Discussed is the "syntax machine," a program for automatically generating syntactically correct programs (test cases) for checking compiler front ends.

The notion of "dynamic grammar" is introduced and is used in a syntax-defining notation that provides for context-sensitivity.

Examples demonstrate use of the syntax machine.

Automatic generation of test cases

by K. V. Hanford

The "syntax machine" discussed here automatically generates random test cases for any suitably defined programming language.[1] The test cases it produces are syntactically valid programs. But they are not "meaningful," and if an attempt is made to execute them, the results are unpredictable and uncheckable. For this reason, they are less valuable than handwritten test cases. However, as an inexhaustible source of new test material, the syntax machine has shown itself to be a valuable tool.

In the following sections, we characterize the use of this tool in testing different types of language processors, introduce the concept of "dynamic grammar" of a programming language, outline the structure of the system, and show what the syntax machine does by means of some examples.

Test cases

Test cases for a language processor are programs written following the rules of the language, as documented. The test cases, when processed, should give known results. If this does not happen, then either the processor or its documentation is in error.

We can distinguish three categories of language processors and assess the usefulness of the syntax machine for testing them. For an interpreter, the syntax machine test cases are virtually useless,

since the result of their execution is unpredictable. For a syntax checker, the syntax machine can do a complete job, including the testing of the diagnostic facilities. For a compiler, the syntax machine can fully test the ability of the input phase to accept syntactically valid input, but does not test the translation process. It can be used to estimate the reliability of the compiler with respect to such problems as getting into an infinite loop, abnormally ending, or diagnosing nonexistent syntax errors.

Although as a writer of test cases, the syntax machine is certainly unintelligent, it is also uninhibited. It can test a processor with many combinations that would not be thought of by a human test case writer. Also, test cases can be created at will, so that at any time new material can be presented to a product under test. This is particularly important for a product that has "learned" its test case library.

In its standard mode, the syntax machine produces programs in a pseudo-random manner. The system has a second mode in which it systematically produces all the possible syntactic forms of certain designated language features. The system can also be directed to produce programs that contain syntactic errors, with annotations giving the position of the errors.

Dynamic grammar

This section discusses how declarations can be considered to modify syntax and how formal notation can be extended to account for this.

Consider a compiler for an algebraic programming language. Its job is to manufacture sequences of elementary machine instructions to simulate the evaluation of expressions in the language. To such a compiler, $2 \times (3 + 4)$ is acceptable for translation—the required simulating sequence can be determined without difficulty. On the other hand, $2 \times) 3 + 4$ and $2 \times 3 (+ 4$ are not acceptable for translation, since the compiler cannot confidently ascribe a meaning to them. Thus we say that the first expression is "well-formed" or "grammatically correct," whereas the other two are not.

Let us define the "acceptable programs" for a given compiler to be those source programs whose compilation is trouble-free and results in no diagnostic messages. Then we find that acceptance depends on fórm rather than meaning: a compiler would accept

$x = 0$

$y = 1/x$

on its form, although it is computationally meaningless. Thus, the notion of "well-formed" applies not only to algebraic expres-

sions but to all the elements of a programming language, including whole statements and whole programs. If we compare two compilers for the same language, we find that the set of acceptable programs for one is almost identical to the set of acceptable programs for the other. There are usually some gray areas of difference, but these serve only to emphasize the high level of agreement on what is acceptable and what is not.

This leads us to ask whether the concept of grammar in programming language can be formalized. Can we make a definition of "well-formed" or "grammatically correct" that is a good, useful approximation to the pragmatic notion of "acceptable" programs or, alternatively, that could be used as an exact official specification of those source programs that should be accepted without diagnostic messages?

The first formal definition of the syntax of a programming language was made by Backus[2] in a notation now known as BNF (Backus normal or Backus-Naur form). A class of strings of a language is given a name enclosed in brackets $<>$, and a class is defined by one or more "writing rules." An example of a rule is:

$$<x> \rightarrow <y> + (<z>)$$

This says that if you write down *any* member of the class $<y>$, followed by the mark $+$, followed by the mark (, followed by *any* member of the class $<z>$, followed by the mark), then what you have written down is some member of the class $<x>$. The classes $<y>$ and $<z>$ must in their turn be defined by further writing rules. The syntax of a language consists of a finite number of such rules.

Following the work of Backus (related to earlier work by Chomsky[3]), a formal approach to syntax in the definition of programming languages has become commonplace. Notations vary but the motivation is always the same—to define as fully as possible the form of a program.

The grammars found in language definitions are recipes for writing down strings. The intention is that the recipe should yield precisely the set of strings that will be considered to be well-formed and must be accepted by a compiler. To meet this intention fully, the following two conditions need to be satisfied:

1. *All* well-formed programs can be written down following the recipe.
2. *Only* well-formed programs can be written down following the recipe.

BNF meets the first condition but fails to meet the second. The set of strings that it defines certainly contains all well-formed programs

but contains other strings as well. In language definition documents, to further restrict the set of defined strings, the BNF grammar is accompanied by extra grammatical rules, stated in English. These rules consist of statements like: "The constituents of an arithmetic expression must have been declared to be of type real or type integer." These extra rules are given in English because it is impossible to express them in BNF.

