A Flexible Notation for Syntactic Definitions

M. HOWARD WILLIAMS
Rhodes University

In view of the proliferation of notations for defining the syntax of programming languages, it has been suggested that a simple notation should be adopted as a standard. However, any notation adopted as a standard should also be as versatile as possible. For this reason, a notation is presented here which is both simple and versatile and which has additional benefits when specifying the static semantic rules of a language.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems

General Terms: Languages

Additional Key Words and Phrases: BNF

1. INTRODUCTION

When ALGOL 60 was developed, its syntax was formally specified using the notation known as BNF [12]. Since then a wide range of different notations has been used by different authors for specifying the syntax of programming languages (e.g., the notation used for defining the syntax of COBOL [10], the two-level grammar approach used in the definition of ALGOL 68 [14, 15], the syntax diagrams used in the specification of PASCAL [4], extended BNF [1, 9], or the canonic system notation [2, 3, 8]). Although in some cases the reasons for the differences in notation can be easily understood (e.g., [9]), in others variations appear to have been introduced merely to satisfy the personal tastes and ego of the author rather than to further any clear objective.

This rapidly expanding plethora of notations is unnecessary, bewildering for the novice, and annoying for the more experienced. In 1977 Wirth [18] attacked the situation and put forward a simple notation suitable for adoption as a standard. The salient features of his notation are that

(1) nonterminals are written as identifiers without enclosing angle brackets;
(2) terminals are contained within quotation marks;
(3) braces—{ }—are used to denote "zero or more repetitions of";
(4) square brackets—[ ]—are used to denote "zero or one occurrence of"; and
(5) parentheses—( )—are used for grouping in the usual way.

This paper presents an alternative notation which is almost as simple but which has some additional advantages.

Author's present address: Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland.

2. THE NOTATION

At the heart of any formal notation for describing the syntax of programming languages must lie a mechanism for specifying repetition. This mechanism may be simple (catering only for the three basic types of repetition or choice which are most common in the specification of programming languages, namely, "zero or one occurrence of," "zero or more occurrences of," and "one or more occurrences of"), or it may be general, taking account of these three special cases and all other cases. In the notation presented here two mechanisms are provided: a simple one to handle the commonly occurring special cases and a more general one to take care of the exceptions.

In this notation terminals are written as character strings within quotation marks. To maintain generality, single or double quotes may be used provided that the use is consistent, for example, "(" or '('. If a quotation mark appears within a literal which is enclosed by the same type of quotation marks, the quotation mark must be written twice, for example, """""".

Again for the purpose of generality, nonterminals may be written as identifiers with or without enclosing angle brackets, provided that, if enclosing angle brackets are used, they are used consistently throughout. An identifier consists of a sequence of upper- or lowercase letters, digits, underlines, and hyphens (starting with a letter and ending with a letter or digit), for example,

assign-statement or ⟨ASSIGN-STATEMENT⟩.

Since there is some confusion over the use of the term "production," the term "string equation" is used here to refer to the composite definition of a nonterminal, while the term "production" is restricted to an instance of that definition which might be substituted directly in a parse. A string equation is written in the following form: nonterminal followed by "::=" followed by a string expression, optionally terminated by a period. The symbol "::=" has been adhered to because the properties of string equations do differ slightly from those of numeric equations ("=") and considerably from those of assignment statements (":="). A string expression consists of a sequence of options separated by vertical bars, for example,

add-operator ::= "+" | "−".

