SETL Newsletter Number 48 — August 5, 1971

Toward a Documentation of the String Project's Program for Parsing English Sentences

Jerry Hobbs

I. Description of the Grammar:

The purpose of the first part of this newsletter is to describe the grammar used by the String Project to parse English setences. In the second part, a number of the basic tree-traversing routines and a few of the restrictions are programmed in SETL. The newsletter is not meant so much to be a documentation of the String Project's program as an indication as to how such a documentation might be carried out.

The grammar consists of four parts:

(1) a BNF grammar,

(2) restrictions,

(3) routines,

(4) a word dictionary.

The BNF grammar employs about two hundred intermediate symbols, e.g. <SENTENCE>, <ASSERTION>, <SUBJECT>, <VERB>, <NSTG>; about thirty atomic symbols, e.g. <*N>, <*V>, <*ADJ>; and about forty literals, e.g. TO, FOR, THAT. There are about two hundred productions, examples of which are as follows:

<SENTENCE>::=<INTRODUCER><CENTER><ENDMARK>.

<CENTER>::=<ASSERTION>|<QUESTION>|<PRESENT>|<PERMUTATION>.

<ASSERTION>::=<SA><SUBJECT><SA><VERB><SA><OBJECT><RV><SA>.

<SUBJECT>::=<NSTG>|<VINGSTG>|<SN>|<*NULLWH>.

(In the above <SA> stands for sentence adjunct, <RV> for adverbial phrase to the right of the verb, <NSTG> for noun string, <VINGSTG> for present participial phrase, <SN> for nominalized sentence, and <*NULLWH> for null pronoun in a wh-string. The others are self-explanatory.) The present implementation uses a straightforward top-down, back-up parser.

The BFN grammar is extended by about 200 restrictions. These restrictions are essentially subroutines which return logical values and which are invoked at various points in the parsing. The purpose of the restrictions is to handle the context-sensitive aspects of the language. The restrictions are of two types: well-formedness and disqualifying. The well-formedness restrictions are applied after the appropriate part of the tree has been built. Thus, for example, well-formedness restriction W26, which is described in the second part of this newsletter, is invoked when all of the subtree depending from <ASSERTION> is formed. If the restriction returns failure, the corresponding subparse of the sentence is destroyed. The disqualifying restrictions are applied before a particular option is tried and generally check, sometimes looking ahead, to see that certain conditions are satisfied. If not, that particular option is not tried. Thus, disqualifying restriction D70, described in the second part, looks ahead for a "TO" before trying the option for <OBJECT> called <TOVO> ("to" - verb - object). Many of the disqualifying restrictions are no more than local optimizations for the top-down back-up parser, and one may question whether they properly belong in the grammar at all. The restrictions are written in a "restriction language", a quite readable sublanguage of English which defines the restrictions precisely and compiles into stacks of routines.

It is because of these routines, many of which define relations between nodes of the parse tree which are arbitrarily distant from each other, that the grammar can cope with the unboundedly context-sensitive nature of natural language. These routines may be classified as primary routines and secondary routines, according to how and where they are defined. The primary routines, mostly logical, tree, and housekeeping operations, are defined

specifically only in long, fluid, and almost undocumented blocks of Fortran code in the bowels of the subroutine ETEST and related subroutines of the String Project's program. In addition, they are described readably but not always currently in Ralph Grishman's documentation on TEST (August, 1970). The secondary routines, such as CORERT and DEEPEST-VERB, are defined precisely in the restriction language as are the restrictions, but in this case the result is not quite so readable, and even after compilation into stacks of primary and other secondary routines, arguments and logical flow is often obscured. In the second part of this newsletter, in order to make these routines more accessible, the most important ones are described roughly and programmed in SETL for precise specification.

