SETL Newsletter # 61 — November 15, 1971

Syntactic Structure of Setl — Kurt Maly

What follows is the syntactic structure of SETL as described in the Postparse metalanguage. It might be worthwhile to remind the reader that the structure therein described is not that of SETL but rather a language into which SETL is transformed by 'lex'. (See Newsletter 58.) The main difference being an explicitly marked block structure in this new language.

As can be seen the 'iff statement' is at this point already in a rather nice tree form which certainly cannot be produced by the programmer. An algorithm producing this tree has yet to be written. At this point we might as well delve into the current status. The procedure tables for the preparser, modifications to the preparser, nextoken, a one line procedure, and scan, the lexical scanner have to be written. Concurrently the parser package for the postparse metalanguage has to be debugged completely. Whereafter the verification and compilation of the following SETL grammar can be done. The side products, namely tables, of this process together with the above mentioned programs will finally allow us to parse SETL programs.

Since the grammar is somewhat lengthy a concordance for the productions is appended at the end.

```
                /th/10
                ///
program =:      block.
                     /a0/ procveds=nl; /*on top of this stack is,for the
                current procedure,the following information: <subroutine or
                function, return statement found yes or no, normal (not external,
                local, or  initial) statement processed  yes or no, within
                store block, entry point to informtree>*/
                informtree = nl; /* contains for each procedure a root node
                from which descend its definition, the symbol table, the
                data table, labels, external statement,local  statement and
                all inner procedures*/
```

```
          iftac = nl; <'sbrt',f,f,f,newat is defpro>stack  proceds;
          formsetup(defpro);
          define formsetup(node); parser external informtree;
          postparse external proceds; <node,1,newat> in informtree;
          (1 < ∀n ≤ 4)   <node,n,nl> in informtree;;
          (5 ≤ ∀n ≤ 6)   <node,n,newat> in informtree;;
          /* set up initial structure of procedure subtree */
          return; end formsetup;
          define a stack b; b(#b+1) = a, return; end stack;
          ///
    block =: ';' statement.1 / statement block. subpart./10
               /a1/ if desc(:1,#desc(:1) is denum)(1:2) eq
          <'var', 'omitted'> then desc = desc lesf <:1, denum>;;
        =: 'LPARRPAR' block. / parenthesized block./
          ///
statement =: 'LABEL'.[label] labl. stat./ statement. subpart. labelled
          statement./10
          stat.
          ///
   labl [ =: ':' [name] 'omitted'/ label.token../ 10,2
               /a2/ desc = desc lesf <:1,2>;
          define labels(nam); parser external informtree;
          postparse external proceds; ok  entry=(topof proceds)(5);
          if nam ε informtree(entry,4) then ok=f; else
          informtree(entry,4) = informtree(entry,4) with nam;;
          /* check for duplication and enter in appropriate label set */
          return; end labels;
               /t2/ labels (desc(:1,1)(2));, 'illegal duplication of label'
       =:] '[ ]' labl. / bracketed label./10
          ///
     stat =: 'IFEND' (exp. 'THEN' (('LPARRPAR' block.) 'ELSE'('LPARRPAR'
          block.))) / single statement. operator. long if statement./
          8,5,1,5,1
        =: 'IFEND' (exp. 'THEN' ('LPARRPAR' block.)) / short if
          statement. / 8,5,2
```

```
=:  'ATLEND' ('LPARRPAR' ('( )' ('AT'[name])) block.)
    /at block. / 8,1,2,7
         /a2/ symbol(desc(:4,1) );
    define symbol(nam); parser external informtree;
    postparse external proceds,  entry=((topof proceds)(5);
    informtree(entry,2) = informtree(entry,2) with nam(2);
    /* add name to appropriate symbol table */
    return; end symbol;
    definef topof  stack; return stack(#stack); end topof;
    definef tk nd; return nd(2); end tk;
=: 'STREND' ('LPARRPAR' ( '( )'('STORE'[name]))block.)