The informal grammatical rule given above says that a variable may occur in an arithmetic expression if and only if it also occurs in some program element declaring it to be of type real or type integer. Thus, what can be written at one place in a program depends in general on what has been written down in other places. A grammatical rule of this kind, which requires relations to hold between parts of a program, is called a "context sensitivity," and languages that have rules of this kind are "context-sensitive." All present-day high-level programming languages are context-sensitive.

The set of strings defined by a BNF grammar is termed a "context-free" language. A BNF formulation for a high-level language defines a context-free language that is a superset of the actual (context-sensitive) language.

The context-sensitive nature of programming languages derives from the fact that these languages allow *declarations*. The programmer uses a declaration to create an object, to give it a name, and to describe what kind of object it is, e.g., an array, a function, etc. Now, the contexts in which an object may occur in a program depend on the kind of object it is. Thus, declarations ascribe certain syntactic qualities to objects and the declarations in a program influence the syntax of that particular program. For example, the real and integer type declarations of a program determine what arithmetic expressions can be written in that program.

The syntax machine involves a method for the description of context-sensitive constraints. The basic idea in our formulation of these constraints is that the effect of writing a declarative statement in a program can be represented as a *change to the context-free grammar*. The rules for writing down a program should, therefore, not only specify what can be written down but also specify what changes to the grammar result from what is written down. By this means, the declarative statements construct the grammar for the other statements. This concept can be stated as follows:

The programs of a declarative programming language can be produced from a context-free grammar that has the ability to modify itself. A grammar of this form will be called a "dynamic grammar."

As an example, consider the FORTRAN declaration

INTEGER XYZ

If this string is written down as a statement of a program, it causes the grammar to be modified by the addition of a rule of the form

<integer variable> → XYZ

Initially, the grammar contains no rules for <integer variable>. Each declaration of the above form causes a new rule for <integer variable> to be added to the grammar. For example, the declarations

INTEGER AY
INTEGER BEE
INTEGER C

cause the dynamic grammar to be enriched by the rules:

<integer variable> → AY
<integer variable> → BEE
<integer variable> → C

When subsequently an arithmetic expression is written, these rules are available for use.

Production systems

In this section, we discuss first the requirements of a system for producing the strings of a context-free language. We then consider how this system can be extended to a context-sensitive production system by using syntax generators with a rewriting algorithm. A more detailed description of the rewriting procedure is then presented, which takes into account the contingencies that may be encountered in rewriting a nonterminal.

context-free production system

Consider a context-free language defined by a BNF grammar. This grammar consists of a set of rules such as

$<x> \rightarrow <y> + (<z>)$

The left-hand side is a class-name or "nonterminal" and the right-hand side is a sequence of symbols and nonterminals.

A process for obtaining a string of the class $<x>$ is described by the following "rewriting" scheme. Starting with the nonterminal $<x>$, we rewrite it using one of the rules for $<x>$. Suppose we choose the above rule. The right-hand side of the rule is a recipe. It uses two methods for obtaining a sequence of symbols to form a string of the $<x>$ class. It directly contributes some symbols (the symbols +, (,) in the example). It obtains other symbols indirectly by rewriting the nonterminals that occur in the recipe ($<y>$, $<z>$ in the example). In general, there is a choice of recipes for the rewriting of a given nonterminal. When a subsidiary recipe has done its job, it returns responsibility for symbol production back to the recipe that called it. To obtain a complete string of the

language, we start the process with the "language nonterminal" (e.g., <program>), which names the class of complete strings.

Thus, the process of writing down a program of a context-free programming language can be mechanized as a rewriting algorithm that interprets a BNF grammar. A rule of the grammar is scanned from left to right. When a nonterminal is met, a pseudo-random choice is made from the rules for that nonterminal and a recursive call to the rewriting algorithm is made. The context-free grammar and the rewriting algorithm together constitute a "production system" for a context-free language.

context-sensitive production system

A "dynamic production system" is a generalization of a context-free production system and again consists of a set of production rules and a rewriting algorithm. The generalizations concern:

- The "activation" during the production process of "syntax generators" for the self-modication of the production system
- A cyclic rewriting algorithm using "delayed nonterminals"

The rewriting algorithm is an extension of the rewriting algorithm for a context-free production system. As we have seen, context-sensitivity constraints are considered to arise from declarations. The effect of a declaration is represented by a change to the context-free grammar. When a declaration is produced, the context-free grammar is immediately modified. This modification is carried out in the following fashion. Certain context-free production rules, generally those concerned with declarations, have syntax generators attached to them. The production algorithm, after using a production rule for rewriting, activates any syntax generator attached to that rule. Syntax generators synthesize new rules and add them to the context-free grammar.

Now a declaration may influence that part of the program that comes before it. For example, we may have a branch statement with a forward reference to a label (the defining occurrence of a label is considered to be an implicit declaration). This problem of "use before declaration" is handled by giving the "delay" qualification to any nonterminal whose rewriting may depend on the rewritten form of nonterminals occurring to its right. When a nonterminal with the delay qualification is met in the left-to-right rewriting scheme, its delay qualification is removed, but it is not rewritten until the remaining nonterminals to its right have been rewritten. Delayed nonterminals are indicated by the character $-$; thus $<-x>$ stands for the nonterminal $<x>$ with the delay qualification.

