```
val put            = 1;
val get            = 2;

val instream       = 0;
val messagestream  = 0;
val binstream      = 512;

val EOF            = 255;

| tree node field selectors |
val t_op           = 0;
val t_op1          = 1;
val t_op2          = 2;
val t_op3          = 3;

| symbols |
val s_null         = 0;
val s_name         = 1;
val s_number       = 2;
val s_lbracket     = 3;
val s_rbracket     = 4;
val s_lparen       = 6;
val s_rparen       = 7;

val s_fncall       = 8;
val s_pcall        = 9;
val s_if           = 10;
val s_then         = 11;
val s_else         = 12;
val s_while        = 13;
val s_do           = 14;
val s_ass          = 15;
val s_skip         = 16;
val s_begin        = 17;
val s_end          = 18;
val s_semicolon    = 19;
val s_comma        = 20;
val s_var          = 21;
val s_array        = 22;
val s_body         = 23;
val s_proc         = 24;
val s_func         = 25;
val s_is           = 26;
val s_stop         = 27;

val s_not          = 32;
val s_neg          = 34;
val s_val          = 35;
val s_string        =  36;

val s_true          =  42;
val s_false         =  43;
val s_return        =  44;

val s_endfile       =  60;

val s_diadic        =  64;

val s_plus          =  s_diadic + 0;
val s_minus         =  s_diadic + 1;
val s_or            =  s_diadic + 5;
val s_and           =  s_diadic + 6;

val s_eq            =  s_diadic + 10;
val s_ne            =  s_diadic + 11;
val s_ls            =  s_diadic + 12;
val s_le            =  s_diadic + 13;
val s_gr            =  s_diadic + 14;
val s_ge            =  s_diadic + 15;

val s_sub           =  s_diadic + 16;

| up instruction codes |
val i_ldam          =  $0_{16}$;
val i_ldbm          =  $1_{16}$;
val i_stam          =  $2_{16}$;
val i_ldac          =  $3_{16}$;
val i_ldbc          =  $4_{16}$;
val i_ldap          =  $5_{16}$;
val i_ldai          =  $6_{16}$;
val i_ldbi          =  $7_{16}$;
val i_stai          =  $8_{16}$;
val i_br            =  $9_{16}$;
val i_brz           =  $A_{16}$;
val i_brn           =  $B_{16}$;
val i_opr           =  $D_{16}$;
val i_pfix          =  $E_{16}$;
val i_nfix          =  $F_{16}$;

val o_brb           =  $0_{16}$;
val o_add           =  $1_{16}$;
val o_sub           =  $2_{16}$;
val o_svc           =  $3_{16}$;

val r_areg          =  0;
val r_breg          =  1;
val m_sp            =  1;
val bytesperword      =  4;

| lexical analyser |
val linemax           =  200;
val nametablesize     =  101;
array nametable[nametablesize];
val nil               =  0;

var outstream;

val treemax           =  20000;
array tree[treemax];
var treep;
var namenode;
var nullnode;
var zeronode;
var numval;
var symbol;

array wordv[100];
var wordp;
var wordsize;

array charv[100];
var charp;
var ch;

array linev[linemax];
var linep;
var linelength;
var linecount;

| name scoping stack |
array names_d[500];
array names_v[500];
var namep;
var nameb;
val pflag             =  1000_16;

var arrayspace;
var arraycount;
var codesize;
var procdef;
var proclabel;
var infunc;

var stackp;
var stk_max;
```

```
| constants, strings and labels |
array consts[500];
var constp;

array strings[1000];
var stringp;

val labval_size       =  2000;
array labval[labval_size];
var labelcount;

val cb_size           =  15000;

| code buffer flags |
val cbf_inst          =  1;
val cbf_lab           =  2;
val cbf_fwdref        =  3;
val cbf_bwdref        =  4;
val cbf_stack         =  5;
val cbf_const         =  6;
val cbf_string        =  7;
val cbf_entry         =  8;
val cbf_pexit         =  9;
val cbf_fnexit        =  10;
val cbf_var           =  11;
val cbf_constp        =  12;
val cb_flag           =  $10000000_{16}$;
val cb_high           =  $1000000_{16}$;
var cbv_flag;
var cbv_high;
var cbv_low;

| code buffer variables |
array codebuffer[cb_size];
var cb_bufferp;
var cb_loadbase;
var cb_entryinstp;
var cb_blockstart;
var cb_loadpoint;
var cb_conststart;
var cb_stringstart;
var entrylab;
var mul_x;
var div_x;

val maxaddr           =  200000;
```

```
proc main() is
var t;
{  selectoutput(messagestream)
;  t := formtree()
;  prints("tree size : ")
;  printn(treep)
;  newline()
;  translate(t)
;  prints("program size : ")
;  printn(codesize)
;  newline()
;  prints("size : ")
;  printn(codesize + mul(arrayspace, 4))
;  newline()
}

proc selectoutput(val c) is
   outstream := c

proc putval(val c) is
   put(c, outstream)

proc newline() is
   putval('\n')

func lsu(val x, val y) is
   if (x < 0) = (y < 0)
   then
      return x < y
   else
      return y < 0
```

```
func $mul\_step$(val $b$, val $y$) is
var $r$;
{  if $(b < 0) \vee (\sim lsu(b, mul\_x))$
   then
      $r := 0$
   else
      $r := mul\_step(b + b, y + y)$
;  if $\sim lsu(mul\_x, b)$
   then
   {  $mul\_x := mul\_x - b$
   ;  $r := r + y$
   }
   else
      skip
;  return $r$
}

func $mul$(val $n$, val $m$) is
{  $mul\_x := m$
;  return $mul\_step(1, n)$
}

func $div\_step$(val $b$, val $y$) is
var $r$;
{  if $(y < 0) \vee (\sim lsu(y, div\_x))$
   then
      $r := 0$
   else
      $r := div\_step(b + b, y + y)$
;  if $\sim lsu(div\_x, y)$
   then
   {  $div\_x := div\_x - y$
   ;  $r := r + b$
   }
   else
      skip
;  return $r$
}
```

```
func $div$(val $n$, val $m$) is
{  $div\_x := n$
;  if $lsu(n, m)$
   then
      return 0
   else
      return $div\_step(1, m)$
}
```

```
func $rem$(val $n$, val $m$) is
var $x$;
{  $x := div(n, m)$
;  return $div\_x$
}
```

```
func $mul2$(val $x$, val $y$) is
var $n$;
var $r$;
{  $r := x$
;  $n := 1$
;  while $n \neq y$ do
   {  $r := r + r$
   ;  $n := n + n$
   }
;  return $r$
}
```

```
func $exp2$(val $n$) is
var $r$;
var $i$;
{  $i := n$
;  $r := 1$
;  while $i > 0$ do
   {  $r := r + r$
   ;  $i := i - 1$
   }
;  return $r$
}
```

```
func $packstring$(array $s$, array $v$) is
var $n$;
var $si$;
var $vi$;
var $w$;
var $b$;
{  $n := s[0]$
;  $si := 0$
;  $vi := 0$
;  $b := 0$
;  $w := 0$
;  while $si \leq n$ do
   {  $w := w + mul(s[si], exp2(mul2(b, 8)))$
   ;  $b := b + 1$
   ;  if $b = bytesperword$
      then
      {  $v[vi] := w$
      ;  $vi := vi + 1$
      ;  $w := 0$
      ;  $b := 0$
      }
      else
         skip
   ;  $si := si + 1$
   }
;  if $b = 0$
   then
      $vi := vi - 1$
   else
      $v[vi] := w$
;  return $vi$
}
```

```
proc $unpackstring$(array $s$, array $v$) is
var $si$;
var $vi$;
var $b$;
var $w$;
var $n$;
{  $si := 0$
;  $vi := 0$
;  $b := 0$
;  $w := s[0]$
;  $n := rem(w, 256)$
;  while $vi \leq n$ do
   {  $v[vi] := rem(w, 256)$
   ;  $w := div(w, 256)$
   ;  $vi := vi + 1$
   ;  $b := b + 1$
   ;  if $b = bytesperword$
      then
      {  $b := 0$
      ;  $si := si + 1$
      ;  $w := s[si]$
      }
      else
         skip
   }
}
```

```
proc prints(array s) is
var n;
var p;
var w;
var l;
var b;
{  n := 1
;  p := 0
;  w := s[p]
;  l := rem(w, 256)
;  w := div(w, 256)
;  b := 1
;  while n ≤ l do
   {  putval(rem(w, 256))
   ;  w := div(w, 256)
   ;  n := n + 1
   ;  b := b + 1
   ;  if b = bytesperword
      then
      {  b := 0
      ;  p := p + 1
      ;  w := s[p]
      }
      else
         skip
   }
}

proc printn(val n) is
   if n < 0
   then
   {  putval('−')
   ;  printn(− n)
   }
   else
   {  if n > 9
      then
         printn(div(n, 10))
      else
         skip
   ;  putval(rem(n, 10) + '0')
   }
```

```
proc printhex(val n) is
var d;
{  d := div(n, 16)
;  if d = 0
   then
      skip
   else
      printhex(d)
