

```

val put           = 1;
val get          = 2;

val instream     = 0;
val messagestream = 0;
val binstream    = 2 << 8;

val EOF          = 255;

val t_op         = 0;
val t_op1       = 1;
val t_op2       = 2;
val t_op3       = 3;

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_mult         = s_diadic + 2;
val s_or           = s_diadic + 5;
val s_and          = s_diadic + 6;
val s_xor          = s_diadic + 7;
val s_lshift       = s_diadic + 8;
val s_rshift       = s_diadic + 9;

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;
val s_lsub         = s_diadic + 17;

val i_asr3i        = #2;
val i_ldr3i        = #D;
val i_lsl3i        = #0;
val i_lsr3i        = #1;
val i_str3i        = #C;

val i_add3         = #C;
val i_ldr3         = #2C;
val i_str3         = #28;
val i_sub3         = #D;

val i_add2i        = #6;
val i_addpci       = #14;
val i_addspi       = #15;
val i_ldrpci       = #11;
val i_ldrspci      = #13;
val i_mov2i        = #4;
val i_strspi       = #12;
val i_sub2i        = #7;

val i_setsp        = #8D;

```

```

val i_incspl           = #160;
val i_decspi          = #161;

val i_and2            = #0;
val i_eor2           = #1;
val i_lsl2           = #2;
val i_lsr2           = #3;
val i_mul2           = #D;
val i_mvn2          = #F;
val i_neg2           = #9;
val i_orr2           = #C;

val i_bc              = #D;
val i_beq            = (i_bc << 4) or #0;
val i_bne            = (i_bc << 4) or #1;
val i_blt            = (i_bc << 4) or #D;
val i_bge            = (i_bc << 4) or #A;

val i_bu              = #1C;
val i_bl1            = #1E;
val i_bl2            = #1F;

val i_bl              = #8F;

val i_pushl          = #B5;
val i_popl           = #BD;

val i_svc            = #DF;

val bytesperword     = 4;

val linemax          = 200;
val nametablesize    = 128;
array nametable[nametablesize];
val nil               = 0;
val hashmask         = 127;

var ostream;

val treemax          = 30000;
array tree[treemax];
var treep;
var namenode;
var nullnode;
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;

array names_d[500];
array names_v[500];
var namep;
var nameb;

array consts[500];
var constp;
var constb;

array strings[1000];
var stringp;
var stringb;

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

var stackp;
var stk_max;

val labval_size      = 1000;
array labval[labval_size];
var labelcount;

val cb_size         = 10000;

val cbf_inst        = 1;
val cbf_blkstrt     = 2;
val cbf_blkend      = 3;
val cbf_cbranch     = 4;
val cbf_branch      = 5;
val cbf_call        = 6;
val cbf_lab         = 7;
val cbf_prog        = 8;
val cbf_const       = 9;
val cbf_string      = 10;

```

```
val cbf_entry          = 11;  
val cbf_exit          = 12;  
val cbf_tcall         = 13;
```

```
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 inblock;
```

```
val maxaddr          = 300000;
```

```
proc main() is
var t;
{ selectoutput(messagestream)
; t := formtree()
; prints("tree size : ")
; printn(treep)
; newline()
; translate(t)
; prints("program size : ")
; printn(codesize × 2)
; newline()
; prints("size : ")
; printn((codesize × 2) + (arrayspace × 4))
; newline()
}
```

```
proc selectoutput(val c) is
  ostream := c
```

```
proc putval(val c) is
  put(c, ostream)
```

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

```

func div(val n, val m) is
  var i;
  var j;
  var b;
  var r;
  { i := m
  ; j := n
  ; b := 1
  ; r := 0
  ; while i < n do
    { i := i << 1
    ; b := b << 1
    }
  ; while b > 0 do
    { if j ≥ i
      then
        { r := r or b
        ; j := j - i
        }
      else
        skip
    ; i := i >> 1
    ; b := b >> 1
    }
  ; return r
  }

```

```

func rem(val n, val m) is
  return n - (div(n, m) × m)

```

```

proc prints(array s) is
var n;
var p;
var w;
var l;
var b;
{ n := 1
; p := 0
; w := s[p]
; l := w and 255
; w := w >> 8
; b := 1
; while n ≤ l do
{ putval(w and 255)
; w := w >> 8
; 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;
{ if n > 15
  then
    printhex(n >> 4)
  else
    skip
; d := n and 15
; 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 < nametables do
  { nametable[i] := nil
; i := i + 1
}
; declsyswords()
; nullnode := cons1(s_null)
; linecount := 0
; rdline()
; rch()
; nextsymbol()
; if (symbol = s_var) or (symbol = s_val) or (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()
}

```

```

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
}
```

```

func lookupword() is
var a;
var hashval;
var i;
var stype;
var found;
var searching;
{ a := wordv[0]
; hashval := (a + (a >> 3) + (wordv[wordsize] << 2)) and hashmask
; namenode := nametable[hashval]
; found := false
; searching := true
; while searching do
    if namenode = nil
    then
    { found := false
; searching := false
}
    else
    { i := 0
; while (i ≤ wordsize) and (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
}

```



```

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 ≤ n do
{ w := w or (s[si] << (b << 3))
; 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 := w and 255
; while vi ≤ n do
{ v[vi] := w and 255
; w := w ≫ 8
; vi := vi + 1
; b := b + 1
; if b = bytesperword
then
{ b := 0
; si := si + 1
; w := s[si]
}
else
skip
}
}

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)
; declare("xor", s_xor)
}

```

```

func getchar() is
    return get(instream)

