Phase one of the SETL compiler

Kurt Maly

Principally it consists of three main routines, namely, 'lex', 'nt' and 'scan'. 'Scan' is the lexical scanner which always just returns the next lexical token of the input string. Between 'lex' and 'scan' is 'nt' which acts as a macro processor and has the additional burden of extracting inverted procedure definitions out of the token stream and saving them until the next semicolon. When required a call is omitted instead. 'lex' can be thought of as a general editor of the token stream it receives from 'nt'.

For the reader's convenience we give a short reference list to the actions of 'lex' and 'nt' and whenever it is deemed necessary we will add some explanations.

'lex': main loop on page 5 under label 'getok'.
Remarks: each token induces either one of the actions described below or emittance to lexl.

1. check label - whether it is referenced by a 'til name' statement on page 5 under label 'tilnam'.
Remarks: Such labels should not be enclosed in brackets.
2. 'til name' statement - on page 6 under label 'ltil'.
3. local on page 6 under label 'lloc'.
4. external on page 6 under label 'lext'.
5. if on page 19 under label 'lif'.

Remarks: 'if', 'then' and 'else' can, in Setl, appear as part of a conditional expression or of an if statement. The main problem is recognizing which case it is. The ambiguous case which cannot be resolved without look ahead, an 'if' at the beginning of a statement, is checked by using a stack and the fact that the expression always contains balanced 'then' and 'else' before a semicolon.

6. then on page 6 under label 'lthen'.
7. else on page 7 under label 'lelse'.

8. initial on page 7 under label 'linit'.
9. definef on page 7 under label 'ldeff'.
10. define on page 7 under label 'ldefs'.
11. left parenthesis on page 8 under label 'lparen'.
Remarks: left parentheses get special care only at the beginning of a statement;
the '∀' set iteration is dealt with on page 9 under label 'fl'.
the while iteration on page 9 under label 'wl';
the at label on page 9 under label 'lat'.
the store name on page 10 under label 'str';
the load on page 10 under label 'ld'.
the numerical range iteration immediately after 'lparen'
If is neither of the above mentioned cases those parentheses which constitute block markers are omitted as 'lpar'.
The rest as normal opening parentheses (remember that a left parenthesis at the beginning of a statement may be either).
12. semicolon on page 10 under label 'semicolon'.
Remarks: if there is any procedure definitions found by 'nt', then they are now inserted and analyzed.
The 'end...' feature of Setl is now dealt with by checking items on a corresponding stack and all necessary block markers and 'and' operators are inserted.
13. iff... on page 11 under label 'liff'.
Remark: because of its peculiar two-dimensional form we have to separate nodes with unambiguous markers.
14. multiple assignment on page 12 under label 'mult'.
Remarks: the ambiguity of '-' has to be resolved.
15. an opening bracket on page 12 under label 'brct'.
Remarks: resolve whether beginning of compound operator, label or expression.
16. blockmarker on page 13 under label 'lparl'.
17. continue or quit statement on page 13 under label 'conquit'.
18. equal sign on page 13 under label 'leq'.
Remark: resolve whether assignment or expression indicator in iff statement.
19. test node marker in iff statement inserted by 'lex' before testnode on page 13 under label 'lexp'.

'nt': main loop on page 16 under label 'main'.