In the following more detailed description of the rewriting algorithm, we discuss the conditions under which syntax generators are activated but not how they create rules. This topic is illustrated by

example in the next section. "Not rules" mentioned in the description are of the same form as rules and mean that a string of the class named by the left-hand nonterminal "cannot be" a member of the class defined by the right-hand expression.

The description of the rewriting algorithm is divided into discussions of the recursive rewriting of a nonterminal, backtracking, and the cyclic rewriting of a complete program.

recursive rewriting

Let $<x>$ be a nonterminal element of an arbitrary string. The recursive rewriting of $<x>$ ought to replace $<x>$ by a string. However, several factors can affect this effort to rewrite $<x>$, including the presence of the delay qualification mentioned above, the possible existence of "not rules," the absence of any rules for $<x>$, and the possibility that a nonterminal in a rule chosen for $<x>$ can itself not be rewritten.

We suppose that $<x>$ is not a delayed nonterminal. A rule for $<x>$ is chosen randomly, and $<x>$ is replaced by the right-hand side of this rule with all its nonterminals recursively rewritten in left-to-right sequence.

If there are no not rules for $<x>$ and if the chosen rule for $<x>$ has a syntax generator, the generator is activated. This completes the recursive rewriting of $<x>$.

Now suppose there exist not rules for $<x>$. Then the string that has replaced $<x>$ is tested for membership of the "not class" for $<x>$. If the string is a member of the not class, it is rejected (and the generator is not activated). The algorithm tries again to rewrite $<x>$, using a new randomly chosen rule. Only when a string is obtained that is not a member of the not class is it accepted. The generator of the successful rule is then activated.

backtracking

There are three possible ways in which the process described above can fail to rewrite a given nonterminal $<x>$. This possibility of failure leads to the need for a "backtracking" facility in the rewriting algorithm. These three possible ways are:

1. There are no rules for $<x>$.
2. The strings obtained for $<x>$ are consistently members of the not class for $<x>$.
3. Some nonterminal in the chosen rule for $<x>$ fails to be rewritten.

In order to explain what happens in these situations, let us suppose that the writing of $<x>$ was called for (directly) in the rewriting of $<y>$. That is, the chosen rule for $<y>$ is of the form

$$<y> \rightarrow \cdots <x> \cdots$$

In Case 1, the production algorithm backtracks. It abandons the attempt to write $<x>$ and goes back to the $<y>$ level. With respect to this level, Case 3 holds.

In Case 2, 25 attempts are made to find a string that is not a member of the not class of $<x>$. At each attempt, a new random choice of a rule is made. After 25 unsuccessful attempts, the attempt to rewrite $<x>$ is abandoned and the algorithm behaves as if there are no rules for $<x>$ (Case 1).

In Case 3, the production algorithm makes a new random choice of a rewriting rule for $<x>$, but debars the abortive rule from being chosen. If Case 3 also arises for the new rule, it in turn is debarred from being chosen at the next attempt. If all the rules for $<x>$ are eventually debarred in this way, the attempt to rewrite $<x>$ is abandoned and the algorithm behaves as if there are no rules for $<x>$ (Case 1).

There are two situations in which backtracking can give erroneous results. First, suppose that backtracking goes back to a nonterminal $<z>$, say, whose rewriting involved the activation of a generator, e.g., the chosen rule for $<z>$ was of the form

$$<z> \rightarrow \cdots <x> \cdots <y> \cdots$$

where the chosen rule for $<x>$ had a generator and $<y>$ could not be written. The effect of the generator is irreversible, so that an invalid change has probably occurred in the grammar. In this situation, a diagnostic message is given, but the production continues. Second, suppose the production process fails to rewrite a nonterminal that was delayed in the previous pass. Then backtracking will not occur and the attempted production is terminated; the syntax machine is unable to backtrack past the start of its current rewriting pass.

cyclic rewriting

Let $<x>$ be a nonterminal. The "cyclic rewriting" of $<x>$ replaces $<x>$ with a string of symbols, as follows. $<x>$ is recursively rewritten. If the string that replaces $<x>$ consists only of symbols, the process is complete. Otherwise, the remaining nonterminals are recursively rewritten, in left-to-right sequence. This step is iterated until no nonterminals remain.

A language string is obtained by cyclically rewriting the language nonterminal.

The syntax machine is such a generalized production system. The actual program consists of two parts: a grammar loader and a production algorithm. The loader stores the grammar in main storage in a form convenient for use by the production algorithm in creating programs.

