SETL Newsletter Number 43 July 13, 1971

A Parsing Scheme for FORTRAN Samson Gruber

This newsletter describes a parsing scheme for FORTRAN (in particular, System/360 Fortran H restricted). It embodies the three phases found in the SETL notes as well as an additional fourth phase, here denoted "kludge", which handles some of FORTRAN's more difficult peculiarities.

In particular, the kludge routine has the following functions:

1) To handle continuation cards and statement numbers: This is necessitated by the fact that nowhere else in the parse are we aware directly of column numbers.

2) To convert the input string to a sequence of characters.

3) To reduce both quoted and h literals to the canonical form: 'number literal, where number is a three character string whose value is dec of the number of characters in the catenated literal. Although this could be done in the nextoken routine, it is necessary for what follows and so to do it there would be redundant.

4) To append a semicolon at the end of all statements but the last, to which a double semicolon is appended. These semicolons disappear after nextoken.

5) To surround all keywords, and only all keywords, with pound signs (#). These, too, go away after nextoken.

6) To insert the special word 'with' between the two halves of a logical if statement.

It works by successively copying the input statement, with modifications each time, from a sequence called sq to a sequence called sql. Since each phase may be called with either an even or ar odd number of previous copies, each uses the routines in and out to characters from the current input sequence and put characters on the current output sequence, respectively. The routine nextchar returns the next character on the input sequence which is not part of a literal and converts any literals it finds to the canonical form, thus accomplishing purpose three. The algorithm used to ascertain whether a statement is an assignment statement consists of scanning for unparenthesized equal signs and commas. If only the former occur the statement is an assignment statement. The code for kludge follows.

```
defiref kludge; initially alph = {'a','b','c','d','e','f','g','h','i',
        'j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y'
        'z'};  s = <input, 1>; string = record (s)::
        litsdone = f; /* signal literals not processed */
        seq = nl; ff = t; /* signal beginning of statement */
        go to jumpin;
read:   string = record (s);
jumpin: if string eq nullc then return (process(seq) withs ':';;
        if (1 first string) eq 'c' then goto read;;
        if ff then ff=f; seq = seq withs ((len string-6) last
        string); (1≤∀i≤5) if ( i elt string ne ' ' and
           (dec( i elt string) eq∩) then print
         'illegal statement number'; quit; end if; end ∀i;
         go to read; end if;
         if 6 elt string eq ' ' then return
         process (seq) withs ':' :: seq = seq withs
         ((len string- 6) last string); go to read;
```

```
definef sa withs st;
        (1≤∀j≤len st); sa(#sa+1) = j elt st;: return sa; end withs;
definef process (sa);
        kludge external alph, litsdone; nextoken external instat;
        preparse external stype;
        state = 'start'; stype = 0; instat='nxt';in=read;out=writel;
        erase=clearl;call erase;flag=t;
define switch;process external in, out, erase, flag, cur,
     readl, writel, read, write, clear, clearl;