1. macro definition on page 18 under label 'lblock'.
2. opening bracket on page 16 under label 'brct'.
   Remarks: if an inverted procedure definition produces a call, only the call itself is emitted, not the semicolon. Hence, if the call should be a statement a semicolon has to be appended to the closing bracket.
   the inverted macro is dealt with on page 17 under label 'blk'. If a call is indicated the just accumulated body is placed there without any change.
   the inverted function definition on page 18 under label 'dfnf'.
   the inverted subroutine definition on page 18 under label 'dfns'.
   the inverted function definition without function name on page 18 under label 'brkt'.
3. macro invocation on page 18 under label 'minvoc'.
   Remarks: if a macro invocation is followed by a semicolon this is skipped.

```
define lex;
        /* lex prepares the input string in such a way that
          it can be parsed by the precedence parser 'preparserl'.
          It gets the input from 'nt' in the form of lexical tokens.
          At this point macros will be already expanded and
          inverted procedure definition will be reverted.*/
        /*initialization block*/
initial elop = <'op','dop'>; lcl=<'local','local'>;
   extnl = <'external','external'>; mns = <'op','mop'>;
   mnskip = <'skip','-'>; lpar=<'lpar','lpar'>;
   rpar = <'rpar','rpar'>; foal = <'op',foal'>;
   whl = <'op','whl'>; atl = <'op','atl'>;
   lod = <'op','lod'>; str = <'op', 'str'>;
   comp = <'op','cop'>;deff =<'op','deff'>;
   defs = <'op','defs'>; udop = <'op','udop'>;
   endv = <'op','end'>; semi=<'del',';'>;
   label = <'label','label'>; com = <'coma','coma'>;
   semicol = <'semicol','semicol'>; eql = <'eqls','eqls'>;
   exp = <'expr','expr'>; ifex =<'ifex','if'>;
   thex = <'thex',then'>; elsex = <'elsex','else'>;
   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>};
   iterset ={<'V',fl>,<'while',wl>,<'at',lat>,
             <'load',ld>,<'store',st>};
rel = {'<','<=','>','>='};
bras = {<'block',blk>,<'define',dfn>,<'definef',dfnf>,
             <],brkt>};
lparset= {'(','[','{','<'}; rparset= {')',']','}','>"};
name =; beg = t; tilswi = f; tilswil = t; lbeg = f;
reswi= f; ifstat = t; end initial;
```

```
            /*start of the main loop*/

getok:  iff testl?
           (go to dolab;), (typ eq name and tilswi  and tilswil)?
            tilnam, (typ eq name and beg)?
              test2? cont,
            actl,(new stack tokstack; go to cont;);
              /*is some special action required */
        testl: token=nt; <typ,tok,->= token; = swiset(tok) is dolab ne Ω;
              /*do we have label*/
        test2: new=nt, = new(2) eq ':';
              /*we have label and add it to lexl and reset
                'beg' (indicator of beginning of statement) to true*/
        actl: add(token); add(new); beg=t; add(label);
           go to getok; end iff;
cont:   add(token); beg=f; tilswil=f; if type eq 'ef' then return;;
        go to getok;
shufle:       /*a look-ahead was done and tokens accumulated
                on 'tstack' they have to be sent back to 'nt'*/
        (#tsta ≥ ∀n ≥ 1) tsta(n) stack tokstack;; go to getok; tsta=nl;
              /*end of main loop*/

 tilnam:      /*we are in the range of a til(tilswi) and after
                semicolon (tilswil) and have to check whether
                current token of type name is on til stack (tilsta)*/
        frst = t; tilfd = f; tstno= #tilsta;
        (tstn ≥ ∀n ≥ 1) if tok eq tilsta(n) then lastil=n;
        tilfd=t; else if frst then frstnot=n; frst=f; end if frst;
        end else; end tstn;
              /*if name was not found on tilstack*/
        if n tilfd then token stack tokstack; tilswil=f;
        go to getok;;   /* in order to catch label the token
        is sent back to 'nt'*/
              /*if there was non-matching name above deepest
                matching label name*/
        if frstnot gt lastil then print 'illegal til nesting';;
```

```
            /*for each closing of a block omit blockmarkers*/
        (lastil ≤ n ≤ tstno) add(rpar); add(endv);
        add(semi); x=topoff blostack; /*blostack contains the first
        five tokens of each 'endable' statement*/
        tilsta(n) = Ω;; if lastil eq 1 then tilswi=f;;
            /*return token to  nt*/
        token stack tokstack; tilswil=f; go to getoken;

ltil:       /*a 'til name;' statement has been found
              the name is stacked on the tilstack and the
              statement itself is skipped*/
        new = nt; if new(1) ne name then print 'illegal til reference';;
        new(2) stack tilsta;   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 ';'
        then add(elop);; token stack tokstack; beg=f; go to getok;
        end exloc;

lloc:       /*a 'local' has been encountered and operator
              tokens are inserted to handle '...; local;'*/
        exloc(lcl);

lext:       /*the same for 'external'*/
        exloc(extnl);

lthen:      /*decide whether 'then' is part of a conditional
              expression or an if statement*/
        if n ifstat then if #ifsta eq 0 then /* it is part of