```

```

proc rdline() is
{ linelength := 1
; linep := 1
; linecount := linecount + 1
; ch := getchar()
; linev[linelength] := ch
; while (ch ≠ '\n') and (ch ≠ EOF) and (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 ≥ ‘A’) and (ch ≤ ‘Z’)) or ((ch ≥ ‘a’) and (ch ≤ ‘z’)) or ((ch ≥ ‘0’) and (ch ≤ ‘9’)) or (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 ≥ base
  then
    cmperror(“error in number”)
  else
    while d < base do
      { numval := (numval × base) + d
      ; rch()
      ; d := value(ch)
      }
}
}

```

```

func value(val c) is
  if (c ≥ '0') and (c ≤ '9')
  then
    return c - '0'
  else
  if (c ≥ 'A') and (c ≤ '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 ≠ ‘\’ 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)
}

```

```

proc nextsymbol() is
{ while (ch = '\n') or (ch = '\r') or (ch = ' ') do
    rch()
; if ch = '|'
    then
    { rch()
    ; while ch ≠ '|' do
        rch()
    ; rch()
    ; nextsymbol()
    }
else
if ((ch ≥ 'A') and (ch ≤ 'Z')) or ((ch ≥ 'a') and (ch ≤ 'z'))
then
{ rdtag()
; symbol := lookupword()
}
else
if (ch ≥ '0') and (ch ≤ '9')
then
{ symbol := s_number
; readnumber(10)
}
else
if ch = '#'
then
{ rch()
; symbol := s_number
; readnumber(16)
}
else
if ch = '['
then
{ rch()
; symbol := s_lbracket
}
else
if ch = ']'
then
{ rch()
; symbol := s_rbracket
}
else
if ch = '('
then
{ rch()
; symbol := s_lparen
}
}

```

```

else
if  $ch = \text{'}'$ 
then
{  $rch()$ 
;  $symbol := s\_rparen$ 
}
else
if  $ch = \text{'{'}$ 
then
{  $rch()$ 
;  $symbol := s\_begin$ 
}
else
if  $ch = \text{'}'$ 
then
{  $rch()$ 
;  $symbol := s\_end$ 
}
else
if  $ch = \text{';'}$ 
then
{  $rch()$ 
;  $symbol := s\_semicolon$ 
}
else
if  $ch = \text{','}$ 
then
{  $rch()$ 
;  $symbol := s\_comma$ 
}
else
if  $ch = \text{'+'}$ 
then
{  $rch()$ 
;  $symbol := s\_plus$ 
}
else
if  $ch = \text{'-'}$ 
then
{  $rch()$ 
;  $symbol := s\_minus$ 
}
else
if  $ch = \text{'*'}$ 
then
{  $rch()$ 
;  $symbol := s\_mult$ 
}
else

```

```

if ch = '='
then
{ rch()
; symbol := s_eq
}
else
if ch = '<'
then
{ rch()
; if ch = '='
  then
    { rch()
    ; symbol := s_le
    }
  else
    if ch = '<'
    then
      { rch()
      ; symbol := s_lshift
      }
    else
      symbol := s_ls
  }
}
else
if ch = '>'
then
{ rch()
; if ch = '='
  then
    { rch()
    ; symbol := s_ge
    }
  else
    if ch = '>'
    then
      { rch()
      ; symbol := s_rshift
      }
    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")
}

```

```
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("name expected")
; 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 := replist()
                  ; 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) or (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 := reexpression()
; 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 (symbol and s_diacic) ≠ 0
  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) and (symbol = s)
then
{ nextsymbol()
; return cons3(s, b, rright(s))
}
else
return b
}

```

```

func associative(val s) is
return (s = s_and) or (s = s_or) or (s = s_xor) or (s = s_plus) or (s = s_mult)

```

```
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
{ if tree[a + t_op] = s_sub
then
tree[a + t_op] := s_lsub
else
skip
; 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) or (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) or (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) or (symbol = s_array) or (symbol = s_proc)
  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) or (symbol = s_var) or (symbol = s_array)
    then
      return cons3(s_semicolon, a, rgdecls())
    else
      return a
  }
}

```

```

func rldecls() is
var a;
{ a := rdecl()
; if (symbol = s_val) or (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 := w and 255
      ; w := w >> 8
      ; b := 1
      ; while n ≤ l do
        { putval(w and 255)
          ; w := w >> 8
          ; 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])
}

```

```

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)
        ; stackp := stackp + 1
      }
}
}

```

```

proc tformals(val x) is
var op;
{ op := tree[x + t.op]
; if op = s_null
  then
    skip
  else
    if op = s_comma
    then
      { tformals(tree[x + t.op1])
      ; tformals(tree[x + t.op2])
      }
    else
      { gen2i(i_strspi, stackp, stackp)
      ; 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 << 2)
; loadconst(7, arraybase - (stackp << 2))
; gensetsp(7)
; while g < namep do
  { name := names_d[g]
  ; if tree[name + t_op] = s_array
    then
      { loadconst(0, arraybase)
      ; gen2i(i_strspi, 0, names_v[g])
      ; arraybase := arraybase + (getval(tree[name + t_op2]) << 2)
      }
    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
              genererror("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) and (n ≥ 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 ≥ nameb

