SETL Newsletter Number 64 January 26, 1972

SETL compiler with elaborated data structures.

Kurt Maly

In this newsletter we will present

a - some comments on the data structure elaboration language;

b - its application to the compiler programs 'scan' up to and including 'preparser'.

The elaboration language.

This language, as suggested in newsletter number 39, is essentially simple and with some changes in its syntax and semantics, as outlined later, should be

a - an extremely useful link between SETL and any lower level language;

b - if implemented for SETL, a considerable improvement of SETL's efficiency.

Instead of discussing all the recommended features of the language we will comment only on those which were found useful in this particular set of programs which certainly are not a representative sample of programs.

Two kinds of sets are distinguished. Sets which are collections of items of which the items commonly are used for indexing mappings or as an iteration domain (we will call them kind A); secondly, sets which represent mappings (kind B). In SETL those mappings may be indexed by any object whatsoever which clearly is bound to produce inefficiencies in most implementations. The obvious programming trick of using reference numbers (i.e., pointers to the object) is mostly what the elaboration language is about.

For instance, we may declare a set a in some way as such a collection - or indexing - set (kind A); then we declare a mapping f as a 'range' and specify that the index is an element of the set a and that not the object but its reference number is used as index. To further enhance efficiency one will want to be able to describe the form of the elements of the set a or the values of the mapping f by giving them some attributes.

After these general remarks let us go into the details of the language itself:

```
pile a;
pile nodup a
hash a;
```

Sets (kind A), thus declared, will be used mainly to address ranges by reference numbers. The semantics of those statements can be found in newsletter number 39. The elements in those sets may be described by a content describer (abbreviated: cd):

```
1 - obj;
2 - int;                    int k;
3 - string;                 string k;
4 - bit;                    bit k;
5 - elt b;                  elt b:k;
6 - subset b;
7 - tupl(cd,..,cd);
8 - tupl(*cd);
9 - tupl(cd,-);
10 - set cd;
```

Except for 7 - and 8 - the semantics remains unchanged. 8 - tupl(*cd) signifies that each element is of the form of a tuple of unspecified dimension but all its elements are of the form described by cd; 9 - tupl(cd,-) means that each element is of the

form of a tuple of unspecified dimension the first element of which is described by cd. (Variations: tupl(cd,cd,-);)

Sets used as mappings will be specified by 'ranges'. Usually, those mappings with a common domain will be stored within one range. If all the functions have n parameters the declaration should look as follows:

range name(x_1,...,x_n);

range is a keyword and 'name' is a name given to this range. Next should follow a description of the nature of the index (i.e., is it a reference number, character string, integer, etc.?) and its exact form:

1 - $x_i$int;$x_i$blank index;$x_i$bit index;$x_i$string index;
2 - $x_i$aselt a;
3 - $x_i$hash a;
4 - $x_i$hash

The names of the functions stored in this range together with the form of their values is given by:

stores name1,name2,...,namek cd,namℓ,...,namem cd;

where cd is a content describer as mentioned above.

Quite often we know exactly the size of sets and further that they do not change very much (e.g., sets used as tables). To encompass this useful information we allow sets to be 'pinned'.

pin a;

For semantics see newsletter number 39. In our opinion it should be employed only for sets of kind A, since it has quite unpleasant side effects for ranges, namely, requiring the programmer to use coding tricks to get around them. Rather, this information should be passed on to the compiler in the form of a somewhat different

'dimension statement'. (Actually, there seems to be no reason for not extending this statement to sets of kind A and dropping the 'pin statement altogether'.)

```
dim rangenamel(expl,...,expn),...,rangenamek(expl,...,expn);
```

Quite a useful extension is the function:

```
newat rangename
```

which returns the smallest value in the domain of the range 'rangename' which is undefined.

In order to nullify certain but not all functions in a range the statement

```
drop f,g;
```

should be used.

Conversion of SETL objects can be done by appending

```
aselt a
ashash a
assub b
```

For extensive, but not commented, examples see the following programs. For the casual reader we give a short, commented, illustration of the above described features.

```
      /* form a set */
ind = {'ab','cd','efg'};
      /* declare it */
pile ind; string;/* the elements are all character strings */
pin ind; /* the set ind is not going to change very much */
bas = {<01b,11b>,<1b,0b>};
```

```
pile bas; tupl(bit 2,bit 2); pin bas;
/* now define
switch1 = {<'ab',label1>,<'efg',label2>};
switch2 = {<'ab','2,3>>,<'cd','1,5,6>};
switch3 = {<'cd',<1b,0b>>};
/* declare a range for the above mappings which have as common
   domain the set ind */
range switch(stri); /* we give it the name switch */
stri aselt ind; /* is to be referenced by a reference number
                   (with respect to ind) */
stores switch1 obj, switch2 tupl (*int), /* the values of switch2
   are tuples of unspecified length containing integers */
switch3 elt bas; /* the value of switch3 are to be given as
   reference numbers with respect to bas */
/* After encountering this statement the functions switch1 switch2
   and switch3 are stored in the range switch by converting 'ab',
   etc., to reference numbers of ind, <01b,11b>,etc, to
   reference numbers of bas, and reserves for the values space
   appropriate to the 'stores' specification */
/* if we know that the mappings will not grow in size we can
   'pin' them by */
dim switch(# switch1);
c = 'ab' aselt ind; /* assigns to the variable c the reference
   number of 'ab' in ind */
to go switch1(c); /* results in transfer to label1 */
```

The compiler programs with data structures elaborated.

The following programs are a recoding of the programs of newsletters number 58 and 62 and provided with the specifications of the data structures according to the above rules. Corrections to those programs written up in various newsletters and newly found ones are incorporated. For the sake of completeness all the programs were written up as a whole without references to already

existing code portions in other newsletters. Comments were eliminated and since we described our use of the data elaboration language above no comments were made with regard to its use either.

