Semantic Specification using Two-Level Grammars: Labels and GOTO Statements

Frank G. Pagan
Department of Computer Science, Southern Illinois University, Carbondale, Illinois 62901, USA

The metalinguistic formalism of two-level grammars (W-grammars) is known to be capable of precisely defining the dynamic semantics of certain features of programming languages in a fairly understandable fashion. This paper demonstrates that its application to low-level control facilities—labels and goto statements—is not only possible but also reasonably straightforward and manageable. Moreover, the extra definitional complexity that arises when there is a mixture of low-level and high-level (if-then-else, while-do, etc.) facilities does not appear to be any worse than it is with other approaches to semantic specification.

INTRODUCTION

In an earlier paper[1] the author presented a two-level grammar (W-grammar) defining the complete syntax and complete semantics of an Algol-like language fragment incorporating block structure, (recursive) procedures, and the three standard parameter mechanisms known as call-by-value, call-by-reference, and call-by-name. The present paper contains a similar treatment of another common but nontrivial-to-define language facility: labels and **goto** statements.

The published use of two-level grammars as a formalism for syntax goes back to the original *Report on Algol 68*.[2] Later, use of the formalism was extended to specification of context conditions ('static semantics')[3] and of dynamic semantics.[4,5] The fact that the latter application is possible raises the method to the level of being a potential competitor for other approaches to formal semantics, such as the Vienna Definition Language and the denotational approach. Whether it will actually see widespread use for this purpose depends in large part on how readily it can be applied to different linguistic features and on how clear and readable are the resulting specifications. The construction of diverse case studies, such as the one presented in the following sections, is a necessary step in the process of making a final judgment on these matters.

To reiterate briefly the basic idea underlying the use of a two-level grammar for defining dynamic semantics, the terminal strings are considered to be of a form such as

P eof F1 eof F2 eof

where P is a syntactically valid program, F1 is an input file, and F2 is an output file, such that the execution of P with F1 terminates normally and results in the creation of F2. The grammar must generate all and only the strings (here termed 'programmes' to distinguish them from the proper 'programs' P contained in them) satisfying these constraints. The semantics of each valid program P is then completely defined by the set of all pairs of files (F1,F2) occurring in those programmes containing P.

As in the previous paper,[1] it is assumed that the reader has a basic knowledge of the two-level grammar formalism and notation. In particular, familiarity with the following metalinguistic concepts and their use is assumed: *protonotion*, *notion*, *metanotion*, *hypernotion*, *hyper-rule*, *metarule*, *predicate*. The following two special devices will serve to enhance the readability of the hyper-rules. (a) Parentheses will be used to enclose the 'noun phrases' that constitute the significant parts of predicates; formally, the parentheses act as extra lower-case letters. (b) Hyphens will frequently be used to join together the words in certain logically connected segments of long hypernotions; formally, the hyphens are like blanks and add nothing to the meaning of the rules.

DEFINITION OF LOW-LEVEL CONTROL FACILITIES

The following is a BNF grammar for a trivially small language in which conditional and unconditional jumps constitute the only control facilities:

⟨program⟩ : : = ⟨series⟩
⟨series⟩ : : = ⟨statement⟩ | ⟨series⟩ ⟨statement⟩
⟨statement⟩ : : = ⟨command⟩ | ⟨label⟩ :
  ⟨command⟩
⟨command⟩ : : = **read** ⟨variable⟩ | **write** ⟨variable⟩ |
  goto ⟨label⟩ | **if** ⟨variable⟩ = 0 **goto** ⟨label⟩
⟨variable⟩ : : = *a* | . . . | *z*
⟨label⟩ : : = ⟨digit⟩ | ⟨label⟩ ⟨digit⟩
⟨digit⟩ : : = 0 | . . . | 9

(All variables initially have the value zero.) If it were not possible to define the semantics of this trivial language by means of a reasonably short and clear two-level grammar, then the whole approach would obviously have to be rejected as inadequate for the treatment of any language containing jumps. The appendix, however, contains a complete grammar for this language, with the rules numbered and cross-referenced in a self-explanatory manner. Some of the salient points are explained below.

The following metarules (1j–n in the appendix) define protonotions which serve as suitable metalinguistic analogs of textual constructs (and hence constitute a kind of abstract syntax):

STMTSETY : : STMTS; EMPTY.
STMTS : : STMT; STMTS STMT.
STMT : : OPTLAB CMD.