Table 1 Syntax of Little PL/I

Symbols, identifiers and numbers			
<letter>	→	a|b|···|z	1.
<digit>	→	0|1|···|9	2.
<identifier>	→	<letter> | <identifier> <letter>	3.
<unsigned integer>	→	<digit> | <unsigned integer> <digit>	4.
Fixed-point variables and expressions			
<fixed-point variable>	→	i|j|k|l|m|n| <fixed-point variable > <letter>	5.
<primary>	→	<unsigned integer> | <fixed-point variable> | (<arithmetic expression>)	6.
<relational operator>	→	<| = |>	7.
<boolean expression>	→	<primary> <relational operator> <primary>	8.
<arithmetic operator>	→	+|−|*|/	9.
<arithmetic expression>	→	<primary> | <arithmetic expression> <arithmetic operator> <primary>	10.
Labels and label variables			
<label>	→	<identifier>	11.
<label variable>	→	<identifier>	12.
Statements			
<arithmetic assign statement>	→	<fixed-point variable> = <arithmetic expression>;	13.
<label assign statement>	→	<label variable> = <label>; | <label variable> = <label variable>;	14.
<go to statement>	→	go to <label>; | go to <label variable>;	15.
<if statement>	→	if <boolean expression> then <nondeclare statement>	16.
<nondeclare statement>	→	<arithmetic assign statement> | <label assign statement> | <go to statement> | <if statement>	17.
<declare statement>	→	declare <label variable> label;	18.
<statement>	→	<nondeclare statement> | <label> : <nondeclare statement>|<declare statement> | <label> : <declare statement>	19.
Programs			
<statement sequence>	→	<statement> | <statement sequence> <statement>	20.
<pl/i program>	→	<label> : procedure; <statement sequence> end <label>;	21.

Using the system

notation

To illustrate the use of the syntax machine, we now develop a definition for the syntax of a subset of PL/I. This subset has been taken from a paper by Donovan and Ledgard,[4] who call it Little PL/I. It includes limited forms of the following types of statements: GOTO, IF, label declaration, label assignment, and arithmetic assignment. A BNF definition for the context-free syntax of this language is given in Table 1. The mark | is used to separate the alternatives for a given class.

Table 2 Little PL/I program due to Donovan and Ledgard

```
Q:  PROCEDURE;
    DECLARE LX LABEL;
    L:  I=I+IA×IB−IC;
        LX=L;
        GO TO CHECK;
    M:  I=I+1;
        LX=M;
    CHECK: IF I< LIMIT
        THEN GO TO LX;
END Q;
```

The following quotation is taken from the Donovan and Ledgard paper. The quotation refers to the example of a Little PL/I program shown in Table 2.

"We define the syntax of a language as the set of rules for specifying the strings that can be recognized by a translator and translated

into some other language. The set of rules excludes strings that the translator would reject solely on their form. The syntax of Little PL/I has the following restrictions, which for all practical purposes make Little PL/I context-sensitive and therefore impossible to completely characterize in Backus-Naur Form:

1. Different declarations of the same identifier are in error, i.e.,
   a. The lists of fix-pt variables, statement labels, and declared label variables for a program must be mutually disjoint;
   b. The label before PROCEDURE must not occur within the procedure block.
2. The label after END must be identical to the label before PROCEDURE.
3. All statement labels must be different.
4. The identifier in a GOTO statement must refer to an existing statement label or a declared label variable.
5. The identifier on the left side of the = in a label assignment statement must refer to a declared label variable; the identifier on the right side of the = must refer to an existing statement label or a declared label variable."

Figure 1 gives a basic grammar for Little PL/I in the notation of the syntax machine. Upper case characters have been listed as lower case characters.

The input is composed of system statements.[5] Each statement begins on a new card and starts with a "master phrase." The end of the statement is indicated by the appearance of the next master phrase (this requires that the master phrases be distinct from all other character sequences at the start of a card). There are two classes of system statement, syntax statements and control statements. Syntax statements are used in defining a grammar. The only control statements in Figure 1 are job and comment statements, with obvious meaning. Further control statements will be introduced in the course of our development of the Little PL/I definition.

Blanks are ignored. Class names are enclosed in brackets <>. Basic symbols are represented by themselves except for the blank and the bracket <, which are represented by the pseudo class names <blank> and <less than>, respectively. This is necessary because these two marks have a special use, viz., blank is ignored and the bracket < introduces a class name. The notation also contains three nongraphic pseudo class names: <eor> terminates a punched card, <eol> terminates a printed line, and <nil> represents the empty string.

The section numbers of Table 1 and Figure 1 correspond. For most sections, Figure 1 is merely a transliteration of the BNF of Table 1. However, for certain sections, the syntax machine defini-

