

The Syntax of Programming Languages—A Survey

ROBERT W. FLOYD

Summary—The syntactic rules for many programming languages have been expressed by formal grammars, generally variants of phrase-structure grammars. The syntactic analysis essential to translation of programming languages can be done entirely mechanically for such languages. Major problems remain in rendering analyzers efficient in use of space and time and in finding fully satisfactory formal grammars for present and future programming languages.

INTRODUCTION

IN RECENT YEARS, few programming languages designed for widespread use have escaped having the more orderly part of their formation rules and restrictions presented in one of several simple tabular forms, somewhat like the axioms of a formal mathematical system. ALGOL, JOVIAL, FORTRAN, NELIAC, COBOL, BALGOL, MAC, APT, and their offshoots have all been defined in such a fashion (see Sections A and B of the bibliography). For some of these languages, the formalism is easy and natural. For others, it is not; FORTRAN [A9] suffers needlessly, bound in the unaccustomed corsetry of her younger rival's design. Whatever the merits of formal grammars in general, some languages are best defined in words. Where formal grammars are appropriate, however, mathematical and linguistic analysis provides compilers of lower cost and high reliability, and theoretical knowledge about the structure and value of the language itself.

PHRASE-STRUCTURE GRAMMARS

The most representative and fruitful example of the use of a formal grammar in defining a programming language is the use of a phrase-structure grammar to specify most of the syntactic rules of ALGOL 60 [A7], [A8], [B1], [B5]. The form for grammatical rules used in the report which officially defines ALGOL 60 is typified [A1] by

```
<(for statement)> ::= <(for clause)> <(statement)>  
| <(label)> : <(for statement)>.
```

This assertion can be read "A for-statement is defined to be a for-clause followed by a statement, or a label followed by a colon ':' followed by a for-statement." The symbol '::=' stands for 'is defined to be'; '|' stands for 'or' and is used to separate alternative forms of the definiendum. The angular brackets '< >' are used to enclose each name of a phrase type, distinguishing it as a name, rather than the thing named. This is the reverse

of the way quotation marks are used in English, for the same purpose, to distinguish

The baby can say "one word"

from

The baby can say one word.

Names of phrase types appearing in the text are hyphenated to show explicitly that the separate words of a name need have no individual meaning and that the name as a whole is used as a technical term, without such connotations as its individual words may suggest. The angular brackets enclosing a name in a grammatical rule share this function. A complete set of grammatical rules for a language written in a format equivalent to that of the example is a *phrase-structure grammar* (PSG); a language definable by a PSG is a *phrase-structure language* (PSL).¹

In general, a phrase-structure grammar, taken as a set of definitions, provides a list of alternative constructions in a definition for each syntactic type, where each construction is a list of characters and syntactic type names. A construction represents the set of phrases which can be formed by replacing each syntactic type name with a phrase of that type; the phrases of a certain type are all those represented by some construction in the definition of that type. There is usually a single syntactic type, called "program" (or "sentence"), which is used in the definition of no other type; the set of phrases of this type is the language defined by the grammar. On the one hand, PSG's can define some languages of considerable complexity; on the other, such simple sets of strings as that consisting of 'abc,' 'aabbcc,' 'aaabbbccc,' etc., are demonstrably not definable by any phrase structure grammar [C4].

It is evident that a complete definition of a programming language may be expressed far more concisely by a PSG than by the corresponding English sentences and that it is humanly impossible to read or write those sentences, with their hundreds of occurrences of 'is defined to be,' 'followed by,' and 'or.' If a phrase-structure grammar is nearly adequate to define a language, then most of the rules defining the language

¹ The type of grammar described here is sometimes called a *context-free* phrase-structure grammar, as distinguished from a more general type of grammar, the *context-dependent* phrase-structure grammar. The latter has no known applications to programming languages, the term "phrase-structure" is not necessarily appropriate for a context-dependent grammar, and the term "context-free" has certain misleading implications; we will therefore use the short term "phrase-structure grammar" for what is sometimes also called a context-free phrase-structure grammar [C6], simple phrase structure grammar [C1], or Type 2 grammar [C4].

can be neatly and compactly listed without explanation, conserving space, time, and clarity; attention may be concentrated on the few syntactic rules which do not fit the pattern of phrase definitions.

As mentioned above, the rules of a PSG are analogous to axioms. One who has somehow obtained a program and an understanding of its structure can use a PSG to prove the program is well formed and to demonstrate the structure to others. The usefulness of a PSG to a programmer writing in the language, or to the compiler which translates it into machine language coding, is less apparent. A grammar does not tell us how to synthesize a specific program; it does not tell us how to analyze a particular given program.²

In order to construct programs in a phrase-structure language, one may interpret every rule of its grammar as a permit to perform certain acts of substitution. Assigning a symbol to each syntactic type of a grammar, let us interpret each rule as allowing the substitution, for the definiendum, of any one of the alternative definiens. Applying these substitution rules repeatedly to the symbol designating the syntactic type 'program' or 'sentence,' we arrive eventually at a sequence of symbols in which no further substitutions can take place; this string is a program or sentence in the language, the process by which it was produced being an abbreviated proof of its sentencehood. The symbols designating syntactic types, for which substitutions may be made, are called *nonterminal characters*; those undefined symbols which form sentences are the *terminal characters*.

Take, for instance, the grammar:

- a) $\langle \text{sentence} \rangle \rightarrow \langle \text{noun} \rangle \langle \text{predicate} \rangle$
- b) $\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{noun} \rangle$
- c) $\langle \text{noun} \rangle \rightarrow \text{John} \mid \text{Mary}$
- d) $\langle \text{verb} \rangle \rightarrow \text{loves}$.