```

```

proc optimise(val x) is
var op;
{ op := tree[x + t_op]
; if (op = s_skip) or (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;
{ op := tree[x + t_op]
; if op = s_name
  then
    { name := findname(x)
    ; if tree[names_d[name] + t_op] = s_val
      then
        return tree[names_d[name] + t_op2]
      else
        return x
    }
  else
    if monadic(op)
    then
      { r := optimiseexpr(tree[x + t_op1])
      ; if isval(r)
        then
          return cons2(s_number, evalmonadic(cons2(op, r)))
        else
          return r
      }
    else
      if op = s_fncall
      then
        return cons3(s_fncall, optimiseexpr(tree[x + t_op1]), optimiseexpr(tree[x + t_op2]))
      else
        if diadic(op)
        then
          { left := optimiseexpr(tree[x + t_op1])
          ; right := optimiseexpr(tree[x + t_op2])
          ; leftop := tree[left + t_op]
          ; rightop := tree[right + t_op]
          ; if (op = s_sub) or (op = s_lsub)
            then
              return cons3(op, left, right)
            else
              if isval(left) and isval(right)
              then
                return cons2(s_number, evaldiadic(cons3(op, left, right)))
              else

```

```

if op = s_eq
then
  if (leftop = s_not) and (rightop = s_not)
  then
    return cons3(s_eq, tree[left + t_op1], tree[right + t_op1])
  else
    return cons3(s_eq, left, right)
else
if op = s_ne
then
  if (leftop = s_not) and (rightop = s_not)
  then
    return cons2(s_not, cons3(s_eq, tree[left + t_op1], tree[right + t_op1]))
  else
    return cons2(s_not, cons3(s_eq, left, right))
else
if op = s_ge
then
  return cons2(s_not, cons3(s_ls, left, right))
else
if op = s_gr
then
  return cons3(s_ls, right, left)
else
if op = s_le
then
  return cons2(s_not, cons3(s_ls, right, left))
else
if (op = s_or) or (op = s_and)
then
  if (leftop = s_not) and (rightop = s_not)
  then
    if op = s_and
    then
      return cons2(s_not, cons3(s_or, tree[left + t_op1], tree[right + t_op1]))
    else
      return cons2(s_not, cons3(s_and, tree[left + t_op1], tree[right + t_op1]))
  else
    return cons3(op, left, right)
else
if op = s_xor
then
  if (leftop = s_not) and (rightop = s_not)
  then
    return cons3(s_xor, tree[left + t_op1], tree[right + t_op1])
  else
    return cons3(s_xor, left, right)
else
if ((op = s_plus) or (op = s_or) or (op = s_xor)) and (iszero(left) or iszero(right))

```

```

then
  if iszero(left)
  then
    return right
  else
    if iszero(right)
    then
      return left
    else
      return cons3(op, left, right)
else
  if ((op = s_minus) or (op = s_lshift) or (op = s_rshift)) and iszero(right)
  then
    return left
  else
    return cons3(op, left, right)
}
else
  if op = s_comma
  then
    return cons3(s_comma, optimiseexpr(tree[x + t_op1]), optimiseexpr(tree[x + t_op2]))
  else
    return x
}

```

```

func isval(val x) is
var op;
{ op := tree[x + t_op]
; return (op = s_true) or (op = s_false) or (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 ~ 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_mult
then
return left × right
else
if op = s_eq
then
return left = right
else
if op = s_ne
then
return left ≠ 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 ≤ right
else
if op = s_ge
then
return left ≥ right
else
if op = s_or
then
return left or right
else
if op = s_and

```

```
then
  return left and right
else
if op = s_xor
then
  return left  $\neq$  right
else
if op = s_lshift
then
  return left  $\ll$  right
else
if op = s_rshift
then
  return left  $\gg$  right
else
{ cmperror("optimise error")
; return 0
}
}
```



```

proc translate(val t) is
var s;
var dlab;
var mainlab;
{ namep := 0
; nameb := 0
; labelcount := 1
; initbuffer()
; arrayspace := 0
; stk_init()
; declglobals(tree[t + t_op1])
; tglobals(tree[t + t_op1])
; declprocs(tree[t + t_op2])
; nameb := namep
; mainlab := getlabel()
; gencall(mainlab, true, 0, false)
; gen1i(i_svc, 0)
; setlab(mainlab)
; genprocs(tree[t + t_op2])
; endblock(0)
; 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]
      ; body := tree[x + t_op3]
      ; stk_init()
      ; declformals(tree[x + t_op2])
      ; genentry()
      ; stackp := 0
      ; tformals(tree[x + t_op2])
      ; 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 (tree[procdef + t_op] = s_func) and 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) and (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 ( $\sim$  funtail(tail)) and ((tree[thenpart + t_op] = s_skip) or (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(0, tree[x + t_op1])
      ; genbr(seq, clab)
    }
}
else
  generror("\return\ expected")
else
if (op = s_while) and (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
  genli(i_svc, 0)
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 gencondjump(val x, val f, val label) is
var cond;
var op;
var cx;
{ cx := x
; cond := f
; op := tree[x + t_op]
; if op = s_not
then
{ cond := ~ cond
; cx := tree[x + t_op1]
}
else
skip
; if tree[cx + t_op] = s_ls
then
{ texp2(s_minus, 0, tree[cx + t_op1], tree[cx + t_op2])
; if cond
then
gencbr(i_blt, label)
else
gencbr(i_bge, label)
}
else
{ if tree[cx + t_op] = s_eq
then
{ if iszero(tree[cx + t_op1])
then
tbexp(0, tree[cx + t_op2])
else
if iszero(tree[cx + t_op2])
then
tbexp(0, tree[cx + t_op1])
else
texp2(s_minus, 0, tree[cx + t_op1], tree[cx + t_op2])
; cond := ~ cond
}
else
tbexp(0, cx)
; if cond
then
gencbr(i_bne, label)
else
gencbr(i_beq, label)
}
}
}