Figure 1 Basic gammar of Little PL/I

```
:job:                                                                    10
:comment:                  little pl/i                                   20
                           ***********                                   30
:begin:                                                                  40
:comment:  symbols, identifiers and numbers                              50
           ********************************                              60
                                                                         70
:rule:       <letter> -> a                                    1.         80
     :begin:                                                  1.         90
     :onlyrule:  <<l>> -> a                                   1.        100
     :end:                                                    1.        110
:rule:       <letter> -> b                                    1.        120
     :begin:                                                  1.        130
     :onlyrule:  <<l>> -> b                                   1.        140
     :end:                                                    1.        150
:rule:       <letter> -> c                                    1.        160
     :begin:                                                  1.        170
     :onlyrule:  <<l>> -> c                                   1.        180
     :end:                                                    1.        190
:rule:       <letter> -> d                                    1.        200
     :begin:                                                  1.        210
     :onlyrule:  <<l>> -> d                                   1.        220
     :end:                                                    1.        230
:rule:       <letter> -> e                                    1.        240
     :begin:                                                  1.        250
     :onlyrule:  <<l>> -> e                                   1.        260
     :end:                                                    1.        270
:rule:       <letter> -> f                                    1.        280
     :begin:                                                  1.        290
     :onlyrule:  <<l>> -> f                                   1.        300
     :end:                                                    1.        310
:rule:       <letter> -> g                                    1.        320
     :begin:                                                  1.        330
     :onlyrule:  <<l>> -> g                                   1.        340
     :end:                                                    1.        350
:rule:       <letter> -> h                                    1.        360
     :begin:                                                  1.        370
     :onlyrule:  <<l>> -> h                                   1.        380
     :end:                                                    1.        390
:rule:       <letter> -> i                                    1.        400
     :begin:                                                  1.        410
     :onlyrule:  <<l>> -> i                                   1.        420
     :end:                                                    1.        430
:rule:       <letter> -> j                                    1.        440
     :begin:                                                  1.        450
     :onlyrule:  <<l>> -> j                                   1.        460
     :end:                                                    1.        470
:rule:       <letter> -> k                                    1.        480
     :begin:                                                  1.        490
     :onlyrule:  <<l>> -> k                                   1.        500
     :end:                                                    1.        510
:rule:       <letter> -> l                                    1.        520
     :begin:                                                  1.        530
     :onlyrule:  <<l>> -> l                                   1.        540
     :end:                                                    1.        550
:rule:       <letter> -> m                                    1.        560
     :begin:                                                  1.        570
     :onlyrule:  <<l>> -> m                                   1.        580
     :end:                                                    1.        590
:rule:       <letter> -> n                                    1.        600
     :begin:                                                  1.        610
     :onlyrule:  <<l>> -> n                                   1.        620
     :end:                                                    1.        630
:rule:       <letter> -> o                                    1.        640
     :begin:                                                  1.        650
     :onlyrule:  <<l>> -> o                                   1.        660
     :end:                                                    1.        670
:rule:       <letter> -> p                                    1.        680
     :begin:                                                  1.        690
     :onlyrule:  <<l>> -> p                                   1.        700
     :end:                                                    1.        710
:rule:       <letter> -> q                                    1.        720
     :begin:                                                  1.        730
     :onlyrule:  <<l>> -> q                                   1.        740
     :end:                                                    1.        750
:rule:       <letter> -> r                                    1.        760
     :begin:                                                  1.        770
     :onlyrule:  <<l>> -> r                                   1.        780
     :end:                                                    1.        790
:rule:       <letter> -> s                                    1.        800
     :begin:                                                  1.        810
     :onlyrule:  <<l>> -> s                                   1.        820
     :end:                                                    1.        830
:rule:       <letter> -> t                                    1.        840
     :begin:                                                  1.        850
     :onlyrule:  <<l>> -> t                                   1.        860
     :end:                                                    1.        870
:rule:       <letter> -> u                                    1.        880
     :begin:                                                  1.        890
     :onlyrule:  <<l>> -> u                                   1.        900
     :end:                                                    1.        910
:rule:       <letter> -> v                                    1.        920
     :begin:                                                  1.        930
     :onlyrule:  <<l>> -> v                                   1.        940
     :end:                                                    1.        950
:rule:       <letter> -> w                                    1.        960
     :begin:                                                  1.        970
     :onlyrule:  <<l>> -> w                                   1.        980
     :end:                                                    1.        990
:rule:       <letter> -> x                                    1.       1000
     :begin:                                                  1.       1010
     :onlyrule:  <<l>> -> x                                   1.       1020
     :end:                                                    1.       1030
:rule:       <letter> -> y                                    1.       1040
     :begin:                                                  1.       1050
     :onlyrule:  <<l>> -> y                                   1.       1060
     :end:                                                    1.       1070
:rule:       <letter> -> z                                    1.       1080
     :begin:                                                  1.       1090
     :onlyrule:  <<l>> -> z                                   1.       1100
     :end:                                                    1.       1110
                                                                       1120
:rule:       <digit> -> 0                                     2.       1130
:rule:       <digit> -> 1                                     2.       1140
:rule:       <digit> -> 2                                     2.       1150
:rule:       <digit> -> 3                                     2.       1160
:rule:       <digit> -> 4                                     2.       1170
:rule:       <digit> -> 5                                     2.       1180
:rule:       <digit> -> 6                                     2.       1190
:rule:       <digit> -> 7                                     2.       1200
:rule:       <digit> -> 8                                     2.       1210
:rule:       <digit> -> 9                                     2.       1220
                                                                       1230