;  d := rem(n, 16)
;  if d < 10
   then
      putval(d + ‘0’)
   else
      putval((d − 10) + ‘a’)
}

func formtree() is
var i;
var t;
{  linep := 0
;  wordp := 0
;  charp := 0
;  treep := 1
;  i := 0
;  while i < nametablesize do
   {  nametable[i] := nil
   ;  i := i + 1
   }
;  declsyswords()
;  nullnode := cons1(s_null)
;  zeronode := cons2(s_number, 0)
;  linecount := 0
;  rdline()
;  rch()
;  nextsymbol()
;  if (symbol = s_var) ∨ (symbol = s_val) ∨ (symbol = s_array)
   then
      t := rgdecls()
   else
      t := nullnode
;  return cons3(s_body, t, rprocdecls())
}
```

```
proc cmperror(array s) is
{ prints(“error near line ”)
; printn(linecount)
; prints(“: ”)
; prints(s)
; newline()
}

| tree node constructors |
func newvec(val n) is
var t;
{ t := treep
; treep := treep + n
; if treep > treemax
    then
        cmperror(“out of space”)
    else
        skip
; return t
}

func cons1(val op) is
var t;
{ t := newvec(1)
; tree[t] := op
; return t
}

func cons2(val op, val t1) is
var t;
{ t := newvec(2)
; tree[t] := op
; tree[t + 1] := t1
; return t
}

func cons3(val op, val t1, val t2) is
var t;
{ t := newvec(3)
; tree[t] := op
; tree[t + 1] := t1
; tree[t + 2] := t2
; return t
}
```

```
func $cons4$(val $op$, val $t1$, val $t2$, val $t3$) is
var $t$;
{  $t$ := $newvec(4)$
;  $tree[t]$ := $op$
;  $tree[t + 1]$ := $t1$
;  $tree[t + 2]$ := $t2$
;  $tree[t + 3]$ := $t3$
;  return $t$
}
```

```
| name table lookup |
func lookupword() is
var a;
var hashval;
var i;
var stype;
var found;
var searching;
{  a := wordv[0]
;  hashval := rem(a, nametablesize)
;  namenode := nametable[hashval]
;  found := false
;  searching := true
;  while searching do
      if namenode = nil
      then
      {  found := false
      ;  searching := false
      }
      else
      {  i := 0
      ;  while (i ≤ wordsize) ∧ (tree[namenode + i + 2] = wordv[i]) do
            i := i + 1
      ;  if i ≤ wordsize
         then
            namenode := tree[namenode + 1]
         else
         {  stype := tree[namenode]
         ;  found := true
         ;  searching := false
         }
      }
;  if found
   then
      skip
   else
   {  namenode := newvec(wordsize + 3)
   ;  tree[namenode] := s_name
   ;  tree[namenode + 1] := nametable[hashval]
   ;  i := 0
   ;  while i ≤ wordsize do
      {  tree[namenode + i + 2] := wordv[i]
      ;  i := i + 1
      }
   ;  nametable[hashval] := namenode
   ;  stype := s_name
   }
;  return stype
}

proc declare(array s, val item) is
{  unpackstring(s, charv)
;  wordsize := packstring(charv, wordv)
;  lookupword()
;  tree[namenode] := item
}

proc declsyswords() is
{  declare("and", s_and)
;  declare("array", s_array)
;  declare("do", s_do)
;  declare("else", s_else)
;  declare("false", s_false)
;  declare("func", s_func)
;  declare("if", s_if)
;  declare("is", s_is)
;  declare("or", s_or)
;  declare("proc", s_proc)
;  declare("return", s_return)
;  declare("skip", s_skip)
;  declare("stop", s_stop)
;  declare("then", s_then)
;  declare("true", s_true)
;  declare("val", s_val)
;  declare("var", s_var)
;  declare("while", s_while)
}

func getchar() is
   return get(instream)

proc rdline() is
{  linelength := 1
;  linep := 1
;  linecount := linecount + 1
;  ch := getchar()
;  linev[linelength] := ch
;  while $(ch \neq \text{'\n'}) \wedge (ch \neq EOF) \wedge (linelength < linemax)$ do
   {  ch := getchar()
   ;  linelength := linelength + 1
   ;  linev[linelength] := ch
   }
}
```

```
proc rch() is
{  if $linep > linelength$
   then
      rdline()
   else
      skip
;  $ch := linev[linep]$
;  $linep := linep + 1$
}

proc rdtag() is
{  $charp := 0$
;  while $((ch \geq `A') \wedge (ch \leq `Z')) \vee ((ch \geq `a') \wedge (ch \leq `z')) \vee ((ch \geq `0') \wedge (ch \leq `9')) \vee (ch = `\_$
   {  $charp := charp + 1$
   ;  $charv[charp] := ch$
   ;  rch()
   }
;  $charv[0] := charp$
;  $wordsize := packstring(charv, wordv)$
}

proc readnumber(val base) is
var d;
{  $d := value(ch)$
;  $numval := 0$
;  if $d \geq base$
   then
      cmperror(“error in number”)
   else
      while $d < base$ do
      {  $numval := mul(numval, base) + d$
      ;  rch()
      ;  $d := value(ch)$
      }
}

func value(val c) is
   if $(c \geq `0') \wedge (c \leq `9')$
   then
      return $c - `0'$
   else
   if $(c \geq `A') \wedge (c \leq `Z')$
   then
      return $(c + 10) - `A'$
   else
      return 500
```

```
func readcharco() is
var v;
{  if ch  = ‘\’
   then
   {  rch()
   ;  if ch  = ‘\’
      then
         v  := ‘\’
      else
      if ch  = ‘\’’
      then
         v  := ‘\’’
      else
      if ch  = ‘\”’
      then
         v  := ‘\”’
      else
      if ch  = ‘n’
      then
         v  := ‘\n’
      else
      if ch  = ‘r’
      then
         v  := ‘\r’
      else
         cmperror(“error  in  character  constant”)
   }
   else
      v  :=  ch
;  rch()
;  return v
}
```

```
proc $readstring()$ is
var $charc$;
{  $charp := 0$
;  while $ch \neq$ ‘\"’ do
   {  if $charp = 255$
      then
         $cmperror(“error\ in\ string\ constant”)$
      else
         skip
   ;  $charc := readcharco()$
   ;  $charp := charp + 1$
   ;  $charv[charp] := charc$
   }
;  $charv[0] := charp$
;  $wordsize := packstring(charv, wordv)$
}
```

```
| lexical analyser main procedure |
proc nextsymbol() is
{  while $(ch = \text{‘}\backslash n\text{’}) \lor (ch = \text{‘}\backslash r\text{’}) \lor (ch = \text{‘ ’})$ do
      rch()
;  if $ch = \text{‘|’}$
   then
   {  rch()
   ;  while $ch \neq \text{‘|’}$ do
         rch()
   ;  rch()
   ;  nextsymbol()
   }
   else
   if $((ch \geq \text{‘}A\text{’}) \land (ch \leq \text{‘}Z\text{’})) \lor ((ch \geq \text{‘}a\text{’}) \land (ch \leq \text{‘}z\text{’}))$
   then
   {  rdtag()
   ;  symbol := lookupword()
   }
   else
   if $(ch \geq \text{‘0’}) \land (ch \leq \text{‘9’})$
   then
   {  symbol := s_number
   ;  readnumber(10)
   }
   else
   if $ch = \text{‘\#’}$
   then
   {  rch()
   ;  symbol := s_number
   ;  readnumber(16)
   }
   else
   if $ch = \text{‘[’}$
   then
   {  rch()
   ;  symbol := s_lbracket
   }
   else
   if $ch = \text{‘]’}$
   then
   {  rch()
   ;  symbol := s_rbracket
   }
   else
   if $ch = \text{‘(’}$
   then
   {  rch()
   ;  symbol := s_lparen
   }
   else
   if ch = ‘)’
   then
   {  rch()
   ;  symbol := s_rparen
   }
   else
   if ch = ‘{’
   then
   {  rch()
   ;  symbol := s_begin
   }
   else
   if ch = ‘}’
   then
   {  rch()
   ;  symbol := s_end
   }
   else
   if ch = ‘;’
   then
   {  rch()
   ;  symbol := s_semicolon
   }
   else
   if ch = ‘,’
   then
   {  rch()
   ;  symbol := s_comma
   }
   else
   if ch = ‘+’
   then
   {  rch()
   ;  symbol := s_plus
   }
   else
   if ch = ‘−’
   then
   {  rch()
   ;  symbol := s_minus
   }
   else
   if ch = ‘=’
   then
   {  rch()
   ;  symbol := s_eq
   }
```

```
else
if ch = ‘<’
then
{  rch()
;  if ch = ‘=’
   then
   {  rch()
   ;  symbol := s_le
   }
   else
      symbol := s_ls
}
else
if ch = ‘>’
then
{  rch()
;  if ch = ‘=’
   then
   {  rch()
   ;  symbol := s_ge
   }
   else
      symbol := s_gr
}
else
if ch = ‘∼’
then
{  rch()
;  if ch = ‘=’
   then
   {  rch()
   ;  symbol := s_ne
   }
   else
      symbol := s_not
}
else
if ch = ‘:’
then
{  rch()
;  if ch = ‘=’
   then
   {  rch()
   ;  symbol := s_ass
   }
   else
      cmperror(“\’= \’ expected”)
}
else
```

```
    if ch = ‘\’’
    then
    {  rch()
    ;  numval := readcharco()