An option consists of a sequence of terms concatenated together. If nonterminal identifiers are not enclosed in angle brackets, then two successive nonterminals concatenated together must be separated by at least one space. Square brackets are used to denote "zero or one occurrence of" whatever is contained within them; for example,

["+" | "−"]

represents a plus sign, or a minus sign, or nothing. The postfix operators "*" and "+" are used to denote, respectively, "zero to ∞ occurrences of" and "one to ∞ occurrences of" the item which each follows. Round brackets are used for grouping; for example,

definition ::= option (" | " option)*.

```
     Syntax ::= production+.
 production ::= nonterminal "::=" definition [";" relations] ["."].
 definition ::= option (" | " option)*.
     option ::= term+.
       term ::= "[" definition "]" | item ["+" | "*" | counter].
       item ::= "(" definition ")" | nonterminal | terminal.
nonterminal ::= "⟨" identifier "⟩" | identifier.
   terminal ::= (""""" character+ """")+ | ("'" character+ "')+.
    counter ::= identifier.
  relations ::= relation ("," relation)*.
   relation ::= limit "≤" counter "≤" limit.
      limit ::= limterm (("+" | "−") limterm)* | "∞".
   limterm ::= integer | identifier.
    integer ::= digit+.
 identifier ::= letter ((hyphen | underline)* (letter | digit)+)*.
```

Figure 1

states that a definition consists of an option followed by a sequence of zero or more groups, each consisting of a vertical bar symbol followed by an option.

The more general mechanism for repetition involves the notion of a counter. A counter is a variable used to indicate repetition and is represented by an identifier without angle brackets. Repetition is denoted by writing a counter as a superscript after the item to be repeated and adding a relation defining the limits of the counter at the end of the string equation. A semicolon is used to separate the definition part from the relations, for example,

$$\text{syntax} ::= \text{production}^{i}; \qquad 1 \leq i \leq \infty.$$

In this definition counters are treated as being local to the productions in which they are used. There may, however, be some point in treating them as global variables, although at this stage no reason can be seen for doing so. Also, where counters are nested, every instance of the inner counter must satisfy the relations given at the end of the production.

The full definition of the notation using itself is given in Figure 1. Spaces are unimportant except within quotation marks.

3. FLEXIBILITY

This notation provides for flexibility with respect to both the lower and upper limits. Thus, besides the three standard cases, the notation also handles cases such as the following:

(1) *Upper Limit* $\neq 1$ *or* ∞. A simple example is the definition of an identifier in FORTRAN, where the length of an identifier is constrained to a maximum of six characters:

$$\text{identifier} ::= \text{letter}\ (\text{letter} \mid \text{digit})^{\text{len}}; \qquad 0 \leq \text{len} \leq 5.$$

(2) *Lower Limit* $\neq 0$ *or* 1. Although this does not occur very often, there are cases where it might be useful to have a lower limit greater than 1. An example

is the switchon command in BCPL [13], which might be defined as follows:

switchon ::= "SWITCHON" expression "INTO $(" case^i "$)"; $2 \leq i \leq \infty$.

case ::= ("CASE" constant ":")+ (command "newline")+.

(3) *More Complex Situations.* A data-name in COBOL [10] is a contiguous sequence of up to 30 characters (letters, digits, or hyphens), provided that a hyphen does not occur in the first or last position of the sequence and at least one of the characters in the sequence is a letter. To define this formally, one may write

data-name ::= (digit (digit | hyphen)^i)^j letter
((digit | hyphen | letter)^m (letter | digit))^k;
$0 \leq i \leq 28, 0 \leq j \leq 1, 0 \leq k \leq 1, 0 \leq m \leq 28 - i - j$.

Another complex situation arises in the case of the multiple assignment statement in BCPL, in which the destinations are written as a list to the left of the assignation symbol (:=) while the values occur in a list to the right of it, for example,

VAL, LEFT, RIGHT := K, 0, I + 1

This causes the value K to be assigned to VAL, 0 to be assigned to LEFT, and I + 1 to RIGHT. One could certainly define this construction as follows:

assign-stm ::= destination "," assign-stm "," value
| destination "=" value.

However, the parse tree which this would produce wrongly associates the value K with the variable RIGHT and the value I + 1 with the variable VAL. A better way of defining this construction would be as follows:

assign-stm ::= destination ("," destination)^i ":=" value ("," value)^i;
$0 \leq i \leq \infty$.