:rule:       <identifier> -> <letter>                         3.       1240
     :begin:                                                  3.       1250
     :onlyrule:  <<lambda>> -> <<l>>                          3.       1260
     :end:                                                    3.       1270
:rule:       <identifier> -> <identifier> <letter>            3.       1280
     :begin:                                                  3.       1290
     :onlyrule:  <<lambda>> -> <<lambda>> <<l>>               3.       1300
     :end:                                                    3.       1310
                                                              3.       1320
:rule:       <declaration identifier> -> <identifier>         3.       1330
     :begin:                                                  3.       1340
     :rule:       <declared identifier> -> <<lambda>>         3.       1350
     :end:                                                    3.       1360
:notrule:    <declaration identifier> -> <declared identifier>3.       1370
                                                                       1380
:rule:       <unsigned integer> -> <digit>                    4.       1390
:rule:       <unsigned integer> -> <unsigned integer> <digit> 4.       1400
                                                                       1410
:comment:  fixed-point variables and expressions                       1420
           ************************************                        1430
                                                                       1440
:rule:       <i_n identifier> -> i                            5.       1450
:rule:       <i_n identifier> -> j                            5.       1460
:rule:       <i_n identifier> -> k                            5.       1470
:rule:       <i_n identifier> -> l                            5.       1480
:rule:       <i_n identifier> -> m                            5.       1490
:rule:       <i_n identifier> -> n                            5.       1500
:rule:       <i_n identifier> -> <i_n identifier> <letter>    5.       1510
                                                              5.       1520
:rule:       <fixed_point variable> -> <i_n identifier>       5.       1530
:notrule:    <fixed_point variable> -> <declared identifier>  5.       1540
                                                                       1550
:rule:       <primary> -> <unsigned integer>                  6.       1560
:rule:       <primary> -> <fixed_point variable>              6.       1570
:rule:       <primary> -> ( <arithmetic expression> )         6.       1580
                                                                       1590
:rule:       <relational operator> -> <less than>             7.       1600
:rule:       <relational operator> -> =                       7.       1610
:rule:       <relational operator> -> >                       7.       1620
                                                                       1630
:rule:       <boolean expression> -> <primary>                8.       1640
                                     <relational operator>     8.       1650
                                       <primary>               8.       1660
                                                                       1670
:rule:       <arithmetic operator> -> +                       9.       1680
:rule:       <arithmetic operator> -> -                       9.       1690
:rule:       <arithmetic operator> -> *                       9.       1700
:rule:       <arithmetic operator> -> /                       9.       1710
                                                                       1720
:rule:       <arithmetic expression> -> <primary>            10.       1730
:rule:       <arithmetic expression> -> <arithmetic expression>10.      1740
                                          <arithmetic operator>10.      1750
                                           <primary>          10.       1760
                                                                       1770
:comment:  labels and label variables                                  1780
           **************************                                  1790
                                                                       1800
:comment:  statements                                                  1810
           **********                                                  1820
                                                                       1830
:rule:       <arithmetic assign statement> -> <fixed_point variable> = 13.      1840
                                                 <arithmetic expression> ; 13.  1850
                                                                       1860
:rule:       <label assign statement> -> <label variable> = <label> ; 14.      1870
:rule:       <label assign statement> -> <label variable> =           14.      1880
                                          <label variable> ;          14.      1890
                                                                       1900
:rule:       <go to statement> -> goto <blank> <label> ;              15.      1910
:rule:       <go to statement> -> goto <blank> <label variable> ;     15.      1920
                                                                       1930
:rule:       <if statement> -> if <blank> <boolean expression> <blank>16.      1940
                               then <blank> <non_declare statement>    16.      1950
                                                                       1960
:rule:       <non_declare statement> -> <arithmetic assign statement> 17.      1970
:rule:       <non_declare statement> -> <label assign statement>      17.      1980
:rule:       <non_declare statement> -> <go to statement>             17.      1990
:rule:       <non_declare statement> -> <if statement>                17.      2000
                                                                       2010
:rule:       <label variable declaration> -> <declaration identifier> 18.      2020
     :begin:                                                          18.      2030
     :rule:       <label variable> -> <<lambda>>                      18.      2040
     :end:                                                            18.      2050
                                                                      18.      2060
:rule:       <declare statement> -> declare <blank> <label variable   18.      2070
                                    declaration> <blank> label ;      18.      2080
                                                                       2090
:rule:       <label declaration> -> <declaration identifier>          19.      2100
     :begin:                                                          19.      2110
     :rule:       <label> -> <<lambda>>                               19.      2120
     :end:                                                            19.      2130
                                                                      19.      2140
:rule:       <statement> -> <- non_declare statement> <eor> <eol>     19.      2150
:rule:       <statement> -> <label declaration> :                     19.      2160
                            <- non_declare statement> <eor> <eol>     19.      2170
:rule:       <statement> -> <declare statement> <eor> <eol>           19.      2180
:rule:       <statement> -> <declaration identifier> :                19.      2190
                            <declare statement> <eor> <eol>          19.      2200
                                                                       2210
:rule:       <statement sequence> -> <statement>                      20.      2220
:rule:       <statement sequence> -> <statement sequence> <statement> 20.      2230
                                                                       2240
:rule:       <procedure name declaration> -> <declaration identifier> 21.      2250
     :begin:                                                          21.      2260
     :rule:       <procedure name> -> <<lambda>>                      21.      2270
     :end:                                                            21.      2280
                                                                      21.      2290
:rule:       <pl/i program> -> <set up> <procedure name declaration> : 21.     2300
                               procedure ; <eor> <eol>                21.      2310
                                <* statement sequence> end <blank>    21.      2320
                                 <procedure name> ;                   21.      2330
                                                                       2340
:rule:       <set up> -> <nil>                                        99.      2350
     :begin:                                                          99.      2360
     :erase:      <declared identifier>                               99.      2370
     :erase:      <label>                                             99.      2380
     :erase:      <label variable>                                    99.      2390
     :erase:      <procedure name>                                    99.      2400
     :end:                                                            99.      2410
:end:                                                                  2420
```