    ;  if ch = ‘\’’
       then
          rch()
       else
          cmperror(“error in character constant”)
    ;  symbol := s_number
    }
    else
    if ch = ‘\”’
    then
    {  rch()
    ;  readstring()
    ;  if ch = ‘\”’
       then
          rch()
       else
          cmperror(“error in string constant”)
    ;  symbol := s_string
    }
    else
    if ch = EOF
    then
       symbol := s_endfile
    else
       cmperror(“illegal character”)
}

| syntax analyser |
proc checkfor(val s, array m) is
    if symbol = s
    then
       nextsymbol()
    else
       cmperror(m)
```

```
func $rname()$ is
var $a$;
{  if $symbol\ =\ s\_name$
   then
   {  $a\ :=\ namenode$
   ;  $nextsymbol()$
   }
   else
      $cmperror(\text{“}name\ \ expected\text{”})$
;  return $a$
}
```

```
func relement() is
var a;
var b;
var i;
{ if symbol = s_name
  then
  { a := rname()
  ; if symbol = s_lbracket
    then
    { nextsymbol()
    ; b := rexpression()
    ; checkfor(s_rbracket, "\']\' expected")
    ; a := cons3(s_sub, a, b)
    }
    else
    if symbol = s_lparen
    then
    { nextsymbol()
    ; if symbol = s_rparen
      then
        b := nullnode
      else
        b := rexplist()
    ; checkfor(s_rparen, "\')\' expected")
    ; a := cons3(s_fncall, a, b)
    }
    else
      skip
  }
  else
  if symbol = s_number
  then
  { a := cons2(s_number, numval)
  ; nextsymbol()
  }
  else
  if (symbol = s_true) ∨ (symbol = s_false)
  then
  { a := namenode
  ; nextsymbol()
  }
  else
  if symbol = s_string
  then
  { a := newvec(wordsize + 2)
  ; tree[a + t_op] := s_string
  ; i := 0
  ; while i ≤ wordsize do
```

```
        {  tree[a + i + 1] := wordv[i]
        ;  i := i + 1
        }
    ;  nextsymbol()
    }
    else
    if symbol = s_lparen
    then
    {  nextsymbol()
    ;  a := rexpression()
    ;  checkfor(s_rparen, "\')\' expected")
    }
    else
        cmperror("error in expression")
;  return a
}

func rexpression() is
var a;
var b;
var s;
    if symbol = s_minus
    then
    {  nextsymbol()
    ;  b := relement()
    ;  return cons2(s_neg, b)
    }
    else
    if symbol = s_not
    then
    {  nextsymbol()
    ;  b := relement()
    ;  return cons2(s_not, b)
    }
    else
    {  a := relement()
    ;  if diadic(symbol)
        then
        {  s := symbol
        ;  nextsymbol()
        ;  return cons3(s, a, rright(s))
        }
        else
            return a
    }
```

```
func $rright$(val $s$) is
var $b$;
{  $b := relement()$
;  if $associative(s) \wedge (symbol = s)$
   then
   {  $nextsymbol()$
   ;  return $cons3(s, b, rright(s))$
   }
   else
      return $b$
}

func $associative$(val $s$) is
   return $(s = s\_and) \vee (s = s\_or) \vee (s = s\_plus)$

func $rexplist$() is
var $a$;
{  $a := rexpression()$
;  if $symbol = s\_comma$
   then
   {  $nextsymbol()$
   ;  return $cons3(s\_comma, a, rexplist())$
   }
   else
      return $a$
}
```

```
func rstatement() is
var a;
var b;
var c;
    if symbol = s_skip
    then
    {  nextsymbol()
    ;  return cons1(s_skip)
    }
    else
    if symbol = s_stop
    then
    {  nextsymbol()
    ;  return cons1(s_stop)
    }
    else
    if symbol = s_return
    then
    {  nextsymbol()
    ;  return cons2(s_return, rexpression())
    }
    else
    if symbol = s_if
    then
    {  nextsymbol()
    ;  a := rexpression()
    ;  checkfor(s_then, “\’then\’ expected”)
    ;  b := rstatement()
    ;  checkfor(s_else, “\’else\’ expected”)
    ;  c := rstatement()
    ;  return cons4(s_if, a, b, c)
    }
    else
    if symbol = s_while
    then
    {  nextsymbol()
    ;  a := rexpression()
    ;  checkfor(s_do, “\’do\’ expected”)
    ;  b := rstatement()
    ;  return cons3(s_while, a, b)
    }
    else
    if symbol = s_begin
    then
    {  nextsymbol()
    ;  a := rstatements()
    ;  checkfor(s_end, “\’}\’ expected”)
    ;  return a
    }
    else
    if $symbol = s\_name$
    then
    {  $a := relement()$
    ;  if $tree[a + t\_op] = s\_fncall$
       then
       {  $tree[a + t\_op] := s\_pcall$
       ;  return $a$
       }
       else
       {  $checkfor(s\_ass,$ “\’:= \’ $expected$”$)$
       ;  return $cons3(s\_ass, a, rexpression())$
       }
    }
    else
    {  $cmperror($“$error\ in\ command$”$)$
    ;  return $cons1(s\_stop)$
    }

func $rstatements()$ is
var $a$;
{  $a := rstatement()$
;  if $symbol = s\_semicolon$
   then
   {  $nextsymbol()$
   ;  return $cons3(s\_semicolon, a, rstatements())$
   }
   else
      return $a$
}

func $rprocdecls()$ is
var $a$;
{  $a := rprocdecl()$
;  if $(symbol = s\_proc) \lor (symbol = s\_func)$
   then
      return $cons3(s\_semicolon, a, rprocdecls())$
   else
      return $a$
}
```

```
func rprocdecl() is
var s;
var a;
var b;
var c;
{  s := symbol
;  nextsymbol()
;  a := rname()
;  checkfor(s_lparen, “\’(\’ expected”)
;  if symbol = s_rparen
   then
      b := nullnode
   else
      b := rformals()
;  checkfor(s_rparen, “\’)\’ expected”)
;  checkfor(s_is, “\’is\’ expected”)
;  if (symbol = s_var) $\vee$ (symbol = s_val)
   then
      c := rldecls()
   else
      c := nullnode
;  c := cons3(s_body, c, rstatement())
;  return cons4(s, a, b, c)
}
```

```
func $rformals()$ is
var $s$;
var $a$;
var $b$;
{  if $(symbol\ =\ s\_val)\ \vee\ (symbol\ =\ s\_array)\ \vee\ (symbol\ =\ s\_proc)\ \vee\ (symbol\ =\ s\_func)$
   then
   {  $s\ :=\ symbol$
   ;  $nextsymbol()$
   ;  if $symbol\ =\ s\_name$
      then
         $a\ :=\ cons2(s,\ rname())$
      else
         $cmperror($“$name\ \ expected$”$)$
   }
   else
      skip
;  if $symbol\ =\ s\_comma$
   then
   {  $nextsymbol()$
   ;  $b\ :=\ rformals()$
   ;  return $cons3(s\_comma,\ a,\ b)$
   }
   else
      return $a$
}

func $rgdecls()$ is
var $a$;
{  $a\ :=\ rdecl()$
;  if $(symbol\ =\ s\_val)\ \vee\ (symbol\ =\ s\_var)\ \vee\ (symbol\ =\ s\_array)$
   then
      return $cons3(s\_semicolon,\ a,\ rgdecls())$
   else
      return $a$
}

func $rldecls()$ is
var $a$;
{  $a\ :=\ rdecl()$
;  if $(symbol\ =\ s\_val)\ \vee\ (symbol\ =\ s\_var)$
   then
      return $cons3(s\_semicolon,\ a,\ rldecls())$
   else
      return $a$
}
```

```
func rdecl() is
var a;
var b;
{  if symbol = s_var
   then
   {  nextsymbol()
   ;  a := cons2(s_var, rname())
   }
   else
   if symbol = s_array
   then
   {  nextsymbol()
   ;  a := rname()
   ;  checkfor(s_lbracket, “\’[\’ expected”)
   ;  b := rexpression()
   ;  checkfor(s_rbracket, “\’]\’ expected”)
   ;  a := cons3(s_array, a, b)
   }
   else
   if symbol = s_val
   then
   {  nextsymbol()
   ;  a := rname()
   ;  checkfor(s_eq, “\’=\’ expected”)
   ;  b := rexpression()
   ;  a := cons3(s_val, a, b)
   }
   else
      skip
;  checkfor(s_semicolon, “\’;\’ expected”)
;  return a
}
```

```
proc $namemessage$(array $s$, val $x$) is
var $n$;
var $p$;
var $w$;
var $l$;
var $b$;
{  $prints(s)$
;  if $tree[x + t\_op] = s\_name$
   then
   {  $n := 1$
   ;  $p := 2$
   ;  $w := tree[x + p]$
   ;  $l := rem(w, 256)$
   ;  $w := div(w, 256)$
   ;  $b := 1$
   ;  while $n \le l$ do
      {  $putval(rem(w, 256))$
      ;  $w := div(w, 256)$
      ;  $n := n + 1$
      ;  $b := b + 1$
      ;  if $b = bytesperword$
         then
         {  $b := 0$
         ;  $p := p + 1$
         ;  $w := tree[x + p]$
         }
         else
            skip
      }
   }
   else
      skip
;  $newline()$
}

proc $generror$(array $s$) is
{  $prints(s)$
;  $newline()$