Successive substitution, starting with $\langle \text{sentence} \rangle$, gives the sequence

- 1) $\langle \text{sentence} \rangle$
- 2) $\langle \text{noun} \rangle \langle \text{predicate} \rangle$
- 3) John $\langle \text{predicate} \rangle$
- 4) John $\langle \text{verb} \rangle \langle \text{noun} \rangle$
- 5) John $\langle \text{verb} \rangle$ Mary
- 6) John loves Mary.

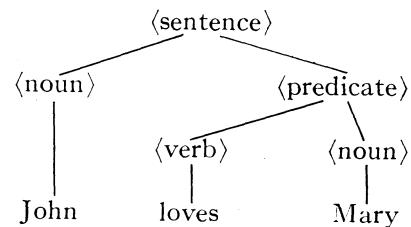
This sequence, a derivation of the sentence "John loves Mary," is an abbreviation of the following proof:

- 1) Any sentence is a sentence.
 - a) A noun followed by a predicate is a sentence.
- 2) Any noun followed by any predicate is a sentence.
 - c) 'John' is a noun.
- 3) 'John' followed by any predicate is a sentence.
 - b) A verb followed by a noun is a predicate.
- 4) 'John' followed by any verb followed by any noun is a sentence.
 - c) 'Mary' is a noun.

- 5) 'John' followed by any verb followed by 'Mary' is a sentence.
- d) 'Loves' is a verb.
- 6) 'John loves Mary' is a sentence.

From this point of view, a sentence in a phrase structure language is the last line of a derivation from the symbol ' $\langle \text{sentence} \rangle$,' provided that no further substitutions are possible.³ The grammar is regarded not as an axiom scheme for validating sentences but as a device for generating them. When a PSG is considered as a generative grammar, its rules are commonly called *productions*. The two viewpoints are substantially equivalent, but the generative viewpoint, by making explicit the process by which sentences are constructed, makes the grammar a more tractable object of study. A writer of programs in a PSL can now be thought of as a device to generate sentences, with choices between alternatives governed, for example, by the structure of a flow chart of the program. Not enough is known about linguistic behavior to specify the mechanism of choice in detail.

A compact representation of a derivation is the syntax tree [C3], [G1]; the syntax tree for the derivation above is:



In general, a syntax tree is like a genealogical tree for a family whose common ancestor is $\langle \text{sentence} \rangle$, where the immediate descendants (sons) of a symbol form one of the alternatives of the definition of that symbol and where only the terminal characters fail to have descendants. Such a tree represents a derivation of the sentence formed by its terminal characters. It also illustrates the structure of the sentence; the terminal descendants of any node on the tree form a phrase in the sentence, of the type designated by that node. In a language satisfactorily described by its grammar, the phrases of a sentence are its meaningful units. Some compilers take advantage of this, creating a syntax tree as a structured representation of the information contained in the source program. Suitable processes then translate the tree into a computer program, or a derivation tree for an equivalent sentence in another language or a related sentence in the same language.

SYNTAX-DIRECTED ANALYSIS

A syntax-directed analyzer might be defined as any procedure capable of constructing a syntax tree for an arbitrary sentence in an arbitrary PSL. This ideal, how-

² See Chomsky, [C3], p. 48.

³ See Chomsky, [C3], ch. 4.

ever, is rarely achieved; most syntax-directed analyzers are restricted to languages whose grammars satisfy certain special conditions. Let us consider a typical procedure for syntax-directed analysis.

Because the analyzer makes use of a complicated hierarchy of subordinate goals in seeking its principal goal, we will introduce it with a metaphor. Suppose a man is assigned the goal of analyzing a sentence in a PSL of known grammar. He has the power to hire subordinates, assign them tasks, and fire them if they fail; they in turn have the same power. The convention will be adhered to that each man will be told only once "try to find a G " where G is a symbol of the language, and may thereafter be repeatedly told "try again" if the particular instance of a G which he finds proves unsatisfactory to his superiors. Depending on the form of the definition of G , each subordinate (e.g., S) should adopt an appropriate strategy:

1) If G is a terminal character, and if it is the next character of the sentence, S must cover the character, and report success to his superior. If it is not the next character of the sentence, S must report failure. After success, if told by his superior to try again, S must report failure and uncover the character.

2) If $G \rightarrow G_1$, S must appoint a subordinate S_1 with the command, "Try to find a G_1 ." S repeats S_1 's report to his superior, firing S_1 on a report of failure. If told to try again, S must tell S_1 to try again, again transmitting the report to his superior and firing S_1 on failure.

3) If $G \rightarrow G_1 G_2 \dots G_n$, S must appoint successively one subordinate S_i for each G_i , with the command, "Try to find a G_i ." If S_i succeeds, i is increased by one, a new subordinate hired and the process repeated until $i > n$, when S reports success. If S_i fails S_i is fired, i is decreased by one and if $i > 0$, the new S_i (predecessor of he who failed) told to try again. If $i = 0$, S reports failure, having exhausted all ways of finding a G . If after success, S is told to try again, he sets $i = n$, tells S_i to try again, and proceeds as before on S_i 's report.

4) If $G \rightarrow G_1 | G_2 | \dots | G_n$, S must appoint successively one subordinate S_i for each G_i , with the command, "Try to find a G_i ." If S_i fails he is fired, i is increased by one, a new subordinate hired, and the process repeated until $i > n$, when S reports failure. If S_i succeeds, S reports success. If after success S is told to try again, he tells S_i (who succeeded) to try again, and proceeds as before on S_i 's report.