For illustrative purposes the data structures were declared at the point of their first appearance. Hence, it occurs that we use 'symbtab' in 'preparser' before we invoked the lexical scanner 'scan' where the symbol table is declared. To make the programs valid SETL programs we would have to shift some of those declarations.

```
definef scan;preparser external separate;

initial setup(type,table);
letype = separate('name operator delimiter bitstring string
integer real nullop objtype uop er ef elop local external mop skip
lpar rpar foal whl atl lod str comp deff defs label coma semicol
eql return expr');pile letype;
string; pin letype;
states = hd tl [table];pile states; string; pin states;
typeset = hd tl [type]; pile typeset; string; pin typeset;
cases = separate ('end end  go skip cont do'); pile cases;
string; pin cases;
nullop = separate('newat null nl nult nulb nullc t f
true false holl');assub symbtab;symbtab = nl; hash symbtab;string;
operator = separate('hd tl abs bitr floor ceiling not n
dec oct hol compile  type atom pair as in eq ne gt lt
ge le max min exp log with less lesf and or nε u is')
assub symbtab;
abre = separate('om nm el fa ex st an qm')
assub symbtab;
range table (type,state);state aselt states;type aselt typeset;
stores table tupl(elt cases,-);
range type(char);char string index;stores type elt typeset;
<end,endl,go,skip,cont,do,next>=;
switch = {<end,endc>,<endl,endlc>,<go,goc>,<skip,
loop>,<cont,contc>,<do,doc>};
range switch (case); case aselt cases; stores switchf obj;
endset = {<'namop','name'>,<'intrealbit','integer'>,<'bitoct',
'bitstring'>,<'octbit','bitstring'>,<'intreal','integer'>;
<'real','real'>,<'quoted','string'>,<'next','delimiter'> ;
range endset(state); state aselt states; stores endset
elt letype;
abret = {<'om','⊼'>; <'nm','#'>,<'el','∈'>,<'fa','∀'>,
<'ex','∃'>,<'st','|'>,<'an','∋'>,<'qm','∀∃'>};
range abret(token);token aselt abre;stores abret elt symbtab;
dim  table(#typeset,#states),type(#type),switchf(#switchf),
endset(#endset),abret(#abret);
```

```
datab = nl; pile datab; obj;
er ='$'; s = <input,1>;cstring=record(s);if cstring eq nulc
or cstring eq ∩ then return <'ef' aselt letype,'ef' aselt symbtab>;;
cstring (#cstring+1) = er;print cstring; end initial;
state = next aselt states;
nn=n-1; data =∩;token=nulc;
loop=nn=nn+1; action=table(type(cstring(nn)),state);
switch: go to switchf(hd action);
goc:  state=tl action;
conc:  token=token+cstring(nn);go to loop;
endc:  ltype=endset(state);
endlc:  n=nn;return if data ne ∩ then <ltype,token aselt
  symbtab, data aselt datab> else <ltype,token aselt symbtab>;
doc:  <-,rout,action> = action; rpak(rout); go to if action
eq ∩ then loop else switch;
end scan;
define rpak(rout); scan local; initial inout={'input','output'}
       assub symbtab;;
obl:   bitn=t;
ob2:   bitn=f;
er2:   print'illegal period';
relbra: if cstring(nn+1) is c eq '/' and cstring(nn+2) ne ' '
        or c eq '.' then token=cstring(nn)+c; nn=nn+2;
        else token=cstring(nn);nn=nn+1;
qm:     if cstring(nn+1) is c eq '⌋' then token=cstring(nn)
        +c; nn=nn+2; else token=cstring(nn);nn=nn+1;;
barcom: if cstring(nn+1) is c eq '/' then token=cstring(nn)
        +c,nn=nn+2; else if c eq '*' then nn=
        endcomment( );action='skip' aselt cases; end if
        else token=cstring(nn); nn=nn+1; end if c;
skipl:  nn=nn+1;
```

```
eof:    token=cstring(nn);ltype='ef' aselt letype;nn=nn+1;
specend: token=cstring(nn);nn=nn+1;
gtest:   nn=nn+1;if cstring(nn) ne '''' then action='end'
         aselt cases;;
ercheck: cstring=record(s); if cstring eq nulc or cstring
         eq ↵ then if token eq nulc then token='ef';
         ltype='ef' aselt letype;action='endl' aselt cases;
         else action = 'end' aselt cases; cstring(1) = er;
         nn=1; end if token; else print cstring; cstring(
        #cstring+1) = er; nn=0; end if cstring;
endo:    [; dum= bit(token(data (2) is x+2:#token-x-1));
         data = (data (1) +dum) as bstring; ltype=
         'bitstring' aselt letype; block bitoctal;-]
endb:    [;data=(data (1)+token(data(2) is x+2:#token
         -x-1)) as bstring; ltype='bitstring' aselt letype;
         block octalbit;-]
octend:  bitoctal; token=token+cstring(nn);
bitend:  octalbit; token=token+cstring(nn);
er3:     print 'illegal bitstring specification';
octo:    bitn=f;
bita:     if n bitn then print 'illegal bitstring specification';
         action='end' aselt cases; else data=<token,#token>;;
oct:     data=<bit(token),#token>;
opcheck: token=token+'.' aselt symbtab;
         iff          (token ∈ nullop)?
([;token=token asobj symbtab;block tokn;-] ltype='nullop' aselt letype;),
                                        (token ∈ operator)?
                (tokn; ltype='operator' aselt letype;), (token ∈ inout)?
                (tokn; ltype='name' aselt letype;),(abret(token)is
                                        tok ne ↵) act1,act2;
         act1:  ltype='delimiter' aselt letype;token=tok asobj symbtab
         act2:  tokn; ltype='uop' aselt letype;
                data=token(1:#token-1);
```

```
        token='uop'; end iff;
        end rpak;
definef endcomment; scan external cstring, nn, er;
initial switch = { <'*',star>, <er,eror>}; ster={'*',er};;
nn=nn+2; [advance:] (while n (cstring (nn) is c) ∈ ster)
nn=nn+1;; go to switch(c); [star:] nn=nn+1; if cstring
(nn) eq '/' then return nn; else go to advance;;
[eror:] cstring=record(s); if cstring eq nulc or cstring