        if statement */ ifstat=t; go to thenl; else add(then);
        go to getok; end if#;;
    thenl:  /*insert 'beginning of block' marker*/
        add(token); add(lpar); beg=t; go to getok;
```

```
linit:          /* insert block marker and stack 'initial' onto blostack*/
        add(token); add(lpar); token(2) stack blostack;
        beg=t; go to getok;

lelse:          /*decide whether part of if statement */
        if n ifstat then /*it is within conditional expression unstack
        corresponding if */ ifsta(#ifsta)=Ω; add(elsex); go to getok;;
                /* 'else' as in if statement*/
        add(rpar); add(token); add(lpar); beg=t;
                /*accumulate the next five tokens (including 'else')
                  into 'contok' to be checked against the possible
                  'end else'   and stack  contok onto blostack*/

      block scoptok; contok=nult;
        (1 ≤ ∀n ≤ 5) contok = contok + <token(2)>; token= nt;
        save(token);; contok stack blostack; go to shufle;
        end scoptok;
        scoptok;

ldeff:  [;/*define(f) has been encountered  put name of
        procedure on block stack*/
        add(deff); add(lpar); add(token);
        token = nt; <typ,tok,-> = token;
        if typ eq 'uop' then tok stack blostack; 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(2) stack blostack;
        else tok stack blostack;; add(token); token=new;
        go to cont; block proc(deff);-]

ldefs:          /* the same as for function above */
        proc(defs);
```

```
lparen:         /*left parenthesis*/
                /*if not beginning of statement continue*/
        if n beg then go to cont;; beg=f; part = 0;
                /*stack parentheses*/
        (while token(2) eq '(' doing token = nt;) token stack
        itsta;   part = part+1;  end while;
                /*check for immediately recognizable iteration header*/
        if iterset(token(2) is labi) ne Ω then go to labi;;
                /*else check for header with numerical range and
                  accumulate control token */
        contok = nult; ct = 0; itfd = f;
        (while part ge 1) ct = ct + 1; tok=token(2); if ct le 5 then
        contok = contok + <tok>;;
        save(token); token = nt; lsem = t;

      iff    (tok eq '(')?
          (part=part+1;), (tok eq ')')?
                      (part=part-1;), (tok ε relat)?
              (if token(2) eq '∀' then itfd=t;;), (tok eq ';' and lsem)?
                          (lblono=part; lsem=f;), (continue while;);
      end iff; end while;
      if itfd then contok stack blostack;
      add(foal); (1 ≤ ∀n ≤ #itsta) add(itsta(n));
      itsta(n) = Ω;;
              /*emit blockmarker after closing parenthesis */
      save(lpar); save(token); go to shufle; end if;
              /*else parentheses are programmer used block markers
                or beginning of left hand side expression */
      if lsem then print 'illegal beginning of statement';
      beg=t; save(token); (1 ≤ ∀n ≤ #itsta) add(lpar); itsta(n) =Ω;;
      go to shufle; else (1 ≤ ∀n ≤ lblono) add(lpar);
      itsta(n) = Ω;; (lblono < ∀n ≤ #itsta) add(itsta(n));
      itsta(n) = Ω; beg=f; save(token); go to shufle;;
```

```
fl:       /* '∀' set iteration header*/
  contok = nult; ct=0;  (while part ge 1 doing ct=ct+1; token=nt;)
  tok = token(2); if ct le 5 then contok=contok+<tok>;;
  if tok eq '(' then part=part+1; else if tok eq ')' then
  part=part-1;; end else;  save(token); end while;
  (1 ≤ ∀n ≤ #itsta) add(itsta(n)); itsta(n) = Ω;;
  contok stack blostack; add(foal); save(lpar); save(token);
  go to shufle;
wl:       /*while iteration header*/
          /*'doswi' will become true if a 'doing' will appear*/
  doswi=f; contok=nult; ct=0; (while part ge 1) ct=ct+1;
  tok=token(2); if ct le 5 then contok = contok+<tok>;; save(token);
  iff   (tok eq '(')?
      (part=part+1;), (tok eq ')')?
          (part=part-1;), (tok eq 'doing' and n doswi)?
                              al, wll;
  al: doswi=t; save(lpar); /*save block of statements follows*/ end iff;

 wll: token = nt; end while;
   if doswi then /* 'doing' occurred must omit block marker*/
   save(rpar); end if;
   save(lpar); /* for block after header*/ save(token);
   contok stack blostack; add(whl); (1 ≤ ∀n ≤ #itsta)
   add(lpar); itsta(n)=Ω;; beg=f; go to shufle;

 block atstr(a); new= nt; if new(1) ne name then print 'illegal
   'at' or 'store' reference';; add(a); (1 ≤ ∀n ≤ #itsta)
   add(itsta(n)); itsta(n)=Ω;; add(token); add(new);
   (<token(2)>+<new(2)>) stack blostack; goto latl; end atstr;

lat:              /* 'at label' statement*/
   atstr(atl);
```

```
str:                    /*'store name' statement*/
        atstr(str);
     latl:              /*insert block marker after 'at', 'store' and
                          'load' statement*/
        token=nt; (while part ge 1 doing token=nt;)
        if token(2) is tok eq '(' then part=part+1; else
        if tok eq ')' then part=part-1;; end else; add(token);;
        token stack tokstack; add(lpar); go to getok;

ld:                     /*'load' statement*/
        add(lod); (1 ≤ ∀n ≤ #itsta) add(itsta(n)); itsta(n)=Ω;;