    /store block. / 8,1,2,7
         /a1/ proceds (#proceds)(4)=t; /*set store block flag*/
         /a2/ proceds(#proceds)(4) = f; symbol(desc(:4,1) );
=: 'LODEND'('LPARRPAR'( '( )' 'LOAD') block.)/load block./8,1,2,7
=: 'INITIALEND'('LPARRPAR' block.) /initial block./9,1
         /a1/ if (topof proceds)(5) then ok2 = f; else ok2=t;;
         /t2/ if n ok2 then ok = f;;, 'illegal statement
    before initial statement'
    /* only name scope changing statements allowed before initial
    statement*/
=:  'WHLEND' ('LPARRPAR' ('LPARRPAR' ('WHILE' whlpart.))
    block.) / while statement. / 8,1,1,8
=:  'FOALEND' ('LPARRPAR' ( '( )' (iter.  '|' exp.))block.)
    /conditional iteration statement. /8,1,1,8
=:  'FOALEND' ['LPARRPAR' ( '( )' iter.) block.) /simple
    iteration statement./8,2,2
=:  'DEFFEND' ('LPARRPAR' dblock.) /function definition./10
         /a1/ <'fn',f,f,f, newat is defproc> stack proceds;
    entproc (defproc); /*start new entry in procedure
    stack, and create corresponding entry in informtree */
    define entproc(node); parser external informtree;
    postparse external proceds; entry=proceds(#proceds-1)(5);
    informtree(entry, #informtree(entry)+1) = node; formsetup(node);
    /* add procedure node to containing procedure and build initial
    structure of this procedure tree */ end entproc;
```

```
        /t2/ ok= (topoff proceds)(2);, 'missing return statement
    in function definition'
=:  'DEFSEND' ('LPARRPAR' dblock.) / subroutine definition./10
        /a1/ <'sbrt', f,f,f, newat is defproc> stack proceds;
    entproc(defproc);
    definef topoff stack; elt = stack(#stack); stack(#stack)=Ω;
    return elt; end topoff;
        /t2/ ok = (topoff proceds)(2);, 'missing return
    statement in subroutine definition'
=:  'IFFEND' ('LPARRPAR' (';' header.  trailer.1))/long
    iff statement./8,2,2
        /t2/ (1 < ∀n < #desc(:3)) if nodtype(desc (:3,n))
    ε {'to', 'eql'} then ok = nodtype (desc(:3,n+1)) eq 'label';
    end if;;, 'illegal position of "to statement" or test
    description in iff trailer'
        /a2/ if desc(:3, #desc(:3) is  desum) (1:2) eq
    <'var', 'omitted'> then desc = desc lesf <:3,desum >;;
=:  'IFFEND' ('LPARRPAR' ( ';' header. 'OMITTED'))
    /short iff statement. / 8,2,2,1
        /a2/ desc = desc lesf <:3,2>;;
    /* those above are compound statements whereas  those
    following are simple statements not containing another
    statement*/
[=: '=' ('< >'(',' multpart.2)) exp./multiple assignment
    statement./8,2,2
        /a2/ [; if n (topof proceds)(3) then proceds(#proceds)(3)=t;;
    block statseq;-] /*this being a 'normal' statement the
    corresponding flag on top of the procedure stack is set */
=:] exp. '=' exp. /essignment statement./10
        /a2/ statseq;
=:  exp. '=' 'Ω'/ assignment statement./ 5,5
        /a2/ statseq;
=:  'IN' exp. [name] / set enlargement. /10
        /a2/ statseq; symbol(desc(:1,2));
=:  'GO' ('TO' exp.) /goto statement./ 10,5
        /a2/ statseq;
[=: 'READ' 'INPUT' cname. / input read statement./10
        /a2/ statseq;
```

```
=:] 'READ' [name] cname. / file read statement. /10
        /a2/ statseq; symbol (desc(:1,1));
[=:  'PRINT' 'OUTPUT' cexp. / output print statement./ 5,5
        /a2/ statseq;
=:] 'PRINT' [name] cexp. /file print statement./10
        /a2/ statseq; symbol(desc(:1,1));
=:  '( )' [name] sexp. /subroutine call./10
        /a2/ statseq; symbol(desc(:1,1));
=:  '[ ]' [name] cexp. /iterated subroutine call./10
        /a2/ statseq; symbol(desc(:1,1));
=:  'ELOP' [name] [local] / short local statement. /10
        /t2/ if (topof proceds)(3) then ok=f;; 'illegal
    statement before local statement'
        /a2/ symbol(desc(:1,1)); locals(:1);
    define locals (node); parser external informtree;
    postparse external proceds; entry = (topof proceds)(5);
    top = informtree(entry,6); <top,#informtree(top)+1,node> in
