiben Sie eine Prozedur

```
closed : exp -> bool
```

die testet, ob ein Ausdruck geschlossen ist.

(b) Seien Typumgebungen durch Prozeduren

```
type env = id -> ty  (* Error *)
```

realisiert, die die Ausnahme

```
exception Error
```

werfen, wenn sie auf Bezeichner angewendet werden, denen sie keine Typen zuordnen. Schreiben Sie eine Prozedur

```
adjoin : env -> id -> ty -> env
```

die zu einer Typumgebung T, einem Bezeichner x und einem Typ t die Typumgebung $T + \{x \mapsto t\}$ liefert.

(c) Schreiben Sie eine Prozedur

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

die zu einer Typumgebung T und einem Ausdruck e den Typ t mit $T \vdash e \Rightarrow t$ liefert. Falls kein solcher Typ existiert, soll die Ausnahme `Error` geworfen werden.

Aufgabe 5: Kontextfreie Syntax von applikativen Ausdrücken (5+10+6+15) Sei die folgende abstrakte Syntax für applikative Ausdrücke gegeben:

```
datatype exp = Id  of string          (* Identifier  *)
             | App of exp * exp       (* Application *)
```

(a) Geben Sie eine eindeutige kontextfreie Grammatik für diese Ausdrücke an, die Applikationen linksassoziativ gruppiert und bei der Teilausdrücke nach Belieben geklammert werden können. Nehmen Sie dabei an, dass das Nonterminal *identifier* für Bezeichner bereits definiert ist.

(b) Schreiben Sie eine Prozedur

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

die Ausdrücke gemäß der obigen Grammatik mit möglichst wenigen Klammern darstellt.

(c) Geben Sie eine modifizierte Form der obigen Grammatik an, die sich für die Ableitung der Parsingprozeduren eignet.

(d) Schreiben Sie eine Prozedur

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

die testet, ob eine Liste von Wörtern einen Ausdruck gemäß der obigen Grammatik darstellt. Dabei sollen Wörter wie folgt dargestellt sein:

```
datatype token = ID of string       (* identifier *)
               | LPAR               (* "("        *)
               | RPAR               (* ")"        *)
```

Aufgabe 6: Übersetzung und Rückübersetzung (8+12) Seien die folgenden Ausdrücke gegeben:

```
datatype exp =
    Con    of int             (* constant           *)
  | Add    of exp * exp       (* addition           *)
  | Mul    of exp * exp       (* multiplication     *)
```

Weiter sei der Typ

```
type code = instruction list
```

gegeben, wobei der Typ `instruction` die Befehle der virtuellen Maschine V darstellt (siehe Seite 2).

(a) Schreiben Sie eine Prozedur

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

die einen Ausdruck in ein V-Programm übersetzt, dessen Ausführung den Wert des Ausdrucks liefert.

(b) Schreiben Sie eine Prozedur

```
decompile : code -> exp
```

sodass für alle Ausdrücke e gilt:

`decompile(compile` e`) =` e

Aufgabe 7: V-Programmierung (10+8+10) Die Prozedur

```
fun gcd(x,y) = if x<>y then
                  if x<=y then gcd(x, y-x) else gcd(x-y, y)
               else x
```

berechnet den größten gemeinsamen Teiler zweier positiven ganzen Zahlen.

(a) Übersetzen Sie die Prozedur `gcd` in eine Befehlssequenz *gcd*, sodass das V-Programm

```
gcd @ [con y, con x, call 0, halt]
```

den größten gemeinsamen Teiler zweier positiven ganzen Zahlen *x* und *y* berechnet. Übersetzen Sie dabei alle Endaufrufe mit `callR`.

(b) Schreiben Sie mithilfe einer Schleife eine nichtrekursive Variante `gcdi` der Prozedur `gcd`.

(c) Schreiben Sie eine Befehlssequenz *gcdi* ohne Prozedurbefehle, sodass das V-Programm

```
[con x, con y] @ gcdi @ [halt]
```

den größten gemeinsamen Teiler zweier positiven ganzen Zahlen *x* und *y* berechnet.

Aufgabe 8: Imperative Listen (10) Seien imperative Listen gemäß der Typdeklaration

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

dargestellt. Schreiben Sie eine Prozedur

```
circle : int -> int ilist
```

die zu $n \in \mathbb{N}^+$ eine zyklische Liste mit n Knoten liefert, die mit den Zahlen $1, \ldots, n$ markiert sind. Der Zeiger des mit n markierte Knotes soll auf den mit 1 markierten Knoten zeigen. Die Prozedur soll den mit 1 markierten Knoten liefern.