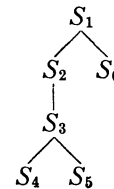
5) All more complicated definitions can be regarded as built up from the first four types.

As an example, take the sentence 'abc' and the grammar

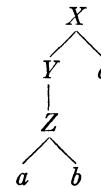
$$\begin{aligned} X &\rightarrow Yc \\ Y &\rightarrow a | Z \\ Z &\rightarrow ab \end{aligned}$$

in which X represents <sentence>. S_1 is appointed to find an X . S_1 appoints S_2 to find a Y . S_2 appoints S_3 to find

'a.' S_3 covers 'a,' reports success. S_2 reports success. S_1 appoints S_4 to find 'c.' S_4 sees 'b' in the sentence, reports failure. S_1 fires S_4 and tells S_2 to try again. S_2 tells S_3 to try again. S_3 uncovers 'a,' reports failure. S_2 fires S_3 , then appoints S_3 to find a Z . S_3 appoints S_4 to find 'a.' S_4 covers 'a,' reports success. S_3 appoints S_5 to find 'b.' S_5 covers 'b,' reports success. S_3 reports success. S_2 reports success. S_1 appoints S_6 to find 'c.' S_6 covers 'c,' reports success. S_1 reports success. The organization chart of S_1 and his subordinates,



when labeled with goals rather than names, gives the syntax tree

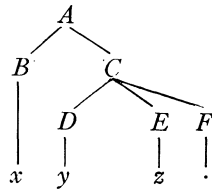


of the sentence 'abc.'

This metaphor conceals certain difficulties by relegating bookkeeping tasks to imaginary men who are assumed to automatically appear when hired, disappear when fired, remember the names of their subordinates and superiors, and so on. It is not difficult, however, by the use of a stack (pushdown list) to simulate the process on a computer, making the entire process explicit. As a convenience for the analyzer, let each definition of the grammar be followed by the additional symbols " $| \neg$," so that " $A \rightarrow B | CD$ " would be rewritten " $A \rightarrow B | CD | \neg$." Each subordinate in the metaphor is represented by an element S_λ of a stack, and contains five fields: $goal_\lambda$, the fixed goal given to S_λ by this superior; i_λ , the place in the definition of $goal$ at which S_λ is reading in the grammar; sup_λ , the name of S_λ 's superior (i.e., his location in the stack); sub_λ , the name of S_λ 's most recently-appointed subordinate; and $pred_\lambda$, the predecessor of S_λ among the subordinates of his superior. For each field, a zero specifies the absence of a value. The chief executive of the process, S_1 , is set initially to have a goal of '<sentence>' with all other fields set to zero. The index λ signifies the subordinate S_λ who is currently active; the index ν signifies the first element of the stack to which no goal is currently assigned. The index j signifies the first uncovered character of the input string. The grammar is represented by the vector $gram$, of which each character either belongs to the language defined or is one of (\rightarrow , $|$, \neg). All occurrences of S , $goal$, i , sup , sub , and $pred$, unless otherwise indexed, are implicitly indexed with λ .

When the algorithm terminates successfully, the con-

tents of the stack represent a syntax tree for the sentence taken from the input string. Each word in the stack represents a node in the tree, where *goal* represents the label of the node, *i* is the index in *gram* of the ']' following the rule of the grammar applied at that node, *sup* designates the parent node, *sub* designates the rightmost son of the node, and *pred* designates the sibling immediately to the left of the node. Only *goal*, *sub*, and *pred* are needed in order to construct the tree. Thus the tree



would be represented by the stack:

	Goal	<i>i</i>	sup	sub	pred
1	A	?	0	4	0
2	B	?	1	3	0
3	x	0	2	0	0
4	C	?	1	9	2
5	D	?	4	6	0
6	y	0	5	0	0
7	E	?	4	8	5
8	z	0	7	0	0
9	F	?	4	10	7
10	.	0	9	0	0

Fig. 1 is a flowchart for this process. There follows an item-by-item explanation of the flowchart.

A = Chief executive S_1 is appointed to find a sentence. S_2 awaits employment.

B = You are a newly appointed subordinate (S_λ); determine whether your goal is a non-terminal (defined) character, or terminal.

C, D = If the first character of the input sentence is your goal, cover it and report success to your superior; otherwise report failure and await temporary unemployment.

E = Find the beginning of the definition of your goal, by means not described here. Prepare to read that definition.

F, G, H = If you have reached a ']' in the definition of your goal, report success unless you are the chief executive, in which case you have analyzed the sentence.

I, J, K = If you have exhausted all alternatives in the definition of your goal, report failure unless you are the chief executive, in which case the input is not a sentence.

L = Otherwise, appoint a subordinate whose goal is the next character in the definition

of your goal. His superior is you, his predecessor your previously junior subordinate. Remember only your most recent subordinate, protect him from other assignments, and await his report.

M = Report success to your superior, who proceeds through the definition of his goal.