The fourth part of the grammar is the word dictionary. The relevant items from the word dictionary are pulled in for each sentence. The word dictionary lists for each word its possible categories (e.g. N, ADJ) and under each category its corresponding attributes (e.g. time noun, etc.) and attribute function values (e.g. NOTNSUBJ of a verb is the list of attributes its subject cannot take.) An example of a dictionary item is given below:

```
<"TYPE"  <N <SINGULAR,NCOUNT1,COLLECTIVE,NPREQ,
                  <NCOUNT2 <"IN">>,NTYPEOF,NMATH>>
         <V <NOT NOBJ <NTIME1,NHUMAN,NSENT3,PROSELF>>,
            <NOT NSUBJ< NTIME1,NSENT1,NSENT2,NSENT3>>,
            <OBJLIST <NULLOBJ,NSTGO>>>
         <TV <PLURAL, <NOTNOBJ<....>> ,....>>>
```

This item says that "type" can be a singular noun, a tenseless verb (V), or a plural tensed verb (TV). Among its attributes as a noun are collective and mathematical. As a verb it cannot take a time noun or a human noun as its object or a time noun as its subject. The String Project has defined about 120 attributes, in some cases extensionally by listing the words with that attribute,

but for the most part in terms of the context in which a word can appear. For example, a singular noun is defined as one which can appear in the context

This-__________- tensed verb - object

but not in

These-__________ - tensed verb-object.

The attributes together with the restrictions allow one to determine grammaticalness in a very strong sense, disallowing not only such sentences as

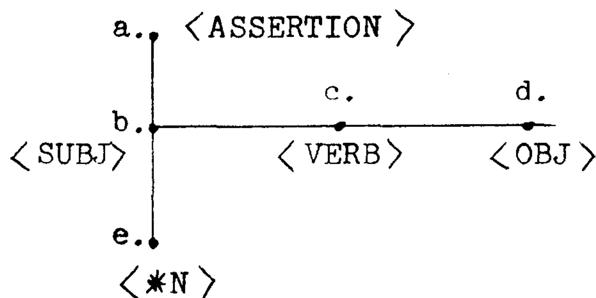
"The student type a paper."

but also such syntactically correct sentences as

"The days type Benjamin Franklin."

II. Some Routines and Restrictions Programmed in SETL

A. Conventions: In order to dodge the question of data structures, we assume certain functions are defined. The parse tree will be thought of as a binary tree. For example, (in a simplified grammar) a partial parse tree may look as follows:



Objects called nodes are created as the parse tree is built, and the following functions are defined on the nodes:

1. name: in the above name(a)=⟨ASSERTION⟩, name(e)=⟨*N⟩.
2. down: e.g. down(a)=b, down(e)=null.

3. up: e.g. up(b)=a, up(c)=null.

4. right: e.g. right(b)=c, right(d)=null.

5. left: e.g. left(c)=b, left(b)=null.

6. wordlist: defined on atomic nodes, takes from the word dictionary item the portion called for by the category of the atomic node. Thus if we have a word dictionary item

⟨word ⟨category1 ⟨attra1,attra2,....⟩⟩
⟨category2 ⟨attrb1,attrb2,....⟩⟩⟩ ,

and the parsing calls for an atomic node of type category2, then

wordlist(node)=⟨word ⟨category2 ⟨attrb1,attrb2,....⟩⟩⟩.

7. sente is the sentence being parsed, stored as a sequence. wn is the number of the word currently being scanned. Thus sente(wn) gives the current word.

In addition we assume the main program has access to certain grammar-defined lists, among which are the following:

string: a large class of intermediate symbols including main clauses, wh-clauses, prepositional phrases, etc.

stgseg: a subclass of string.

adjsetl: a class of intermediate symbols which can appear as sentence adjuncts or as left or right adjuncts of other symbols.

ladjset: the class of intermediate symbols which can appear as left adjuncts.

atomic: the atomic symbols.

B. The Simpler Primary Routines: It would seem reasonable in any specification of this part of the grammar to dispense entirely with many of the simpler primary routines used in the String Project's program by incorporating their actions into the actual SETL codes for more complex routines. The rest of section B will be devoted to general indications as to how this might be carried out. The meanings of the primary routines should be sufficiently clear from the corresponding SETL code.