eq ∧ then print 'illegally structured comment';
cstring (1) = er; nn=0; return nn; else nn=1; print cstring;
cstring(#cstring+1) = er; go to advance;
end endcomment;
```

```
definef nt; scan external letype, symbtab, datab;
nextoken external defstack, tokstack, restack,lparset,rparset,
name, calstack, mbody, blocks, bl, sc, pl, co, pr, ms,
br, uop, end, semi, defi, defif, ef, ends;
initial bras = {<'block',blk>,<'define', dfnf>,<'definef,dfns>,
        <']',brkt>};
        range bras(tok);tok aselt symbtab; stores bras obj;
        macro=nl;
        range macro(tok);tok aselt symbtab; stores
        macro tupl(set afn, set restack);
        range afn(tok); tok aselt symbtab; stores afn int;
        range argstack(i); i int; stores argstack tupl
        (* tupl (elt letype, elt symbtab, elt datab));
        end initial;
main:   [;  if# tokstack eq 0 then token=scan( ); else token =
        topo tokstack;; block tokget(token);-]
        <typ,tok,-> = token; minvo; end iff;
        iff          (tok eq blocks)?
                lblock,     (tok eq bl)?
                        brct,      (typ eq name)?
                         (macro(tok) is mac eq ↵)? (return token;),
                       (return token;),
brct:   tokget(new); if new(2) ne sc then new stack tokstack;
        return token; end if;
        call=f;[bt:] tokget(token); (while token(2) ne sc
        doing tokget(token);) if token(2) eq blocks then go
        to blk;; resta(token); end while;
        resta(token); tokget(token); if bras(token(2) is labb
        ne ↵ then go to labb;; resta(token); go to bt;
        block mget(a); afn=nl;mok=t; tokget(token);
        mname=token(2); if token(1) ne name then print
        'illegal macro-name'; mok=f;;tokget(new);
        if new(2) is tok eq sc then go to a; end if;
```

```
      if tok ne pl then mok=f; print 'illegal beginning
      of macro argument list'; (while new(2) ne sc)
      tokget(new); end while; go to a; end if tok;
      tokget(n1); tokget(n2);
      (while n1(1) eq name and n2(2) eq co doing
      tokget(n1); tokget(n2);) if afn(n1(2)) ne Λ
      then print 'illegal duplication of arguments';
      mok=f; else afn(n1(2)) =#afn+1;; end while;
      if afn(n1(2)) ne Λ then print 'illegal duplication
      of arguments'; mok=f; else afn(n1(2)) =#afn+1;;
      tokget(n3); if n2(2) eq pr and n3(2) eq sc then
      go to a; else mok=f;
      iff          (n2(2) eq sc)?
        (n3 stack tokstack; go to a;), (n3(2) eq sc)?
                                    a,  (tokget(new);
        (while new(2) ne sc)        tokget(new); end while;
       go to a;); end iff; end if; end mget;
blk:  mget(b4);
b4:   if mok then macro(mname) = <afn,restack>;
      end if; tokget(new); if new(2) eq ms then
      (≠ restack ≥ ∀n≥1) restack(n) stack tokstack;;
      tokget(new); end if;
      if new(2) ne br then print 'illegal ending
      of inverted macro definition'; new stack
      tokstack;; drop restack; go to main;
      block namget(a,b);
      tokget(token); <typ,tok,-> = token;
      if typ eq uop then pname = token(3) asobj
      datab aselt symbtab; calsta(token); go to b;
      end if; tokget(new); if new(1) eq uop then
      pname = new(3) asobj datab aselt symbtab;
      else pname = tok;; end if new; calsta(token);
      calsta(new);
      [b:] tokget(token);(while token(2) ne sc doing
```

```
tokget(token);) calsta(token);; tokget(new);
if new(2) eq ms then call=t; tokget(new);
end if;
if new(2) ne br then print 'illegal ending of
inverted procedure definition'; reswi=t;
else reswi=f; end if;
block reshufle(a); defstack(1) = <name,a>;
copy(calstack,defstack);defsta(semi);copy
(restack,defstack);defsta(end); defsta(<name,
pname>);defsta(semi); if reswi then new
stack tokstack; reswi=f;;
if call then ( #calstack≥ ∀n≥1) calstack(n)
stack tokstack;; go to main; end reshufle;
reshufle(a); end namget;
[dfns:]  nameget(defi,b5);
[dfnf:]  namget(defif,b6);
[brkt:]  calsta(<name,newat as cstring is pname
aselt symbtab>); call=t; reshufle(defif);

ℓblock:  mget(lbl);
[lbl:] tokget(token); (while n token(1) eq ef)
iff              (token(2) eq ends)?
            namtest?  (resta(token);tokget(token);continue
                  while;),
      mstore, (resta(token);token=new; continue while;);
namtest:  tokget(new);=new(2) eq mname;
mstore:  tokget(skip); if mok then macro(mname)
=<afn,restack>; end if; drop restack;
go to main; end iff; end while;
print 'illegal macro ending'; return token;
```

```
minvoc:   nargs=0; tokget(new); if new(2) is tok eq pl
          then go to arg; end if;
          if tok eq n then go to expand;;
          print 'illegal macro use'; new stack tokstack;
          return token;
          [arg:] argtok=nult; argstack=nl; tokget(token);

          [argbeg:] iff      (token(2) is tok ∈ lparset)?
                        (mpart=mpart+1;),      (tok ∈ rparset)?
                     stex, (tok eq co and mpart eq 0)?
                    argend,  (tok eq sc and mpart eq 0)?
                         merl, anad;
          stex:  mpart=mpart+1; if mpart eq -1 then
          argtok stack argstack; narg=narg+1; tokget(new);
          if new(2) ne sc then token stack tokstack; end if;
          go to expand; end if mpart;

          tokget(token); go to argbeg;
          merl:  print 'illegal macro call ending';
          go to expand;
          end iff;
          anad:  argtok=argtok+<token>; tokget(token);
          go to argbeg;
          [expand:]  <argfn,mbody>=mac;