```

```

proc tcall(val x, val seq, val clab, val tail) is
var sp;
var entry;
var def;
{ sp := stackp
; prepareaps(tree[x + t_op2])
; setstack()
; stackp := sp
; loadaps(tree[x + t_op2])
; if isval(tree[x + t_op1])
  then
    gen1i(i_svc, getval(tree[x + t_op1]))
  else
    { entry := findname(tree[x + t_op1])
    ; def := names.d[entry]
    ; if islocal(entry)
      then
        { gen2i(i_ldrspi, 6, names.v[entry])
        ; gen1(i_bl, 6)
        }
      else
        { checkps(tree[def + t_op2], tree[x + t_op2])
        ; gencall(names.v[entry], seq, clab, tail)
        }
    }
; stackp := sp
}

```

```

proc prepareaps(val aplist) is
var x;
var reg;
{ x := aplist
; reg := 0
; while tree[x + t_op] = s_comma do
  { prepareap(reg, tree[x + t_op1])
  ; x := tree[x + t_op2]
  ; reg := reg + 1
  }
; prepareap(reg, x)
}

```

```

proc prepareap(val reg, val x) is
var op;
var vn;
{ op := tree[x + t_op]
; if op = s_null
  then
    skip
  else
    if (reg > 0) and (containscall(x) or (regsfor(x) > (7 - reg)))
    then
      { texp(0, x)
      ; gen2i(i_strspi, 0, stackp)
      ; stackp := stackp + 1
      }
    else
      skip
  }
}

```

```

proc loadaps(val aplist) is
var x;
var reg;
{ x := aplist
; reg := 0
; while tree[x + t_op] = s_comma do
  { loadap(reg, tree[x + t_op1])
  ; x := tree[x + t_op2]
  ; reg := reg + 1
  }
; loadap(reg, x)
}

```



```

proc loadap(val reg, val x) is
var op;
var vn;
var aptype;
{ op := tree[x + t_op]
; if op = s_null
then
skip
else
if op = s_name
then
{ vn := findname(x)
; aptype := tree[names_d[vn] + t_op]
; if aptype = s_val
then
loadconst(reg, names_v[vn])
else
if aptype = s_proc
then
loadproc(reg, vn)
else
loadvar(reg, vn)
}
else
if (reg > 0) and (containscall(x) or (regsfor(x) > (7 - reg)))
then
{ gen2i(i_ldrsp, reg, stackp)
; stackp := stackp + 1
}
else
texp(reg, x)
}
}

```

```

proc checkps(val alist, val flist) is
  var ax;
  var fx;
  { 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 iszero(val x) is
  return isval(x) and (getval(x) = 0)

```

```

func immop5(val x) is
  var value;
  { value := getval(x)
    ; return isval(x) and (value ≥ 0) and (value < 32)
  }

```

```

func immop8(val x) is
var value;
{ value := getval(x)
; return isval(x) and (value ≥ 0) and (value < 256)
}

```

```

func regsfor(val x) is
var op;
var rleft;
var rright;
{ op := tree[x + t_op]
; 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
}

```

```

func containscall(val x) is
var op;
{ op := tree[x + t_op]
; if monadic(op)
  then
    return containscall(tree[x + t_op1])
  else
    if diadic(op) or (op = s_comma)
      then
        return containscall(tree[x + t_op1]) or containscall(tree[x + t_op2])
      else
        if op = s_fncall
          then
            return true
          else
            return false
        }
}

```

```

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(0, right)
    ; storevar(0, name)
    }
  else
    { base := tree[left + t_op1]
    ; offset := tree[left + t_op2]
    ; if isval(offset)
      then
        { value := getval(offset)
        ; if value < 32
          then
            { texp(0, right)
            ; loadbase(1, base)
            ; gen3i(i_str3i, 0, 1, value)
            }
          else
            { texp(0, right)
            ; texp(1, left)
            ; gen3i(i_str3i, 0, 1, 0)
            }
          }
        }
      }
    }
else
if containscall(right)
then
{ if containscall(offset)
  then
    { sp := stackp
    ; texp(0, offset)
    ; stackp := stackp + 1
    ; setstack()
    ; gen2i(i_strspi, 0, sp)
    ; texp(0, right)
    ; gen2i(i_ldrspi, 1, sp)
    ; stackp := sp
    }
  else
    { texp(0, right)

```

```

    ; txp(1, offset)
  }
; subassign(0, base, 1)
}
else
if containscall(offset)
then
{ txp(0, offset)
; txp(1, right)
; subassign(1, base, 0)
}
else
if regsfor(right) > regsfor(offset)
then
{ txp(0, right)
; txp(1, offset)
; subassign(0, base, 1)
}
else
{ txp(0, offset)
; txp(1, right)
; subassign(1, base, 0)
}
}
}

```

```

proc subassign(val source, val base, val sub) is
{ loadbase(2, base)
; gen3i(i_lsl3i, sub, sub, 2)
; gen3(i_str3, source, 2, sub)
}

```

```

proc texp(val reg, val x) is
var op;
var name;
{ op := tree[x + t_op]
; texp(reg, x)
; if op = s_name
then
{ name := findname(x)
; if tree[names_d[name] + t_op] = s_var
then
gen2i(i_add2i, reg, 0)
else
skip
}
else
if op = s_sub
then
gen2i(i_add2i, reg, 0)
else
skip
}
}

