Programmierung WS 2002 / 03:
Musterlösung zum 14. Übungsblatt

Prof. Dr. Gert Smolka, Dipl.-Inform. Thorsten Brunklaus

Aufgabe 14.1: Imperative Listen $(25 = 5 + 10 + 10)$

(a)

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

fun ilist () = ref Nil

fun cons x xr = ref(N(x, xr))

fun circle' n xs =
  if n<1 then xs else circle' (n-1) (cons n xs)

fun circle n =
  let val xs = ilist() in xs := !(circle' n xs) ; xs end
```

(b)

```
fun empty (ref(Nil)) = true
  | empty _          = false

fun tail (ref(N(_, xr))) = xr
  | tail _               = raise Empty

fun sizeR' rs r =
  if empty r then 1 + length rs
  else if List.exists (fn r' => r'=r) rs then length rs
       else sizeR' (r::rs) (tail r)

fun sizeR xs = sizeR' nil xs
```

(c)

```
fun empty (ref(Nil)) = true
  | empty _          = false

fun tail (ref(N(_, xr))) = xr
  | tail _               = raise Empty

fun sizeN' ns r =
  if empty r then length ns
  else if List.exists (fn n' => n'=(!r)) ns then length ns
       else sizeN' ((!r)::ns) (tail r)

fun sizeN xs = sizeN' nil xs
```

Aufgabe 14.2: Größte gemeinsame Teiler mit V $(20 = 10 + 10)$

(a)

```
fun gcdi (x0, y0) =
    let
        val x = ref x0
        val y = ref y0
    in
        while !x <> !y do
          if !x <= !y then y:= !y - !x else x:= !x - !y ;
        !x
    end
```

(b)

```
[getS 1, getS 0, sub, cbranch 15,
 getS 1, getS 0, leq, cbranch 6,
 getS 0, getS 1, sub, putS 1, branch ~12,
 getS 1, getS 0, sub, putS 0, branch ~17]
```

Aufgabe 14.3: Übersetzung und Rückübersetzung $(15 = 5 + 10)$

(a)

```
fun compile’(Con i)      = [con i]
  | compile’(Add(e1,e2)) = compile’ e2 @ compile’ e1 @ [add]
  | compile’(Sub(e1,e2)) = compile’ e2 @ compile’ e1 @ [sub]
  | compile’(Mul(e1,e2)) = compile’ e2 @ compile’ e1 @ [mul]

fun compile e = compile’ e @ [halt]
```

(b)

```
fun decompile’ (con n::is, es)      = decompile’(is, Con n::es)
  | decompile’ (add::is, e::e’::es) = decompile’(is, Add(e,e’)::es)
  | decompile’ (sub::is, e::e’::es) = decompile’(is, Sub(e,e’)::es)
  | decompile’ (mul::is, e::e’::es) = decompile’(is, Mul(e,e’)::es)
  | decompile’ ([halt], [e])        = e
  | decompile’     _                = raise Error "cannot decompile"

fun decompile code = decompile’(code,nil)
```

Aufgabe 14.4: Endaufrufe (5)

fun f(x,y) = if x<y then y else $\overline{\text{f}}$(x+1,y)

fun g x = if x<0 then x else $\overline{\text{f}}$(x, 2*x)

fun h (1::xs) = $\overline{\text{h}}$ xs | h xs = $\overline{\text{p}}$ xs
and p (2::xs) = $\overline{\text{h}}$ xs | p (x::xs) = g x + 5

Aufgabe 14.5: Fibonacci-Prozedur in V $(20 = 10 + 10)$

```
[proc(1,17),
 con 1, getF ~1, leq, cbranch 3,
 getF ~1, return,
 con 1, getF ~1, sub, call 0,
 con 2, getF ~1, sub, call 0,
 add,
 return]
```

```
[proc(3,15),
 con 0, getF ~1, leq, cbranch 3,
 getF ~2, return,
 getF ~3, getF ~2, add, getF ~3, con 1, getF ~1, sub, callR 0,
 proc(1,5),
 con 1, con 0, getF ~1, callR 0]
```

Aufgabe 14.6: Verschränkte Rekursion in V (15)