     <in,out,erase,flag> ={<t,<readl,write,clear,f>>,
     <f,<read,writel,erasel,t>>} (flag);call erase;
      cur=0; parent=0; end switch;
definef read; process external sa, cur;
        (while c is sa(cur is (cur+1)) eq ' '); return c; end read;
definef readl; process external sal, cur;
        (while c is sal (cur is(cur+1)) eq ' ');return c; end readl;
define write(char); process external sa; if char ne ↵ then
        sa(#sa+1) = char;; return; end write;

define writel(char);process external sal; if char ne ↵then

        sa(#sa+1)=char;; return;end writel;

define clear; process external sa; sa=nl; return; end clear;
define clearl; process external sal; sal=nl; return; end clearl;
definef length; process external sa, sal; return(if flag
                        then #sa else #sal); end length;
definef ms str; return {<n,n elt str>, 1≤n≤len str}; end ms;
definef outret c; process external out; call out(c); return c;
                        end outret;
```

```
definef par c; process external parcnt;
        if n is ({<'(',1>, <')',-1>}(c)) ne Λ then
        parcnt = parcnt+n;; return c; end par;
define finish; c = nextchar; (while c ne Λ) call out(c); c=nextchar;;
                        call switch; end finish;
definef nextchar; /* literal handler: returns next character in
        input which is not part of a literal.  Outputs any
        intervening literals */
        process external cur, in, out, outret, state, clear,
                        litsdone, par;
        initially strl = nulc;;
        go to {<t,nc>,<f,ncl>} (litsdone);
nc:     char=in; if char ne '''' then return
        par (char);;
        call out(char); str=nulc; (∀i∈{1,2,3})
        str=str+outret in;;
        (1≤∀j≤dec str) call out(in);; goto nc;
ncl:    if strl ne nulc then char=1 elt strl;
        strl = (len strl-1) last strl; return char;;
        char = in; if char eq Λ then litsdone=t; return(char);;
        if char eq '''' then goto quotlit;;
        /* continue check for h literal */
        definef type(c); return if t is ({<dec n, 'dig'>,0≤n≤9} u
        {<x,'delim'>,x∈{'+','-','*','/','=',',','(',')'}}
                        u{<'p','pos let'>,
        <'x','poslet'>,<'h','h'>} )(c) eq Λ then 'other' else t;
        end type;
```

```
        state= {<'start','dig','start'>,<'start','delim','delim'>,
               <'start','poslet','start'>,<'start','h','start'>,
               <'start','other','start'>,<'delim','dig','check'>,
               <'delim','delim','delim'>,<'delim','poslet','start'>,
               <'delim','h','start'>,<'delim','other','start'>}
                (state,type(char)); if state ne 'check'
               then return par (char);;
        strl=nulc; save=cur; (while type(char eq 'dig');
               strl=strl+char; char=in;; goto{<'h','hlit'>,
        <'poslet','saveup'>,<'other','restore'>,<'delim','restore'>}
               (type(char));
restcre: char=1 elt strl; strl=nulc; cur=save; state='start';
               return char;
saveup:  strl=strl+char; state='delim'; go to ncl;
hlit:    call out(''''); (1≤∀i≤3) call out (i elt(3 last('000'+strl)));
         (1≤∀i≤dec strl); call out(in);; state='delim';strl=nulc;
         go to ncl;
quotlit: j=0; str=nulc; char=in;
loop:    (while char ne '''')j=j+1; str=str+char;
         char=in;; if in eq '''' then j=j+1; str=str+'''';
         char=in; go to loop;; cur=cur-1;
         strl=3 last('000'+dec j); call out('''');
         (1≤∀i≤3) call out(i elt strl);;