          if narg eq 0 then (#mbody≥∀n≥1) mbody(n)
          stack tokstack; end  mbody; go to main; end if;
          (#mbody≥∀n≥1) token=mbody(n);
          if argfn(token(2)) is argno) eq ⌐⌐ then token
          stack tokstack; else if argno gt narg then
          token stack tokstack; else(# ( argstack(argno)
          is argtok)≥∀n≥1) argtok(n) stack tokstack;;
          end if argno; end if (argfn; end mbody;
          go to main;
end nt;
define defsta(token); nextoken external defstack;
```

```
token stack defstack; return; end defsta;
define calsta(token); nextoken external calstack;
token stack calstack; return; end calsta;
define resta(token); nextoken external restack;
token stack restack; return; end resta;
define copy(st1,st2); (1≤∀n≤#st1) st1(n) stack
st2;; return; end copy;
definef topo stack; token=stack(#stack); stack(#stack)
=⌐; return token; end topo;
```

```
definef  nextoken; scan external symbtab, letype, datab;

        initial z=nl; z(1) = 'elop'; z(2)='local'; z(3)='external';
        z(4)='mop';z(5)='-';z(6)='lpar';z(7)='rpar';z(8)='foal';
        z(9)='whl';z(10)='atl';z(11)='lod';z(12)='str';z(13)='comp';
        z(14)='deff';z(15)='defs';z(16)='end';z(17)=';';z(18)='label';
        z(19)='coma';z(20)='semicol';z(21)='eql';z(22)='return';
        z(23)='expr';z(24)='if';z(25)='then';z(26)='else';
        zx=z;zx(5)='skip';zx(16)='name';zx(17)='delimiter';
        zx(24)='ifex';zx(25)='thex';zx(26)='elsex';
        (1≤∀i≤#z)z(i)=z(i) aselt symbtab; zx(i)=zx(i) aselt letype;;
        <elop,lcl,extnl,mns,mnskip,lpar,rpar,foal,whl,
        atl,lod,str,comp,deff,defs,endv,semi,label,
        com,semicol,eql,ret,exp,ifex,thex,elsex,->=
        [+: 1≤i≤#z]<<zx(i),z(i)>>;
        swiset={<'<',mult >,<'[',brckt>,<',',comma>,<';',
        semicolon>,<'(',lparen>,<'define',ldefs>,<'definef',ldeff>,
        <'then',lthen>,<'else',lelse>,<'if',lif>,<'initial',
        linit>,<'external',lext>,<'local',lloc>,<'til',ltil>,
        <'lpar',lparl>,<'iff',liff>,<'=/',leq>,<'expr',lexpr>,
        <'continue',conquit>,<'quit',conquit>,<'return',retrn>};
        range swiset(tok);tok aselt symbtab; stores swiset obj;
        dim  swiset(# swiset);
        iterset={<'∀',fl>,<'while',whl>,<'at',lat>,<'load',
        ld>,<'store',st>};
        range iterset(tok);tok aselt symbtab; stores iterset obj;
        dim iterset(#iterset);
        rel={'<.','</','>.','>/'}  assub symbtab;
        opset= {'*','/','//','+','-'}  assub symbtab;
        bras={<'block',blk>,<'define',dfn>,<'definef',dfnf>,
          <']',brkt>};
        range bras(tok);tok aselt symbtab; stores bras obj;
        lparset= {'(','[','{','<'}  assub symbtab;
        rparset= {')',']','}','>'}  assub symbtab;
```

```
        name='name' aselt letype; beg=t; tilswi=f;
        tilswil=t;lbeg=f;reswi=f;ifstat=t;
        lexl=nl;restack=nl;defstack=nl;tokstack=nl;itstack=
        nl;calstack=nl;mbody=nl;
        range stack(i);i int; stores restack,defstack,tokstack
        itstack,calstack,mbody,lexl,tstack,tupl(elt letype,
        elt symbtab, elt datab);
        blostack=nl;ifstack=nl;
        range blostack(i); i int; stores blostack tupl( *elt
        symbtab);
        range ifstack(i); i int; stores ifstack, expstack int;
        sc=';' aselt symbtab; p<='(' aselt symbtab; pr=')'
        aselt symbtab; br=']' aselt symbtab; bl='[' aselt
        symbtab; cl='{' aselt symbtab; cr='}' aselt symbtab;
        pbl='<' aselt symbtab; pbr='>' aselt symbtab;
        co=',' aselt symbtab; ms='-' aselt symbtab;
        col=':' aselt symbtab; op='operator' aselt letype;
        uop='uop'aselt letype; ifs='if' aselt symbtab; th='then' aselt
        symbtab; doing='doing' aselt symbtab;
        rpr='rpar' aselt symbtab; els='else' aselt symbtab;
        ends='end' aselt symbtab; qem='?' aselt symbtab;
        lpr='lpar' aselt symbtab; epr='expr' aselt symbtab;
        blocks = 'block' aselt symbtab; defi='define' aselt
        symbtab; defif='definef' aselt symbtab;
        ef='ef' aselt letype; delimiter='delimiter' aselt
        letype;lexpt=1;tilstack=nl;
        range tilstack(i);i int; stores tilstack elt symbtab;
        end initial;

getok:  if   lexl ne 0 then token=lexl(lexpt);lexpt=lexpt+1;
        if lexl(lexpt) eq ↩ then drop lexl; lexpt=1;; return token;
        end if lexpt;
            /*  instead of one complete pass over the input
                we return a token at a time to preparser */
```

```
        iff                  testl?
              (go to dolab;),  (typ eq name and tilswi and tilswil)?
                    tilname,   (typ eq name and beg)?
                        test2?,     cont,
                    actl, (new stack tokstack;go to cont;);
        testl:  token=nt( );/* procedures with no arguments have
        to be called as 'procedurename( )' */
        <typ,tok,->=token;=swiset(tok) is dolab ne Λ;
        test2:  new=nt( );=new(2) eq col;
        actl:  add(token);add(new);beg=t;add(label);
        go to getok;
        end iff;
 cont:  add(token);beg=f;tilswil=f;go to getok;
 shufle: (# tstack≥∀n≥1)tstack(n) stack tokstack;;


tilname: frst=t;tilfd=f;tstno=# tilstack;
        (tstno≥∀n≥1) if tok eq tilstack(n) then lastil=n;
        tilfd=t; else if frst then frstnot=n;frst=f;end if frst;
        end else; end tstno;
        if n tilfd then token stack tokstack;tilswil=f;
        go to getok;;
        if frstnot gt lastil then print 'illegal til nestring';;