```
[proc(1,9),
 getF ~1, cbranch 5,
 con 1, getF ~1, sub, callR 9,
 con 1,
 return,
 proc(1,9),
 getF ~1, cbranch 5,
 con 1, getF ~1, sub, callR 0,
 con 0,
 return]
```

Aufgabe 14.7: Syntax und Semantik von W

(a)

$c \in \mathit{Con} = \mathbb{Z} \mid \texttt{unit}$	**Konstanten**
$x \in \mathit{Id} = \mathbb{N}$	**Bezeichner**
$o \in \mathit{Opr} = + \mid - \mid * \mid \leq$	**Operatoren**
$t \in \mathit{Ty} = \texttt{unit} \mid \texttt{int}$	**Typen**
$e \in \mathit{Exp} =$	**Ausdrücke**
c	**Konstante**
$\mid x$	**Bezeichner**
$\mid e_1\ o\ e_2$	**Operatoranwendung**
$s \in \mathit{Stmt} =$	**Statements**
$x := e$	**Zuweisung**
$\mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$	**Konditional**
$\mid \texttt{while}\ e\ \texttt{do}\ s_2\ \texttt{end}$	**Schleife**
$\mid s_1\ s_2$	**Sequenz**
$d \in \mathit{Decl} =$	**Deklarationen**
$\texttt{var}\ x := e$	**Deklaration**
$\mid d_1\ d_2$	**Declarationssequenz**
$p \in \mathit{Prog} = d\ s\ \texttt{return}\ e$	**Programm**

(b)

$$
\begin{array}{rcl}
\mathit{exp} & = & \mathit{asexp}\ \ [\ \texttt{"<="}\ \ \mathit{asexp}\ ] \\
\mathit{asexp} & = & \mathit{mulexp}\ \ \mathit{asexp}' \\
\mathit{asexp}' & = & [\ (\texttt{"+"}\ \mid\ \texttt{"-"})\ \ \mathit{mulexp}\ \ \mathit{asexp}'\ ] \\
\mathit{mulexp} & = & \mathit{atexp}\ \ \mathit{mulexp}' \\
\mathit{mulexp}' & = & [\ \texttt{"*"}\ \ \mathit{atexp}\ \ \mathit{mulexp}'\ ] \\
\mathit{atexp} & = & \mathit{identifier}\ \mid\ \mathit{integer}\ \mid\ \texttt{"("}\ \ \mathit{exp}\ \ \texttt{")"} \\
\mathit{stmt} & = & \mathit{identifier}\ \ \texttt{":="}\ \ \mathit{exp} \\
 & \mid & \texttt{"if"}\ \mathit{exp}\ \texttt{"then"}\ \mathit{stmt}\ \texttt{"then"}\ \mathit{stmt} \\
 & \mid & \texttt{"while"}\ \mathit{exp}\ \texttt{"do"}\ \mathit{stmts}\ \texttt{"end"} \\
\mathit{stmts} & = & \mathit{stmt}\ \ \mathit{stmts}' \\
\mathit{stmts}' & = & [\ \texttt{";"}\ \ \mathit{stmt}\ \ \mathit{stmts}'\ ] \\
\mathit{decl} & = & \texttt{"var"}\ \ \mathit{identifier}\ \ \texttt{":="}\ \ \mathit{exp} \\
\mathit{decls} & = & [\ \mathit{decl}\ \ \mathit{decls}\ ] \\
\mathit{prog} & = & \mathit{decls}\ \ \mathit{stmt}\ \ \texttt{"return"}\ \ \mathit{exp}
\end{array}
$$

(c)

$$\frac{T \vdash e_1 \Rightarrow \mathit{int} \qquad T \vdash e_2 \Rightarrow \mathit{int}}{T \vdash e_1\ o\ e_2 \Rightarrow \mathit{int}} \qquad \frac{}{T \vdash x \Rightarrow T x}$$

$$\frac{T \vdash e \Rightarrow \mathit{int} \qquad T x = \mathit{int}}{T \vdash x\ \texttt{:=}\ e \Rightarrow \mathit{unit}}$$

$$\frac{T \vdash e \Rightarrow \mathit{int} \qquad T \vdash s_1 \Rightarrow \mathit{unit} \qquad T \vdash s_2 \Rightarrow \mathit{unit}}{T \vdash \texttt{if} e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \mathit{unit}}$$