4. EXTENSION TO STATIC SEMANTIC RULES

Besides the syntax there are two other aspects of programming languages which need formal specification: the static semantics and the semantics. The static semantic rules are closely related to the syntax rules, and the formal specification of the static semantics of a language is usually an extension of the formal specification of the syntax.

Any notation for formally specifying the static semantic rules of a language must provide some mechanism for counting [17]. This is necessary in order to check the length of a formal parameter list against the length of each list of actual parameters or to ensure that a subscripted variable has the correct number of subscripts.

However, the counting mechanism in some static semantic notations is fairly crude and tends to complicate the specification. For example, in the two-level grammar notation [6, 14, 15] counting is performed by successively concatenating one or more symbols to a nonterminal name. A simple illustration is the require-

ment that an identifier be no more than six characters in length. In this case the symbols of the identifier are accumulated within the nonterminal name, for example,

Metasyntax

LETTER :: a; b; . . . ; z.

VAR :: LETTER; LETTER VAR.

Syntax

VAR partid : LETTER symbol; LETTER symbol, VAR partid.

and the check on the length of an identifier would be specified by enumerating the possible differing-length sequences.

Using a variation of the notation proposed here, this specification might be written as

Metasyntax

LETTER ::= a | b | · · · | z.

VAR ::= LETTER+.

Syntax

VAR-id ::= LETTER-symbol^{len}; $1 \leq len \leq 6$.

This approach has also been used to count the number of dimensions in an array definition and check this against the number of subscripts in a subscripted variable in the definition for ALGOL 68. The resulting specification requires fewer string equations. But what is more important is that the counting function has been disentangled from the rest of the specification and set apart in an easily recognizable form, thereby making it more readable.

Another important notation for specifying static semantics is attribute grammars [5, 7, 11, 16]. There, counting is performed by explicitly performing an action which causes 1 to be added to a variable. For example, the previous illustration of an identifier of not more than six characters in length may be written as an attribute grammar in which (1) *condition* precedes a predicate on attributes, (2) ↑ prefixes attribute names passed up the parse tree and ↓ prefixes attributes passed down the tree, (3) attributes can be subscripted to distinguish occurrences, and (4) attribute evaluation rules have the form of function calls where ↓ precedes the arguments and ↑ precedes the newly defined attribute.

⟨ID⟩ ↑name ::= ⟨IDENTIFIER⟩ ↑name ↑noletters
condition: noletters < 7

⟨IDENTIFIER⟩ ↑name$_1$ ↑noletters$_1$ ::= ⟨LETTER⟩ ↑name$_1$
give value to attribute ↓1 ↑noletters$_1$
| ⟨IDENTIFIER⟩ ↑name$_2$ ↑noletters$_2$
⟨LETTER⟩ ↑name$_3$
concatenate ↓name$_2$ ↓name$_3$ ↑name$_1$
add one letter ↓noletters$_2$ ↑noletters$_1$

add one letter ↓noletters$_1$ ↑noletters$_2$ ⇒ noletters$_2$ = noletters$_1$ + 1

This example may be rewritten in a variation of the notation proposed here as follows:

$$\langle \text{ID} \rangle \uparrow \text{name}_1 ::= \langle \text{LETTER} \rangle \uparrow \text{name}_1 \; (\langle \text{LETTER} \rangle \uparrow \text{name}_2 \; \textit{concatenate} \downarrow \text{name}_1 \downarrow \text{name}_2 \uparrow \text{name}_1)^i; \quad 0 \leq i \leq 5.$$

Once again, separation of the counting mechanism can simplify the notation, thereby improving readability.

5. CONCLUSIONS

A notation for defining the syntax of programming languages which is both simple and flexible has been presented. It could also have certain advantages when extending a syntax specification to include static semantic rules.

From the point of view of teaching, the notation is an extension of the notation for regular expressions which makes it easier for students to see the relationship between string equations and regular expressions. It is important that these concepts, which form part of a continuum, are not regarded as completely separate entities, each with its own notation.