;  $namemessage(``in\ function\ ", tree[procdef + t\_op1])$
}
```

```
| translator |
proc declprocs(val x) is
    if tree[x + t_op] = s_semicolon
    then
    { declprocs(tree[x + t_op1])
    ; declprocs(tree[x + t_op2])
    }
    else
        addname(x, getlabel())

proc declformals(val x) is
var op;
{ op := tree[x + t_op]
; if op = s_null
    then
        skip
    else
    if op = s_comma
    then
    { declformals(tree[x + t_op1])
    ; declformals(tree[x + t_op2])
    }
    else
    { if op = s_val
        then
            tree[x + t_op] := s_var
        else
            skip
    ; addname(x, stackp + pflag)
    ; stackp := stackp + 1
    }
}
```

```
proc declglobals(val x) is
var op;
{  op := tree[x + t_op]
;  if op = s_semicolon
   then
   {  declglobals(tree[x + t_op1])
   ;  declglobals(tree[x + t_op2])
   }
   else
   if op = s_var
   then
   {  addname(x, stackp)
   ;  stackp := stackp + 1
   }
   else
   if op = s_val
   then
   {  tree[x + t_op2] := optimiseexpr(tree[x + t_op2])
   ;  if isval(tree[x + t_op2])
      then
         addname(x, getval(tree[x + t_op2]))
      else
         generror("constant expression expected")
   }
   else
   if op = s_array
   then
   {  tree[x + t_op2] := optimiseexpr(tree[x + t_op2])
   ;  if isval(tree[x + t_op2])
      then
      {  arrayspace := arrayspace + getval(tree[x + t_op2])
      ;  addname(x, stackp)
      ;  stackp := stackp + 1
      }
      else
         generror("constant expression expected")
   }
   else
      skip
}
```

```
proc tglobals() is
var g;
var arraybase;
var name;
{  g := 0
;  arraybase := maxaddr − arrayspace
;  gen(cbf_var, 0, arraybase − 2)
;  while g < namep do
   {  name := names_d[g]
   ;  if tree[name + t_op] = s_array
      then
      {  gen(cbf_var, 0, arraybase)
      ;  arraybase := arraybase + getval(tree[name + t_op2])
      }
      else
      if tree[name + t_op] = s_var
      then
         gen(cbf_var, 0, 0)
      else
         skip
   ;  g := g + 1
   }
}
```

```
proc decllocals(val x) is
var op;
{  op := tree[x + t_op]
;  if op = s_null
   then
      skip
   else
   if op = s_semicolon
   then
   {  decllocals(tree[x + t_op1])
   ;  decllocals(tree[x + t_op2])
   }
   else
   if op = s_var
   then
   {  addname(x, stackp)
   ;  stackp := stackp + 1
   }
   else
   if op = s_val
   then
   {  tree[x + t_op2] := optimiseexpr(tree[x + t_op2])
   ;  if isval(tree[x + t_op2])
      then
         addname(x, getval(tree[x + t_op2]))
      else
         generror("constant expression expected")
   }
   else
      skip
}

proc addname(val x, val v) is
{  names_d[namep] := x
;  names_v[namep] := v
;  namep := namep + 1
}
```

```
func $findname$(val $x$) is
var $n$;
var $found$;
{  $found$ := false
;  $n := namep - 1$
;  while ($found$ = false) $\wedge$ ($n \geq 0$) do
      if $tree[names\_d[n] + t\_op1] = x$
      then
         $found$ := true
      else
         $n := n - 1$
;  if $found$
   then
      skip
   else
   {  $namemessage$("$name\ \ not\ \ declared$ ", $x$)
   ;  $namemessage$("$in\ \ function$", $tree[procdef + t\_op1]$)
   }
;  return $n$
}

func $islocal$(val $n$) is
   return $n \geq nameb$
```

```
proc $optimise$(val $x$) is
var $op$;
{  $op := tree[x + t\_op]$
;  if $(op = s\_skip) \lor (op = s\_stop)$
   then
      skip
   else
   if $op = s\_return$
   then
      $tree[x + t\_op1] := optimiseexpr(tree[x + t\_op1])$
   else
   if $op = s\_if$
   then
   {  $tree[x + t\_op1] := optimiseexpr(tree[x + t\_op1])$
   ;  $optimise(tree[x + t\_op2])$
   ;  $optimise(tree[x + t\_op3])$
   }
   else
   if $op = s\_while$
   then
   {  $tree[x + t\_op1] := optimiseexpr(tree[x + t\_op1])$
   ;  $optimise(tree[x + t\_op2])$
   }
   else
   if $op = s\_ass$
   then
   {  $tree[x + t\_op2] := optimiseexpr(tree[x + t\_op2])$
   ;  $tree[x + t\_op1] := optimiseexpr(tree[x + t\_op1])$
   }
   else
   if $op = s\_pcall$
   then
   {  $tree[x + t\_op2] := optimiseexpr(tree[x + t\_op2])$
   ;  $tree[x + t\_op1] := optimiseexpr(tree[x + t\_op1])$
   }
   else
   if $op = s\_semicolon$
   then
   {  $optimise(tree[x + t\_op1])$
   ;  $optimise(tree[x + t\_op2])$
   }
   else
      skip
}
```

```
func optimiseexpr(val x) is
var op;
var name;
var r;
var temp;
var left;
var right;
var leftop;
var rightop;
{  r := x
;  op := tree[x + t_op]
;  if op = s_name
   then
   {  name := findname(x)
   ;  if tree[names_d[name] + t_op] = s_val
      then
         r := tree[names_d[name] + t_op2]
      else
         skip
   }
   else
   if monadic(op)
   then
   {  tree[x + t_op1] := optimiseexpr(tree[x + t_op1])
   ;  if isval(tree[x + t_op1])
      then
      {  tree[x + t_op1] := evalmonadic(x)
      ;  tree[x + t_op] := s_number
      }
      else
      if op = s_neg
      then
         r := cons3(s_minus, zeronode, tree[x + t_op1])
      else
         skip
   }
   else
   if op = s_fncall
   then
   {  tree[x + t_op2] := optimiseexpr(tree[x + t_op2])
   ;  tree[x + t_op1] := optimiseexpr(tree[x + t_op1])
   }
   else
   if diadic(op)
   then
   {  tree[x + t_op2] := optimiseexpr(tree[x + t_op2])
   ;  tree[x + t_op1] := optimiseexpr(tree[x + t_op1])
   ;  left := tree[x + t_op1]
   ;   $right := tree[x + t\_op2]$
   ;   $leftop := tree[left + t\_op]$
   ;   $rightop := tree[right + t\_op]$
   ;   if $op = s\_sub$
      then
         skip
      else
      if $isval(left) \wedge isval(right)$
      then
      {   $tree[x + t\_op1] := evaldiadic(x)$
      ;   $tree[x + t\_op] := s\_number$
      }
      else
      if $op = s\_eq$
      then
         if $(leftop = s\_not) \wedge (rightop = s\_not)$
         then
         {   $tree[x + t\_op1] := tree[left + t\_op1]$
         ;   $tree[x + t\_op2] := tree[right + t\_op1]$
         }
         else
             skip
      else
      if $op = s\_ne$
      then
      {   $tree[x + t\_op] := s\_eq$
      ;   $r := cons2(s\_not, x)$
      ;   if $(leftop = s\_not) \wedge (rightop = s\_not)$
         then
         {   $tree[x + t\_op1] := tree[left + t\_op1]$
         ;   $tree[x + t\_op2] := tree[right + t\_op1]$
         }
         else
             skip
      }
      else
      if $op = s\_ge$
      then
      {   $tree[x + t\_op] := s\_ls$
      ;   $r := cons2(s\_not, x)$
      }
      else
      if $op = s\_gr$
      then
      {   $temp := tree[x + t\_op1]$
      ;   $tree[x + t\_op1] := tree[x + t\_op2]$
      ;   $tree[x + t\_op2] := temp$
      ;   $tree[x + t\_op] := s\_ls$
      }
```

```
  else
  if $op = s\_le$
  then
  { $temp := tree[x + t\_op1]$
  ; $tree[x + t\_op1] := tree[x + t\_op2]$
  ; $tree[x + t\_op2] := temp$
  ; $tree[x + t\_op] := s\_ls$
  ; $r := cons2(s\_not, x)$
  }
  else
  if $(op = s\_or) \vee (op = s\_and)$
  then
    if $(leftop = s\_not) \wedge (rightop = s\_not)$
    then
    { $r := cons2(s\_not, x)$
    ; if $tree[x + t\_op] = s\_and$
      then
        $tree[x + t\_op] := s\_or$
      else
        $tree[x + t\_op] := s\_and$
    ; $tree[x + t\_op1] := tree[left + t\_op1]$
    ; $tree[x + t\_op2] := tree[right + t\_op1]$
    }
    else
      skip
  else
  if $((op = s\_plus) \vee (op = s\_or)) \wedge (iszero(tree[x + t\_op1]) \vee iszero(tree[x + t\_op2]))$
  then
    if $iszero(tree[x + t\_op1])$
    then
      $r := tree[x + t\_op2]$
    else
    if $iszero(tree[x + t\_op2])$
    then
      $r := tree[x + t\_op1]$
    else
      skip
  else
  if $(op = s\_minus) \wedge iszero(tree[x + t\_op2])$
  then
    $r := tree[x + t\_op1]$