$$\frac{T \vdash e \Rightarrow \mathit{int} \qquad T \vdash s \Rightarrow \mathit{unit}}{T \vdash \texttt{while}\ e\ \texttt{do}\ s\ \texttt{end} \Rightarrow \mathit{unit}}$$

$$\frac{T \vdash s_1 \Rightarrow \mathit{unit} \qquad T \vdash s_2 \Rightarrow \mathit{unit}}{T \vdash s_1\ s_2 \Rightarrow \mathit{unit}}$$

$$\frac{T \vdash e \Rightarrow \mathit{int}}{T \vdash \texttt{var}\ x\ \texttt{:=}\ e \Rightarrow \mathit{unit}, T[x := \mathit{int}]}$$

$$\frac{T \vdash d_1 \Rightarrow \mathit{unit}, T' \qquad T' \vdash d_2 \Rightarrow \mathit{unit}, T''}{T \vdash d_1\ d_2 \Rightarrow \mathit{unit}, T''}$$

$$\frac{T \vdash d \Rightarrow \mathit{unit}, T' \qquad T' \vdash s \Rightarrow \mathit{unit} \qquad T' \vdash e \Rightarrow \mathit{int}}{T \vdash d\ s\ \texttt{return}\ e \Rightarrow \mathit{int}}$$

$$\frac{S \vdash e_1 \Rightarrow v_1 \qquad S \vdash e_2 \Rightarrow v_2}{S \vdash e_1 \; o \; e_2 \Rightarrow v \qquad v = v_1 \; o \; v_2} \qquad \frac{}{S \vdash x \Rightarrow Sx}$$

$$\frac{S \vdash e \Rightarrow v}{S \vdash x \; \texttt{:=} \; e \Rightarrow \mathit{unit}, S[x := v]}$$

$$\frac{S \vdash e \Rightarrow 0 \qquad S \vdash s_2 \Rightarrow \mathit{unit}, S'}{S \vdash \texttt{if} e \; \texttt{then} \; s_1 \; \texttt{else} \; s_2 \Rightarrow \mathit{unit}, S'} \qquad \frac{S \vdash e \Rightarrow v \qquad S \vdash s_1 \Rightarrow \mathit{unit}, S' \qquad v \neq 0}{S \vdash \texttt{if} e \; \texttt{then} \; s_1 \; \texttt{else} \; s_2 \Rightarrow \mathit{unit}, S'}$$

$$\frac{S \vdash e \Rightarrow 0}{S \vdash \texttt{while} \; e \; \texttt{do} \; s \; \texttt{end} \Rightarrow \mathit{unit}, S}$$

$$\frac{S \vdash e \Rightarrow v \qquad S \vdash s \Rightarrow \mathit{unit}, S' \qquad S' \vdash \texttt{while} \; e \; \texttt{do} \; s \; \texttt{end} \Rightarrow \mathit{unit}, S'' \qquad v \neq 0}{S \vdash \texttt{while} \; e \; \texttt{do} \; s \; \texttt{end} \Rightarrow \mathit{unit}, S''}$$

$$\frac{S \vdash s_1 \Rightarrow \mathit{unit}, S' \qquad S' \vdash s_2 \Rightarrow \mathit{unit}, S''}{S \vdash s_1 \; s_2 \Rightarrow \mathit{unit}, S''}$$

$$\frac{S \vdash e \Rightarrow v}{S \vdash \texttt{var} \; x \; \texttt{:=} \; e \Rightarrow \mathit{unit}, S[x := v]}$$

$$\frac{S \vdash d_1 \Rightarrow \mathit{unit}, S' \qquad S' \vdash d_2 \Rightarrow \mathit{unit}, S''}{S \vdash d_1 \; d_2 \Rightarrow \mathit{unit}, S''}$$

$$\frac{S \vdash d \Rightarrow \mathit{unit}, S' \qquad S' \vdash s \Rightarrow \mathit{unit}, S'' \qquad S'' \vdash e \Rightarrow v}{S \vdash d \; s \; \texttt{return} \; e \Rightarrow v}$$

(d)