        add(token); token(2) stack blostack; go to latl;

semicolon:              /*a semicolon has been found*/
        add(token); /*if there is a procedure definition waiting
        (from an inverted definition) place it hereafter*/
        if #defsta ne 0 then (#defsta ≥  n ≥ 1) defsta(n)
        stack tokstack; defsta(n)=Ω;; tilswil = t; beg=t;
        ifstat=t; go to getok;   end if;
                        /*replace right parenthesis by block marker*/
        token=nt; (while token(2) eq ')' doing token=nt;)
        add(rpar);; beg=t; ifstat=t;

     block commatch; token=nt; contok=nult; ct=0;
        (while token(2) is tok ne ';' doing token=nt; ct=ct+1;)
        if ct le 5 then contok=contok+<tok>;; 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;; endt = endv + <contok>;
                  /*we found on the block stack a matching series
                    of tokens*/
      iff               (n eq #blostack is blon)?
           sl, (n eq (blon-1))?
         (blosta(blon)(1) eq 'else')?  al,
        (contok(1) eq 'if')?  al,
       sl, al;
      al:  k = blon; (while k gt n) intm = topoff blostack;
                  /*for each terminator emit blockmarker and
                    'statement-ender', skipping the
                    'else' terminator'*/
      k = k - intm; add(rpar); add(end); add(semi);
      print 'illegal block nesting'; end while; go to sl;
      end iff; end conmatch;

      if token(2) is tok eq 'end' then /*check correct ending
      of compound statement*/ commatch;
      if tok ne ';' then token stack tokstack; go to getok;;
      endt = endv;
    sl:add(rpar); add(endt); skip = topoff blostack;
      go to semicolon;

liff:               /*iff has been found*/
                    /*commas following action mode are marked,
                      parentheses surrounding nodes are replaced
                      by 'lpar' and 'rpar' and semicolon ending
                      header is marked */
      add(token); add(lpar); token(2) stack blostack;
      ifct=0;  token=nt;  nswi=f;  (while  n (token(2) is tok
      eq ';' and ifct = 0)
      iff                (tok eq '(')?
                al,                  (tok eq ')')?
                       a2,     (tok eq ',' and ifct eq 0)?
                            (save(com);), (save(token););
```

```
      a1: ifct = ifct+1;  ifct eq 1 then save(lpar); lastdef = #tsta;
          else save(token);;
      a2: ifct=ifct-1;  if ifct eq 0 then save(rpar); else
          save(token);; new=nt;  nswi=t;  if new(2) eq '?'
          then lastdef stack expstack;; end iff;
      if nswi then token = new; nswi = f; else token = nt;;
      end while;
      save(semicol); beg = f; (#tsta ≥ ∀n ≥ 1) if
      n eq expstack(#expstack) then expstack(#expstack)=Ω;
      exp stack tokstack;; tsta(n) stack tokstack;; go to getok;

mult:                 /*multiple assignment starts, replace 'skip'
                        operator (-) by operator followed by dummy
                        argument*/
      if n beg then go to cont;;
      mct = 1; add(token); token = nt;
   m1:if mct eq 0 then token stack tokstack; go to getok;;
      if token(2) is tok eq ';' then print 'illegal statement';
      token stack tokstack; go to getok;
      iff               (tok eq '-')?
                   a1,          (tok eq '<')?
                           (mct=mct+1;),   (tok eq '>')?
                                        (mct=mct-1;),   m2;
      a1: new=nt;  if new(2) eq ',' or new(2) eq '>'
        then add(mns);  add (mnskip);; token = new;
        go to m1; end iff;
   m2: add(token); token = nt; go to m1;

brct:                 /*'[' has been found*/
                      /*if label?*/
      if lbeg then go to lbl1;;
      n1 = nt; n2=nt; /*check for compound operator*/

   block multcom; add(token); add(n1); add(n2);
      token = nt; mbct = 1; (while  mbct ge 1 doing token = nt;)
      if token(2) eq '[' then mbct=mbct+1; else if token(2)
      eq ']' then mbct=mbct -1;  end if; end else;
      save(token);; save(comp);  save(token); go to shufle; end multcomp;
```

```
      if nl(1) eq 'op' or (nl(1) eq 'uop') and (n2(2) eq ':'
      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;
   lbll: lbeg = f; bct=1; add(token); token=nt;
                  /*differentiate between label and '[a] op b=...'*/
      (while bct ge 1 doing token = nt;) if token(2) is tok
      eq '[' then bct = bct+1; else if tok eq ']' then bct=bct-1;;
      end else; 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); beg =t; go to getok;

conquit:              /*'continue...' or 'quit...' statement the
                        scope markers are added as datum to the token*/
      new = nt; contok = nult; ct=0;
      (while new(2) ne ';' doing new = nt; ct=ct+1;)
      if ct le 5 then contok = contok + <new(2)>;; end while;
      token(3) = token(3) + contok; add(token); token = new;
      go to semicolon;

leq:                  /*'=' at the begin of expression in 'iff-trailer'*/
      if beg then add(eql);  beg = f; go to getok; else go to cont;;

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