    informtree; /* enter root of local statement with respect
    to 'desc' into appropriate place in informtree */
    return; end locals;
=:  'ELOP' [local] /abbreviated local statement./ 10
        /t2/ if(topof proceds)(3) then ok = f;;, 'illegal
    statement before local statement'
        /a2/ locals(:1);
=:  'ELOP' [local] cname. /local statement./10
        /t2/ if (topof proceds)(3) then ok = f;;,'illegal
    statement before local statement'
        /a2/ locals(:1);
=:  'ELOP' [name] [local] cname. / long local statement. /10
        /t2/ if (topof proceds)(3) then ok = f;;, 'illegal
    statement before local statement'
        /a2/ symbol(desc(:1,1)); locals(:1);
=:  'ELOP' [name][external] cname2./external statement./10
        /t2/ if (topof proceds)(3) then ok = f;;, 'illegal
    statement before external statement'
```

```
          /a2/ symbol(desc(:1,1)); externals(:1);
        define externals(node); parser external informtree;
        postparse external proceds; entry = (topof proceds)(5);
        top = informtree(entry,5); <top,#informtree(top)+1,node> in
        informtree;  /*enter root of external statement with respect
        to 'desc' into appropriate place in informtree */
        return; end externals;
    =:  'ELOP' [external] cname2. / simple external statement./10
          /t2/ if(topof proceds)(3) then ok = f;;, 'illegal
        statement before external statement'
          /a2/ externals (:1);
    =:  'RETURN' exp. /function return statement./10
          /t2/ if topoff proceds is flags eq Ω then ok=f;;,
        'illegal ending of main program',
        if flags(1) eq 'sbrt' then ok = f;; flags(2) = t;
        flags stack proceds;, 'illegal return within subroutine',
        if flags(4) then ok = f;; flags stack proceds;,
        'illegal return within store block'
          /a2/ statseq;
    =:  'RETURN'
          /t2/ if topoff proceds is flags ne Ω then
        iff                     (flags(1) eq 'fn')?
                     (flags(4))?              a1,
              a1,  (ok=f; flags(2)=t; flags stack proceds;);
        a1:  flags(2) = t; flags stack proceds; end iff;
        end if topoff;, 'illegal return within function'
          /a2/ statseq;
    =:  {'CONTINUE','QUIT'}
        ///
defstat =:  {'DEFINEF','DEFINE'} ('( )' [name][name])/procedure
        definition statement./8,2
          /a2/ definit(desc(:2,1)(2), if nodtgre(:1) eq 'define'
        then 'sbrt' else 'fn', {desc(:2,2)(2)}); symbol(desc(:2,1));
        symbol(desc(:2,2));
```

```
          define definit(name,code, arg); parser external
          informtree; postparse external proceds; entry=(topof proceds)(5);
          top = informtree(entry,1); <top,1,name> in informtree;
          <top,2,code> in informtree; <top,3,arg> in informtree;
          /* enter name,  code and arguments of procedure in
          appropriate place in informtree */
          return; end definit;
      =:  {'DEFINEF','DEFINE'}( '( )'[name]  [','[name].2))
          /procedure definition statement./ 8,2,2
               /a2/ definit (desc(:2,1)(2) is pname, if nodtype(:1)
          eq 'define' then 'sbrt' else 'fn', {desc(:4,k)(2),
          1 < k < #desc(:4)} is args); symbol(pname); symbol[args];
      =:  {'DEFINEF','DEFINE'} [name] / procedure definition statement./10
               /a2/ definit (desc(:1,1)(2) is pname, if nodtype(:1)
          eq 'define' then 'sbrt' else 'fn', nl); symbol(pname);
      =:  {'DEFINEF','DEFINE'} ('UOP'.[uop] [name].1)
          /user defined procedure statement./ 8,2
               /a2/ definit(nodtype(:2)(3) is pname,
          if nodtype(:1) eq 'define' then 'suop' else 'fuop',
          {desc(:2,k), 1 < k < #desc(:2)} is args); symbol(pname),
          symbol[args];
          ///
multpart =:  'MOP' [skip] / part of multiple assignment statement.