         (1≤∀i≤len str) call out (i elt str);;
        state='delim'; strl=nulc; go to ncl;
               end nextchar;
        definef n subseq seq; return {<k,seq(k+n-1)>,
               1≤k≤#seq-n+1}; end subseq;
        definef sl withseq s2; return sl u {<k,sq(k-#sl)>,
              #sl+1≤k≤#sl+#s2}; end withseq;
        /* interesting execution starts here */
        /* check for if( */
```

```
            if c is outret nextchar ne 'i' then goto norm;;
            if c is outret nextchar ne 'f' then goto norm;;
            if c is outret nextchar ne '(' then goto norm;;
          /* we may be looking at an if statement, or
            maybe at if(i)=2. scan for ) to see */
            (while c ne ')' or parent ne 0)c = outret nextchar;;
            c=nextchar; if c eq '=' then goto ret;;
            /* i.e., if this is an assignment statement */
            if n(c ∈ alph) /* arithmetic if, do nothing
                    special */ then goto norm;;
            return (ms '#if#') withseg (3 subseg
            (if flag then sgl else sg)) withseg
            (ms '#with#') withseg (process(cur subseg(
            if flag then sg  else sgl)));
norm:       com=f; eg  =f;(while c ne ↑);
            if parent eg 0 then if c eg '=' then eg  =t;
            else if c=',' then com=t;; end else;
            end if parent; c=outret nextchar; end while c;
            if n com and eg  then goto ret;
            call switch; call out ('#');
nl:         if n ((c is nextchar) ∈ alph) then call out (c); goto reg;;
            call out (c);
            if (ℓ is {<ms '#data',decl>, <ms '#integer',decℓ>,
                      <ms '#real',decl>, <ms '#logical',decl>,
                      <ms '#dimension',reg>, <ms'#common',decl>,
                      <ms '#external',reg>, <ms'#subroutine',decl>,
                      <ms '#entry',reg>, <ms'#implicit',impl>,
                      <ms '#format',form>,<ms '#assign',asgn>,
                      <ms '#backspace',reg>, <ms'#call',reg>,
                      <ms '#endfile',reg>,<ms '#goto',reg>,
                      <ms '#print',reg>,<ms '#punch',reg>,
                      <ms '#read',reg>, <ms '#return',reg>,
                      <ms '#rewind',reg>, <ms'#write',reg>}
                      (if flag then sgl else sg)) ne ↑then goto ℓ:
                      goto nl;
```

```
form:   instat=formnxt;
reg:    call out ('#');
ret:    call finish; return if flag then sq else sql;
decl:   call out ('#'); save=cur;
        if nextchar ne 'f' then goto dout;;
        if nextchar ne 'u' then goto dout;;
        if nextchar ne 'n' then goto dout;;
        if nextchar ne 'c' then goto dout;;
        if nextchar ne 't' then goto dout;;
        if nextchar ne 'i' then goto dout;;
        if nextchar ne 'o' then goto dout;;
        if nextchar ne 'n' then goto dout;;
        (1≤∀i≤10) call out (i elt '#function#');; goto ret;
dout:   cur=save; stype=1;goto ret;
asgn:   call out('#'); c=nextchar;
              (while c∈{'0','1','2','3','4','5','6','7','8','9'})
        call out (c); c=nextchar;; call out ('#');
        (while c∈alph) call out(c); c=nextchar; cur=cur-1;
        go to reg;
impl:   call out ('#'); c=nextchar;
loop:   call out ('#'); (while c∈alph) call out(c);
        c=nextchar;; call out ('#');
        (while n (c∈alph)or
               (parent ne 0) doing if c eq /\ then stype=2;goto ret;;)
        call out(c); c=nextchar;; goto loop;
        end process; end kludge;
```