```

```

proc txp(val reg, val x) is
var op;
var left;
var right;
var offs;
var value;
var def;
{ op := tree[x + t_op]
; if isval(x)
  then
    { value := getval(x)
    ; loadconst(reg, value)
    }
  else
    if op = s_name
    then
      { left := findname(x)
      ; def := names_d[left]
      ; if tree[def + t_op] = s_val
        then
          loadconst(reg, names_v[left])
        else
          if tree[def + t_op] = s_var
          then
            loadvar(reg, left)
          else
            skip
          }
      else
        if op = s_string
        then
          genstring(reg, x)
        else
          if op = s_neg
          then
            { txp(reg, tree[x + t_op1])
            ; gen2(i_neg2, reg, reg)
            }
          else
            if op = s_not
            then
              { left := tree[x + t_op1]
              ; if tree[left + t_op] = s_eq
                then
                  { txp2(s_minus, reg, tree[left + t_op1], tree[left + t_op2])
                  ; gen1i(i_beg, 1)
                  ; gen2i(i_mov2i, reg, 1)
                  ; gen2(i_neg2, reg, reg)
                  }
                }
            }

```



```

    }
    else
    { txp(reg, tree[x + t_op1])
    ; gen2(i_mvn2, reg, reg)
    }
}
else
if (op = s_sub) or (op = s_lsub)
then
{ left := tree[x + t_op1]
; def := names_d[left]
; if isval(tree[x + t_op2])
then
{ loadbase(reg, left)
; value := getval(tree[x + t_op2])
; if value < 32
then
    if op = s_sub
    then
        gen3i(i_ldr3i, reg, reg, value)
    else
        gen2i(i_add2i, reg, value  $\ll$  2)
    else
    { loadconst(reg + 1, value  $\ll$  2)
    ; if op = s_sub
    then
        gen3(i_ldr3, reg, reg, reg + 1)
    else
        gen3(i_add3, reg, reg, reg + 1)
    }
}
}
else
if containscall(tree[x + t_op2])
then
{ txp(0, tree[x + t_op2])
; gen3i(i_lsl3i, 0, 0, 2)
; loadbase(1, left)
; if op = s_sub
then
    gen3(i_ldr3, 0, 1, 0)
else
    gen3(i_add3, 0, 1, 0)
}
else
{ loadbase(reg, left)
; txp(reg + 1, tree[x + t_op2])
; gen3i(i_lsl3i, reg + 1, reg + 1, 2)
; if op = s_sub
then

```

```

        gen3(i_ldr3, reg, reg, reg + 1)
    else
        gen3(i_add3, reg, reg, reg + 1)
    }
}
else
if op = s_fncall
then
    tcall(x, true, 0, false)
else
if op = s_ls
then
{ texp2(s_minus, reg, tree[x + t_op1], tree[x + t_op2])
; gen3i(i_asr3i, reg, reg, 31)
}
else
if op = s_eq
then
{ texp2(s_minus, reg, tree[x + t_op1], tree[x + t_op2])
; gen1i(i_beq, 1)
; gen2i(i_mov2i, reg, 1)
; gen2i(i_sub2i, reg, 1)
}
else
    texp2(op, reg, tree[x + t_op1], tree[x + t_op2])
}

```

```

proc txp2(val op, val reg, val left, val right) is
var sp;
  if (op = s_plus) and immop8(left)
  then
    { txp(reg, right)
    ; genopimm(op, reg, getval(left))
    }
  else
  if ((op = s_plus) or (op = s_minus)) and immop8(right)
  then
    { txp(reg, left)
    ; genopimm(op, reg, getval(right))
    }
  else
  if ((op = s_lshift) or (op = s_rshift)) and immop5(right)
  then
    { txp(reg, left)
    ; genopimm(op, reg, getval(right))
    }
  else
  if containscall(right)
  then
    if containscall(left)
    then
      { sp := stackp
      ; txp(0, right)
      ; stackp := stackp + 1
      ; setstack()
      ; gen2i(i_strspi, 0, sp)
      ; txp(0, left)
      ; gen2i(i_ldrspi, 1, sp)
      ; genop(op, 0, 1)
      ; stackp := sp
      }
    else
    { txp(0, right)
    ; if (op = s_lshift) or (op = s_rshift)
      then
        { gen3i(i_lsl3i, 6, 0, 0)
        ; txp(0, left)
        ; genop(op, 0, 6)
        }
      else
      { txp(1, left)
      ; if op = s_minus
        then
          gen3(i_sub3, 0, 1, 0)
        else

```

```

        genop(op, 0, 1)
    }
}
else
if containscall(left)
then
{ texp(0, left)
; texp(1, right)
; genop(op, 0, 1)
}
else
if (op = s_lshift) or (op = s_rshift)
then
{ texp(reg, left)
; texp(reg + 1, right)
; genop(op, reg, reg + 1)
}
else
if regsfor(left) > regsfor(right)
then
{ texp(reg, left)
; texp(reg + 1, right)
; genop(op, reg, reg + 1)
}
else
{ texp(reg, right)
; texp(reg + 1, left)
; if op = s_minus
then
gen3(i_sub3, reg, reg + 1, reg)
else
genop(op, reg, reg + 1)
}
}

```

```

proc genop(val op, val dreg, val sreg) is
  if op = s_plus
  then
    gen3(i_add3, dreg, dreg, sreg)
  else
    if op = s_minus
    then
      gen3(i_sub3, dreg, dreg, sreg)
    else
      if op = s_mult
      then
        gen2(i_mul2, dreg, sreg)
      else
        if op = s_and
        then
          gen2(i_and2, dreg, sreg)
        else
          if op = s_or
          then
            gen2(i_orr2, dreg, sreg)
          else
            if op = s_xor
            then
              gen2(i_eor2, dreg, sreg)
            else
              if op = s_lshift
              then
                gen2(i_lsl2, dreg, sreg)
              else
                if op = s_rshift
                then
                  gen2(i_lsr2, dreg, sreg)
                else
                  skip