  else
    skip
}
else
if $op = s\_comma$
then
{ $tree[x + t\_op2] := optimiseexpr(tree[x + t\_op2])$
; $tree[x + t\_op1] := optimiseexpr(tree[x + t\_op1])$
```

```
    }
    else
        skip
;   return $r$
}

func $isval$(val $x$) is
var $op$;
{   $op := tree[x + t\_op]$
;   return $(op = s\_true) \vee (op = s\_false) \vee (op = s\_number)$
}

func $getval$(val $x$) is
var $op$;
{   $op := tree[x + t\_op]$
;   if $op = s\_true$
    then
        return 1
    else
    if $op = s\_false$
    then
        return 0
    else
    if $op = s\_number$
    then
        return $tree[x + t\_op1]$
    else
        return 0
}
```

```
func $evalmonadic$(val $x$) is
var $op$;
var $opd$;
{  $op := tree[x + t\_op]$
;  $opd := getval(tree[x + t\_op1])$
;  if $op = s\_neg$
   then
      return $-opd$
   else
   if $op = s\_not$
   then
      return $\sim opd$
   else
   {  $generror($“$compiler\ \ error$”$)$
   ;  return 0
   }
}
```

```
func $evaldiadic$(val $x$) is
var $op$;
var $left$;
var $right$;
{  $op := tree[x + t\_op]$
;  $left := getval(tree[x + t\_op1])$
;  $right := getval(tree[x + t\_op2])$
;  if $op = s\_plus$
   then
      return $left + right$
   else
   if $op = s\_minus$
   then
      return $left - right$
   else
   if $op = s\_eq$
   then
      return $left = right$
   else
   if $op = s\_ne$
   then
      return $left \neq right$
   else
   if $op = s\_ls$
   then
      return $left < right$
   else
   if $op = s\_gr$
   then
      return $left > right$
   else
   if $op = s\_le$
   then
      return $left \leq right$
   else
   if $op = s\_ge$
   then
      return $left \geq right$
   else
   if $op = s\_or$
   then
      return $left \vee right$
   else
   if $op = s\_and$
   then
      return $left \wedge right$
   else
   {  $cmperror(``optimise\ error")$
   ;  return 0
   }
}

proc translate(val t) is
var s;
var dlab;
var mainlab;
var link;
{  namep := 0
;  nameb := 0
;  labelcount := 1
;  initlabels()
;  initbuffer()
;  arrayspace := 0
;  stk_init(m_sp + 1)
;  declglobals(tree[t + t_op1])
;  tglobals(tree[t + t_op1])
;  gen(cbf_constp, 0, 0)
;  declprocs(tree[t + t_op2])
;  nameb := namep
;  entrylab := getlabel()
;  mainlab := getlabel()
;  link := getlabel()
;  setlab(entrylab)
;  genref(i_ldap, link)
;  genref(i_br, mainlab)
;  setlab(link)
;  geni(i_ldac, 0)
;  geni(i_opr, o_svc)
;  setlab(mainlab)
;  genprocs(tree[t + t_op2])
;  flushbuffer()
}
```

```
proc $genprocs$(val $x$) is
var $body$;
var $savetreep$;
var $pn$;
   if $tree[x\ +\ t\_op]\ =\ s\_semicolon$
   then
   {  $genprocs(tree[x\ +\ t\_op1])$
   ;  $genprocs(tree[x\ +\ t\_op2])$
   }
   else
   {  $savetreep\ :=\ treep$
   ;  $namep\ :=\ nameb$
   ;  $pn\ :=\ findname(tree[x\ +\ t\_op1])$
   ;  $proclabel\ :=\ names\_v[pn]$
   ;  $procdef\ :=\ names\_d[pn]$
   ;  $infunc\ :=\ tree[procdef\ +\ t\_op]\ =\ s\_func$
   ;  $body\ :=\ tree[x\ +\ t\_op3]$
   ;  if $infunc$
      then
         $stk\_init(2)$
      else
         $stk\_init(1)$
   ;  $declformals(tree[x\ +\ t\_op2])$
   ;  $setlab(proclabel)$
   ;  $genentry()$
   ;  $stk\_init(1)$
   ;  $decllocals(tree[body\ +\ t\_op1])$
   ;  $setstack()$
   ;  $optimise(tree[body\ +\ t\_op2])$
   ;  $genstatement(tree[body\ +\ t\_op2]$, true, 0, true)
   ;  $genexit()$
   ;  $treep\ :=\ savetreep$
   }

func $funtail$(val $tail$) is
   return $infunc\ \wedge\ tail$
```

```
proc genstatement(val x, val seq, val clab, val tail) is
var op;
var op1;
var lab;
var thenpart;
var elsepart;
var elselab;
{  op := tree[x + t_op]
;  if op = s_semicolon
   then
   {  genstatement(tree[x + t_op1], true, 0, false)
   ;  genstatement(tree[x + t_op2], seq, clab, tail)
   }
   else
   if (op = s_if) ∧ (clab = 0)
   then
   {  lab := getlabel()
   ;  genstatement(x, true, lab, tail)
   ;  setlab(lab)
   }
   else
   if op = s_if
   then
   {  thenpart := tree[x + t_op2]
   ;  elsepart := tree[x + t_op3]
   ;  if (∼ funtail(tail)) ∧ ((tree[thenpart + t_op] = s_skip) ∨ (tree[elsepart + t_op] = s_skip))
      then
      {  gencondjump(tree[x + t_op1], tree[thenpart + t_op] = s_skip, clab)
      ;  if tree[thenpart + t_op] = s_skip
         then
            genstatement(elsepart, seq, clab, tail)
         else
            genstatement(thenpart, seq, clab, tail)
      }
      else
      {  elselab := getlabel()
      ;  gencondjump(tree[x + t_op1], false, elselab)
      ;  genstatement(thenpart, false, clab, tail)
      ;  setlab(elselab)
      ;  genstatement(elsepart, seq, clab, tail)
      }
   }
   else
   if funtail(tail)
   then
      if op = s_return
      then
      {  op1 := tree[x + t_op1]
      ; if $tree[op1 + t\_op] = s\_fncall$
         then
            $tcall(op1, seq, clab, tail)$
         else
         { $texp(tree[x + t\_op1])$
         ; $genbr(seq, clab)$
         }
      }
      else
         generror(“\”return\” expected”)
   else
   if $(op = s\_while) \wedge (clab = 0)$
   then
   { $lab := getlabel()$
   ; $genstatement(x$, false, $lab$, false$)$
   ; $setlab(lab)$
   }
   else
   if $op = s\_while$
   then
   { $lab := getlabel()$
   ; $setlab(lab)$
   ; $gencondjump(tree[x + t\_op1]$, false, $clab)$
   ; $genstatement(tree[x + t\_op2]$, false, $lab$, false$)$
   }
   else
   if $op = s\_pcall$
   then
      $tcall(x, seq, clab, tail)$
   else
   if $op = s\_stop$
   then
   { $geni(i\_ldac, 0)$
   ; $geni(i\_opr, o\_svc)$
   }
   else
   { if $op = s\_skip$
      then
         skip
      else
      if $op = s\_ass$
      then
         $genassign(tree[x + t\_op1], tree[x + t\_op2])$
      else
      if $op = s\_return$
      then
         generror(“misplaced \”return\””)
      else
         skip
```

```
    ;   $genbr(seq,\ clab)$
    }
}
```

```
proc tbool(val x, val cond) is
var op;
var lab;
{  op := tree[x + t_op]
;  if op = s_not
   then
      tbool(tree[x + t_op1], ∼ cond)
   else
   if (op = s_and) ∨ (op = s_or)
   then
   {  lab := getlabel()
   ;  gencondjump(x, cond, lab)
   ;  geni(i_ldac, 0)
   ;  geni(i_br, 1)
   ;  setlab(lab)
   ;  geni(i_ldac, 1)
   }
   else
   if op = s_eq
   then
   {  if iszero(tree[x + t_op1])
      then
         texp(tree[x + t_op2])
      else
      if iszero(tree[x + t_op2])
      then
         texp(tree[x + t_op1])
      else
         texp2(s_minus, tree[x + t_op1], tree[x + t_op2])
   ;  if cond
      then
      {  geni(i_brz, 2)
      ;  geni(i_ldac, 0)
      ;  geni(i_br, 1)
      ;  geni(i_ldac, 1)
      }
      else
      {  geni(i_brz, 1)
      ;  geni(i_ldac, 1)
      }
   }
   else
   if op = s_ls
   then
   {  if iszero(tree[x + t_op2])
      then
         texp(tree[x + t_op1])
      else
```

```
            $texp2(s\_minus,\ tree[x\ +\ t\_op1],\ tree[x\ +\ t\_op2])$
    ;   if $cond$
        then
        {   $geni(i\_brn,\ 2)$
        ;   $geni(i\_ldac,\ 0)$
        ;   $geni(i\_br,\ 1)$
        ;   $geni(i\_ldac,\ 1)$
        }
        else
        {   $geni(i\_brn,\ 2)$
        ;   $geni(i\_ldac,\ 1)$
        ;   $geni(i\_br,\ 1)$
        ;   $geni(i\_ldac,\ 0)$
        }
    }
    else
    {   $texp(x)$
    ;   if $cond$
        then
            skip
        else
        {   $geni(i\_brz,\ 2)$
        ;   $geni(i\_ldac,\ 0)$
        ;   $geni(i\_br,\ 1)$
        ;   $geni(i\_ldac,\ 1)$
        }
    }
}
```

```
proc gencondjump(val x, val cond, val target) is
var op;
var lab;
{  op := tree[x + t_op]
;  if op = s_not
   then
      gencondjump(tree[x + t_op1], ∼ cond, target)