          operator. skip operator. /10
      =:  [name]
               /a2/ symbol( :1);
      =:  exp.
          ///
 whlpart =:  exp. 'WHEN' (exp. 'DOING' ('LPARRPAR' block.))
          /while iteration header. operator. complete iteration./ 8,8,2
      =:  exp. 'DOING' ('LPARRPAR' block.) / doing option. /8,2
      =:  exp. 'WHEN' exp. /when option. /10
      =:  exp.
          ///
```

```
   iter =:  ',' ielt' elt.1/iteration header. subpart. long form. /10
        =:  ielt.
            ///
   ielt =:  '∀' ('ε'[name] exp.) /range restriction. operator.set
            iteration. /8,2
                 /a2/ symbol(desc(:2,1));
        =:  {'>=', '>', '<', '<='} exp. ({'>=', '>', '<', '<='}
            ('∀'[name])exp.) / numerical range. /5,5,5
                 /t2/ ok=[; if  nodtype(:1) ε {'<=','<'} then
            nodtype(:3)  ε {'<=','<'} else nodtype(:3)  ε {'>=','>'};
            block rancheck;-], 'illegal numerical range restriction
            in iteration header'
                 /a2/ symbol(desc(:4,1));
            ///
 header =:  test. '?' desc. / iffheader. subpart../10
            ///
   test =:  'LPARRPAR' exp. / testnode. subpart. explicit testnode./10
        =:  [name]
                 /a2/ symbol ( :1);
            ///
   desc =:  'CATENATION' header.2  / testnode descendants. subpart.
            two testnodes. /10
        =:  ',' header. action. / mixed descendants. /10
        =:  ',' action. header. /mixed descendants. /10
        =:  ',' action.2 /  two action nodes. / 10
            ///
 action =:  'LPARRPAR' iffblock. /action node-subpart. explicit
            action node. /10
        =:  [name]
                 /a2/ symbol ( :1);
            ///
trailer =:  'LABEL'.[label] labl.('EQL'.[eql] eqs.)/ trailer statement.
            operator. labelled testnode . / 8,2
        =:  'TO' exp. / to statement. /10
        =:  'EQL'.[eql] exp. / test expression. /10
        =:  statement.
        =:  'OMITTED'.[var]
            ///
```

```
  dblock =:  ';' defstat. statement.1 / procedure block. subpart../10
                /a1/ if desc(:1,#desc(:1))(1:2) eq<'var','omitted'>
             then desc = desc lesf <:1,#desc(:1)>;;
                /a2/ if n (topof proceds)(2) then
             if (topof proceds)(1) eq 'fn' then <:1,#desc(:1)+1,
             newat is nodn> in desc; <nodn,1,<'var','Ω'>> in desc;
             <nodn, 'return'> in nodtype; /* a return statement
             within a function is missing we insert a 'return Ω' in the
             trace */  else <:1,#desc(:1)+1, 'return'> in desc;
             /* for missing subroutine return insert 'return' */;
             end if n;
             ///
iffblock =:  ';' iffstat.1 / action. subpart../10
                /a1/ f stack iftac; if desc (:1,#desc(:1) is deno)(1:2)
             eq <'var','omitted'> then desc = desc lesf <:1,deno>;;
                /t2/ if topoff iftact  then ok = nodtype (desc (:1,
             #desc(:1))) eq 'to';;, 'illegal position of to statement'
             ///
 iffstat =:  'TO' exp. / action statement. subpart. to statement. /10
                /a2/ iftac(#iftac) = t;
         =:  statement.