```

```

proc genopimm(val op, val reg, val v) is
  if op = s_plus
  then
    gen2i(i_add2i, reg, v)
  else
    if op = s_minus
    then
      gen2i(i_sub2i, reg, v)
    else
      if op = s_lshift
      then
        gen3i(i_lsl3i, reg, reg, v)
      else
        if op = s_rshift
        then
          gen3i(i_lsr3i, reg, reg, v)
        else
          skip

```

```

proc stk_init() is
{ stackp := 0
; stk_max := 0
}

```

```

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

```

```

proc loadconst(val reg, val value) is
var v;
var shift;
  if value  $\geq$  0
  then
    if value < 256
    then
      gen2i(i_mov2i, reg, value)
    else
      { v := value
      ; shift := 0
      ; while (v and 1) = 0 do
      { v := v  $\gg$  1
      ; shift := shift + 1
      }
      ; if v < 256
      then
        { gen2i(i_mov2i, reg, v)
        ; gen3i(i_lsl3i, reg, reg, shift)
        }
      else
        genconst(reg, value)
      }
    else
      if value > (- 256)
      then
        { gen2i(i_mov2i, reg, - value)
        ; gen2(i_neg2, reg, reg)
        }
      else
        genconst(reg, value)

proc loadproc(val reg, val vn) is
  if islocal(vn)
  then
    loadvar(reg, vn)
  else
    genpref(reg, names_v[vn])

```

```

proc loadvar(val reg, val vn) is
var offs;
{ offs := names_v[vn]
; if islocal(vn)
  then
    gen2i(i_ldrspi, reg, offs)
  else
    if offs < 32
    then
      gen3i(i_ldr3i, reg, 7, offs)
    else
      { loadconst(reg, offs << 2)
      ; gen3(i_ldr3, reg, 7, reg)
      }
}

```

```

proc storevar(val reg, val vn) is
var offs;
{ offs := names_v[vn]
; if islocal(vn)
  then
    gen2i(i_strspi, reg, offs)
  else
    if offs < 32
    then
      gen3i(i_str3i, reg, 7, offs)
    else
      { loadconst(reg + 1, offs << 2)
      ; gen3(i_str3, reg, 7, reg + 1)
      }
}

```

```

func monadic(val op) is
  return (op = s_not) or (op = s_neg)

```

```

func diadic(val op) is
  return (op and s_diadic) ≠ 0

```

```

proc gen3i(val op, val rd, val rm, val imm) is
  geni((op << 11) or (imm << 6) or (rm << 3) or rd)

```

```

proc gen3(val op, val rd, val rm, val rn) is
  geni((op << 9) or (rn << 6) or (rm << 3) or rd)

```



```
proc gen2(val op, val rd, val rm) is
  geni(((op or 256) << 6) or (rm << 3) or rd)
```

```
proc gen2i(val op, val rd, val imm) is
  geni((op << 11) or (rd << 8) or imm)
```

```
proc gensetsp(val rm) is
  geni((i_setsp << 7) or (rm << 3) or 5)
```

```
proc gen1i(val op, val imm) is
  geni((op << 8) or imm)
```

```
proc gen1(val op, val rm) is
  geni((op << 7) or (rm << 3))
```

```
proc geni(val i) is
  gen(cbf_inst, 0, i)
```

```
proc genibr(val i, val lab) is
  gen(cbf_cbranch, i, lab)
```

```
proc genbr(val seq, val lab) is
  if seq
  then
    skip
  else
    { gen(cbf_branch, 0, lab)
      ; endblock(400)
    }
```

```
proc gencall(val lab, val seq, val clab, val tail) is
  if tail and (lab = proclabel)
  then
    gen(cbf_tcall, 0, lab)
  else
    { gen(cbf_call, 0, lab)
      ; genbr(seq, clab)
    }
```

```

proc genconst(val reg, val n) is
var i;
var cp;
var found;
{ startblock()
; found := false
; i := constb
; while (i < constp) and (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
    }
; gen(cbf_const, (i_ldrpci << 3) or reg, cp)
}

```

```

proc genstring(val reg, val x) is
var i;
var sp;
{ startblock()
; sp := stringp
; i := 0
; while i ≤ ((tree[x + 1] and 255) >> 2) do
    { strings[stringp] := tree[x + i + 1]
    ; stringp := stringp + 1
    ; i := i + 1
    }
; gen(cbf_string, (i_addpci << 3) or reg, sp)
}

```

```

proc genpref(val reg, val lab) is
{ gen(cbf_prog, reg, constp)
; consts[constp] := lab
; constp := constp + 1
}

proc gen(val t, val h, val l) is
{ cb_loadpoint := cb_loadpoint + 1
; codebuffer[cb_bufferp] := (t << 28) or (h << 16) or l
; cb_bufferp := cb_bufferp + 1
; if cb_bufferp = cb_size
  then
    generor("code buffer overflow")
  else
    skip
}

func getlabel() is
{ if labelcount < labval_size
  then
    labelcount := labelcount + 1
  else
    generor("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
; labval[proclabel] := cb_loadpoint
; gen(cbf_entry, 0, proclabel)
}

proc genexit() is
{ cb_sethigh(cb_entryinstp, stk_max)
; gen(cbf_exit, stk_max, 0)
; endblock(300)
}

```

```

proc initbuffer() is
{ cb_loadpoint := 0
; cb_loadbase := 0
; constp := 0
; constb := 0
; stringp := 0
; stringb := 0
; cb_bufferp := 0
; cb_blockstart := 0
}