```
'if:                    /* we got an if we have to differentiate
                           between conditional expression and 'if statement'*/
      if n beg then ifstat = f; add(ifex); 1 stack ifsta;
      go to getok;;
      1 stack ifsta; new = nt; (while new(2) is tok ne ';'
      doing new = nt;)
      iff                          (tok eq 'else')?
                    (ifsta(#ifsta)=Ω;),     (tok eq 'if')?
                                        (1 stack ifsta;),  ℓ1; end iff;
      ℓ1: save(new); end while; save(new);
      if #ifsta eq 0 then ifstat = f; add (ifex);
      1 stack ifsta;  beg = f;  go to shufle;  else
      (1 ≤ ∀n ≤ #ifsta) ifsta(n) = Ω;; add(token); ifstat = t;
      beg = f; go to shufle;;
   end lex;
```

```
define save(token); lex external tstack; token stack tstack; end save;
define a stack b;  b(#b+1) = a;  end stack;
define add(token);  external lex1;  token stack lex1; end add;
define topoff blostack; blon=#blostack;  blotok=blostack(blon);
   blostack(blon)=Ω;
   iff                    (blotok(1) ne 'else')?
                  (return 1;),      (blostack(blon-1)(1) eq 'if')?
                        (blostack(blon-1)=Ω;return 2;),  (print
   'illegal else option', return 1;); end iff; end topoff;
define 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 nt; lex external defsta, tokstack, bras, lparset,rparset;
   initially name=;;
                /*provides the next token for lex by collecting
                  and expanding macros (definition before call),
                  reverting inverted procedure definitions and
                  storing them away in 'defstack' emitting a call
                  when necessary*/