        (lastil≤∀n≤tstno)add(rpar);add(endv);
        add(semi);x=topoff blostack;tilstack(n)=Λ;;
        if lastil eq 1 then tilswi=f;; token stack tokstack;
        tilswil=f;go to getok;

   ltil: new=nt( );if new(1) ne name then print 'illegal
        til reference';; new(2) stack tilstack; skip=nt( );
        if skip(2) ne ';' then skip stack tokstack; beg=t;
        go to getok;

        block exloc(a);token=nt( );add(elop);add(a);
        if token(2) ne sc then add(elop);; token stack
        tokstack; beg=f;go to getok; end exloc;
```

```
lloc:    exloc(lcl);

lext:    exloc(extnl);

lthen:   if n ifstat then if# ifstack eq 0 then ifstat=t;
         go to thenl; else add(thex); go to getok; end if   ;
    thenl: add(token); add(lpar);beg=t;go to getok;

linit:   add(token); add(lpar);<token(2)>stack blostack;
         beg=t; go to getok;

lelse:   if n ifstat then ifstack(#ifstack)=Ω;add(elsex);
         go to getok;;
         add(rpar);add(token);add(lpar);beg=t;
         [;contok=nult;(1≤∀n≤5)contok=contok+if token(1) eq uop
         then   <token(3) asobj datab aselt symbtab> else
                <token(2)>;token=nt( );save(token);;contok stack
         blostack;go to shufle;block scoptok;-]

ldeff:   [;add(deff);add(lpar);add(token);
         token=nt( );<typ,tok,->=token;
         if typ eq uop then              <token(3) asobj datab aselt symbtab>
                 stack blockstack; go to cont;;
         if typ ne name then print 'illegal name of procedure';
         go to cont;;new=nt( );
         if new(1) eq uop then        <new(3) asobj datab aselt
         symbtab>                        else<tok>stack blostack;
         add(token);token=new;go to cont;block proc(deff);-]

ldefs:   proc(defs);
```

```
lparen: if n beg then go to cont;; beg=f;part=0;
        (while token(2) eq pl doing token=nt( );)token stack
        itstack;part=part+1;end while;contok=nult;ct=0;
        if iterset(token(2) is labi) ne Λ then go to labi;;
        itfd=f;(while n (part eq 0 or (partc eq 0 and itfd)))
      [;ct=ct+1;tok=token(2);if ct le 5 then contok=contok+if token(1) eq uop
        then        <token(3) asobj datab aselt symbtab> else
                    <tok>;              end if ct;block scope;-] save
        (token);token=nt( );partc=1;ifswi=f;
        iff                  (tok,eq pl)?
            (part=part+1;partc=partc+1;),(tok eq pr)?
                  (part=part-1;partc=partc-1;),(tok ∈ relat)?
                             ifcheck, (tok eq sc)?
                                 lpl, (tok eq ifs)?
                            (ifswi=t;),(tok eq th)?
                                  (ifswi=f;),(continue while;);
        ifcheck: if token(2) eq '∀' aselt symbtab then if n itfd
        then if n ifswi then pnum=part-1;partc=1;itfd=t;
        end if n ifswi; end if n itfd;end if token;
        end iff;
        end while;
        if itfd then contok stack blostack;
        (1≤∀n≤pnum)add(lpar);;add(foal);
                 (pnum<∀n≤#itstack)add(itstack(n));;
        drop itstack;save(lpar);save(token);go to shufle;
        end if;
        (1≤∀n≤#itstack)add(itstack(n));;drop itstack;
        save(token);go to shufle;
 lpl:   (1≤∀n≤partc) add(lpar);;
        (part<∀n≤#itstack)add(itstack(n));;drop itstack;
        save(token);go to shufle;
  fl:   part=1;(while part ge 1 doing token=nt( );)
        scope; if tok eq pl then part=part+1; else if
        token eq pr then part=part-1;;end else if; save
        (token); end while;
```

```
        (1≤∀n≤ #itstack)add(lpar);;add(foal);
                        add(itstack(n));drop itstack;
        save(lpar);save(token);contok stack blostack;
        go to shufle;
    wl: part=1;doswi=f;(while part ge 1) scope;
        save(token);
        iff         (tok eq pl)
            (part=part+1;),(tok eq pr)?
                    (part=part-1;),(tok eq doing and n doswi)?
                                doblo,wll;
        doblo: doswi=t;save(lpar);end iff;
    wll:token=nt( );end while
        contok stack blostack;
        (1≤∀n≤ #itstack)add(lpar);;add(whl);
                        add(lpar);drop itstack;
        tstack(#tstack)=rpar;
        if doswi then save(rpar);;save(lpar);save(token);
        go to shufle;
        block atstr(a);new=nt( );if new(1) ne name
        then print 'illegal at or store reference';;
        add(a);(1≤∀n≤# itstack)add(itstack(n));;
        drop itstack;add(token);add(new);
        (<token(2),new(2)>aselt datab)stack blostack;
        go to latl;end atstr;
    lat: atstr(atl);
     st: atstr(str);
   latl: token=nt( );(while part ge 1 doing token=nt( );)
        if token (2) is tok eq pl then part=part+1;else
        if tok eq pr then part=part-1;end else if;add
        (token);;token stack tokstack;add(lpar);go to getok;

semicolon:  add(token);if # defstack ne 0 then (# defstack≥∀n≥1)
        defstack(n) stack tokstack; drop defstack;; tilswil=t;
        beg=t;ifstat=t;go to getok;end if# ;
        token=nt( );(while token(2) eq pr or token(1) eq rpr
```

```
        doing token=nt( );)add(rpar);;beg=t;ifstat=t;
        block conmatch;token=nt( );contok=nult;ct=0;
        (while token(2) ne sc doing token=nt( );)
        scope; end while;
        if n (#blostack≥∃[n]≥1 | match(blostack(n),contok))
        then add(rpar);add (end); print 'no match for
        scope terminator found'; go to semicolon; end if n;
        endt=endv;
        iff       (n eq # blostack is blon)?
              sadd,   (n eq (blon-1))?