The states used in nextoken have the following significance:

nxt - the initial state

key - having seen a #, look for closing #

nam - scanning a name

int - scanning some sort of number

de ρop - having seen a '.', scanning either a decimal or a period delimited operator

star - having seen a '*', scanning either a multiplication or exponentiation symbol

intpt - we have seen the combination integer

ptint - we have seen the combination . integer

decl - we have seen integer integer

exps - while in int, intpt, ptint, or decl, we have seen an E, possibly an exponential form. Check for sign.

expl - this may not be an exponential form, we may have to do fixup.

exp - this is an exponential form, on error stop.

formnxt - the initial state when scanning a format statement.

vl - scanning a letter

il - scanning an integer or a decimal

dec2 - having seen integer ., maybe scanning valid decimal

dec - having seen integer. integer, scanning a valid decimal.

The definitions of character types and the state table follow:

```
     ['''', '=', 'a', 'l', '+', '.', 'E', 'er', '*', '/']
{<'''',''''>,<'#', '#'>,<'n', 'ABCDFGHIJKLMNOPQRSTUVWXYZ'>,
 <'l', '0123456789'>, <'+','+-'>,<'.','.'>,
 <'E','E'>, <'*', '*'>, <'/', '/=(),'>}
```

state	ctype	'	#	a	l	+	.	er	E	*	/
nxt	do lit/litend end	do litend end	do key go key	go nam	go int	do opend end	go dec-pop	do erend end	go var	go star	do opend end
key		do lit cont	do key end	cont	cont	cont	cont	cont	cont	cont	cont
nam		end	end	cont	cont	end	end	end	end	end	end
int		end	end	end	cont	end	go intpt	end	do emaybe go exps	end	end
decpop		end	end	go pop	go ptint	end	end	end	go pop	end	end
star		end	end	end	end	end	end	end	end	do opend end	end
intpt		end	end	do afix end	go decl	end	end	end	do emaybel go exps	end	end
ptint		end	end	end	cont	end	end	end	do emaybe go exps	end	end
decl		end	end	end	cont	end	end	end	do emaybe go exps	end	end
exps		do efix end	do efix end	do efix end	go exp	go expl	do efix end	do efix end	do efix end	do efix end	do efix end
expl		do efix end	do efix end	do efix end	go exp	do efix end	do efix end	do efix end	do efix end	do efix end	do efix end
exp		end	end	end	cont	end	end	end	end	end	end
formnxt		do lit do litend	do key go key	go vl	go il	do opend end	do opend end	do erend end	go vl	go star	do opend end
vl		end	end	cont	end	end	end	end	end	end	end
il		end	end	end	cont	end	go dec2	end	end	end	end
dec2		end	end	end	go dec	end	end	end	end	end	end
dec		end	end	end	cont	end	end	end	end	end	end
pop		end	end	cont	cont	end	do opend end	end	cont	end	end

```
<lit, '(curpointer  ≤ ∀j ≤ k is(curpointer+2+dec(cstring(curpointer+1)
          + cstring(curpointer+2) + cstring(curpointer+3))))
          token = token + cstring(j);;
          curpointer = k+1;'>,
        <litend,'state = "lit";'>

        <key,'curpointer = curpointer+1;'>
        <opend,'token=token+cstring(curpointer);>
                curpointer -- curpointer +1;'
        <erend, 'parser external endf if cstring(curpointer+1) eq ";"
                then endf = t;
        else cstring = kludge; curpointer=1;; state = "er";
        token= ";";>
        <emaybe, 'saved = <curpointer,state,token,data>;'>
        <afix, 'curpointer=curpointer-1; state="int";
                token = ((len(token)-1) first token;'>
        <emaybe1, 'saved = <curpointer-1, 'int',
                ((len token)-1) first token, data>;'>
        <efix, '<curpointer,state,token,data> = saved;'>
```

The preparser in the notes also has to be modified slightly. In particular, the order of statements in getkind must be changed to:

```
return if (x is typkind (token)) ne Ω then x else
       if (x is symbkind (token)) ne Ω then x else 'var';
```

The code for the blocks usercode, restart, and give follows:

```
block give;
     definef tilb(x); y=x; seq=nl;
(while 1 ≤ ∃[x] ≤ len y|(k elt y) eq ' ' or k eq len y)
 <seq(#seq+1),y> = <if k lt len y then k-1 else k first y,
     (len y)-k last y>;; return seq; end tilb;
typinf = {<'nam', 'var'>};
tokinf = {<y tl [tilb(x)], tilb(x)(1),'o'>, x ∈
  {'with','=','implicit','assign backspace call common
   continue data dimension do end endfile
   entry equivalence external format function
   goto if integer logical pause print
   punch read real return rewind stop
   subroutine to write', ',', '+ -', '* /', '**'}}
  u {<'(', '('>,<')', ')'>,< ';', 'er'>};
lprinf = {<tl(x), hd(x)>, x ∈ tilb('( with # implicit var assign =
   + * ** var var var var)')};
lprinf = lprinf lessf 'var';
<lprinf('var'), lprinf('er')> = <13,1>;
<rprinf('var'), rprinf('**'), rprinf('('), rprinf(')')>=
 <14,11,15,1>;
 triple = Ω; endfilesigns = nl; end give;
block restart; <lprec(','),rprec(',')>={<0,<7,7>>,<1,<12,12>>,
   <2,<5,5>>}(stype); end restart;
block usercode; end usercode;
```