```

```

proc startblock() is
  if inblock
  then
    skip
  else
    { cb_loadbase := cb_loadpoint
    ; constb := constp
    ; stringb := stringp
    ; cb_blockstart := cb_bufferp
    ; gen(cbf_blkstrt, 0, 0)
    ; inblock := true
    }

```

```

proc endblock(val n) is
var codelength;
var constlength;
var stringlength;
{ codelength := cb_loadpoint - cb_loadbase
; constlength := constp - constb
; stringlength := stringp - stringb
; if inblock and ((codelength + ((constlength + stringlength) << 1)) > n)
then
  { cb_sethigh(cb_blockstart, codelength)
  ; cb_setlow(cb_blockstart, (constlength << 8) or stringlength)
  ; gen(cbf_blkend, 0, 0)
  ; inblock := false
  }
else
  skip
}

```

```

proc cb_setflag(val p, val f) is
  codebuffer[p] := (codebuffer[p] and #FFFFFFF) or (f << 28)

```

```

func cb_flag(val p) is
  return codebuffer[p] >> 28

proc cb_sethigh(val p, val f) is
  codebuffer[p] := (codebuffer[p] and #F) or (f << 16)

func cb_high(val p) is
  return (codebuffer[p] >> 16) and #FFF

proc cb_setlow(val p, val f) is
  codebuffer[p] := (codebuffer[p] and #0) or f

func cb_low(val p) is
  return codebuffer[p] and #FFFF

func cbrlen(val offset) is
  if (offset > (-127)) and (offset < 128)
  then
    return 1
  else
    return 1 + brlength(offset - 1)

func brlength(val offset) is
  if (offset > (-1023)) and (offset < 1023)
  then
    return 1
  else
    return 2

func cb_entrylen(val p) is
  if cb_high(p) = 0
  then
    return 1
  else
    return 2

```

```

proc expand() is
var bufferp;
var offset;
var entrysize;
var flag;
{ bufferp := 0
; while bufferp < cb.bufferp do
  { flag := cb.flag(bufferp)
  ; if flag = cb.entry
    then
      { if labval[cb.low(bufferp)] ≠ cb.loadpoint
        then
          labval[cb.low(bufferp)] := cb.loadpoint
        else
          skip
      ; entrysize := cb.entrylen(bufferp)
      ; cb.loadpoint := cb.loadpoint + entrysize
      }
    else
      if flag = cb.exit
      then
        cb.loadpoint := cb.loadpoint + entrysize
      else
        if flag = cb.inst
        then
          cb.loadpoint := cb.loadpoint + 1
        else
          if flag = cb.blkstrt
          then
            { cb.loadbase := cb.loadpoint
            ; cb.conststart := cb.loadpoint + cb.high(bufferp)
            ; cb.stringstart := cb.conststart + ((cb.low(bufferp) ≫ 8) ≪ 1)
            ; cb.blockstart := bufferp
            }
          else
            if flag = cb.blkend
            then
              { if (cb.loadpoint and 1) = 0
                then
                  skip
                else
                  cb.loadpoint := cb.loadpoint + 1
              }
            ; if (cb.loadpoint - cb.loadbase) = cb.high(cb.blockstart)
            then
              skip
            else
              cb.sethigh(cb.blockstart, cb.loadpoint - cb.loadbase)
            ; cb.loadpoint := cb.loadpoint + (((cb.low(cb.blockstart) ≫ 8) + (cb.low(cb.blockstart) and #FF

```

```

}
else
if flag = cbf_lab
then
  if labval[cb_low(bufferp)] ≠ cb_loadpoint
  then
    labval[cb_low(bufferp)] := cb_loadpoint
  else
    skip
else
if flag = cbf_cbranch
then
{ offset := labval[cb_low(bufferp)] - (cb_loadpoint + 1)
; cb_loadpoint := cb_loadpoint + cbrlength(offset)
}
else
if flag = cbf_branch
then
{ offset := labval[cb_low(bufferp)] - (cb_loadpoint + 1)
; cb_loadpoint := cb_loadpoint + brlength(offset)
}
else
if flag = cbf_tcall
then
{ offset := (labval[cb_low(bufferp)] + entrysize) - (cb_loadpoint + 1)
; cb_loadpoint := cb_loadpoint + brlength(offset)
}
else
if flag = cbf_call
then
  cb_loadpoint := cb_loadpoint + 2
else
if flag = cbf_prog
then
  cb_loadpoint := cb_loadpoint + 3
else
if (flag = cbf_const) or (flag = cbf_string)
then
  cb_loadpoint := cb_loadpoint + 1
else
  cmperror("code buffer error")
; bufferp := bufferp + 1
}
}

```

```

proc flushbuffer() is
var bufferp;
var last;
var offset;
var entrysize;
var flag;
{ cb_loadpoint := 0
; last := 0
; expand()
; while cb_loadpoint ≠ last do
  { last := cb_loadpoint
  ; cb_loadpoint := 0
  ; expand()
  }
; outhdr()
; constb := 0
; stringb := 0
; bufferp := 0
; cb_loadpoint := 0
; while bufferp < cb_bufferp do
  { flag := cb_flag(bufferp)
  ; if flag = cbf_entry
  then
    { outbin(i_pushl ≪ 8)
    ; entrysize := cb_entrylen(bufferp)
    ; if entrysize = 1
    then
      skip
    else
      outbin((i_decspi ≪ 7) or cb_high(bufferp))
    ; cb_loadpoint := cb_loadpoint + entrysize
    }
  else
  if flag = cbf_exit
  then
  { if entrysize = 1
  then
    skip
  else
    outbin((i_incspi ≪ 7) or cb_high(bufferp))
  ; cb_loadpoint := cb_loadpoint + entrysize
  ; outbin(i_popl ≪ 8)
  }
  else
  if flag = cbf_inst
  then
  { outbin(cb_low(bufferp))
  ; cb_loadpoint := cb_loadpoint + 1