1. Logical operations: AND and ANDPTH become and, providing one takes care with parameters. IMPLY and IMPLYPTH become imp, NOT becomes not, OR and ORPTH become or. ITERT (y_1, y_2), which means roughly "do y_2 until y_1 fails", may be translated as

```
iff            y2?
     (return t),     y1?
                to y2,   (return f);
```

although it can frequently be streamlined.

2. Tree operations: DOWN and VALUE become down, RIGHT becomes right, LEFT becomes left.

3. Other: TEXTX becomes hd wordlist(atom).
NAMEX becomes name(node) eq nl.
WORDL becomes sente(wn).
NEXTL becomes sente(wn+1).
EXECUTE, CANDO, and PRESENT can be ignored.
LOOK and STORE merely keep track of arguments.
IS is translated as an equality or set membership test.

C. More Complex Primary Routines: The SETL functions defined in this and the following section would form the backbone of any specification of this part of the grammar. The name used in the present program and a description of the action of the routine are followed by a definition in SETL.

1. ATTRB picks off the list of attributes for the word corresponding to the given atom.

```
definef attrblist(atom); external wordlist;
        return<-<- * >> wordlist(atom);
end attrblist;
```

2. DNTRN descends in the following manner:

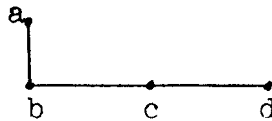
   (1) sets m=1,

   (2) scans the nodes m levels from the current node from left to right and returns the first node whose name is on yeslist,

   (3) if it finds none, sets m=m+1 and returns to (2). During the descent, it does not go below any node whose name is on the list blocklist, or any node whose name is on the list string unless it is also on the list exceptlist. Returns nl when no further descent is possible.

```
definef dntrn(node,yeslist,exceptlist,blocklist);
            external down,right,name,string;
       tryseq=nl;
    [labela:] tryseq=tryseq+<down(node) is y>;
      (while right(y) is y ne nl) tryseq=tryseq+<y>;
    [labelb:] if (tryseq is <node,tryseq> eq nl) or (name(node) is
                                namnod ∈ yeslist) then return node;;
       if namnod ∈ blocklist u (string-exceptlist) then go to labelb;
              else go to labela;;
end dntrn;
```

3. UPONE returns the parent node, e.g. in




```
upone(c)=upone(b)=a.
definef upone(node); external right,up;
   [labela:] x=up(node);
       if x eq nl then node=right(node); go to labela;
             else return x;;
end upone;
```

4. UPTRN goes up until it finds a node on yeslist and returns that node. However, it does not go above any node on blocklist, or above any node on string unless it is also on exceptlist.

```
definef uptrn(node,yeslist,exceptlist,blocklist); external name,string;
       (while upone(node) is node ne nl)
            if name(node) is namnod ∈ yeslist then return node:;
            if namnod ∈ blocklist u (string-exceptlist)then return nl:;
       return nl;
end uptrn;
```

D. Secondary Routines:

1. $DOWN1: downl and isit are used to go down one level and return the first node whose name is on list. The complication depending from nultest in isit is because in the grammar there are productions of the sort

A ::= (B/C/D)EF. (D.1)

and we wish B or C or D to be thought of as on the same level as E and F. We assume (B/C/D) is represented in the parse tree by a node x whose name is null and for which the name of down(x) is either B or C or D.

```
definef downl(node,list); external right,down;
       node=down(node)
[labela:] if isit(node,list) is node ne nl then return node:;
       if right(node) is node eq nl then return node;;
       go to labela;
end downl;
```