            (blostack(blon)(1) eq els)? emit,
          (contok(1) eq ifs)?  emit,
          sadd,   emit;
        emit: k=blon;(while k gt n) intm=topoff blostack;
              k=k-intm; add(rpar);add(end);add
              (semi);print 'illegal block nesting'; end while;
              go to emit; end iff; end conmatch;
        if token(2) is tok eq ends then token=nt( );
        conmatch; end if token;
        if tok ne sc then token stack tokstack; go to getok;;
        endt=endv;
        sadd: add(rpar);add(endt);skip=topoff blostack;
        go to semicolon;

liff:   add(token);add(lpar);<token(2)>stack
        blostack;ifct=0;token=nt( );nswi=f;
        (while n (token(2) is tok eq sc and ifct eq 0)
        iff              (tok eq pl)?
             lplus,                (tok eq pr)?
                      rplus,   (tok eq co and ifct eq 0)?
                        (save(com);), (save(token););
        lplus:  ifct=ifct+1; if ct eq 1 then save(lpar);
                lastdef=#tstack; else save(token);;
        rplus:  ifct=ifct-1;if ct eq 0 then save(rpar);else
                save(token);;new=nt( );nswi=t;if new(2)
                eq qem then lastdef stack expstack;;
```

```
        end iff;
        if nswi then token=new;nswi=f;else token=nt( );;
        end while;
        save(semicol);beg=f;(# tstack≥∀n≥1) if n eq
        expstack(#expstack) then expstack(#expstack)=Ω;
        exp stack tokstack;;tstack(n) stack tokstack;;
        go to getok;

mult:   if n beg then go to cont;;
        mct=1; add(token); token=nt( );
   ml:  if mct eq 0 then save(token); beg=f;go to shufle;;
        if token(2) is tok eq sc then print 'illegal statement';
        save(token); go to shufle;;
        iff             (tok eq ms)?
                  mad,          (tok eq pbl)?
                          (mct=mct+1;),(tok eq pbr)?
                                  (mct=mct-1;), mad2;
        mad: new=nt( );if new(2) eq co or new(2) eq
            pbr then save(mns);save(mnskip);;
            token=new;go to ml;
        end iff;
        mad2: save(token);token=nt( );go to ml;

brct:   if lbeg then go to lbll;;
        nl=nt( );n2=nt( );
        block multcom;add(token);nl(1)=('c'+(nl(1) asobj
        letype)) aselt letype; add(nl);add(n2);
        token=nt( );mbct=1;(while mbct ge 1 doing token
        =nt();) if token(2) eq bl then mbct=mbct+1;
        else if token(2) eq br then mbct=mbct-1;end if;
        end else if;save(token);end while;save(comp);
```

```
        save(token);go to shufle;end multcomp;
        if nl(1) eq op or nl(1) eq uop or nl(2) ∈ opset
        and n2(2) eq col then multcomp; else if
        n beg then add(token);else lbeg=t;save
        (token);end if n; save(nl);save(n2);go to
        shufle; end else if;
        [lbll:] lbeg=f;bct=1;add(token);token=nt( );
        (while bct ge 1 doing token=nt( );) if token(2)
        is tok eq bl then bct=bct+1;else if tok eq br
        then bct=bct-1;;end else if;add(token);end while;
        token stack tokstack;if token(1) is typ eq op
        or typ eq uop then beg=f;else beg=t;add
        (label);;go to getok;

lparl:  add(token); if token(1) eq lpr then beg=t;;
        go to getok;

conquit: oldto=token; contok=nult; ct=0;token=nt( );
        (while token(2) ne sc doing token=nt( );) scope;
        end while;
        if n (#blostack≥∃n≥1 | match(blostack(n),contok))
        then print 'illegal continue or quit statement';;
        add(oldto);go to semicolon;

  leq:  if beg then add(eql);beg=f;go to getok;
        else go to cont;end if beg;

  lexp: if token(1) ne epr then go to cont;;
        token=nt( );add(token);beg=f;go to getoken;
```

```
lif:    if n beg then ifstat=f;add(ifex);1 stack ifstack;
        go to getok;;
        1 stack ifstack;oldto=token;token=nt( );
        contok=nult;ct=0;(while token(2) ne sc doing
        token=nt( );)scope;
        iff             (tok eq els)?
                  (ifstack(#ifstack)=Ω;), (tok eq ifs)?
                           (1 stack ifstack;), lsav; end iff;
        lsav: save(token);end while;
        save(token);
        if  ifstack eq 0 then ifstat=f;add(ifex);
        1 stack ifstack;beg=f;go to shufle;
        else drop ifstack;add(oldto);ifstat=t;
        beg=f;contok stack blostack;go to shufle;
retr:   new=nt( ); if new(2) eq sc then add(ret); else
        add(token);; new stack tokstack; go to getok;

        end nextoken;
define  save(token);nextoken external tstack;token stack
tstack; return; end save;
  define a stack b;b(#b+1)=a;return;end stack;
    define add(token);nextoken external lex1;token stack lex1;return;
       end add;
   definef topoff blostack;blon=#blostack;blotok=blostack
   (blon); blostack(blon)=Ω;
    iff             (blotok(1) ne els)?
              (return 1;), (blostack(blon-1)(1) eq ifs)?
                  (blostack(blon-1)=Ω;return 2;), (print 'illegal
               else option';return 1;); end iff;
end topoff;
definef match(a,b);cok=t; (1≤∀i≤#b) if a(i)
eq Ω then quit;; if a(i) ne b(i) then cok=f;; end 1;
return cok; end match;
```

```
define preparser(treetop);scan external letype,symbtab,datab;
block setup;<on,beg,var,o,er,>=;
begend=<beg,nnon,beg,on,beg,f,parcas,beg,f,mispar,
beg,specialcases,f,beg,f>;
bend=<f,nnon,f,on,f,f,parcas,f,f mispar,f,
specialcases,f,users,f>;
start =beg locsin begend;
finish={nnon,on,parcas,mispar,specialcases,users} is
labs locsin bend;
label= {<x locsin begend,x>,x ∈ labs | n x ∈ {on,parcas}} ;