   else
   if $(op = s\_and) \lor (op = s\_or)$
   then
      if $((op = s\_and) \land cond) \lor ((op = s\_or) \land (\sim cond))$
      then
      {  lab := getlabel()
      ;  gencondjump(tree[x + t_op1], ∼ cond, lab)
      ;  gencondjump(tree[x + t_op2], ∼ cond, lab)
      ;  genref(i_br, target)
      ;  setlab(lab)
      }
      else
      {  gencondjump(tree[x + t_op1], cond, target)
      ;  gencondjump(tree[x + t_op2], cond, target)
      }
   else
   if op = s_eq
   then
   {  if iszero(tree[x + t_op1])
      then
         texp(tree[x + t_op2])
      else
      if iszero(tree[x + t_op2])
      then
         texp(tree[x + t_op1])
      else
         texp2(s_minus, tree[x + t_op1], tree[x + t_op2])
   ;  genjump(i_brz, cond, target)
   }
   else
   if op = s_ls
   then
   {  if iszero(tree[x + t_op2])
      then
         texp(tree[x + t_op1])
      else
         texp2(s_minus, tree[x + t_op1], tree[x + t_op2])
   ;  genjump(i_brn, cond, target)
   }
   else
   {  texp(x)
```

```
    ;  $genjump(i\_brz,\ \sim\ cond,\ target)$
    }
}

proc $genjump$(val $inst$, val $cond$, val $target$) is
var $lab$;
    if $cond$
    then
        $genref(inst,\ target)$
    else
    {  $lab\ :=\ getlabel()$
    ;  $genref(inst,\ lab)$
    ;  $genref(i\_br,\ target)$
    ;  $setlab(lab)$
    }
```

```
proc tcall(val x, val seq, val clab, val tail) is
var sp;
var entry;
var actuals;
var def;
{  sp := stackp
;  actuals := tree[x + t_op2]
;  if isval(tree[x + t_op1])
   then
   {  tactuals(actuals, 2)
   ;  texp(tree[x + t_op1])
   ;  geni(i_opr, o_svc)
   ;  geni(i_ldam, m_sp)
   ;  geni(i_ldai, 1)
   }
   else
   {  entry := findname(tree[x + t_op1])
   ;  def := names_d[entry]
   ;  if tree[def + t_op] = s_func
      then
      {  tactuals(actuals, 2)
      ;  gencall(entry, actuals)
      ;  geni(i_ldai, 1)
      }
      else
      {  tactuals(actuals, 1)
      ;  gencall(entry, actuals)
      }
   ;  genbr(seq, clab)
   }
;  stackp := sp
}

proc tactuals(val aps, val n) is
var sp;
{  sp := stackp
;  preparecalls(aps)
;  loadaps(aps, n)
;  stackp := stackp + numps(aps) + n
;  setstack()
;  stackp := sp
;  loadcalls(aps, n)
;  stackp := sp
}
```

```
func numps(val x) is
   if tree[x + t_op] = s_null
   then
      return 0
   else
   if tree[x + t_op] = s_comma
   then
      return 1 + numps(tree[x + t_op2])
   else
      return 1

proc gencall(val entry, val actuals) is
var link;
var def;
{  link := getlabel()
;  genref(i_ldap, link)
;  if islocal(entry)
   then
   {  loadvar(r_breg, entry)
   ;  geni(i_opr, o_brb)
   }
   else
   {  def := names_d[entry]
   ;  checkps(tree[def + t_op2], actuals)
   ;  genref(i_br, names_v[entry])
   }
;  setlab(link)
}

proc preparecalls(val x) is
   if tree[x + t_op] = s_comma
   then
   {  preparecalls(tree[x + t_op2])
   ;  preparecall(tree[x + t_op1])
   }
   else
      preparecall(x)
```

```
proc $preparecall$(val $x$) is
var $op$;
var $vn$;
var $sp$;
{  $op$ := $tree[x + t\_op]$
;  if $op = s\_null$
   then
      skip
   else
   if $containscall(x)$
   then
   {  $sp$ := $stackp$
   ;  $texp(x)$
   ;  $stackp$ := $stackp + 1$
   ;  $setstack()$
   ;  $geni(i\_ldbm, m\_sp)$
   ;  $gensref(i\_stai, sp)$
   }
   else
      skip
}

proc $loadcalls$(val $x$, val $n$) is
   if $tree[x + t\_op] = s\_comma$
   then
   {  $loadcalls(tree[x + t\_op2], n + 1)$
   ;  $loadcall(tree[x + t\_op1], n)$
   }
   else
      $loadcall(x, n)$
```

```
proc $loadcall$(val $x$, val $n$) is
var $op$;
var $vn$;
var $sp$;
{  $op\ :=\ tree[x\ +\ t\_op]$
;  if $op\ =\ s\_null$
   then
      skip
   else
   if $containscall(x)$
   then
   {  $geni(i\_ldam,\ m\_sp)$
   ;  $gensref(i\_ldai,\ stackp)$
   ;  $stackp\ :=\ stackp\ +\ 1$
   ;  $geni(i\_ldbm,\ m\_sp)$
   ;  $geni(i\_stai,\ n)$
   }
   else
      skip
}

proc $loadaps$(val $x$, val $n$) is
   if $tree[x\ +\ t\_op]\ =\ s\_comma$
   then
   {  $loadaps(tree[x\ +\ t\_op2],\ n\ +\ 1)$
   ;  $loadap(tree[x\ +\ t\_op1],\ n)$
   }
   else
      $loadap(x,\ n)$
```

```
proc $loadap$(val $x$, val $n$) is
var $op$;
var $vn$;
var $aptype$;
{  $op := tree[x + t\_op]$
;  if $op = s\_null$
   then
      skip
   else
   if $containscall(x)$
   then
      skip
   else
   {  if $op = s\_name$
      then
      {  $vn := findname(x)$
      ;  $aptype := tree[names\_d[vn] + t\_op]$
      ;  if $aptype = s\_val$
         then
            $loadconst(r\_areg, names\_v[vn])$
         else
         if $aptype = s\_func$
         then
            if $islocal(vn)$
            then
               $loadvar(r\_areg, vn)$
            else
               $genref(i\_ldap, names\_v[vn])$
         else
            $loadvar(r\_areg, vn)$
      }
      else
         $texp(x)$
   ;  $geni(i\_ldbm, m\_sp)$
   ;  $geni(i\_stai, n)$
   }
}
```

```
proc $checkps$(val $alist$, val $flist$) is
var $ax$;
var $fx$;
{  $ax$ := $alist$
;  $fx$ := $flist$
;  while $tree[fx + t\_op] = s\_comma$ do
      if $tree[ax + t\_op] = s\_comma$
      then
      {  $checkp(tree[ax + t\_op1], tree[fx + t\_op1])$
      ;  $fx := tree[fx + t\_op2]$
      ;  $ax := tree[ax + t\_op2]$
      }
      else
         $cmperror$("$parameter\ \ mismatch$")
;  $checkp(ax, fx)$
}

proc $checkp$(val $a$, val $f$) is
   if $tree[f + t\_op] = s\_null$
   then
      skip
   else
   if $tree[f + t\_op] = s\_val$
   then
      skip
   else
   if $tree[f + t\_op] = s\_array$
   then
      skip
   else
   if $tree[f + t\_op] = s\_proc$
   then
      skip
   else
      skip
```

```
func $containscall$(val $x$) is
var $op$;
{  $op := tree[x + t\_op]$
;  if $op = s\_null$
   then
      return 0
   else
   if $monadic(op)$
   then
      return $containscall(tree[x + t\_op1])$
   else
   if $diadic(op)$
   then
      return $containscall(tree[x + t\_op1]) \lor containscall(tree[x + t\_op2])$
   else
      return $op = s\_fncall$
}

func $iszero$(val $x$) is
   return $isval(x) \land (getval(x) = 0)$

func $immop$(val $x$) is
var $value$;
{  $value := getval(x)$
;  return $isval(x) \land (value > (-65536)) \land (value < 65536)$
}

func $needsareg$(val $x$) is
var $op$;
{  $op := tree[x + t\_op]$
;  return $\sim (isval(x) \lor (op = s\_string) \lor (op = s\_name))$
}
```

```
func regsfor(val x) is
var op;
var rleft;
var rright;
{  op := tree[x + t_op]
;  if op = s_fncall
   then
      return 10
   else
   if monadic(op)
   then
      return regsfor(tree[x + t_op1])
   else
   if diadic(op)
   then
   {  rleft := regsfor(tree[x + t_op1])
   ;  rright := regsfor(tree[x + t_op2])
   ;  if rleft = rright
      then
         return 1 + rleft
      else
      if rleft > rright
      then
         return rleft
      else
         return rright
   }
   else
      return 1
}

proc loadbase(val reg, val base) is
var name;
var def;
   if isval(base)
   then
      loadconst(reg, getval(base))
   else
   {  name := findname(base)
   ;  def := names_d[name]
   ;  if tree[def + t_op] = s_array
      then
         loadvar(reg, name)
      else
         namemessage("array expected", tree[def + t_op1])
   }
```

```
proc genassign(val left, val right) is
var sp;
var leftop;
var name;
var base;
var offset;
var value;
{  leftop := tree[left + t_op]
;  if leftop = s_name
   then
   {  name := findname(left)
   ;  texp(right)
   ;  storevar(name)
   }
   else
   {  base := tree[left + t_op1]
   ;  offset := tree[left + t_op2]
   ;  if isval(offset)
      then
      {  value := getval(offset)