             ///
   cname =:  ',' [name].2 / list of names. delimiter../10
                /a2/ symbol [{desc(:1,k), 1 ≤ k ≤ #desc(:1)}];
         =:  [name]
                /a2/ symbol (:1);
             ///
    sexp =:  ',' sexp1.2 / argument list. delimiter../ 10
         =:  exp.
             ///
   sexp1 =:  '[ ]' (',' exp.2) / set argument iteration. delimiter.
             list of sets. / 8,2
         =:  '[ ]' exp. / set argument. / 10
         =:  exp.
             ///
  cname2 =:  ',' cname3.2  / external name list. delimiter../ 10
         =:  cname3.
             ///
```

```
cname3 =:  '( )' (',' [name].2) / external name list. delimiter.
           external name mapping. / 8,2
                /a2/ symbol [{desc(:2,1), desc(:2,2)}];
       =:  [name]
                /a2/ symbol(:1);
           ///
   exp =:  {'∍', 'HD', 'TL', '#', 'ABS', 'BITR', 'FLOOR', 'CEILING',
           'NOT', 'N', 'DEC', 'OCT', 'HOL','COMPILE', 'TYPE',
           'ATOM', 'PAIR'} expl. / expression. operator or some token.
           monadic expression. /10
       =:  {'+','-'} expl. / monadic arithmetic expression. /10
      [=:  {'EQ','NE'} {'Ω','SET','INTEGER','TUPL','BSTRING',
           'CSTRING','LABEL','BLANK','REAL','SUBROUTINE',
           'FUNCTION'}·[objtype] exp. / type test. / 8,2
       =:  {'EQ','NE'} exp. {'Ω','SET','INTEGER','TUPL','BSTRING'
           'CSTRING','LABEL','BLANK','REAL','SUBROUTINE',
           'FUNCTION'}·[objtype] / type test. /8,2
       =:] {'EQ','NE','LT','GT','LE','GE','MAX','MIN','*','/','//',
           '+','-','EXP','LOG','WITH','LESS','LESF','AND','OR',
           'nε','ε','U','IS'} expl.2 / diadic expression. /10
       =:  'UOP'.[uop] expl.2 / dyadic expression with user defined
           operator. / 10
                /a2/ symbol (nodtype(:1)(2:2));
       =:  'UOP'.[uop] expl. / monadic expression with user defined
           operator. /10
                /a2/ symbol (nodtype(:1)(2:2)); /* whenever the lexical
           scanner finds a user defined operator it creates the token
           <'uop','uop',name> */
       =:  'COMP' ('[ ]' (':'[cop,cuop] range. )) exp. / compound
           operator expression. /8,2,2
                /a2/ if desc(:3,1)(1) eq 'cuop' then /* operator
           is user defined */ symbol(desc(:3,1)(2:2));;
       =:  'AS' [name][objtype] / type conversion expression. /10
                /a2/ symbol (desc(:1,1));
       =:  quantifier.'|' exp. / quantified boolean expression. /10
```

```
        =:  'IF'.[ifex] (exp. 'THEN'.[thex]  (exp. 'ELSE'.[elsex] exp.))