CCC–0010–4620/82/0025–0063 $02.50

OPTLAB : : EMPTY; lab LABEL.
CMD : : read TAG; write TAG; goto LABEL; test TAG goto LABEL.

(There are additional metarules (1a–i) which define TAGs as analogs of variables, LABELs as analogs of labels, and **EMPTY** as the empty protonotion.) An abstract syntax for files is provided by the metarules (1u–w)

FILE : : DATETY eof.
DATETY : : DATA; EMPTY.
DATA : : datum VALUE; datum VALUE DATA.

Most of the hyper-rules for the program portion of a programme (2d–m) define hypernotions of the form

VARS STMT part-of-STMTS statement

where **STMT** is the metalinguistic analog of the textual statement and **STMTS** is the metalinguistic analog of the entire program. For example, the syntax of **goto** statements is given by the hyper-rule (2i)

```
VARS goto-LABEL part-of-STMTS statement:
  goto symbol,
    LABEL string,
    where (LABEL) defined once in (STMTS).
```

The last part of the rule refers to a set of auxiliary rules (3e–m) for predicates which enforce the context condition that there must be precisely one statement in the program with the specified label. A **VARS** protonotion serves to record the names and values of all variables used in a program and is characterized by the metarules (1q–s)

VARS : : VAR; VARS VAR.
VAR : : var TAG value VALUE.
VALUE : : EMPTY; VALUE i.

Another set of auxiliary rules (3a–d) constrains all initial values to be zero (**EMPTY**).

The 'topmost' hyper-rule of the grammar is (2a)

```
programme:
  VARS1 STMTS part-of-STMTS series,
  eof symbol,
  FILE1 file,
  FILE2 file,
  where (state VARS1 stcount-i infile-FILE1 outfile-
    eof) becomes (state VARS2 stcount-COUNT infile-
    FILE1 outfile-FILE2) given (STMTS).
```

The last part of this rule is a predicate referring to a set of hyper-rules (4a–d) which define the semantics of the language in terms of state transformations. Here there is a noteworthy contrast with previous case studies. If the language had high-level control structures (**if-then-else**, **while-do**, etc.) instead of low-level ones, then a **STATE** protonotion could simply record the values of variables, the unread portion of the input file, and the portion of the output file produced so far, and the semantic predicates could take a form such as

where (STMT) transforms (STATE1) into (STATE2).

But when control is expressed in terms of **goto** statements, the semantics of an individual statement cannot be expressed in isolation from the rest of the program. Thus the predicates contain a metalinguistic analog of the entire program and a **STATE** (1o,t) includes a statement counter giving the ordinal number of the statement to be executed:

STATE : : state VARS stcount COUNT infile FILE outfile FILE.
COUNT : : i; COUNT i.

Now in a semantic predicate of the form

where (STATE1) becomes (STATE2) given (STMTS),

STATE2 always represents the final state of execution rather than the state that exists after the execution of the current statement.

The effect of the first part of rule 4a is to define program termination in terms of the statement counter in the current state exceeding the number of statements in the program. The other five parts of the rule define the semantics of the four types of command (the fifth and sixth parts both deal with conditional jumps). For example, the second part specifies that the effect of executing an input statement (**OPTLAB-read-TAG**) is to remove a value (**VALUE2**) from the input file (**infile-datum-VALUE2-FILE1**), assign it to the relevant variable (**var-TAG**), and increment the statement counter (**stcount-COUNT**) by one (**i**). The fourth part specifies that the effect of executing an unconditional jump (**OPTLAB-goto-LABEL**) is to reset the statement counter (**stcount-COUNT1**) to the ordinal number (**COUNT2**) of the statement (**lab-LABEL-CMD**) with the relevant label.

MIXED LOW- AND HIGH-LEVEL CONTROL FACILITIES

Thus the semantics of languages with only low-level control facilities can be successfully defined by means of two-level grammars. The question now is whether the same is true when there is a mixture of low-level and high-level control, which is the case in many common languages and which is known to give rise to appreciable complexity in other approaches to formal semantics. To explore this a little way, let us consider the effect of adding a command of the form **loop** ⟨series⟩ **end** to the language treated above. Both jumps into and jumps out of any number of such loops are allowed (in fact, jumping out is the only means of termination).