2. $ISIT, $ISIT1, ISIT: as described above except that it doesn't go down first.

```
definef isit(node,list); external name,right,down;
              isit external snode,n;
       initially n=1;;
       snode(n)=node; n=n+1;
```

```
        iff                 nametest?
                       ret,         nultest?
                              isittest?    righttest?
                           ret,    reset, to nametest, ret;
        nametest:= name(node) ∈ list;
        nultest:= name(node) eq nl;
        isittest:= isit(node,string) is node ne nl;
        reset: node=snode(n);n=n-1;to righttest;
        righttest:= right(node) is node ne nl;
        ret: return node;
        end iff;
end isit;
```

3. STARTAT: returns the current node or the leftmost node one level down whose name is on list.

```
definef startat (node,list); external name;
   if name(node) ∈ list then return node;
        else return downl(node,list);;
end startat;
```

4. CORERT: applies dntrn until it reaches an atomic node if there is one, or if not, a string. Returns that node.

```
definef corert(node); external name; atomic,string,adjsetl
   if node ∈ atomic then return node;;
   if dntrn(node,atomic,nl,adjsetl)is core ne nl then
              return core;;
   if dntrn(node,string,nl,adjsetl)is core ne nl then
              return core; else return nl;;
end corert;
```

5. ELEM: goes down one level to find a node of the given type. If there is none, and there is a node whose name is on stgseg, it goes down one more level to look again for a node of the given type.

```
definef elem(node,type); external stgseg;
   if  downl(node,{type} ) is y ne nl then go to ret;
       else x=downl(node,stgseg);
            y=downl(x, {type} );;
   [ret:] return y;
end elem;
```

6. COEL: goes to the right and left to find a node of the given type. The complications are to handle productions of the sort (D.1).

```
definef coel(node,type); external right,left,name,up;
     snode=node;
     (while right(node) is node ne nl) if name(node)eq type then
                                       return node;;;
     node=snode;
     (while left(node) is x ne nl doing snode=node;node=x;)
          if name(node) eq type then return node;;;
     if name(up(snode) is x)eq nl then return coel(x,type);
end coel;                              else return node;;
end coel;
```

7. ELEMOF: goes up through any node in stgseg looking for a node whose name is on stglist. Again complications arise in treating case (D.1).

```
definef elemof(node,stglist); external name,stgseg;
[labela:] if upone(node) is node eq nl then return f;;
          if name(node)∈ stglist then return t;;
          if name(node) eq nl then go to labela;;
     [labelb:] if upone(node) is node eq nl then return f;;
          if name(node) ∈ stgseg then go to labela;;
          if name(node) ne nl then return f; else go to labelb;;
end elemof;
```

8. DEEPEST-VERB: passes through all object strings which contain verbs (such as TOVO) and which are the coelements of other verbs to find the deepest verb. For example, in the sentence

"He had given up trying to learn to program in SETL."

the deepest verb is "program".

```
definef deepestverb(node);
   [recur:] deepverb=xvbstg(node);
      if xobjv(deepverb)is x ne nl then node=x; go to recur;
          else return deepverb;;
   definef xvbstg(node); external name,down  elem;
          deepverb=hd
                             'lvr','verb1','verb2','verb3}])is x);
          if (name(deepverb)='verb1' and name(down(deepverb))
                ne 'ltvr') then deepverb= <-*-> x;;
          return deepverb;
   end xvbstg;
   definef xobjv(vnode); external corert,coel,isit;
      x=corert(coel(vnode,'object'));
      y=isit(x,[ {{'vo'},{'vingo'},{'veno'},{'venpass'},{'tovo'}}]);
      return hd y;
   end xobjv;
end deepestverb;
```

9. DEEPEST-OBJBE: first finds the deepest verb. If it is a form of "be" or a representative of "be" or if it is a passive verb, the node named OBJBE depending from its coelement object is returned.

```
definef deepestobjbe(node);
    x=deepestverb(node);
    if ∃[z]∈{xdeepbe(node), elem(node,'objbe'),xpassobjbe(node)}|
           z ne nl then return z; else return nl;;
    definef xdeepbe(node); external attrblist,deepestverb,corert,
                                    dntrn,coel;
       deepestobjbe external x;
       y=attrblist(corert(x));
       if not (('vbe' ∈ y) or ('berep' ∈ y)) then return nl;
          else return dntrn(coel(x,'object'), {'objbe'} ,nl,nl);;
    end xdeepbe;
```

```
   definef xpassobjbe;deepestobjbe external x; external elemof,
                                            dntrn,coel;
        if elemof(x,'venpass') eq f then return nl;
            else return dntrn(coel(x,'passobj'), {'objbe'}, nl,
                                            {'asobjbe'});;
   end xpassobjbe;
end deepestobjbe;
```