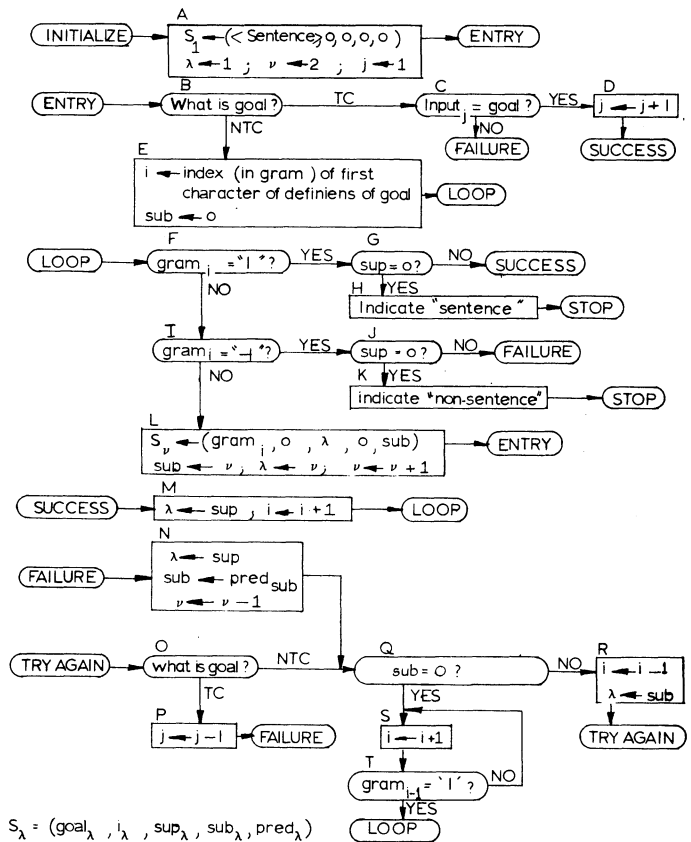
N = Report failure to your superior, who will take your predecessor as his junior subordinate and fire you.

O = When told to try again, determine again whether your local is terminal or not.

P = If terminal, uncover the input character you previously covered, and report failure.

Q, R = If goal is nonterminal, tell your junior subordinate to try again.

S, T = If you have no junior subordinate, you have exhausted an alternative; try the next one.



$$S_\lambda = (\text{goal}_\lambda, i_\lambda, \text{sup}_\lambda, \text{sub}_\lambda, \text{pred}_\lambda)$$

Fig. 1—Flowchart for syntax-directed analysis.

The serious and intrinsic flaw of the algorithm is that it fails for grammars whose rules contain certain types of cyclic formations. If the definition of A contains an alternative beginning with A, or if one of the alternatives for A begins with B, one of those for B begins with C, and one of these for C begins with A, e.g., then certain choices of input string lead the procedure into an infinite loop. A grammar containing such formations is called *left-recursive* [G3]. It is possible, at some cost to the explanatory power of a grammar, to reformulate it ex-

cluding left-recursive definition [C11]. This has been successfully done for several programming languages in the Compass compiler [D5], [D15]. A second type of syntax-directed analyzer, which is free of the left-recursion problem, constructs the syntax tree not from the top down, but from the bottom up [C11], [D2], [D5], [D8], [G4]; it is, however, as presently formulated, subject to other restrictions. All syntax-directed analyzers currently known are further restricted in practice to nonpathological languages; if a sentence is chosen at random from the grammar

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow (A) \mid (B) \mid x \\ B &\rightarrow (A) \mid (B) \mid y, \end{aligned}$$

analysis will require a time which increases exponentially with the length of the sentence, if read from left to right. At each character of the first half of the sentence, choice must be made between two alternatives. Not until the second half of the sentence is read is any information gained about the correctness of these choices. Typical processes will back up and try again many times before hitting on the right pattern. A related grammar may be designed to exhaust the patience of all known syntax-directed analyzers, whether their prejudices be left, right, or center.

It is not known whether a method of syntactic analysis is possible for which the time required for analysis does not increase exponentially with the length of the sentence, even for pathological languages. Known methods of full generality, such as the systematic generation of all sentences until a match is found, would be unacceptably slow even for short sentences. The properties of programming languages which make them legible to human readers, however, allow them to be analyzed by simple and efficient methods. A case in point is COBOL, for which a syntax-directed analyzer is greatly simplified because each choice among alternative constructions can be decided by examining the first character or word of that construction [D6].

SYNTAX-CONTROLLED ANALYSIS

An alternative approach to syntactic analysis of phrase structure languages, sometimes called *syntax-controlled* analysis, entails a preliminary processing of a grammar during which matrices, tables, and lists are constructed describing in some sense the possible constructions of the grammar. Analysis of sentences then makes use of these tabulations, and may even dispense entirely with the original grammar.

As an example, let us consider precedence analysis [E2], which is a formalization and extension of methods of analysis which were used in compilers even before formal grammars were employed in defining programming languages. From the grammar of ALGOL it is possible to deduce that in any ALGOL program if a left parenthesis '(' is followed by a multiplication sign '×,'

separated by at most one phrase, then there is some phrase to which the multiplication sign and any phrase adjacent to it belong, but not containing the left parenthesis. The relation is symbolized, '(<×,' where the sign '<' is read 'yields precedence to.' Similarly, if '×' is followed by '+' with at most one phrase between, some phrase contains the multiplication sign and any phrases adjacent to it, but not the plus sign. This relation is symbolized '×>+,' where '>' is read 'takes precedence over.' We may deduce that whenever

$$\dots (a \times b + \dots)$$

occurs in an ALGOL program, and a and b are arbitrary phrases, then ' $a \times b$ ' is a phrase. A third relation, '(=),' applies to characters of equal precedence. While analysis based on precedence relations does not yield a complete derivation of a sentence, it determines the phrases of the sentence and the operators connecting them, which is normally sufficient information for use by a compiler.