It turns out that this extension does not lengthen the grammar by an undue amount (most of the expansion is in the auxiliary predicate rules 3e–m and 4b–c), but several alterations to the semantic rule 4a are necessary. The simple counting of statements must be abandoned in favor of a counting of 'positions', where a position is either the beginning of a non-loop command, the beginning of a loop command, or the end of a loop command. A **STMTS** protonotion can no longer be decomposed as **STMTSETY1 STMT STMTSETY2**; it now has the form **PRELUDE STMT POSTLUDE**, where **PRELUDE** and **POSTLUDE** are defined by the additional metarules

PRELUDE : : STMTSETY; PRELUDE loop STMTSETY.
POSTLUDE : : STMTSETY; STMTSETY end POSTLUDE.

The semantics of the loop construct is defined by the following additional hyper-rules (which could be combined with the modified rule 4a):

```
where (STATE1) becomes (STATE3) given (STMTS):
  where (STMTS) is (PRELUDE OPTLAB-loop-
      STMTS2-end POSTLUDE),
    where (PRELUDE OPTLAB-loop) has length
        (COUNT),
    where (STATE1) is (state VARS psncount-
        COUNT infile-FILE1
        outfile-FILE2),
    where (STATE2) is (state VARS psncount-
        COUNT-i infile-FILE1 outfile-FILE2),
    where (STATE2) becomes (STATE3) given
        (STMTS).
where (STATE1) becomes (STATE3) given (STMTS):
  where (STMTS) is (PRELUDE OPTLAB-loop-
      STMTS2-end POSTLUDE),
    where (PRELUDE OPTLAB-loop-STMTS2-
        end) has length (COUNT1),
    where (STATE1) is (state VARS psncount-
        COUNT1 infile-FILE1 outfile-FILE2),
    where (STATE2) is (state VARS psncount-
        COUNT2 infile-FILE1 outfile-FILE2),
    where (PRELUDE OPTLAB-loop) has length
        (COUNT2),
    where (STATE2) becomes (STATE3) given
        (STMTS).
```

In effect, then, **loop** is treated as a null statement and **end** is treated as a jump back to the corresponding **loop**. A **while** construct would have to be dealt with in a similar way, whereas in the absence of **goto**'s it would be defined in terms of high-level composition of statements, with no position counts involved. It is not surprising that the low-level facilities should drag the high-level ones down to their semantic level—a similar effect is seen in other contexts.

CONCLUSION

A metalanguage should have wide applicability as far as the range of language features it is readily able to define is concerned; the preceding two sections help to establish such a property for the two-level grammar formalism as applied to semantics. The first section together with the appendix treats the specification of languages with only low-level control facilities. The simple exercise with mixed control described in the second section encourages one to believe that the formalism could cope with other mixed control regimes, such as jumps out of procedures, with no more extra complexity than is involved in other approaches to semantic specification.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. MCS-7902962.

REFERENCES

1. F. G. Pagan, Semantic specification using two-level grammars: blocks, procedures, and parameters, *Computer Languages* **4**, 171–185 (1979).
2. A van Wijngaarden *et al.*, Report on the algorithmic language ALGOL 68. *Numerische mathematik* **14**, 79–218 (1969).
3. A. van Wijngaarden *et al.*, *Revised Report on the Algorithmic Language ALGOL 68*, Springer-Verlag, Berlin (1976). Also in *Acta Informatica* **5**, 1–236 (1975) and *ACM Special Interest-Group on Programming Languages (SIGPLAN) Notices*, **12** (No. 5), 1–70 (1977).
4. J. C. Cleaveland and R. C. Uzgaliş, *Grammars for Programming Languages*, Elsevier North-Holland, New York (1977).
5. M. Marcotty, H. F. Ledgard and G. V. Bochmann, A sampler of formal definitions, *Computing Surveys* **8**, 191–276 (1976).

Received November 1980

APPENDIX

1. Metarules

a) **ALPHA :: a; b; . . .; z.**
b) **BETA :: zero; one; . . .; nine.**
c) **LETTER :: letter ALPHA.**
d) **DIGIT :: digit BETA.**
e) **EMPTY :: .**
f) **NOTION :: ALPHA; NOTION ALPHA.**
g) **NOTETY :: NOTION; EMPTY.**
h) **TAG :: LETTER.**
i) **LABEL :: DIGIT; LABEL DIGIT.**
j) **STMTSETY :: STMTS; EMPTY.**
k) **STMTS :: STMT; STMTS STMT.**
l) **STMT :: OPTLAB CMD.**
m) **OPTLAB :: EMPTY; lab LABEL.**
n) **CMD :: read TAG; write TAG; goto LABEL; test TAG goto LABEL.**
o) **STATE :: state VARS stcount COUNT infile FILE outfile FILE.**
p) **VARSETY :: VARS; EMPTY.**
q) **VARS :: VAR; VARS VAR.**
r) **VAR :: var TAG value VALUE.**
s) **VALUE :: EMPTY; VALUE i.**
t) **COUNT :: i; COUNT i.**
u) **FILE :: DATETY eof.**
v) **DATETY :: DATA; EMPTY.**
w) **DATA :: datum VALUE; datum VALUE DATA.**