main:  [; if #tokstack eq 0 then token = scan; else token =
       topo tokstack;; block tokget(token);-]
       <typ, tok,-> = token;
       iff     (tok eq 'block')?
           lblock,      (tok eq '[')?
                  brct,      (typ eq name)?
                   (macro(tok) is mac eq Ω)? (return token;),
                 (return token;), minvo; end iff;

brct:             /*look for inverted procedure definition*/
       tokget(new); if new(2) ne ';' then new stack tokstack;
       return token; end if;
       call = f; /* will become true if there is a call to be emitted*/
    bt: tokget(token); (while token(2) ne ';' doing tokget(token);)
                  /*accumulate procedure body*/
       resta(token);; resta(token); tokget(token); if bras(token(2))
       is labb ne Ω then go to labb;; resta(token); go to bl;

    block mget(a); /* we have just passed the keyword block
       and get the macro name and its arguments*/
       afn = nl; mok = t; tokget(token); mname=token(2);
       if token(1) ne name then print 'illegal macroname';
       mok = f;; tokget(new); if new(2) is tok eq ';' then
       go to a; /* i.e. macro has no arguments*/;
       if tok ne '(' then mok = f; print 'illegal beginning
       of macro argument list'; (while new(2) ne ';')
       tokget(new);; go to a; end if;
                  /* else correct starting, get arguments*/
       tokget(n1); tokget(n2);
```

```
   (while nl(1) eq name and n2(2) eq ',' doing
   tokget(nl); tokget(n2);)
   if afn(nl(2)) ne Ω then print 'illegal duplication
   of arguments'; mok = f; else afn(nl(2)) = #afn+1;;
   end while;
                /*check correct ending of argument list*/
   if afn(nl(2)) ne Ω then print 'illegal duplication of
   arguments'; mok = f; else afn(nl(2)) = #afn+1;;
   tokget(n3); if n2(2) eq ')' and n3(2) eq ';'
   then go to a; else mok = f;
   iff     (n2(2) eq ';')?
                  (n3 stack tokstack; go to a;),  (n3(2) eq ';')?
                                                a,  (tokget(new);
            (while new(2) ne ';')  tokget(new);; go to a;);
   end iff; end if;

   end mget;

blk:            /* we recognized the procedure as macro
                   definition*/
   mget(b4);
b4: if mok then macro(mname) = <afn,resta>;;
   tokget(new); /* is it invoked now*/  if new(2) eq '-'
   then (#resta ≥ ∀n ≥ 1) resta(n) stack tokstack;;
   tokget(new); end if;
   if new(2) ne ']' then print 'illegal ending
   of inverted macro definition'; new stack tokstack;;
   resta = nl; go to main;

block namget(a,b);
                /*get the name and the arguments of an inverted
                  function or subroutine definition */
   tokget(token); <typ,tok,-> = token;
   if typ eq 'uop' then pname = tok; calsta(token);
   go to b;; tokget(new); if new(1) eq 'uop' then
   pname = new(2); else pname = tok;; calsta(token);
   calsta(new);
 b: tokget(token); (while token(2) ne';' doing tokget(token);)
```