      ;  texp(right)
      ;  loadbase(r_breg, base)
      ;  geni(i_stai, value)
      }
      else
      {  sp := stackp
      ;  texp(offset)
      ;  loadbase(r_breg, base)
      ;  geni(i_opr, o_add)
      ;  stackp := stackp + 1
      ;  setstack()
      ;  geni(i_ldbm, m_sp)
      ;  gensref(i_stai, sp)
      ;  texp(right)
      ;  geni(i_ldbm, m_sp)
      ;  gensref(i_ldbi, sp)
      ;  geni(i_stai, 0)
      ;  stackp := sp
      }
   }
}
```

```
proc $texp$(val $x$) is
var $op$;
var $left$;
var $right$;
var $offs$;
var $value$;
var $def$;
var $sp$;
{  $op := tree[x + t\_op]$
;  if $isval(x)$
   then
   {  $value := getval(x)$
   ;  $loadconst(r\_areg,\ value)$
   }
   else
   if $op = s\_string$
   then
      $genstring(x)$
   else
   if $op = s\_name$
   then
   {  $left := findname(x)$
   ;  $def := names\_d[left]$
   ;  if $tree[def + t\_op] = s\_val$
      then
         $loadconst(r\_areg,\ names\_v[left])$
      else
      if $tree[def + t\_op] = s\_var$
      then
         $loadvar(r\_areg,\ left)$
      else
         skip
   }
   else
   if $(op = s\_not) \vee (op = s\_and) \vee (op = s\_or) \vee (op = s\_eq) \vee (op = s\_ls)$
   then
      $tbool(x,$ true$)$
   else
   if $op = s\_sub$
   then
   {  $left := tree[x + t\_op1]$
   ;  $def := names\_d[left]$
   ;  if $isval(tree[x + t\_op2])$
      then
      {  $loadbase(r\_areg,\ left)$
      ;  $value := getval(tree[x + t\_op2])$
      ;  $geni(i\_ldai,\ value)$
      }
```

```
        else
        {  $texp(tree[x + t\_op2])$
        ;  $loadbase(r\_breg, left)$
        ;  $geni(i\_opr, o\_add)$
        ;  $geni(i\_ldai, 0)$
        }
    }
    else
    if $op = s\_fncall$
    then
        $tcall(x$, true, 0, false)
    else
        $texp2(op, tree[x + t\_op1], tree[x + t\_op2])$
}
```

```
proc texp2(val op, val op1, val op2) is
var left;
var right;
var sp;
{  left := op1
;  right := op2
;  if (op = s_plus) ∧ (regsfor(left) < regsfor(right))
   then
   {  left := op2
   ;  right := op1
   }
   else
      skip
;  if needsareg(right)
   then
   {  sp := stackp
   ;  texp(right)
   ;  stackp := stackp + 1
   ;  setstack()
   ;  geni(i_ldbm, m_sp)
   ;  gensref(i_stai, sp)
   ;  texp(left)
   ;  geni(i_ldbm, m_sp)
   ;  gensref(i_ldbi, sp)
   ;  stackp := sp
   }
   else
   {  texp(left)
   ;  tbexp(right)
   }
;  if op = s_plus
   then
      geni(i_opr, o_add)
   else
   if op = s_minus
   then
      geni(i_opr, o_sub)
   else
      skip
}
```

```
proc tbexp(val x) is
var op;
var left;
var value;
var def;
{  op := tree[x + t_op]
;  if isval(x)
   then
   {  value := getval(x)
   ;  loadconst(r_breg, value)
   }
   else
   if op = s_string
   then
      genstring(x)
   else
   if op = s_name
   then
   {  left := findname(x)
   ;  def := names_d[left]
   ;  if tree[def + t_op] = s_val
      then
         loadconst(r_breg, names_v[left])
      else
      if tree[def + t_op] = s_var
      then
         loadvar(r_breg, left)
      else
         skip
   }
   else
      skip
}

proc stk_init(val n) is
{  stackp := n
;  stk_max := n
}

proc setstack() is
   if stk_max < stackp
   then
      stk_max := stackp
   else
      skip
```

```
proc loadconst(val reg, val value) is
    if $(value > (-65536)) \wedge (value < 65536)$
    then
        if reg = r_areg
        then
            geni(i_ldac, value)
        else
            geni(i_ldbc, value)
    else
        gen(cbf_const, reg, genconst(value))

proc loadvar(val reg, val vn) is
var offs;
{   offs := names_v[vn]
;   if islocal(vn)
    then
        if reg = r_areg
        then
        {   geni(i_ldam, m_sp)
        ;   gensref(i_ldai, offs)
        }
        else
        {   geni(i_ldbm, m_sp)
        ;   gensref(i_ldbi, offs)
        }
    else
    if reg = r_areg
    then
        geni(i_ldam, offs)
    else
        geni(i_ldbm, offs)
}

proc storevar(val vn) is
var offs;
{   offs := names_v[vn]
;   if islocal(vn)
    then
    {   geni(i_ldbm, m_sp)
    ;   gensref(i_stai, offs)
    }
    else
        geni(i_stam, offs)
}
```

```
func $monadic$(val $op$) is
    return $(op\ =\ s\_not)\ \vee\ (op\ =\ s\_neg)$

func $diadic$(val $op$) is
    return $div(op,\ s\_diadic)\ \neq\ 0$

proc $geni$(val $i$, val $opd$) is
    $gen(cbf\_inst,\ i,\ opd)$

proc $genref$(val $inst$, val $lab$) is
    if $labval[lab]\ =\ 0$
    then
        $gen(cbf\_fwdref,\ inst,\ lab)$
    else
        $gen(cbf\_bwdref,\ inst,\ lab)$

proc $gensref$(val $i$, val $offs$) is
    $gen(cbf\_stack,\ i,\ offs)$

proc $genbr$(val $seq$, val $lab$) is
    if $seq$
    then
        skip
    else
        $genref(i\_br,\ lab)$
```

```
func $genconst$(val $n$) is
var $i$;
var $cp$;
var $found$;
{  $found$ := false
;  $i$ := 0
;  while ($i < constp$) $\wedge$ ($found$ = false) do
      if $consts[i] = n$
      then
      {  $found$ := true
      ;  $cp := i$
      }
      else
         $i := i + 1$
;  if $found$
   then
      skip
   else
   {  $consts[constp] := n$
   ;  $cp := constp$
   ;  $constp := constp + 1$
   }
;  return $cp$
}

proc $genstring$(val $x$) is
var $i$;
var $sp$;
{  $sp := stringp$
;  $i$ := 0
;  while $i \leq div(rem(tree[x + 1], 256), 4)$ do
   {  $strings[stringp] := tree[x + i + 1]$
   ;  $stringp := stringp + 1$
   ;  $i := i + 1$
   }
;  $gen(cbf\_string, 0, sp)$
}
```

```
proc gen(val t, val h, val l) is
{  cb_loadpoint := cb_loadpoint + 1
;  codebuffer[cb_bufferp] := mul2(t, cb_flag) + mul2(h, cb_high) + l + 65536
;  cb_bufferp := cb_bufferp + 1
;  if cb_bufferp = cb_size
   then
      generror("code buffer overflow")
   else
      skip
}

proc initlabels() is
var l;
{  l := 0
;  while l < labval_size do
   {  labval[l] := 0
   ;  l := l + 1
   }
}

func getlabel() is
{  if labelcount < labval_size
   then
      labelcount := labelcount + 1
   else
      generror("too many labels")
;  return labelcount
}

proc setlab(val l) is
{  labval[l] := cb_loadpoint
;  gen(cbf_lab, 0, l)
}

proc genentry() is
{  cb_entryinstp := cb_bufferp
;  gen(cbf_entry, 0, 0)
}
```

```
proc genexit() is
{  cb_setlow(cb_entryinstp, stk_max)
;  if tree[procdef + t_op] = s_proc
   then
      gen(cbf_pexit, 0, 0)
   else
      gen(cbf_fnexit, 0, 0)
}

proc initbuffer() is
{  cb_loadpoint := 0
;  constp := 0
;  stringp := 0
;  cb_bufferp := 0
}

proc cb_unpack(val p) is
var x;
{  x := codebuffer[p]
;  cbv_flag := div(x, cb_flag)
;  x := rem(x, cb_flag)
;  cbv_high := div(x, cb_high)
;  x := rem(x, cb_high) − 65536
;  cbv_low := x
}

proc cb_setlow(val p, val f) is
var t;
{  t := div(codebuffer[p], cb_high)
;  codebuffer[p] := mul2(t, cb_high) + f + 65536
}
```

```
func $instlength$(val $opd$) is
var $v$;
var $n$;
{  if ($opd \ \geq\ 0$) $\wedge$ ($opd\ <\ 16$)
   then
      $n\ :=\ 1$
   else
   {  $n\ :=\ 8$
   ;  if $opd\ <\ 0$
      then
      {  $v\ :=\ mul2(div(opd,\ 256),\ 256)$
      ;  while $div(v,\ 10000000_{16})\ =\ \mathrm{F}_{16}$ do
         {  $v\ :=\ mul2(v,\ 16)$
         ;  $n\ :=\ n\ -\ 1$
         }
      }
      else
      {  $v\ :=\ opd$
      ;  while $div(v,\ 10000000_{16})\ =\ 0$ do
         {  $v\ :=\ mul2(v,\ 16)$
         ;  $n\ :=\ n\ -\ 1$
         }
      }
   }
;  return $n$
}

func $cb\_laboffset$(val $p$) is
   return $labval[cbv\_low]\ -\ (cb\_loadpoint\ +\ cb\_reflength(p))$

func $cb\_reflength$(val $p$) is
var $ilen$;
var $labaddr$;
{  $ilen\ :=\ 1$
;  $labaddr\ :=\ labval[cbv\_low]$
;  while $ilen\ <\ instlength(labaddr\ -\ (cb\_loadpoint\ +\ ilen))$ do