Not every grammar is amenable to precedence analysis. Yet, like phrase-structure grammars, matrices representing precedence relations are generally adequate for the description of the structure of programs in standard programming languages. Because a precedence matrix can be derived from a grammar, and applied to syntactic analysis, by a completely mechanical process, precedence analysis offers much the same flexibility and universality as does syntax-directed analysis.

Neither syntax-directed nor syntax-controlled analyzers are capable, by themselves, of dealing with non-sentences. Syntax-directed analyzers are usually incapacitated by syntactic errors in their input sentences [D5], [D9]. Precedence methods are more flexible, but still require explicit specification of error recovery policies. Chomsky has proposed that an adequate grammar for a natural language must account for our ability to interpret ungrammatical sentences [G3]. Such grammars are doubly necessary for programming languages, at least to the extent of localizing the effects of programming errors.

ADEQUACY

The phrase-structure grammar, though developed as a model for natural language, is generally considered inadequate to represent either the structure or the constraints imposed on sentences in most natural languages [C3], [C5]. Nor is the PSG sufficient to fully describe the formation rules of most programming languages. Most require, for example, that the arithmetic type of each variable be declared before using it in a formula or that dimensions of an array be specified before referring to one of its elements. Rules of this type cannot be incorporated in a PSG [C7]; nor can the rules for writing DO-loops in FORTRAN [A9]. Any rule requiring that two or more constituent phrases of a construction be identical (or different) is almost certainly beyond the scope of phrase-structure definition, as is the indication of scope of nested loops by indentation.

The PSG is nonetheless a valuable tool for describing languages, both natural and artificial. Chomsky has described it as the only theory of grammar with any linguistic motivation that is sufficiently simple to permit serious abstract study. Most published PSG's for programming languages, while not serving as complete definitions, define languages which include the programming languages as subsets satisfying simple restrictions, and correctly account for the structure of programs.

EXTENSIONS

The use of curly brackets around a part of a rule in a PSG is sometimes used to signify an arbitrary number, possibly zero, of occurrences of the form described within the brackets. As a refinement, super- and subscripts on the closing bracket, if present, signify upper and lower limits on the number. A variant uses square brackets to signify an optional single occurrence of the form described within the brackets. The COBOL syntax uses a two-dimensional display of alternatives and options. While none of these operators extends the generative power of PSG's, they all increase the convenience and explanatory power. For example, a phrase-structure description of the function $f(a, b, c, d)$, if general enough to deal with functions of arbitrarily many variables, leads to such absurdities as assertions that ' a, b ' is a phrase but ' c, d ' is not. A definition using curly brackets, $\langle \text{function} \rangle \rightarrow \langle \text{function name} \rangle (\langle \text{expression} \rangle \{, \langle \text{expression} \rangle \})$

avoids designating as phrases any parts of the function except those which serve as names or have values.

An extension to permit specification that two component phrases of a construction must be identical increases the generative power of PSG's. Such a mechanism is used in Input Language (Siberian ALGOL) [A5], to permit programs to contain relations like " $\text{Alpha}_1 \leq \dots \leq \text{Alpha}_n$ " but not " $\text{Alpha}_1 \leq \dots \leq \text{Beta}_n$." It seems unlikely, however, that extensions will be found which, while retaining the explanatory power of PSG's, permit the complete description of even the present generation of computer languages.

THEORY OF FORMAL LANGUAGES

There exists a rapidly growing body of theory of PSG's and other formal models of language. Some of the results are of interest to the designer of compilers and the writer of programming manuals, such as the possibility of listing the allowed character pairs which may occur in programs, or the possible initial characters of each phrase type [C1], [E2]. Others pertain to the design of programming languages, such as the absence of a general procedure to determine whether a PSG generates ambiguous sentences [C2], [C6], [C8] [C10], the existence of recognizable classes of grammars which are free from ambiguity [E1], [E2], [E3] and the existence of languages for which all PSG's are ambiguous

[C5], [C14]. Chomsky [C4] and Bar-Hillel, Perles, and Shamir [C1] are important original papers on the general theory of PSG's; Chomsky [C5] is a thorough survey of known results about PSG's and related language-generating devices.

UNSOLVED PROBLEMS

Many questions of practical importance in the design of programming languages and their compilers are unanswered; some have not, to the writer's knowledge, been stated in print. It is not known, for example, how to synthesize a phrase-structure grammar for a programming language, given the precedence relations of its operators. Such a synthesis method would have prevented the costly ambiguities originally present in ALGOL 60. For a given language, it is not known how to synthesize a grammar which best displays the structure of its sentences, best accommodates a particular method of syntactic analysis, or best accounts for the structure of sentences containing slight syntactic errors. It is not known whether an analyzer is possible which would not consume excessive space and time, even for pathological languages. Some of these questions are capable of precise formulation, but even rule-of-thumb solutions for any of them would be valuable.

BIBLIOGRAPHY

The bibliography which follows includes subjects related to the syntax of programming languages insofar as they illuminate the problems of analysis and synthesis of formally defined programming languages. The bibliography is arranged by subjects, alphabetically by author within each subject. Particularly recommended as introductions to their subjects are [A7], [A12], [B1], [C1], [C3], [C4], [C5], [D5], [D12], [E2], [E4], [F10], [G1], [G3] (subjects are designated by letter, individual papers by number).