            /conditional expression. /8,5,5
       [=:  '{ }' range. / formed set expression. /10
                 /ob/2
        =:  '{ }' (',' exp.2   / enumerated  set expression. /8,2
        =:] '{ }' exp. /simple set. /10
        =:  '{ }' (srange. '|' exp.) / set expression. /8,2
        =:  '< >' (',' exp.2) / tuple expression. /8,2
        =:  '< >' exp. / tuple. / 10
        =:  '( )' exp. / parenthesized expression. /10
        =:  [name]
                 /a2/ symbol(:1);
        =:  [integer, real]
                 /a2/ data(dec tk :1) /* store actual number */
            define data(node); parser external informtree,
            postparse external proceds; entry = (topof proceds)(5);
            informtree(entry,3) = informtree(entry,3) with node;
            /add datum   constant to appropriate table */
            return; end data;
        =:  [boolean]
                 /a2/ data (oct tk :1); /* store octal constant */
        =:  [string]
                 /a2/ data (tk :1);
        =:  {'NEWAT','NULL','NULT','NULC','T','F','TRUE','NL',
            'FALSE','HOLL'}.[nullop]
        =:  indname.
            ///
    expl =:  '[ ]'exp.  / operand. operator. iteration operand. / 10
        =:  exp.
            ///
   range =:  restrict. '|' exp. / range. delimiter. conditional range./10
        =:  restrict.
            ///
restrict =:  ',' elt.2/restriction sequence. delimiter../10
        =:  elt.
            ///
```

```
      elt =:  'ε' [name] exp. / setformer iteration. subpart../ 10
                   /a2/ symbol(desc(:1,1));
          =:  {'<=','<','>=','>'} exp. ({'<=','<','>=','>'} [name]
              exp.) / numerical range restriction. /5,5
                   /t2/ ok = rancheck;, 'illegal numerical range
              restriction'
              ///
quantifier=:  ',' quelt.2 / quantifier. delimiter. quantifier sequence./10
          =:  quelt.
              ///
    quelt =:  '∃'(qname.'ε' exp.) / quantified boolean element
              subpart. existential quantified element. / 8,2
          =: '∀' ('ε' [name] exp.) / universal quantified element. / 8,2
                   /a2/ symbol(desc(:2,1));
          =: {'<=','<','>=','>'} exp. ({'<=','<','>=','>'} ('∀'[name])exp.)
              / universal numerical range. / 4,4,8
                   /t2/ ok = rancheck;, 'illegal numerical range
              restriction in quantified boolean expression'
                   /a2/ symbol (desc(:4,1));
          =:  {'<=','<','>=','>'} exp. ({'<=','<','>=','>'} ('∃'qname.)exp.)
              /existential numerical range./ 4,4,8
                   /t2/ ok = rancheck;, 'illegal numerical range
              restriction in quantified boolean expression'
          =:  elt.
              ///
    qname =:  '[ ]' [name] / search variable. token. search with
              assignment. / 10
                   /a2/ symbol (desc(:1,1));
          =:  [name]
                   /a2/ symbol (:1);
              ///
   srange =:  exp. ',' elt.1  / setrange. subpart. sequence of restrictions./10
          =:  elt.
              ///
```

```
indname =:  '{ }' exp. multexp. / functional application. parenthesis.
            image set. / 10
        =:  '[ ]' exp. cexp. / image set of set. / 10
        =:  '( )' exp. (':' exp.1) / indexed tuple or string. /5,5
                /a1/ if desc(:3,#desc(:3) is des )(1:2) eq
            <'var','omitted'> then desc = desc lesf <:3,des >;;
                /t2/ ok = #desc(:3) lt 3;, 'illegal number of indices'
        =:  '( )' exp. multexp. / image point. / 10
        =:  '( )' ('POW' exp.) / power set. / 8,8
        =:  '( )'('NPOW'(','exp.2)) / restricted power set. / 2,8,8
            ///
multexp =:  ',' exp2.2 / arguments. delimiter, sequence of arguments. / 10
        =:  exp2.
            ///
   exp2 =:  '[ ]'(',' exp.2) / argument set. delimiter sequence of sets../5,5
        =:  '[ ]' exp. / single argument set. / 10
        =:  exp.
            ///
   cexp =:  ',' exp.2 / list of expressions. delimiter../ 10
        =:  exp.
            ///
  /end/
```

Note: a lexically specified literal in the tree-describer part is written as: 'literal'.[lexical type]

Concordance for the Productions