label (on locsin begend)=nnon; label( {mispar,parcas}
locsin  begend)=parcas;label(00000o)=getoken;nnmask=
on locsin begend;
kindz=nl; gross= {<x,x>,x ∈ {var,o,'(',')',er}} ;
give; symbkind=analyze(tokinf);typkind=analyze
(typinf);
definef analyze(inf);preparser external gross,kindz;
map=nl;(∀x ∈ inf)<set,kind,grossk,->=x;kind in kindz;
if atom set then map (set)=kind; else (∀y ∈ set) map(y)=
kind;; end if;if grossk ne Ω then gross(kind)=grossk;;
end ∀ x; return map;end analyze;
kinds=kindz;[;lprec=analyse(lprinf);rprec=analyse
(rprinf);(∀ x∈ hd [lprec] | rprec(x) eq Ω)rprec(x)=lprec(x);;
(∀x∈ hd[rprec] | lprec(x) eq Ω)lprec(x)=rprec(x);;
(∀x ∈ kinds | lprec(x) eq Ω) lprec(x)=lprec(gross(x));
rprec(x)=rprec(gross(x));;block precedence(lprec,rprec,
lprinf,rprinf);-] precedence(lprecl,rprecl,lprinfl,rprinfl
);
symbtuple=<var,var,o,var,'(',var')', {'(',er},var,{er,')'},
{'(',er,o},{')',er}>
(∀x ∈ tl[symbkind] u tl [typkind] with var) mask(x)=
gross(x) locsin symbtuple+x locsin triple;;
```

```
end setup;
block give; /* given in newsletter 62, pages 10-11 */
end give;
nextoken external delimiter,sc,uop,name,co,qem;
initial setup;
kinds=kinds assub symbtab;var='var' aselt kinds;
o='o' aselt kinds;
symbols=hd [symbkind] assub symbtab;
types=hd [typkind] assub letype;
range symbkind(token);token aselt symbols; stores symbkind
elt kinds;
range typkind(type);type aselt types;stores typekind elt kinds;
range mask(x);x aselt kinds;stores mask bit 15;
labels=hd [label];hash labels;bit 15;
range label(x);x aselt labels; stores label obj;
range prec(x);x aselt kinds;stores rprecl,rprec,
lprec,lprecl int 12;
range gross(x);x aselt kinds;stores gross elt kinds;
switch= <er,')',erp>,<'(',ev,per>,<'(',')',pp>,
<o, er,oer>,<o,')', oer>,<er,er, erer> ;
range switch(kind1,kind2);kind1 aselt kinds;kind2
aselt kinds;stores switch obj;
uswi= <'∀∃','∀∃',11>,<'∀∃',',',12>,<',','∀∃',13>,
<',',',',14>,<',',';',14> ;
range uswi(t1,t2);t1 aselt symbtab;t2 aselt symbtab;
stores uswi obj;
catenation='catenation' aselt symbtab; omitted=
'omitted' aselt symbtab;
dim symbkind(#symbkind),typkind(#typkind),
mask(#mask),label(#label),prec(#rprec),
gross(#gross),switch(#switch,#switch),uswi(#uswi,
#uswi);end initial;
```

```
statstak=nl;bakstak=nl;
range statstak(i);i int;stores statstak tupl(bit 15,
elt kinds,obj);
range bakstack(i);i int; stores bakstack tupl(elt
kinds,tupl(elt letype,elt symbtab,elt datab));
nodplace=nl;nodarray=nl;
range nodeplace(node);node int;stores nodplace
int 12;
range nodarray(i);i int;stores nodarray tupl
(bit 1, int 12, int 12, int 12, int 12);
iffq=nl;
range iffq(i);i int;stores iffq int 12;

definef   top stk; return stk(#stk);end top;
definef   topoff stk; x=stk(#stk);stk(#stk)=  ;return x; end topoff;
define  a onto stk; stk(#stk+1)=a;return; end onto;
definef n elm stk;return stk(#stk-n+1);end elt;

          zero=00000b;state=zero;<er,<er,er>>onto bakstak;
          go to jumpin;
getoken:  <state,kind,tokdat>onto statstak;
jumpin:   if bakstack ne nl then <kind,tokdat>=topoff
          bakstak;<type,token,->=tokdat;else tokdat
          =nextoken( );<type,token,->=tokdat;kind=getkind
          (tokdat);end else;
definef getkind(tokdat);preparse external symbkind,typkind,var;
<type,token,->=tokdat;return if typkind(type)is x ne Ω then x
else if symbkind(token) is x ne Ω then x else var;end getkind;
newstate:  if token eq 'iff' aselt symbtab then iffbeg=#statstak+2;;
          state=0b+state(1:#state-1)or starts and mask(kind);
          go to label ((state and finish)aselt labels);
```

```
definef kind stelt;return stelt(2);end kind;
definef tokof stelt;return stelt(3)(2);end tokof;
definef tdat stelt;return stelt(3);end tdat;

nnon:     if bakstak eq nl then <(getkind(nextoken( )is x),x>
          onto bakstak;;<kind2,->=top bakstak;
          <-,kind1,->=top statstak;
          condd=rprec(kind2)gt lprec(kind1);
          condm=rprec1(kind2)gt lprec1(kind1);
          if state and nnmask eq zero then if condd then
          go to getoken; else condensenn;end if condd;end if;
          iff     (gross (knd (2 elm statstak)) eq var)?
               (condd)?                          (condm)?
          getoken, (condensenon;),  getoken,(new=newat nodplace;
                                              condenseon;);
          end iff;
newcycle: tokdat=new;
newcyc:   kind=var;
restart:  state=hd top statstak;go to newstate;

definef nodtype(node);preparser external nodarray,nodeplace,
symbtab;
(load)  return if nodarray (nodeplace(node))is x (1) eq 0
          then x(2) else Ω ; end load;
(store t) nodeplace(node)=newat nodarray is x; nodarray
          (x)(1)=0;nodarray (x)(2)=t;
          nodarray (x)(3)=node;return;;end nodtype;
definef desc(x,i);preparser external nodarray,
nodeplace;
(load)    if nodarray(nodeplace(x)) (4)is n eq Ω then return Ω;;
          (1≤∀k≤i) if n eq Ω then return Ω;;
          n=nodarray(n)(5);end 1≤∀k;
          return if nodarray(n) is z(1) eq 0 then z(3) else
          z(2:3); end load;
```

```
(store t) n=nodplace(x);
          iff        (i eq 1)?
                tutest,          chain,
               setn,    (nodarray(n)(4)=nodplace(t););
          chain:  k=nodarray(n)(4);if k eq Ω then nodarray
          (n)(4)=newat nodarray is k;end if k;
          (1≤∀1<i) k1=nodarray(k)(5);if k1 eq Ω then
          nodarray(k)(5)=newat nodarray is k1;k2=k;k=k1;end 1<  1;
          if type t eq tupl then nodarray (k)(1)=1;