2. Main syntactic hyper-rules

a) **programme:**
  VARS1 STMTS part-of-STMTS series {d,e},
  eof symbol,

```
       FILE1 file {b,c},
       FILE2 file {b,c},
       where (state VARS1 stcount-i infile-FILE1 outfile-
            eof) becomes (state VARS2 stcount-COUNT
            infile-FILE1 outfile-FILE2) given (STMTS)
            {4a}.
b)  datum-VALUE-FILE file:
       VALUE constant {. . .},
         point symbol,
         FILE file {b,c}.
c)  eof file: eof symbol.
d)  VARS STMTS1-STMT part-of-STMTS2 series:
       VARS STMTS1 part-of-STMTS2 series {d,e},
         VARS STMT part-of-STMTS2 statement {f-j}.
e)  VARS STMT part-of-STMTS series:
       VARS STMT part-of-STMTS statement {f-j}.
f)  VARS lab-LABEL-CMD part-of-STMTS
         statement:
       LABEL string {l,m},
         colon symbol,
         VARS CMD part-of-STMTS statement {f-j},
         where (LABEL) defined once in (STMTS) {3e}.
g)  VARS read-TAG part-of-STMTS statement:
       read symbol,
         VARS identifier with TAG {k}.
h)  VARS write-TAG part-of-STMTS statement:
       write symbol,
         VARS identifier with TAG {k}.
i)  VARS goto-LABEL part-of-STMTS statement:
       goto symbol,
         LABEL string {l,m},
         where (LABEL) defined once in (STMTS) {3e}.
j)  VARS test-TAG-goto-LABEL part-of-STMTS
         statement:
       if symbol,
         VARS identifier with TAG {k},
         equals symbol,
         zero symbol,
         goto symbol,
         LABEL string {l,m},
         where (LABEL) defined once in (STMTS) {3e}.
k)  VARS identifier with TAG:
       TAG symbol,
         where (TAG) occurs once in (VARS) {3a}.
l)  LABEL-DIGIT string:
       LABEL string {l,m},
         DIGIT symbol.
m)  DIGIT string:
       DIGIT symbol.
```

3. Syntactic predicates

```
a)  where (TAG) occurs once in (VARSETY1 var-TAG-
         value-EMPTY VARSETY2):
       where (TAG) not occurring in (VARSETY1)
            {b-d},
         where (TAG) not occurring in (VARSETY2)
            {b-d}.
b)  where (TAG) not occurring in (VARS VAR):
       where (TAG) not occurring in (VARS) {b,d},
         where (TAG) not occurring in (VAR) {d}.
c)  where (TAG) not occurring in (EMPTY):
       EMPTY.
d)  where (TAG1) not occurring in (var TAG2 value
         EMPTY):
       where (TAG1) is not (TAG2) {j}.
e)  where (LABEL) defined once in (STMTSETY1 lab-
         LABEL-CMD STMTSETY2):
       where (LABEL) not defined in (STMTSETY1)
            {f-i},
         where (LABEL) not defined in (STMTSETY2)
            {f-i}.
f)  where (LABEL) not defined in (STMTS STMT):
       where (LABEL) not defined in (STMTS) {f,h,i},
         where (LABEL) not defined in (STMT) {h,i}.
g)  where (LABEL) not defined in (EMPTY):
       EMPTY.
h)  where (LABEL) not defined in (CMD):
       EMPTY.
i)  where (LABEL1) not defined in (lab-LABEL2-
         CMD):
       where (LABEL1) is not (LABEL2) {j}.
j)  where (NOTETY1 ALPHA1) is not (NOTETY2
         ALPHA2):
       where (NOTETY1) is not (NOTETY2) {j-l};
       where (ALPHA1) precedes (ALPHA2) in
            (abcdefghijklmnopqrstuvwxyz) {m};
       where (ALPHA2) precedes (ALPHA1) in
            (abcdefghijklmnopqrstuvwxyz) {m}.
k)  where (NOTION) is not (EMPTY):
       EMPTY.
l)  where (EMPTY) is not (NOTION):
       EMPTY.
m)  where (ALPHA1) precedes (ALPHA2) in
         (NOTETY1 ALPHA1 NOTETY2 ALPHA2
         NOTETY3):
       EMPTY.
```