```



```

}
else
if flag = cbf_blkstrt
then
{ cb_conststart := cb_loadpoint + cb_high(bufferp)
; cb_stringstart := cb_conststart + ((cb_low(bufferp) >> 8) << 1)
; cb_blockstart := bufferp
}
else
if flag = cbf_blkend
then
{ if (cb_loadpoint and 1) = 0
then
skip
else
{ cb_loadpoint := cb_loadpoint + 1
; out2i(i_add2i, 0, 0)
}
; outconsts(cb_low(cb_blockstart) >> 8)
; outstrings(cb_low(cb_blockstart) and #FF)
; cb_loadpoint := cb_loadpoint + (((cb_low(cb_blockstart) >> 8) + (cb_low(cb_blockstart) and #FF)
}
else
if flag = cbf_lab
then
skip
else
if flag = cbf_cbranch
then
{ offset := labval[cb_low(bufferp)] - (cb_loadpoint + 1)
; outcbr(cb_high(bufferp), offset)
; cb_loadpoint := cb_loadpoint + cbrlength(offset)
}
else
if flag = cbf_branch
then
{ offset := labval[cb_low(bufferp)] - (cb_loadpoint + 1)
; outbr(offset)
; cb_loadpoint := cb_loadpoint + brlength(offset)
}
else
if flag = cbf_tcall
then
{ offset := (labval[cb_low(bufferp)] + entrysize) - (cb_loadpoint + 1)
; outbr(offset)
; cb_loadpoint := cb_loadpoint + brlength(offset)
}
else
if flag = cbf_call

```

```

then
{ offset := labval[cb_low(bufferp)] - (cb_loadpoint + 2)
; outcall(offset)
; cb_loadpoint := cb_loadpoint + 2
}
else
if flag = cb_prog
then
{ offset := ((cb_conststart  $\gg$  1) + (cb_low(bufferp) - constb)) - ((cb_loadpoint + 1)  $\gg$  1)
; outofs(cb_high(bufferp), offset, cb_low(bufferp))
; cb_loadpoint := cb_loadpoint + 3
}
else
if flag = cb_const
then
{ offset := ((cb_conststart  $\gg$  1) + (cb_low(bufferp) - constb)) - ((cb_loadpoint + 1)  $\gg$  1)
; outcref(cb_high(bufferp), offset)
; cb_loadpoint := cb_loadpoint + 1
}
else
if flag = cb_string
then
{ offset := ((cb_stringstart  $\gg$  1) + (cb_low(bufferp) - stringb)) - ((cb_loadpoint + 1)  $\gg$  1)
; outcref(cb_high(bufferp), offset)
; cb_loadpoint := cb_loadpoint + 1
}
else
skip
; bufferp := bufferp + 1
}
; codesize := cb_loadpoint
}

```

```

proc outcbr(val inst, val offset) is
  if (offset > (- 127)) and (offset < 128)
  then
    outbin((inst << 8) or (offset and #FF))
  else
    { outbin((invert(inst) << 8) or brlength(offset))
    ; outbr(offset - 1)
    }

func invert(val inst) is
  if inst = i_beq
  then
    return i_bne
  else
    if inst = i_bne
    then
      return i_beq
    else
      if inst = i_blt
      then
        return i_bge
      else
        return i_blt

proc outbr(val offset) is
  if (offset > (- 1023)) and (offset < 1023)
  then
    outbin((i_bu << 11) or (offset and #7FF))
  else
    outcall(offset)

proc outcall(val offset) is
  { outbin((i_bl1 << 11) or ((offset >> 11) and #7FF))
  ; outbin((i_bl2 << 11) or (offset and #7FF))
  }

```

```

proc outoffs(val reg, val offset, val c) is
var poffs;
{ if (cb_loadpoint and 1) = 0
  then
    poffs := cb_loadpoint + 2
  else
    poffs := cb_loadpoint + 1
; consts[c] := (labval[consts[c]] - poffs) << 1
; outcref((i_ldrpci << 3) or reg, offset)
; out2i(i_addpci, 6, 0)
; out3(i_add3, reg, reg, 6)
}

proc outcref(val inst, val offset) is
  outbin((inst << 8) or offset)

proc out3(val op, val rd, val rm, val rn) is
  outbin((op << 9) or (rn << 6) or (rm << 3) or rd)

proc out2i(val op, val rd, val imm) is
  outbin((op << 11) or (rd << 8) or imm)

proc outconsts(val n) is
var count;
{ count := 0
; while count < n do
  { outbin(consts[constb + count] and 65535)
  ; outbin(consts[constb + count] >> 16)
  ; count := count + 1
  }
; constb := constb + n
}

proc outstrings(val n) is
var count;
{ count := 0
; while count < n do
  { outbin(strings[stringb + count] and 65535)
  ; outbin(strings[stringb + count] >> 16)
  ; count := count + 1
  }
; stringb := stringb + n
}

```

```
proc outbin(val d) is
{ selectoutput(binstream)
; putval(d and 255)
; putval(d  $\gg$  8)
; selectoutput(messagestream)
}
```

```
proc outhdr() is
var wordsize;
{ wordsize := cb_loadpoint  $\gg$  1
; outbin(wordsize and 65535)
; outbin(wordsize  $\gg$  16)
}
```