A. Formal Grammars for Programming Languages

- [1] J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference," *Proc. Internat'l. Conf. Information Processing*, UNESCO, Paris, pp. 125-132; June, 1959.
- [2] R. Berman, J. Sharp, and L. Sturges, "Syntactical charts of COBOL 61," *Comm. ACM*, vol. 5, p. 260; May, 1962.
- [3] R. A. Brooker and D. Morris, "A description of the Mercury Autocode in terms of a phrase structure language," in "Annual Review in Automatic Programming," Pergamon Press, The Macmillan Co., New York, N. Y., vol. 2, pp. 29-66; 1961.
- [4] S. A. Brown, C. E. Drayton, and B. Mittman, "A description of the APT language," *Comm. ACM*, vol. 6, pp. 649-658; Nov., 1963.
- [5] A. P. Ershov, G. I. Kohuzhin, and Yu. M. Voloshin, "Input language for a system of automatic programming," Academy of Science U.S.S.R. Computing Center, Moscow; 1961 (Russian). Academic Press, London; 1963 (English).
- [6] H. D. Huskey, R. Love, and N. Wirth, "A syntactic description of BC NELIAC," *Comm. ACM*, vol. 6, pp. 367-375; July, 1963.
- [7] P. Naur, *et al.*, "Report on the algorithmic language ALGOL 60," *Comm. ACM*, vol. 3, pp. 299-314; May, 1960. "Annual Review of Automatic Programming," Pergamon Press, The Macmillan Co., New York, N. Y., vol. 2, pp. 351-390; 1961. *Numerische Mathematik*, vol. 2, pp. 106-136; March, 1960.
- [8] P. Naur, *et al.*, "Revised report on the algorithmic language ALGOL 60," *Comm. ACM*, vol. 6, pp. 1-7; January, 1963. *Numerische Mathematik*, vol. 4, pp. 420-452; March, 1963. *Computer Journal*, vol. 5, pp. 349-367; January, 1963.

- [9] I. N. Rabinowitz, "Report on the algorithmic language FORTRAN II," *Comm. ACM*, vol. 5, pp. 327-337; June, 1962.
- [10] C. J. Shaw, "A specification of JOVIAL," *Comm. ACM*, vol. 6, pp. 721-736; December, 1963.
- [11] C. J. Shaw, "JOVIAL—a programming language for real-time command systems," in "Annual Review in Automatic Programming," Pergamon Press, The Macmillan Co., New York, N. Y., vol. 3, pp. 53-119; 1963.
- [12] W. Taylor, L. Turner, and R. Waychoff, "A syntactical chart of ALGOL 60," *Comm. ACM*, vol. 4, p. 393; September, 1961. (See [A7].)
- [13] N. Wirth, "A generalization of ALGOL," *Comm. ACM*, vol. 6, pp. 547-554; September, 1963.
- [14] W. W. Youden, "An analysis of ALGOL 60 syntax," Data Proc. Systems Div., Nat. Bureau of Standards, Washington, D. C.; August 15, 1961. (See [A7].)
- [15] "Index to ALGOL 60 syntactical chart," Training and Education Dept., E.D.P., RCA, Camden, N. J.; October 20, 1961. (See [A12].)
- [16] "COBOL 61, revised specifications for a common business-oriented language," U. S. Govt. Printing Office, Washington, D. C., O-598941; 1961.
- [9] S. Gorn, "Detection of generative ambiguities in context-free mechanical languages," *J. ACM*, vol. 10, pp. 196-208; April, 1963.
- [10] S. A. Greibach, "The undecidability of the ambiguity problem for minimal linear grammars," *Inf. and Control*, vol. 6, pp. 119-125; June, 1963.
- [11] S. A. Greibach, "Inverses of phrase structure generators," Ph.D. dissertation, Harvard University, Cambridge, Mass.; June, 1963.
- [12] P. S. Landweber, "Three theorems on phrase structure grammars of type 1," *Inf. and Control*, vol. 6, pp. 131-136; June, 1963.
- [13] G. H. Matthews, "Discontinuity and asymmetry in phrase structure grammars," *Inf. and Control*, vol. 6, pp. 137-146; June, 1963.
- [14] R. J. Parikh, "Language-generating devices," Res. Lab. of Electronics, MIT, Cambridge, Mass., Quarterly Progress Rept., No. 60, pp. 199-212; January 15, 1961.
- [15] M. P. Schützenberger, "On context-free languages and push-down automata," *Inf. and Control*, vol. 6, pp. 246-264; September, 1963.
- [16] See also [E2], [G3], [G4].

B. Expositions of Languages Defined by Formal Grammars

- [1] H. Bottenbruch, "Structure and use of ALGOL 60," *J. ACM*, vol. 9, pp. 161-221; April, 1962.
- [2] E. W. Dijkstra, "A primer of ALGOL 60 programming," Academic Press, New York, N. Y.; 1962.
- [3] H. D. Huskey, M. H. Halstead, and R. McArthur, "Neliac—a dialect of ALGOL," *Comm. ACM*, vol. 3, pp. 463-468; August, 1960.
- [4] D. E. Knuth and J. N. Merner, "ALGOL 60 confidential," *Comm. ACM*, vol. 4, pp. 268-272; June, 1961.
- [5] D. D. McCracken, "A Guide to ALGOL Programming," John Wiley and Sons, Inc., New York, N. Y.; 1962.
- [6] D. D. McCracken, "A Guide to COBOL Programming," John Wiley and Sons, Inc., New York, N. Y.; 1963.
- [7] P. Naur, "A Course of ALGOL 60 Programming," Regnecentralen, Copenhagen, Denmark; 1961.
- [8] D. T. Ross, "The design and use of the APT language for automatic programming of numerically controlled machine tools," *Proc. Computer Applications Symposium*, ITT Res. Inst., Chicago, Ill., pp. 80-99; 1959.
- [9] J. E. Sammet, "Basic elements of COBOL 61," *Comm. ACM*, vol. 5, pp. 237-253; May, 1962.
- [10] J. E. Sammet, "Detailed description of COBOL," in "Annual Review in Automatic Programming," Pergamon Press, The Macmillan Co., New York, N. Y., vol. 2, pp. 197-230; 1961.
- [11] H. Schwarz, "An Introduction to ALGOL," *Comm. ACM*, vol. 5, pp. 82-95; February, 1962.
- [12] Reference Manual, 709/7090 FORTRAN Programming System, IBM Form No. C28-6054-2.
- [13] L. Bolliet, N. Gastinel, and P. J. Laurent, "Un Nouveau Langage Scientifique—ALGOL—Manual Pratique," Hermann, Paris, France; 1964.