      $ilen\ :=\ ilen\ +\ 1$
;  return $ilen$
}
```

```
func $cb\_stackoffset$(val $p$, val $stksize$) is
var $offs$;
{  $offs := cbv\_low$
;  if ($offs - pflag) < 0$
    then
        return $stksize - offs$
    else
        return $stksize + (offs - pflag)$
}
```

```
proc expand() is
var bufferp;
var offset;
var stksize;
var flag;
{ bufferp := 0
; while bufferp < cb_bufferp do
    { cb_unpack(bufferp)
    ; flag := cbv_flag
    ; if flag = cbf_constp
        then
        { cb_conststart := div(cb_loadpoint, 4)
        ; cb_stringstart := cb_conststart + constp
        ; cb_loadpoint := cb_loadpoint + mul2(constp + stringp, 4)
        }
        else
        if flag = cbf_entry
        then
        { stksize := cbv_low
        ; cb_loadpoint := cb_loadpoint + instlength(− stksize) + 4
        }
        else
        if flag = cbf_pexit
        then
            cb_loadpoint := cb_loadpoint + instlength(stksize) + 5
        else
        if flag = cbf_fnexit
        then
            cb_loadpoint := cb_loadpoint + instlength(stksize) + instlength(stksize + 1) + 5
        else
        if flag = cbf_inst
        then
            cb_loadpoint := cb_loadpoint + instlength(cbv_low)
        else
        if flag = cbf_stack
        then
        { offset := cb_stackoffset(bufferp, stksize)
        ; cb_loadpoint := cb_loadpoint + instlength(offset)
        }
        else
        if flag = cbf_lab
        then
            labval[cbv_low] := cb_loadpoint
        else
        if flag = cbf_bwdref
        then
            cb_loadpoint := cb_loadpoint + cb_reflength(bufferp)
        else
        if flag = cbf_fwdref
        then
        {   offset := cb_laboffset(bufferp)
        ;   if offset > 0
            then
                cb_loadpoint := cb_loadpoint + cb_reflength(bufferp)
            else
                cb_loadpoint := cb_loadpoint + 1
        }
        else
        if flag = cbf_const
        then
        {   offset := cbv_low + cb_conststart
        ;   cb_loadpoint := cb_loadpoint + instlength(offset)
        }
        else
        if flag = cbf_string
        then
        {   offset := cbv_low + cb_stringstart
        ;   cb_loadpoint := cb_loadpoint + instlength(offset)
        }
        else
        if flag = cbf_var
        then
            cb_loadpoint := cb_loadpoint + 4
        else
        {   cmperror(“code buffer error ”)
        ;   printn(bufferp)
        ;   newline()
        }
    ;   bufferp := bufferp + 1
    }
}
```

```
proc flushbuffer() is
var bufferp;
var last;
var offset;
var stksize;
var flag;
var loadstart;
{ loadstart := mul2(m_sp, 4)
; cb_loadpoint := loadstart
; last := 0
; expand()
; while cb_loadpoint ≠ last do
    { last := cb_loadpoint
    ; cb_loadpoint := loadstart
    ; expand()
    }
; codesize := cb_loadpoint
; outhdr()
; bufferp := 0
; cb_loadpoint := loadstart
; while bufferp < cb_bufferp do
    { cb_unpack(bufferp)
    ; flag := cbv_flag
    ; if flag = cbf_constp
        then
        { cb_conststart := div(cb_loadpoint, 4)
        ; cb_stringstart := cb_conststart + constp
        ; cb_loadpoint := cb_loadpoint + mul2(constp + stringp, 4)
        ; outconsts()
        ; outstrings()
        }
        else
        if flag = cbf_entry
        then
        { stksize := cbv_low
        ; outinst(i_ldbm, m_sp)
        ; outinst(i_stai, 0)
        ; outinst(i_ldac, − stksize)
        ; outinst(i_opr, o_add)
        ; outinst(i_stam, m_sp)
        ; cb_loadpoint := cb_loadpoint + instlength(− stksize) + 4
        }
        else
        if flag = cbf_pexit
        then
        { outinst(i_ldbm, m_sp)
        ; outinst(i_ldac, stksize)
        ; outinst(i_opr, o_add)
;  $outinst(i\_stam,\ m\_sp)$
;  $outinst(i\_ldbi,\ stksize)$
;  $outinst(i\_opr,\ o\_brb)$
;  $cb\_loadpoint\ :=\ cb\_loadpoint\ +\ instlength(stksize)\ +\ 5$
}
else
if $flag\ =\ cbf\_fnexit$
then
{  $outinst(i\_ldbm,\ m\_sp)$
;  $outinst(i\_stai,\ stksize\ +\ 1)$
;  $outinst(i\_ldac,\ stksize)$
;  $outinst(i\_opr,\ o\_add)$
;  $outinst(i\_stam,\ m\_sp)$
;  $outinst(i\_ldbi,\ stksize)$
;  $outinst(i\_opr,\ o\_brb)$
;  $cb\_loadpoint\ :=\ cb\_loadpoint\ +\ instlength(stksize)\ +\ instlength(stksize\ +\ 1)\ +\ 5$
}
else
if $flag\ =\ cbf\_inst$
then
{  $outinst(cbv\_high,\ cbv\_low)$
;  $cb\_loadpoint\ :=\ cb\_loadpoint\ +\ instlength(cbv\_low)$
}
else
if $flag\ =\ cbf\_stack$
then
{  $offset\ :=\ cb\_stackoffset(bufferp,\ stksize)$
;  $outinst(cbv\_high,\ offset)$
;  $cb\_loadpoint\ :=\ cb\_loadpoint\ +\ instlength(offset)$
}
else
if $flag\ =\ cbf\_lab$
then
   skip
else
if $(flag\ =\ cbf\_bwdref)\ \vee\ (flag\ =\ cbf\_fwdref)$
then
{  $offset\ :=\ cb\_laboffset(bufferp)$
;  if $cb\_reflength(bufferp)\ >\ instlength(offset)$
   then
      $out1(i\_pfix,\ 0)$
   else
      skip
;  $outinst(cbv\_high,\ offset)$
;  $cb\_loadpoint\ :=\ cb\_loadpoint\ +\ cb\_reflength(bufferp)$
}
else
if $flag\ =\ cbf\_const$
then
```

```
        {  $offset := cbv\_low + cb\_conststart$
        ;  if $cbv\_high = r\_areg$
           then
              $outinst(i\_ldam, offset)$
           else
              $outinst(i\_ldbm, offset)$
        ;  $cb\_loadpoint := cb\_loadpoint + instlength(offset)$
        }
        else
        if $flag = cbf\_string$
        then
        {  $offset := cbv\_low + cb\_stringstart$
        ;  $outinst(i\_ldac, offset)$
        ;  $cb\_loadpoint := cb\_loadpoint + instlength(offset)$
        }
        else
        if $flag = cbf\_var$
        then
        {  $outvar(cbv\_low)$
        ;  $cb\_loadpoint := cb\_loadpoint + 4$
        }
        else
           skip
     ;  $bufferp := bufferp + 1$
     }
}
```

```
proc $outinst$(val $inst$, val $opd$) is
var $v$;
var $n$;
    if $(opd \geq 0) \wedge (opd < 16)$
    then
        $out1(inst, opd)$
    else
    {   $n := 28$
    ;   if $opd < 0$
        then
        {   $v := mul2(div(opd, 256), 256)$
        ;   while $div(v, 10000000_{16}) = \mathrm{F}_{16}$ do
            {   $v := mul2(v, 16)$
            ;   $n := n - 4$
            }
        ;   $out1(i\_nfix, div(opd, exp2(n)))$
        ;   $n := n - 4$
        }
        else
        {   $v := opd$
        ;   while $div(v, 10000000_{16}) = 0$ do
            {   $v := mul2(v, 16)$
            ;   $n := n - 4$
            }
        }
    ;   while $n > 0$ do
        {   $out1(i\_pfix, div(opd, exp2(n)))$
        ;   $n := n - 4$
        }
    ;   $out1(inst, opd)$
    }

proc $outconsts()$ is
var $count$;
{   $count := 0$
;   while $count < constp$ do
    {   $outword(consts[count])$
    ;   $count := count + 1$
    }
}
```

```
proc outstrings() is
var count;
{  count := 0
;  while count < stringp do
     {  outword(strings[count])
     ;  count := count + 1
     }
}

proc outvar(val d) is
   outword(d)

proc outword(val w) is
{  outbin(w)
;  outbin(div(w, $100_{16}$))
;  outbin(div(w, $10000_{16}$))
;  outbin(div(w, $1000000_{16}$))
}

proc out1(val inst, val opd) is
   outbin(mul2(inst, 16) + rem(opd, 16))

proc outbin(val d) is
{  selectoutput(binstream)
;  putval(rem(d, 256))
;  selectoutput(messagestream)
}

proc outhdr() is
var w;
var entrypoint;
var offset;
{  w := div(cb_loadpoint + 3, 4)
;  entrypoint := labval[entrylab]
;  outword(w)
;  offset := entrypoint − 4
;  out1(i_pfix, div(offset, $1000_{16}$))
;  out1(i_pfix, div(offset, $100_{16}$))
;  out1(i_pfix, div(offset, $10_{16}$))
;  out1(i_br, offset)
}
```