The postparse grammar is rather straightforward. The postparser must be modified slightly to accept it in that the simple successor alternative must be treated like any other alternative, in particular, an's and tn's must be allowed and it must be permissible to have more than one such alternative. When reading the grammar it should be remembered that upon seeing any sort of declarative statement, except IMPLICIT, the precedence of comma has been placed above that of slash and that upon seeing an IMPLICIT statement, the precedence of comma has been lowered below that of everything except IMPLICIT and WITH. The grammar follows:

```
stat  [=:  'WITH' ('IF' ('(' lexp.) stat.)/statement.
                keyword.logical if statement/4,4,0/
                t2/ok=n(sttp lt 17);,'does not have a valid
                second half' /a2/sttp=1;/a0/switch=t;sl=t;

      =:  'DO' ('=' ('CATENATION' [int] [nam]) (',' ind. ind.))/
                do statement/6,0,0,4/a2/sttp=2;

      =:  'DO' ('=' ('CATENATION' [int] [nam]) (',' ind.ind.ind.))/
                do statement with step option/6,0,0,4/a2/sttp=2;

      =:] '=' qualname. exp./assignment statement/10/a2/sttp=17;

      =:  'DATA' ('/' datpart.3)/data statement/10,0/a2/sttp=3;
                /t2/ok=nswitch;

      =:  {'REAL','INTEGER','LOGICAL'}, datpart./
                declaration statement/10/a2/sttp=5;

      =:  {'REAL','INTEGER','LOGICAL'} ('/' datpart.3)
                declaration statement with initialization/
                10,0/a2/sttp=5;/t2/ok=n switch;,
                'lacks terminating slash'

      =:  'DIMENSION' datpart./dimension statement/10/a2/sttp=7;
                /t2/ok=n switch;

      =:  'END' /a2/sttp=8; parser external ends; ends=t;

      =:  'COMMON' ('/' compart.2)/labeled common statement/5,5/
                a2/sttp=9;

      =:  'EXTERNAL' [nam]/external statement/10/a2/sttp=10;/ob/2

      =:  'EXTERNAL' (',' [nam].2)/external statement/
                10,0/a2/sttp=10;
```

```
=:  'EQUIVALENCE' (',' eqpart.2)/equivalence statement/10,0/
        a2/sttp=11;

=:  'EQUIVALENCE' eqpart./equivalence statement/10/a2/sttp=11;

=:  'SUBROUTINE' ('(' [nam] (',' fmprm.2))/subroutine state
        subroutine statement/10,0,0/a2/sttp=12;

=:  'SUBROUTINE' ('(' [nam] fmprm.)/subroutine statement/
        10,0/a2/sttp=12;

=:  'SUBROUTINE' [nam]/subroutine statement/10/a2/sttp=12;

=:  'ENTRY' ('(' [nam] (',' fmprm.2))/entry statement/10,0,0/
        a2/sttp=14;

=:  'ENTRY' ('(' [nam] fmprm.)/entry statement/10,0/
        a2/sttp=14;

=:  'ENTRY' [nam]/entry statement/10/a2/sttp=14;

=:  'IMPLICIT' (',' impart.2)/implicit statement/10,0/
        a2/sttp=15;

=:  'IMPLICIT' impart./implicit statement/10/a2/sttp=15;

=:  'FORMAT' formins./format statement/10/a2/sttp=16;

=:  'TO' ('ASSIGN' [int])[nam]/assign statement/5,10/
        a2/sttp=18;

=:  'BACKSPACE' ind./backspace statement/10/a2/sttp=19;

=:  'CALL' ('(' [nam] (',' arg.2))/call statement/
        10,0,0/a2/sttp=20;

=:  'CALL' ('(' [nam] arg.)/call statement/10,0/a2/sttp=20;

=:  'CALL' [nam]/call statement/10/a2/sttp=20;

=:  'CONTINUE' /a2/sttp=21;

=:  'ENDFILE' ind./endfile statement/10/a2/sttp=22;

=:  'GOTO' [int]/goto statement/10/a2/sttp=23;

=:  'GOTO' (',' '(' '(' ',' [int])[nam])/computed goto statement/
        4,0,4,0/a2/sttp=24;

=:  'GOTO' (',' '(' '(' [int])[nam])/computed goto statement/
        4,0,4/a2/sttp=24;
```

```
        =:  'GOTO' [nam]/assigned goto statement/10/ob/2/a2/sttp=25;

        =:  'GOTO' (',' [nam] ('()'(',' [int].2\))/assigned goto
                statement/4,0,4,0/a2/sttp=25;

        =:  'GOTO' (','[nam] ('()' [int]))/assigned goto statement/