C. General Theory of Phrase-Structure Grammars

- [1] Y. Bar-Hillel, M. Perles, and E. Shamir, "On formal properties of simple phrase structure grammars," Applied Logic Branch, Hebrew Univ. of Jerusalem, Tech. Rept. No. 4; 1960. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, vol. 14, pp. 143-172; 1961. Summarized in *Comp. Rev.*, vol. 4, pp. 213-214; September-October, 1963.
- [2] D. G. Cantor, "On the ambiguity problem of Backus systems," *J. ACM*, vol. 9, pp. 477-479; October, 1962.
- [3] N. Chomsky, "Syntactic Structures," Mouton and Co., The Hague, Netherlands; 1957.
- [4] N. Chomsky, "On certain formal properties of grammars," *Inf. and Control*, vol. 2, pp. 137-167; June, 1959. (Addendum) "A note on phrase structure grammars," *Inf. and Control*, vol. 2, pp. 393-395; December, 1959.
- [5] N. Chomsky, "Formal Properties of Grammars," in "Handbook of Mathematical Psychology," John Wiley and Sons, Inc., New York, N. Y., vol. 2, pp. 323-418; 1963.
- [6] N. Chomsky and M. P. Schützenberger, "The Algebraic Theory of Context-Free Languages," in "Computer Programming and Formal Systems," North-Holland, Amsterdam, pp. 118-161; 1963.
- [7] R. W. Floyd, "On the non-existence of a phrase structure grammar for ALGOL 60," *Comm. ACM*, vol. 5, pp. 483-484; September, 1962.
- [8] R. W. Floyd, "On ambiguity in phrase structure languages," *Comm. ACM*, vol. 5, pp. 526, 534; October, 1962.

D. Syntax-Directed Analysis

- [1] M. P. Barnett and R. P. Futrelle, "Syntactic analysis by digital computer," *Comm. ACM*, vol. 5, pp. 515-526; October, 1962.
- [2] A. L. Bastian, Jr., "A phrase-structure language translator," Air Force Cambridge Res. Labs., Hanscom Field, Mass., Rept. No. AF-CRL-69-549; August, 1962.
- [3] R. A. Brooker and D. Morris, "A general translation program for phrase-structure languages," *J. ACM*, vol. 9, pp. 1-10; January, 1962.
- [4] R. A. Brooker and D. Morris, "A compiler for a self-defining phrase structure language," Univ. of Manchester, England (undated).
- [5] T. E. Cheatham, Jr., and K. Sattley, "Syntax-directed compiling," *Proc. Spring Joint Computer Conf.*, Spartan Books, Baltimore, Md., vol. 25, pp. 31-57; 1964.
- [6] M. E. Conway, "Design of a separable transition-diagram compiler," *Comm. ACM*, vol. 6, pp. 396-408; July, 1963.
- [7] P. Z. Ingerman, "A syntax oriented compiler . . .," Moore School of Elec. Engineering, Univ. of Penn., Philadelphia; April, 1963.
- [8] E. T. Irons, "A syntax directed compiler for ALGOL 60," *Comm. ACM*, vol. 4, pp. 51-55; January, 1961. (See also reference [D13].)
- [9] E. T. Irons, "An error-correcting parse algorithm," *Comm. ACM*, vol. 6, pp. 669-673; November, 1963.
- [10] E. T. Irons, "The structure and use of the syntax-directed compiler," in "Annual Review in Automatic Programming," Pergamon Press, The Macmillan Company, New York, N. Y., vol. 3, pp. 207-227; 1963.
- [11] R. S. Ledley and J. B. Wilson, "Automatic-programming-language translation through syntactical analysis," *Comm. ACM*, vol. 5, pp. 145-155; March, 1962.
- [12] P. Lucas, "The structure of formula-translators," Mailüfterl, Vienna, Austria, ALGOL Bulletin Suppl. No. 16; September, 1961. *Elektronische Rechenanlagen*, vol. 3, pp. 159-166; August, 1961.
- [13] B. H. Mayoh, "Irons' procedure DIAGRAM," *Comm. ACM* (letter of correction), vol. 4, p. 284; June, 1961.
- [14] J. C. Reynolds, "A compiler and generalized translator," Applied Math. Div., Argonne Natl. Lab., Argonne, Ill., (undated).
- [15] S. Warshall, "A syntax-directed generator," *Proc. Eastern Joint Computer Conf.*, Spartan Books, Baltimore, Md., vol. 20, pp. 295-305; 1961.
- [16] See also [C11], [G4].