10. LADJ: goes left and then up looking for a left adjunct. It doesn't pass through any node whose name is on string.

```
definef ladj(node); external left,name,ladjset,string,
    iff                  lefttest?
              ladjtest?               uptest?
         rett,    to uptest,     stringtest?    retf,
                            to lefttest,   retf;
       lefttest:= left(node) is node ne nl;
       ladjtest:= name(node)  ladjset;
       uptest:= upone(node) is node ne nl;
       stringtest:= not(name(node) ∈ string);
       rett:  return node;
       retf:  return nl;
    end iff;
end ladj;
```

11. IMMSTRING: goes up until it finds a string.

```
definef immstring(node);
    return upturn(node,string,nl,nl);
end immstring;
```

E. Global Abbreviations: The restrictions frequently refer to tests, called "address sentences" by the String Project and "abbreviations" here (since they could be inserted in line in the

statement of the restrictions.) Their names begin with "$" in the program and with "x" below. Some of the abbreviations are local to the particular restriction and in the SETL documentation could either be written in line or as local subroutines. Other abbreviations are referred to by many restrictions. Programs for some of these global abbreviations are given here. For each the definition in the String Project's "restriction language" is followed by the SETL code. This is especially interesting since it shows SETL in competition with English in expressive power.

1. $LADJNOTPLUR=IN THE LEFT ADJUNCT OF X_1 NEITHER $TPLUR NOR $QPLUR IS TRUE.

```
definef  xladjnot plur(node);
   x=ladj(node);
   return not (xtplur(x) or xqplur(x));
end xladjnotplur;
```

2. $LADJNOTSING=IN THE LEFT ADJUNCT OF X_1 NEITHER $TSING NOR $QSING IS TRUE.

```
definef xladjnotsing(node);
   x=ladj(node);
   return not (xtsing(x) or xqsing(x));
end xladjnotsing;
```

3. $LADJPLUR=IN THE LEFT ADJUNCT OF X_3 EITHER $TPLUR OR $QPLUR IS TRUE.

```
definef xladjplur(atom);
   x=ladj(atom);
   return xtplur(atom) or xqplur(atom);
end xladjplur;
```

4. $LADJSING=IN LEFT ADJUNCT OF X_3 EITHER $TSING OR $QSING IS TRUE.

```
definef xladjsing(node);
   x=ladj(node);
   return xtsing(x) or xqsing(x);
end xladjsing;
```

```
5.  $NOTPLURN=BOTH X1 IS NOT PLURAL, AND IF $NUMBERLESS THEN
         $LADJNOTPLUR IS TRUE.
    definef xnotplurn(atom);
       return(not('plural'∈ attrblist(atom))) and
             (xnumberless(atom)imp xladjnotplur(atom));
    end xnotplurn;
```

```
6.  $NOTSINGN=BOTH X1 IS NOT SINGULAR, AND IF $NUMBERLESS THEN
         $LADJNOTSING IS TRUE.
    definef xnotsingn(atom);
       return(not('singular"∈ attrblist(atom)))and
             (xnumberless(atom) imp xladjnotsing(atom));
    end xnotsingn;
```

```
7.  $NOTSIG=X1 IS NOT OF TYPE STRING.
    definef xnotstg(node); external string;
       return not(isit(node,string));
    end xnotstg;
```

```
8.  $NUMBERLESS=EITHER X3 IS NULLN, OR X3 IS NOT SINGULAR
          OR PLURAL.
    definef xnumberless(atom);
       return(isit(atom, {'nulln'} )ne nl) or
            (( {'singular','plural'} int attrblist(atom))eq nl);
    end xnumberless;
```

```
9.  $PLURTEST=IF $PREDPLUR IS TRUE THEN $SUBJNOTSG IS TRUE.
    definef xplurtest(node,atom);
       return xpredplur(atom) imp xsubjnotsg(node);
    end xplurtest;
```