```
datatype token = LEQ | ASSIGN | ADD | SUB | MUL
               | COLON | SEMICOLON | LPAR | RPAR
               | ICON of int
               | VAR
               | IF | THEN | ELSE
               | WHILE | DO | END
               | RETURN
               | ID of string

exception Error

val ord0 = ord #"0"

fun lex ts nil                  = rev ts
  | lex ts (#" " ::cr)          = lex ts cr
  | lex ts (#"\n"::cr)          = lex ts cr
  | lex ts (#"\t"::cr)          = lex ts cr
  | lex ts (#"<" :: #"=" ::cr) = lex (LEQ::ts) cr
  | lex ts (#":" :: #"=" ::cr) = lex (ASSIGN::ts) cr
  | lex ts (#"+" ::cr)          = lex (ADD::ts) cr
  | lex ts (#"-" ::cr)          = lex (SUB::ts) cr
  | lex ts (#"*" ::cr)          = lex (MUL::ts) cr
  | lex ts (#":" ::cr)          = lex (COLON::ts) cr
  | lex ts (#";" ::cr)          = lex (SEMICOLON::ts) cr
  | lex ts (#"(" ::cr)          = lex (LPAR::ts) cr
  | lex ts (#")" ::cr)          = lex (RPAR::ts) cr
  | lex ts (#"~" ::c::cr) =
      if Char.isDigit c then lexN ts ~1 0 (c::cr) else raise Error
  | lex ts (c::cr) =
      if Char.isDigit c then lexN ts 1 0 (c::cr)
      else if Char.isAlpha c then lexA ts [c] cr
      else raise Error
and lexN ts s n cs =
    if not(null cs) andalso Char.isDigit(hd cs)
    then lexN ts s (10*n + (ord(hd cs) - ord0)) (tl cs)
    else lex (ICON(s*n)::ts) cs
and lexA ts xs cs =
  if not(null cs) andalso Char.isAlphaNum(hd cs)
  then lexA ts (hd cs::xs) (tl cs)
  else lex (lexA’(implode(rev xs))::ts) cs
and lexA’ "if"     = IF
  | lexA’ "then"   = THEN
  | lexA’ "else"   = ELSE
  | lexA’ "while"  = WHILE
  | lexA’ "do"     = DO
  | lexA’ "end"    = END
  | lexA’ "var"    = VAR
  | lexA’ "return" = RETURN
  | lexA’ x        = ID x
```

```
type id = string

datatype exp = Con of int
             | Var of id
             | Add of exp * exp
             | Sub of exp * exp
             | Mul of exp * exp
             | Leq of exp * exp

datatype sta = Assign of id * exp
             | If of exp * sta * sta
             | While of exp * sta
             | Seq of sta list

datatype declaration = id * exp

datatype program = declaration list * sta * exp

fun match (a,ts) t = if null ts orelse hd ts <> t
                     then raise Error
                     else (a, tl ts)

fun combine a ts p f = let val (a’,tr) = p ts
                       in (f(a,a’), tr)
                       end

fun exp ts = case asexp ts of
               (a, LEQ::tr) => combine a tr asexp Leq
             |     ats      => ats
and asexp ts = asexp’ (mulexp ts)

and asexp’(a, ADD::ts) = asexp’(combine a ts mulexp Add)
  | asexp’(a, SUB::ts) = asexp’(combine a ts mulexp Sub)
  | asexp’    ats      = ats

and mulexp ts = mulexp’(atexp ts)
and mulexp’(a, MUL::ts) = mulexp’(combine a ts atexp Mul)
  | mulexp’    ats      = ats

and atexp ((ICON n)::ts) = (Con n, ts)
  | atexp (ID s  ::ts) = (Var s, ts)
  | atexp (LPAR  ::ts) = match (exp ts) RPAR
  | atexp      ts       = raise Error
```

```
and stmt ((ID s)::ASSIGN::ts) = (case exp ts of
                                  (e, tr) => (Assign(s, e), tr))
  | stmt (IF ::ts) = let
                       val (e1, ts1) = match (exp ts) THEN
                       val (e2, ts2) = match (stmt ts1) ELSE
                       val (e3, ts3) = stmt ts2
                     in
                       (If(e1, e2, e3), ts3)
                     end
  | stmt (WHILE ::ts) = let
                          val (e1, ts1) = match (exp ts) DO
                          val (e2, ts2) = match (stmts ts1) END
                        in
                          (While(e1, e2), ts2)
                        end
  | stmt ts = raise Error

and stmts ts = (case stmt ts of
                  (s, SEMICOLON::tr) =>
                  combine s tr stmts' (fn (s, ts) => Seq (s::ts))
                | sts => sts)

and stmts' ts = (case stmt ts of
                  (s, SEMICOLON::tr) =>
                  combine s tr stmts' (fn (s, ts) => s::ts)
                | (s,tr) => ([s], tr))

and decls ds (VAR::(ID s)::ASSIGN::tr) =
    let
      val (e, ts) = exp tr
    in decls ((s, e)::ds) ts
    end
  | decls ds  ts = (rev ds, ts)

and parse ts = let
                 val (ds, tr) = decls nil ts
                 val (s, tr)  = match (stmt tr) RETURN
                 val (e, tr)  = exp tr
               in
                 (case tr of nil => (ds, s, e)
                           | _   => raise Error)
               end
```