E. Syntax-Controlled Analysis

- [1] J. Eickel, M. Paul, F. L. Bauer, and K. Samelson, "A syntax-controlled generator of formal language processors," *Comm. ACM*, vol. 6, pp. 451-455; August, 1963.
- [2] R. W. Floyd, "Syntactic analysis and operator precedence," *J. ACM*, vol. 10, pp. 316-333; July, 1963.
- [3] R. W. Floyd, "Bounded context syntactic analysis," *Comm. ACM*, vol. 7, pp. 62-67; February, 1964.
- [4] R. Graham, "Bounded context translation," *Proc. Spring Joint Computer Conf.*, Spartan Books, Baltimore, Md., vol. 25, pp. 17-29; 1964.
- [5] M. Paul, "ALGOL 60 processors and a processor generator," *Proc. IFIP Congress*, North Holland, Amsterdam, pp. 493-497; 1962.

F. Non-Syntactic Methods of Analysis

- [1] E. W. Dijkstra, "Making a translator for ALGOL 60," Annual Review in Automatic Programming, Pergamon Press, The

- Macmillan Company, Inc., New York, N. Y. vol. 3, pp. 347-356; 1963.
- [2] E. W. Dijkstra, "ALGOL 60 translation," Stichting Mathematisch Centrum, Amsterdam, The Netherlands, ALGOL Bulletin Suppl. No. 10; November, 1961.
- [3] A. Evans, Jr., "An ALGOL 60 compiler," Computation Center, Carnegie Inst. of Technology, Pittsburgh, Pa., Rept. No. C.R.O-4; August 27, 1963.
- [4] R. W. Floyd, "A descriptive language for symbol manipulation," *J. ACM*, vol. 8, pp. 579-584; October, 1961.
- [5] A. A. Grau, "Recursive processes and ALGOL translation," *Comm. ACM*, vol. 4, pp. 10-15; January, 1961.
- [6] A. A. Grau, "The structure of an ALGOL translator," Oak Ridge Natl. Lab., Oak Ridge, Tenn., Rept. No. ORNL-3054; February 9, 1961.
- [7] A. A. Grau, "A translator-oriented symbolic language programming language," *J. ACM*, vol. 9, pp. 480-487; October, 1962.
- [8] P. Naur, "The design of the GEIR ALGOL compiler," *Nordisk Tidsskrift for Informations, Behandling*, pt. I, vol. 3, p. 124; 1963.
- [9] D. T. Ross, "An algorithmic theory of language," Electronic Systems Lab., MIT, Cambridge, Mass., Rept. No. ESL-TM-156; November, 1962.
- [10] K. Samelson, "Programming languages and their processing," *Proc. IFIP Congress*, pp. 487-492; 1962.
- [11] K. Samelson and F. L. Bauer, "Sequential formula translation," *Comm. ACM*, vol. 3, pp. 76-83; February, 1960.
- G. Related Work on Analysis of Natural Languages**
- [1] D. G. Bobrow, "Syntactic analysis of English by computer—a survey," *Proc. Fall Joint Computer Conf.*, Spartan Books, Baltimore, Md., vol. 24, pp. 365-387; 1963.
- [2] T. E. Cheatham, Jr. and S. Warshall, "Translation of retrieval requests couched in 'semi-formal' English-like language," *Comm. ACM*, vol. 5, pp. 34-39; January, 1962.
- [3] N. Chomsky and G. A. Miller, "Introduction to the formal analysis of natural languages," in "Handbook of Mathematical Psychology," John Wiley and Sons, Inc., New York, N. Y.; vol. 2, pp. 269-322; 1963.
- [4] S. Kuno and A. G. Oettinger, "Multiple-path syntactic analyzer," *Proc. IFIP Congress*, North Holland, Amsterdam, pp. 306-312; 1962.
- [5] See also [C3].
- H. Miscellaneous Devices Useful in Performing Syntactic Analysis**
- [1] R. W. Floyd, "Ancestor" (Algorithm 96), *Comm. ACM*, vol. 5, pp. 344-345; June, 1962.
- [2] A. W. Holt, "A mathematical and applied investigation of tree structures for computer syntactic analysis," Ph.D. dissertation, Univ. of Penna., Philadelphia; 1963.
- [3] S. Warshall, "A theorem on Boolean matrices," *J. ACM*, vol. 9, pp. 11-12; January, 1962.
- I. Supplementary Bibliographies**
- [1] R. A. Kirsch, "The application of automata theory to problems in information retrieval (with selected bibliography)," National Bureau of Standards, Washington, D. C., Rept. No. 7882; March 1, 1963.
- [2] O. Kesner, "Bibliography: ALGOL references," *Comp. Revs.*, vol. 3, pp. 37-38; January-February, 1962.
- [3] U. M. Voloshin, "Bibliography on automatic programming," Institut Matematiki Sibirskogo Otdeleniia Akademii Nauk S.S.S.R., Novosibirsk; 1961.
- [4] V. H. Yngve, *et al.*, "Towards better documentation of programming languages," (ALGOL 60, COBOL, COMIT, FORTRAN, IPL-V, JOVIAL, NELIAC), *Comm. ACM*, vol. 6, pp. 76-92; March, 1963.
- [5] W. W. Youden, "Index to the Communications of the ACM volumes 1-5, 1958-1962," *Comm. ACM*, vol. 6, pp. I 1-32; March, 1963.
- [6] "ALGOL references in the Communications of the ACM, 1960-1961," *Comm. ACM*, vol. 4, p. 404; September, 1961.
- [7] "Automatic programming—a short bibliography," in "Annual Review in Automatic Programming," Pergamon Press, The Macmillan Co., New York, N. Y., vol. 1, pp. 291-294; 1960.
- [8] See also [B13], [C5], [C6], [E1], [E5], [F9], [F10], [G1], [G3], which contain extensive bibliographies relevant to their subjects.