10. $PREDPLUR=EITHER X_3 IS PLURAL, OR IF $NUMBERLESS THEN $LADJPLUR IS TRUE.

```
definef xpredplur(atom);
   return('plural' ∈ attrblist(atom)) or
         (xnumberless(atom) imp xladjplur(atom));
end xpredplur;
```

11. $PREDSING=IF X_6 HAS VALUE NSTG THEN EITHER CORE X_3 OF X_6 IS SINGULAR, OR IF $NUMBERLESS THEN $LADJSING IS TRUE.

```
definef xpredsing(node); external down;
   x=isit(down(node) {'nstg'});
   if x eq nl then return t;;
   y=corert(x);
   return('singular' ∈ attrblist(y)) or
         (xnumberless(y) imp xladjsing(y));
end xpredsing;
```

12. $QPLUR=EITHER THE CORE X_4 OF QPOS IS PLURAL, OR ELSE EITHER X_4 IS CPDNUMBR, OR X_4 IS CONJOINED BY AN ANDSTG.

```
definef xqplur(node);
   x=corert(startat(node,'qpos'));
   return (('plural' ∈ attrblist(x)) or (isit
         (x, {'cpdnumber'})ne nl)or conjoined by*
         (x,'andstring'));
end xqplur;
```

13. $QSING=BOTH THE CORE X_4 OF QPOS IS SINGULAR, AND X_4 IS NOT CONJOINED BY AN ANDSTG.

```
definef xqsing(node);
   x=corert(startat(node,'qpos'));
   return('singular' ∈ attrblist(x)) and
         (not conjoined by* (x,'andstring'));
end xqsing;
```

* not yet defined by String Project.

14. $SINGTEST=IF $PREDSING IS TRUE THEN $NOTPLURN IS TRUE.

```
definef xsingtest(node,objbenode);
   return(xpredsing(objbenode) imp xnotplurn(node));
end xsingtest;
```

15. $SUBJNOTSG=EITHER BOTH $NOTSINGN AND $NOTSTG ARE TRUE OR ELSE EITHER X_1 IS AGGREGATE, OR THE SUBJECT IS CONJOINED BY AN ANDSTG.

```
definef xsubjnotsg(node);
   return(xnotsign(node) and xnotstg(node))
      or ('aggregate' ∈ attrblist(node))
      or (conjoined by* (startat(node,'subject'),
            'andstring'));
end xsubjnotsg;
```

16. $TPLUR=THE CORE OF TPOS IS PLURAL.

```
definef xtplur(node);
   return('plural' ∈ attrblist(corert(startat(node,'tpos')))
end xtplur;
```

17. $TSING=THE CORE OF TPOS IS SINGULAR.

```
definef xtsing(node);
   return 'singular' ∈ attrblist(corert(startat
            (node,'tpos')));
end xtsing;
```

F. Restrictions: The few examples of SETL specifications of well-formedness and disqualifying restrictions given below are included to impart the flavor of the restrictions and to indicate how these restrictions may be encoded in SETL. The phrase before the colon in the "restriction language" specification of the restrictions indicates where it is housed. At present the restrictions are translated into stacks of routines by the RLS B.N.F.

* not yet defined by String Project.

grammar. Presumably one could devise a modified RLS B.N.F. grammar which could translate the restrictions language into SETL routines, if he thought that effort worthwhile.

In each example below a brief description of the restriction is given, followed by its specification in the "restriction language", followed by its specification in SETL.