```
      calsta(token);; tokget(new); if new(2) eq '-' then
      call = t; tokget(new);;
      if new(2) ne ']' then print 'illegal ending of inverted
      procedure definition';reswi = t;  else reswi = f;;
      block reshufle(a); defsta(1) = <name,a>;
      copy (calstack, defsta ; defstk(semi); copy(restack,
      defsta);   defstk(end);  defstk(<name, pname>);
      defstk (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:          /*procedure in subroutine definition*/
      namget('define',b5);
   dfnf:          /*procedure in function definition*/
      namget('definef',b6);
   brkt:          /* function definition with no name*/
      calsta(<name, newat is pname>); call = t;
      reshufle('definef');

lblock:           /*we are at the beginning of a normal macro
                    definition, get name and arguments*/
      mget(lbl);
   lbl:           /*accumulate macro body*/
      tokget(token); (while n token(1) eq 'ef')
      iff              (token(2) eq 'end')?
                    tl?   (resta(token); tokget(token); continue while;),
             al, (resta(token); token=new; continue while;);
      tl: tokget(new); = new(2) eq mname;
      al: tokget(skip); if mok then macro(mname) =
        <afn, resta>;; resta=nl; go to main; end iff;
      end while; print 'illegal macro ending'; return token;

minvoc:           /*macro is being invoked*/
      nargs = 0; /* get the arguments */
      tokget(new); if new(2) is tok eq '(' then go to arg;;
```

```
   if tok eq ';' then go to expand;;  print 'illegal macro use';
   new stack tokstack; return token;
arg: argtok = nult; argstack = nl; tokget(token); mpart = 0;
argbeg:          /* get one argument and store it in argtok*/
   iff               (token(2) is tok ε lparset)?
          (mpart=mpart+1;),              (tok ε rparset)?
                        al, (tok eq ',' and mpart eq 0)?
                      argend, (tok eq ';' and mpart eq 0)?
                                merl, anad;
   al: mpart = mpart -1; if mpart eq -1 then argtok stack argstack;
     narg = narg+1; tokget(new); if new(2) ne ';'
     then token stack tokstack;; go to expand; end if mpart;
   argend: argtok stack argstack; narg = narg+1;
     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:          /*arguments are now on argstack and we will
                   expand macro*/
   <argfn, mbody> = mac;
   if narg eq 0 then (#body ≥ ∀n ≥ 1) mbody(n)
   stack tokstack; end #mbody; go to main; end if;
                /*macro has arguments*/
   (#mbody ≥ ∀n ≥ 1) token = mbody(n);
   if (argfn(token(2)) is argno) eq Ω 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 defstk(token);  lex external defsta;
   token stack defsta;   end defstk;
define calsta (token);  nt external calstack;
   token stack calstack; end calsta;
define copy(st1,st2) (1 ≤ ∀n ≤ #st1) st1(n) stack st2;;
   end copy;
definef topo stack; token = stack(#stack);
   stack(#stat) = Ω; return token; end topo;
```

Example of by 'lex' transformed Setl program

input to scan:

```
define at;  initially((a) o b = 1; if x then b else c
= if y then 2 else3;);
local; [; x=2; if if x then y else z then x = z; end if;
block b;-]
(1 ≤ ∀b ≤ 5)(while x doing z=ax;) til ℓ; <a,-> = u;
x = [o:cr] f(a);
if x then continue while;; x = a + [; y=5; return 2*y;]+c;
ℓ: a = e;  end 1 ≤  b; end at;
```

output of lex:

```
defs(define at; initially (((a) o b=1; if x then b else
c = if y then 2 else 3; )) end; elop local; x = 2;
if if x then y else z then (x = 2;) end;
foal (1 ≤  b ≤ 5) (whl (while x doing (z = ax;))) (<a,->=u;x=[o:ex]
comp f(a); if x then (continue;) end; x = a + name + c; defs(
define name; y = 5; return 2 * y;)end;) end; ℓ: a = e;) end;) end;
```

Items underlined free-hand are either marked of inserted by 'lex'.

SETL Newsletter Number 58a
Phase one of the SETL compiler -
Addenda and Errata

November 15, 1971
Kurt Maly

Page	Line	Addenda and Errata
4	8	<'op','elop'>,...
4	11	....<'op','foal'>;
4	18	...<'semicol',';'>;...
4	20	....<'thex','then'>;...
5	19	reverse order: ...tsta=nl; go to getok;
6	12	....skip=nt;....
6	25	....add(thex);
7	21	....then token(3) stack...
7	24	....then new(3) stack...
10	19	....eq ')' or token(1) eq 'rpar' doing token=nt;)
11	22	....skip=topoff blostack;
11	25	....action node....
12	31	....add(token);nl(1)='c'+nl(1); add(nl);add(n2);
13	18	insert the following between token(3)+contok; and add(token); : if n(#blostack≥] n≥1) | match(blostack(n),contok) then print 'illegal continue or quit statement';;
13	22	strike add(token);
16	20	if bras(token(2)) is labb ne ⌐ then go to labb;; which appears in lines 22 and 23 should also be added to the end of line 20
17	31	....pname=token(3);....
17	33	....pname=new(3);....
4	30	relat={'<.','<=','>.','>='};

Please replace pages 8 and 9 with the following three pages.