tion is an elaboration of the BNF definition, where the new material concerns the context-sensitive constraints of Little PL/I. Note also that, in Figure 1, sections 11 and 12 are empty—originally there are no rules for <label> and <label variable>.

The concept of dynamic syntax is implemented in the syntax machine by allowing "syntax generators" to be attached to the rules of a BNF grammar. These generators require an "environment" for use as a working store. An environment consists of "metaclasses" whose names are written between double pointed brackets <<,>>. When a rule has a syntax generator attached to it, this is given following the rule and is bracketed by begin and end statements. An example of a generator occurs in section 19:

```
:rule:   <label declaration> → <declaration identifier>
    :begin:
    :rule:   <label> → <<lambda>>
    :end:
```

The first line is a rule of the context-free grammar, and the following lines are the associated syntax generator. When the rule is used and produces the label declaration abc (say), the syntax generator is "activated," causing the new rule <label> → abc to be added to the context-free grammar.

In the grammar for identifiers, sections 1 and 3, the generators are designed to place a copy of a produced identifier in the metaclass <<lambda>>. Consider, for example, the production of the identifier abc. The rules for <identifier> use left recursion, and abc is formed piecemeal as a, ab, abc. This results in the activation of the sequence of generators.

```
1.  :only rule:  <<1>> → a
2.  :only rule:  <<lambda>> → <<1>>
3.  :only rule:  <<1>> → b
4.  :only rule:  <<lambda>> → <<lambda>> <<1>>
5.  :only rule:  <<1>> → c
6.  :only rule:  <<lambda>> → <<lambda>> <<1>>
```

An "only rule" for the metaclasses <<1>> or <<lambda>> erases the old values before assigning the new values. At the end of the above sequence of generator activations, <<lambda>> has the single value abc.

The grammar for <declaration identifier> also contains the "not rule":

:not rule: <declaration identifier> → <declared identifier>

The definition of a class <*x*> is that it consists of all those strings that are allowed by the rules for <*x*> and are not disallowed

by the not rules for <x>. (The class of strings defined by the not rules for <x> is called the "not class" for <x>.) By the use of not rules, we can arrange that the same identifier cannot be declared twice.

We saw above how section 19 arranges that the implicit declaration of a label α, say, causes the rule <label> $\rightarrow \alpha$ to be added to the grammar. Similarly sections 18 and 21 arrange that the declaration of a label variable and the procedure name cause rules for <label variable> and <procedure name> to be added to the grammar. Initially there are no rules for <label>, <label variable>, and <procedure name>; rules for these are created as a result of declarations.

In section 99, the "set-up generator" attached to a dummy rule ensures that the dynamic part of the grammar is initialized to be empty at the start of a program production.

Examples

We now describe the effect of providing to the syntax machine a number of variations on the definition of Little PL/I given in Figure 1. Updates to the definition are given card sequence numbers to show where they are to be inserted.

```
:rewrite: <+pl/i program> 10      4000
```

The system will produce 10 programs. (If the + were omitted, only one rewrite cycle would be performed for each production, so that the output strings would contain some nonterminals that had not been rewritten.)

```
:weight:  1                       1685
:weight: 10                       1715
```

The first of these statements assigns "weights" of 1 to the second, third, and fourth rules for <arithmetic operator>. The second statement resets the "weight register" to its default value of 10. The current value of the weight register is used in assigning a weight to each new rule. In the 10 produced programs, the operators +,−,*,/ will occur with relative frequences of approximately 10/13, 1/13, 1/13, and 1/13, respectively.

```
:rule: <statement sequence> → <# 10,20,25> <statement>  2220
       remove this statement                             2230
```

In the 10 produced program, the procedure body will consist of from 10 to 25 statements, with a mean of 20.