          nodarray (k)(2:4)=t; else nodarray(k2)(5)=nodplace(t);
          end if type;
          tutest:=type t eq tupl;
          setn:  nodarray(n)(4)=newat nodarray is k;
                 nodarray(k)(1)=1;nodarray(k)(2:4)=t;end iff;
          end desc;
          block condensenn; oldel  =tdat topoff statstak;if nodtype
          (oldel) ne catenation then new=newat nodplace;
          nodtype(new)=catenation;else desc(oldel,descnum
          oldel+1)=tokdat;new=oldel;end else;end condensenn;
          definef descnum node;preparser external nodarray,nodeplace;
          if nodarray(nodeplace(node))(4)is n eq Ω then
          return 0;;i=0;(while n ne Ω );i=i+1; n=
          nodarray(n)(5);end while;return i;end descnum;
          block condensenon;
          tkdat=tpdat topoff statstack;oldel=tdat topoff
          statstak;
          if tkdat(1) is optok eq uop then new=newat nodplace;
          desc(new,1)=<name,tdat(2)>;desc(new,2)=
          oldel;desc(new,3)=tokdat;nodtype(new)=uop;;
          if nodtype(oldel) ne optok then [;new=newat nodeplace;
          desc(new,1)=oldel;desc(new,2)=tokdat;
          nodtype(new)=optok;block newco;-]else if descnum oldel
```

```
          eq 1 then newco;else desc(oldel,descnum oldel+1)=
          tokdat;new=oldel;end else desc;end else if;
          end condensenon;
          definef tpdat stelt;return stelt(3)(2:);end tpdat;
          block condenseon;
          if tpdat topoff statstak is tkdat(1) eq uop then
          desc(new,1)=<name,tkdat(2)>;desc(new,2)=
          tokdat;else desc(new,1)=tokdat;end if tpdat;
          nodtype(new)=thdat(1);end condenseon;
          block condense3(t);
          desc(new,1)=tdat topoff statstak;nodtype(new)=[;
          ((tokof topoff statstak)asobj symbtab+t asobj symbtab)
          aselt symbtab;block tokas;-]
          block condense4(t);
          desc(new,2)=tdat topoff statstak;nodtype(new)=[;
          tokas;desc(new,1)=tdat topoff statstak;end condense4;

parcas:   new=newat nodplaces;if gross(knd (3 elm statstak))
          eq var then condense4(token) else condense3(token);;
          go to newcycle;
mispar:   dum=new;new=newat nodeplace; if kind ne er
          then desc(new,1)=tdat topoff statstak;nodtype
          (new)=('missing(,'+token asobj symbtab) aselt symbtab;
          else if gross(knd (2 elm statstak)) eq er then return
          dum;end if gross;<kind,tokdat>onto bakstack;
          if gross(knd(3 elm statstak)) eq var then condense4
          (',missing)'aselt symbtab);else condense3(',missing)'
          aselt symbtab);end if gross;end if kind;
          go to newcycle;
specialcases: kind1=knd top statstak;
          go to switch(gross(kind1,gross(kind));
```

```
exp:      kind=o; go to restart;
per:      <kind,tokdat>onto bakstack;tokdat=tdat  topoff statstak;
          <type,token,->=tokdat;kind=o;go to restart;
pp:       tokdat=<name,((tokof topoff statstak)asobj symbtab+
          token asobj symbtab) aselt symbtab>;go to newcyc;
oer:      <kind,to dat> onto bakstak;tokdat=<name,'omitted'
          aselt symbtab>;go to newcyc;
ever:     if n tokdat ϵ endfilesigns then treetop=<name,
          'blank statement' aselt symbtab>;return;else
          print 'parse terminated by illegal end of file';
          exit;;
users:    <state,kind,<delimiter,sc>>onto statstak;iffend=
          #statstak;cur=iffbeg;iffq=nl;root=newat
          nodplace;desc(root,1)=tdat statstak(cur);
          nodtype(root)=qem;root onto iffq;cur=cur-2;
          [contin: ] cur=cur+4;if cur+3 gt iffend then go to
          err;;(0≤ ∀n≤2) if tokof statstack(cur+i) eq sc then
          go to err;; if next iffq is top eq Ω then go to err;;
          typel=tokof statstack(cur+1);type2=tokof statstak
          (cur+3);go to if uswi(typel,type2)is lab ne Ω
          then lab else err;
          [11:] desc(top,2)=[;newat nodeplace block nw;-]is top;
          nodetype(top)=catenation;desc(top,1)=nw;is topl;nodtype
          (topl)=qem;desc(topl,1)=tdat statstak(cur);topl onto iffq;
          desc(top,2)=nw;is top2;nodtype(top2)=qem;
          desc(top2,1)=tdat statstak(cur+2);top2 onto iffq;
          go to contin;
          [12:] desc(top,2)=nw;is top;nodtype(top)=co;
          desc(top,1)=nw;is topl;nodtype(topl)=qem;
          desc(top,1)=tdat statstak(cur);topl onto iffq;
          desc(top,2)=tdat statstak(cur+2);go to contin;
          [13:] desc(top,2)=nw;is top;nodtype(top)=co;
```

```
            desc(top,1)=tdat statstak(cur);
            desc(top,2)=nw;is topl;nodetype(topl)=qem;
            desc(topl,1)=tdat statstak(cur+2);topl onto iffq;
            go to contin;
            [14:] desc(top,2)=nw;is top ; nodtype(top)=co;
            desc(top,1)=tdat statstak(cur);
            desc(top,2)=tdat statstak(cur+2);
            if type2 eq sc and iffend eq(cur+3) then go
            to headend; else go to contin;;
            [err:] print 'probable illegal or missing delimiter
            in iff header';
            [headend:] (iffbeg≤∀n≤iffend) topoff statstak;;
            new=root;<';'aselt kinds,<delimiter,sc>> onto
            bakstak;go to newcycle;
            definef next queue;initial n=0;;
            n=n+1;return queue(n);end next;
definef setat locsin tuple; bito=nulb;
(while tuple ne Ω) <ent,tuple>=tuple;
bitr=bitr+if atom setat then if atom ent then setat eq ent
else setat ∈ ent else if atom ent then ent ∈ setat else∃x ∈ ent
| x ∈ setat;;return bitr;end locsin;
end preparse;
```