```
lparen:         /*left parenthesis*/
                / if not beginning of statement continue /
        if n beg then go to cont;; beg=f; part = 0;
                /*stack parentheses*/
        (while token(2) eq '(' doing token = nt;) token stack
        itsta;  part = part+1;  end while;
                /*check for immediately recognizable iteration header*/
        if iterset(token(2) is labi) ne ∩then go to labi;;
                /*else check for header with numerical range and
                  accumulate control token*/
        contok = nult; ct = 0; 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 + <tok>;;
        save(token); token = nt; partc=1;

        iff                     (tok eq '(' )?
                (part=part+1;partc=partc+1;), (tok eq ')' )?
                      (part=part-1;partc=partc-1;), (tok ∈ relat)?
                      (if token(2) eq '∀' then if ifswi then
                                              pnum=part-1;
                      partc=1;itfd=t;;end if token;),(tok eq ';')?
                           lpl, (tok eq 'if')?
                              (ifswi=t;), (tok eq 'then')?
                                  (ifswi=f;), (continue while;);
        end iff; end while;
        if itfd then contok stack blostack;
        (1≤∀n≤pnum) add (lpar);; add (foal);
        lpnum<∀n≤#itsta) add(itsta(n));; itsta=nl;
        save(lpar); save(token); go to shufle; end if;
        /*else expression on left hand side expression*/
        (1≤∀n<if itsta) add(itsta(n));; itsta=nl;
        save(token); go to shufle;
```

```
lpl:    /* parentheses are programmer used block markers
           and beginning of left-hand side expression*/
        (1≤∀n≤part) add(lpar);;
        (part<∀n≤#itsta) add(itsta(n));;itsta=nl;
        save(token); go to shufle;
fl:     /* '∀' set iteration header */
        part=1;
        contok=nult; ct=0; (while part ge 1 doing ct=ct+1; token=nt;)
        tok=token(2); if ct le 5 then contok=contok+<tok ;;
        if tok eq '(' then part=part+1; else if tok eq ')' then
        part=part-1;; end else; save(token); end while;
        (1≤∀n<#itsta) add(lpar);; add(foal);
        add(itsta(n)); itsta=nl; save(lpar);
        save(token); contok stack blostack; go to shufle;
wl:     /* while iteration header*/
        /* 'doswi' will become true if a 'doing' will appear*/
        part=1;
        doswi=f; contok=nult; ct=0; (while part ge 1) ct=ct+1;
        tok=token(2); if ct le 5 then contok=contok+<tok>;;save(token);
        iff   (tok eq '(')?
            (part=part+1;), (tok eq ')')?
                (part=part-1;), (tok eq 'doing' and n doswi)?
                                    al, wll;
        al:  doswi=t; save(lpar); /* save block of statement
                follows*/ end iff;

wll:    token=nt; end while;
        contok stack blostack;
        (1≤∀n<#itsta) add(lpar);; add(whl);
        add(lpar); itsta=nl; tstack(#tstack)=rpar;
        if doswi then /* 'doing' occurred must omit block marker*/
             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≤#itsta)
```

```
        add(itsta(n)); itsta(n)=↑; add(token); add(new);
        (<token(2)>+<new(2)>) stack blostack; go to latl; end atstr;

lat:                        /* 'at label' statement */
        atstr(atl);
```