<#10,20,25> is an "iteration nonterminal"; these may be used

Table 3 Example of output obtained with Little PL/I

```
SCS:PROCEDURE;
DECLARE IAK LABEL;
H:I=1155;
IF 25>(IT) THEN IF (3)>9 THEN JB=03+26+K+K;
ES:DECLARE ZMQZW LABEL;
XN:DECLARE E LABEL;
IF 5>27 THEN J=Z;
DECLARE EG LABEL;
DECLARE J LABEL;
HMKKHEPAVVC:I=(K)/27+I;
MKBP:DECLARE JK LABEL;
IF (N+4+I+(I+24+N+KJ)-I)>((7+(L/L+((311))/(235+7/((N+4)))))) THEN IF 53>: THEN IF I<N THEN K=(I)+7;
C:DECLARE IQLTVJ LABEL;
DECLARE T LABEL;
F:DECLARE M LABEL;
GOTO HMKKHEPAVVC;
ITV:DECLARE Z LABEL;
X:IF ((I))>1 THEN IAK=HMKKHEPAVVC;
IF (JT/(5+(L+(I))+1820))<0276 THEN E=K;
K=I;
ADM:DECLARE OP LABEL;
XBJ:IF ((N)+(K+(4+(((3)+(?6+/Q4))-(5)+(4)/((383))+3*ND)+82+((N+(8))+2600)/KV))/0)<(I) THEN GOTO JK;
IF 2=32 THEN IF (KY)<0 THEN IF 2=4 THEN IF I>(IXR+0) THEN IF K>L THEN IVH=(K)*(3+L);
END SCS;

IJIVIAE:PROCEDURE;
E:DECLARE N LABEL;
GOTO ZS;
GOTO WEVL;
ZS:IF (O6+6)>I THEN OSW=ND;
ND:GOTO Q;
LGFA:Q=M;
DECLARE Q LABEL;
RIGI:DECLARE OWE LABEL;
IOD:DECLARE B LABEL;
IF 6<(L) THEN OWE=B;
TO:DECLARE T LABEL;
IF 6<15 THEN IF 5=ML THEN GOTO OWE;
DECLARE OSW LABEL;
T=R;
X:IF 1>3 THEN GOTO T;
B=X;
M:K=6;
GOTO OSW;
WEVL:GOTO M;
J:DECLARE R LABEL;
END IJIVIAE;

X:PROCEDURE;
IF 915=3 THEN XD=SV;
MQD:GOTO F;
A:FF=A;
U:DECLARE F LABEL;
Z:IF 98<900274 THEN IF (67+(M-(((KN))+4+L+7)-1+N+39776))<( THEN GOTO A;
RJHB:DECLARE JFEQ LABEL;
DECLARE C LABEL;
SV:MQ=N+((L+M)*4);
GT:DECLARE K LABEL;
N=MK+6;
C=K;
PRLD:DECLARE FF LABEL;
E:IF O>MCWC THEN L=N;
GIV:DECLAPE S LABEL;
J:DECLARE WYKEL LABEL;
UVGTI:DECLARE H LABEL;
EY:FF=SV;
KS:DECLARE XD LABEL;
DE:DECLARE IC LABEL;
END X;
```

in place of formulations with direct recursion (e.g., the original statements 2220, 2230).

Table 3 gives part of the output obtained from running the system with Little PL/I updated by the above statements.

:recursion limit: <arithmetic expression> 2 3000

In the 10 produced programs, the nesting of arithmetic expressions within arithmetic expressions will be limited to a depth of 2.

:rewrite trace: 3001

The system will "trace rewriting" by printing each nonterminal that is rewritten and the rule selected for it.

:print generated rules: 3001

The system will "trace rule generation" by printing all rules that are dynamically created.

```
:backtrack trace:                                          3001
```

The system will "trace backtracking" by printing all nonterminals whose rewriting directly causes backtracking, with the reason for this backtracking.

```
:expansion depth: 2                                        3001
:expansion dictionary: each <boolean expression>           3002
:rewrite systematic: <+ pl/i program> 10                   4000
```

The system will be put into the "systematic mode" and will produce a set of at most 10 programs in which every possible form of Boolean expression occurs in every possible context. The expansion depth statement (together with the language definition) defines when two Boolean expressions are to be considered to have different form.

```
:error group: <arithmetic expression>, <boolean expression>
                                                           3001
:error percent: 50                                         3002
:error dictionary: all <boolean expression>                3003
:rewrite: <+ pl/i program> 10                              4000
```

In the 10 produced programs, some Boolean expressions will be invalidly rewritten as arithmetic expressions. The first statement says that arithmetic expressions and Boolean expressions are to be considered invalid forms of each other. The second statement says that fifty percent of Boolean expressions are to be invalid, and the third statement says that invalid Boolean expressions may occur in all possible contexts. The produced programs could be used as diagnostic test cases.

Conclusion

The machine production of programs for testing certain aspects of programming products has been achieved. The system has been successfully used on a number of products to establish their reliability in accepting new test cases without error. The input to the system is a syntax definition in a formal notation. The construction of such a definition for a high-level language is an exacting task. It yields important bonuses by deepening knowledge of the structure of the language and showing up obscurities or ambiguities in the existing documentation. Definitions exist for ECMA Algol, FORTRAN IV, and a major subset of PL/I. The future effort required to adapt the definitions to particular versions of these languages should be small.

ACKNOWLEDGEMENTS

The PL/I implementation of the syntax machine was designed and programmed by S. M. Glassover, K. V. Hanford, and C. B. Jones.

The systematic and error modes were added by P. D. Wright. P. D. Wright was also responsible for the design and implementation of the assembler version, with the assistance of Miss B. Conyers, a vacation student from Brighton College of Technology, and Mrs. J. Moss.

FOOTNOTES AND CITED REFERENCES

1. The syntax machine was implemented for use on System/360 computers.
2. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM Conference," *Proceedings of the International Conference on Information Processing*, 125–132 (June 1959).
3. N. Chomsky, "Three models for the description of language," *I.R.E. Transactions on Information Theory* **IT-2,** *Proceedings of the Symposium on Information Theory* (September 1956).
4. J. J. Donovan and H. F. Ledgard, "A formal system for the specification of the syntax and translation of computer languages," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **31,** 553–580 (1967).
5. The input philosophy and the structure of grammars in memory were influenced by the compiler-compiler of Brooker et al.[6]
6. R. A. Brooker et al., "The compiler-compiler," R. Goodman, editor, *Annual Review in Automatic Programming* **3,** Pergamon Press, New York (1963).