                4,0,4/a2/sttp=25;

        =:  "IF'(',' ('CATENATION' ('()' aexp.)[int])[int][int])/
                arithmetic if statement/4,0,4,0/a2/sttp=26;
        =:  'PAUSE'/a2/sttp=27

        =:  'PAUSE' [lit]/pause statement/10/a2/sttp=27;

        =:  'PAUSE' [int]/pause statement/10/a2/sttp=27;

        =:  'PRINT' iopart./print statement/10/a2/sttp=28;

        =:  'PUNCH' iopart./punch statement/10/a2/sttp=29;

        =:  'READ' iopart./read statement/10/a2/sttp=30;

        =:  'RETURN'/a2/sttp=31;

        =:  'RETURN' ind./return statement/10/a2/sttp=31;

        =:  'REWIND' ind./rewind statement/10/a2/sttp=32;

        =:  'STOP'/a2/sttp=33;

        =:  'STOP' [int]/stop statement/10/a2/sttp=33;

        =:  'WRITE' iopart./write statement/10/a2/sttp=34;

        =:  {'REAL, 'INTEGER', 'LOGICAL'} funcstat./typed function
                statement/0/ob/2/a2/sttp=13;

        =:  funcstat./a2/sttp=13;///

funcstat    =:  'FUNCTION' ('()' [nam] (',' [nam]2\))/function statement.
                formal parameter or function name./10,0,0

        =:  'FUNCTION' ('()' [nam][nam]\/function statement/10,0///

    lexp    =:  '.AND.','.OR.','.EQ.','.NE.' lexp.2/logical
                expression. operator.dyadic expression/10/ob/2,3

        =:  qualname.

        =:  lnum.

        =:  '.NOT.' lexp./negation/10

        =:  {'.EQ.','.NE.','.LT.','.LE.','.GT!','.GE.'} aexp.2/
                comparison expression/10/ob/2,3

        =:  '()' lexp./parenthesized expression/10/ob/2///
```

```
      aexp    =:  '*','/','+','-' aexp.2/arithmetic expression.
                      operator.dyadic expression/10

              =:  qualname.

              =:  anum.

              =:  '+','-' aexp./monadic expression/10

              =:  '*' aexp. aexp./dyadic expression/10

              =:  '()' aexp./parenthesized expression/10///

      exp     =:  aexp.

              =:  lexp.///

      ind     =:  [int]

              =:  [nam]///

qualname      =:  '()' [nam] (',' exp.2)/qualified name.subscript or
                      argument./10,0

              =:  '()' [nam] exp./qualified name/10

              =:  [nam]///

datpart       =:  ',' dimnam.2/declaration.name or initial value.
                      variable list/t1/ok=switch;/a2/switch=f;

              =:  dimnam./t1/ok=switch;/a2/switch=f;

              =:  'OMITTED'/t1/ok=switch;/a2/switch=f;

              =:  initval./t1/ok=n switch;/a2/switch=t;

              =:  ',' initval.2/initial value list/10/t1/ok=n switch:/
                      a2/switch=t;///

compart       =:  '/' [nam]/declarative part.name or label name.
                      label name/10/t1/ok=s1;/a2/s1=f;

              =:  numnam./t1/ok=switch;/a2/s1=f;switch=f;

              =:  ',' numnam.2/variable list/10/t1/ok=switch;/
                      a2/s1=f;switch=f;

              =:  [nam]/t1/ok=n switch;/a2/s1=f;switch=t;

              =:  '/' numnam./blank part/10/t1/ok=n switch and n s1:/
                      a2/s1=f;
```

```
                =:  '/'('(',' numnam.2)/blank part/10,0/t1/ok=n switch and n s1;;
                        /a2/s1=f;///

eqpart          =:  '()'('(',' numnam.2)/equivalence group.variable./10,10///

fmprm           =:  '*'

                =:  [nam]///

impart          =:  {'INTEGER','REAL','LOGICAL'} ('()'range.)/'range/type
                        list'.range./10,10

                =:  {'INTEGER','REAL','LOGICAL} ('()'('(',' range.2))
                        /'range/type list'/10,10,10///

range           =:  '-' alph. alph./range.letter./10

                =:  alph.///

alph            =:  {A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z}///