4. Semantic hyper-rules

```
a)  where (STATE1) becomes (STATE3) given
         (STMTS):
       where (STMTS) has length (COUNT) {b,c},
         where (STATE1) is (state VARS stcount-
            COUNT-i infile-FILE1 outfile-FILE2)
            {d},
         where (STATE3) is (STATE1) {d};
       where (STMTS) is (STMTSETY1 OPTLAB-read-
          TAG STMTSETY2) {d},
         where (STMTSETY1 OPTLAB-read-TAG) has
            length (COUNT) {b,c},
         where (STATE1) is (state VARSETY1-var-
            TAG-value-VALUE1-VARSETY2 stcount-
            COUNT infile-datum-VALUE2-FILE1
            outfile-FILE2) {d},
         where (STATE2) is (state VARSETY1-var-
            TAG-value-VALUE2-VARSETY2 stcount-
            COUNT-i infile-FILE1 outfile-FILE2)
            {d},
         where (STATE2) becomes (STATE3) given
            (STMTS) {a};
       where (STMTS) is (STMTSETY1 OPTLAB-
          write-TAG STMTSETY2) {d},
         where (STMTSETY1 OPTLAB-write-TAG)
            has length (COUNT) {b,c},
         where (STATE1) is (state VARSETY1-var-
            TAG-value-VALUE-VARSETY2 stcount-
            COUNT infile-FILE1 outfile-DATETY-
            eof) {d},
         where (STATE2) is (state VARSETY1-var-
            TAG-value-VALUE-VARSETY2 stcount-
            COUNT-i infile-FILE1 outfile-FILE2) {d},
```

where (STATE2) becomes (STATE3) given (STMTS) {a};
where (STMTS) is (STMTSETY1 OPTLAB-goto-LABEL STMTSETY2) {d},
where (STMTSETY1 OPTLAB-goto-LABEL) has length (COUNT1) {b,c},
where (STATE1) is (state VARS stcount-COUNT1 infile-FILE1 outfile-FILE2) {d},
where (STATE2) is (state VARS stcount-COUNT2 infile-FILE1 outfile-FILE2) {d},
where (STMTS) is (STMTSETY3 lab-LABEL-CMD STMTSETY4) {d},
where (STMTSETY3 lab-LABEL-CMD) has length (COUNT2) {b,c},
where (STATE2) becomes (STATE3) given (STMTS) {a};
where (STMTS) is (STMTSETY1 OPTLAB-test-TAG-goto-LABEL STMTSETY2) {d},
where (STMTSETY1 OPTLAB-test-TAG-goto-LABEL) has length (COUNT1) {b,c},
where (STATE1) is (state VARSETY1-var-TAG-value-EMPTY-VARSETY2 stcount-COUNT1 infile-FILE1 outfile-FILE2) {d},
where (STATE2) is (state VARSETY1-var-TAG-value-EMPTY-VARSETY2 stcount-COUNT2 infile-FILE1 outfile-FILE2) {d},
where (STMTS) is (STMTSETY3 lab-LABEL-CMD STMTSETY4) {d},
where (STMTSETY3 lab-LABEL-CMD) has length (COUNT2) {b,c},
where (STATE2) becomes (STATE3) given (STMTS) {a};
where (STMTS) is STMTSETY1 OPTLAB-test-TAG-goto-LABEL STMTSETY2) {d},
where (STMTSETY1 OPTLAB-test-TAG-goto-LABEL) has length (COUNT) {b,c},
where (STATE1) is (state VARSETY1-var-TAG-value-VALUE-i-VARSETY2 stcount-COUNT infile-FILE1 outfile-FILE2) {d},
where (STATE2) is (state VARSETY2-var-TAG-value-VALUE-i-VARSETY2 stcount-COUNT-i infile-FILE1 outfile-FILE2) {d},
where (STATE2) becomes (STATE3) given (STMTS) {a}.

b) **where (STMTS STMT) has length (COUNT i): where (STMTS) has length (COUNT)** {b,c}.

c) **where (STMT) has length (i): EMPTY.**

d) **where (NOTETY) is (NOTETY): EMPTY.**