In this notation as it has been presented, counters have been written as superscripts in productions to aid readability. However, if one wanted to enter a definition in this notation into a computer system, one would need a few minor modifications. First, the symbol "↑" (or even "**") could be used to precede a counter to indicate a superscript (as it is used in the case of exponentiation in computer languages). Second, "≤" should be written as "<=". Finally, if the upper limit in a relation is infinity, the limit and its preceding "≤" can be omitted. None of these alterations causes any ambiguity. The definition in Figure 1 need only be modified by substituting for the string equations for term, relation, and limit the equations

```
          term ::= "[" definition "]" | item ["+"| "*"| superscript-op counter].
superscript-op ::= "↑" | "**".
      relation ::= limit "<=" counter ["<=" limit].
         limit ::= limterm (("+" | "−") limterm)*.
```

It is hoped that this paper will not be viewed simply as a presentation of yet another notation for syntactic definitions. The main purpose of the paper has been to look closely at the advantages of the notation proposed, and it is hoped that in the future, before adopting any syntactic notation, readers will give careful consideration to the advantages of such a notation and avoid the introduction of new notations or variations on existing ones unless the advantages can be clearly spelled out.

REFERENCES

1. BULL, G.M., FREEMAN, W., AND GARLAND, S. *Specification for Standard BASIC.* NCC Publications, Manchester, England, 1973.
2. DONOVAN, J.J. *Systems Programming.* McGraw-Hill, New York, 1972.

3. DONOVAN, J.J., AND LEDGARD, H.F. A formal system for the specification of the syntax and translation of computer languages. In Proc. 1967 Fall Jt. Computer Conf., vol. 31. AFIPS Press, Arlington, Va., 1967, pp. 553–569.
4. JENSEN, K., AND WIRTH, N. *Pascal User Manual and Report.* Springer-Verlag, New York, 1976.
5. KNUTH, D.E. Semantics of context free languages. *Math. Syst. Theory 2* (1968), 127–145.
6. KOSTER, C.H.A. Two-level grammars. In *Compiler Construction, An Advanced Course*, G. Goos and J. Hartmanis (Eds.). Springer-Verlag, New York, 1974, pp. 146–156.
7. KOSTER, C.H.A. Affix grammars. In *Algol68 Implementation*, J.E.L. Peck (Ed.). Elsevier North-Holland, New York, 1971, p. 95.
8. LEDGARD, H.F. A Formal System for Defining the Syntax and Semantics of Computer Languages. Ph.D. dissertation, M.I.T., Cambridge, Mass., 1969.
9. LEE, J.A.N. The formal definition of the BASIC language. *Comput. J. 15*, 1 (Feb. 1972), 37–41.
10. MAGINNIS, J.B. *Fundamental ANSI COBOL Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
11. MARCOTTY, M., LEDGARD, H.F., AND BOCHMANN, G.V. A sampler of formal definitions. *Comput. Surv.* (ACM) *8*, 2 (June 1976), 191–276.
12. NAUR, P., ET AL. Revised report on the algorithmic language ALGOL60. *Comput. J. 5*, 4 (Jan. 1963), 349–367.
13. RICHARDS, M. The BCPL programming manual. Univ. Cambridge Computer Laboratory, Cambridge, England, 1973.
14. VAN WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A., SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., AND FISKER, R.G. Revised report on the algorithmic language ALGOL68. Springer-Verlag, New York, 1976.
15. VAN WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A., SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., AND FISKER, R.G. Report on the algorithmic language ALGOL68. *Numer. Math. 14* (1969), 79–218.
16. WATT, D.A. An extended attribute grammar for Pascal. *SIGPLAN Notices* (ACM) *14*, 2 (Feb. 1979), 60–74.
17. WILLIAMS, M.H. Methods for specifying static semantics. *Comput. Lang. 6*, 1 (1981), 1–17.
18. WIRTH, N. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM 20*, 11 (Nov. 1977), 822–823.

Received December 1980; revised July 1981; accepted August 1981