formins         =:  '()' formins./format list.format specification.sublist/10

                =:  '()' formins. formins./sublist/10

                =:  '/' formins./skip specification/10

                =:  '/' formins. formins./skip specification/10

                =:  ',' formins. formins./combination of specifications/10

                =:  '()' [int] formins./multiple sublist/10

                =:  'CATENATION' [int] formins./multiple specification/10

                =:  '/'

                =:  '+','-' formins. pspec./sign/10

                =:  pspec.

                =:  'CATENATION' [int] 'X'/space specification/0,10

                =:  'CATENATION' [int] ('CATENATION' 'X' formins.)/
                        space specification/0,0,10
```

```
              =:  'CATENATION' 'T' [int]/position specification/0,10

              =:  'CATENATION' 'T' ('CATENATION' [int] formins.)/
                      position specification/0,10,0

              =:  [lit]

              =:  'CATENATION' [lit] formins./hollerith specification/2

              =:  'CATENATION' 'I' [int]/integer specification/0,10

              =:  'CATENATION' 'I' ('CATENATION' [int]formins.)/
                      integer specification/0,10,0

              =:  'CATENATION' 'F' [dec]/floating specification/0,10

              =:  'CATENATION' 'F' ('CATENATION' [dec]formins.)/
                      floating specification/0,10,0

              =:  'CATENATION' 'E' [dec]/exponential specification/0,10

              =:  'CATENATION' 'E' ('CATENATION' [dec] formins.)/
                      exponential specification/0,10,0

              =:  'CATENATION' 'L' [int]/logical specification/0,10

              =:  'CATENATION' 'L' ('CATENATION' [int]formins.)/
                      logical specification/0,10,0

              =:  'CATENATION' 'A' [int]/alphameric specification/0,10

              =:  'CATENATION' 'A' ('CATENATION' [int] formins.)/
                      alphameric specification/0,10,0///

pspec         =:  'CATENATION' [int] 'p'/scale factor.ill-formed integer./
                      0,10

              =:  'CATENATION' [int] ('CATENATION' 'p' formins.)/
                      scale factor/0,0,10///

arg           =:  'a' [int]/argument.argument.statement number/10

              =:  exp.///

iopart        [=: 'CATENATION'('()'(',' ind.ind.)) item./
                      formatted specification/2,2,1

              =:  ','('CATENATION'('()'(',' ind.ind.))item.) item.1/
                      formatted specification/1,1,2,1
```

```
dimnam    =:  '()' [nam]ind./dimensioned name.delimiter.dimensioned
                  name/10

          =:  '()' [nam] (',' ind.2)/dimensioned name/3,3

          =:  [nam]

initval   =:  '*' [int] inval./initial value.delimiter.repeated
                  value/10

          =:  inval.///

inval     =:  'CATENATION' 'Z' [int]/initial value.character.hex
                  constant/3,3

          =:  'T'

          =:  'F'

          =:  [lit]

          =:  lnum.

          =:  anum.///

numnam    =:  '()' [nam] [int]/subscripted name.subscript or argument./10

          =:  '()' [nam] (',' [int].2)/subscripted name/3,3

          =:  [nam]///


  /end/
```

Finally, we come to a proposed definition of the routine parser, often referenced but never defined. It is:

```
Definef parser;  good = t ;
     loop:  call preparse (treetop);  good = good  and  postparse
            ('stat', treetop, 0);  if endf then if ends return good;
            else print 'end of file found before logical end of program';
            return (f); end else; end if; go to loop; end parser;
```