1. W26 checks for a suitable subject.

W26=IN ASSERTION, YESNOQ, SENTNOM, PERMUTLIST: IF ALL $SUBJN, $NOTPASS, $HASAT ARE TRUE, THEN $NOCOMMON15 IS TRUE.

$SUBJN=THE CORE X_1 OF THE SUBJECT IS N OR PRO.

$NOTPASS=THE DEEPEST VERB IS NOT AN ELEMENT OF VENPASS.

$HASAT=THE CORE OF THE DEEPEST VERB HAS THE ATTRIBUTE NOTNSUBJ X_5.

$NOCOMMON15=LISTS X_1 AND X_5 HAVE NO COMMON ATTRIBUTE.

```
definef W26(node); external name;
   if ((name(corert(start(node,'subject'))is x)∈{'n','pro'})and
       (elemof((deepestverb(node)is y),'venpass')) and
       (∃[z] ∈ attrblist(corert(y))|  hd z eq 'notnsubj'))
   then return (attrblist(x) int tl z) eq nl;
   else return t;;
end W26;
```

2. W67 checks that a subject pronoun is not accusative except in the nominalization strings ⟨FOR TOVO⟩, ⟨NTOVO⟩, etc. (e.g. "for him to go").

W67=IN NSTG AFTER LPROR: IF NSTG IS OCCURRING AS SUBJECT X_1, THEN BOTH $ACC AND $NOM ARE TRUE.

$ACC=IF THE CORE X_2 OF NSTG IS ACCUSATIVE, THEN X_1 IS AN ELEMENT OF FORTOVO OR NTOVO OR SOBJBE OR SVEN OR SASOBJBE OR SVO OR STOVO-N.

$NOM=IF X_2 IS NOMINATIVE, THEN X_1 IS NOT AN ELEMENT OF FORTOVO OR NTOVO OR SOBJBE OR SVEN OR SASOBJBE OR SVO OR STOVO-N.

```
definef W67(node);
   y=startat(node,'nstg');
   x=uptrn(y, {'subject'},nl,nl);
   z=elemof(node,[{'fortovo','ntovo','sobjbe','sven',
                               'sasobjbe','svo','stovo-n'}]);
   return (('accusative' ∈ attrblist(y)) imp (t ∈ z))
      and (('nominative' ∈ attrblist(y)) imp not (t ∈ z));
end W67;
```

3. W96 checks that the subject and noun object of "be" agree in number.

W96=IN ASSERTION, YESNOQ, SENTNOM, PERMUTLIST: IF BOTH CORE X_1 OF SUBJECT IS NOT ↓IT↓, AND PRESENT STRING HAS DEEPEST OBJBE X_6, THEN EITHER $RARE OR BOTH $SINGTEST AND $PLURTEST ARE TRUE.

```
definef W96(node); external wordlist,rareswitch;
   x=corert(start(node,'subject'));
   if hd wordlist(x) eq 'it' then return t;;
   y=deepestobjbe(node);
   if y eq nl then return t; else z=corert(y);;
   return rareswitch or (xsingtest(x,y) and xplurtest(x,z));
end W96;
```

4. D50 checks that the adjective option of OBJBE does not occur in SUBO if CSO is CSNOTA ("since", "as"). ("As a young man, he liked sports." but not "As young, he liked sports.")

D50=IN OBJBE RE ASTG: IF IMMEDIATE STRING IS SUBO → THEN CSO IS NOT CSNOTA.

```
definef d50(node);
   return ((node is isit(immstring(node),{'subO'})) ne nl)
     imp (not ('csOas'∈ attrblist(startat(node,'csO'))));
end d50;
```

5. D70 checks that there is "TO" ahead in the sentence.

D70=IN TOVO, FORTOVO, NTOVO: THERE IS A ↓TO↓ AHEAD.

```
definef d70; external xn, sente, wordlist;
   w=wn; x=sente(wn)
   (while x ne nl doing w=w+1; x=sente(w);)
       if hd wordlist(x) eq 'to' then return t;;;
   return f;
end d70;
```