*Floyd:* People are already confused about this. In fact, I saw a report recently in which it was denied that ALGOL was a context-free language because one has to use context to analyze it. So maybe it's better to bring out into the open the fact that it has two meanings.

*Gorn:* But while the terminology is changing, do we have to go through the dodge that symbolic logicians have to go through in switching bound variables—substituting other names temporarily until we can return to the original names when people have forgotten their original meanings?

*Ross:* I have a handout, which I will read at a later session, which I am afraid offers still another definition for context.

*Gorn:* Good, the more the merrier. The worse the problem is the sooner someone will do something about it. Those of you who have had the patience to look at some of my definitions will know that I take an even more radical view of context. We have been talking about purely syntactic context, the surrounding characters in a string. But from my point of view the context of an expression is the processor that happens to be working on it at the moment. And at that point context transcends purely syntactical questions.

# "Structural Connections" in Formal Languages*

E. T. Irons
*Institute For Defense Analyses, Princeton, New Jersey*

**This paper defines the concept of "structural connection" in a mechanical language in an attempt to classify various formal languages according to the complexity of parsing structures on strings in the languages. Languages discussed vary in complexity from those with essentially no structure at all to languages which are self-defining. The relationship between some existing recognition techniques for several language classes is examined, as well as implications of language structure on the complexity of automatic recognizers.**

Probably the most popularly accepted definition of a *context-free* language stems from a definition by Chomsky for a context-free grammar [1], which may be summarized (though not precisely defined) as follows:

A *grammar* (type 0) consists of a set of productions of the form $\varphi \rightarrow \psi$ (meaning $\varphi$ may be rewritten as $\psi$, or $\varphi$ generates $\psi$), where $\varphi$ and $\psi$ are strings formed from a vocabulary B consisting of basic symbols $V_T$, meta variables $V_N$ and the null string I.

A *type 1 grammar* is a grammar in which all productions take the form $w_1\varphi w_2 \rightarrow w_1\psi w_2$ where $\varphi \in V$, $\varphi \neq I$, $w_1$, $\psi$, $w_2$ are strings over V.

A *type 2* (*context-free*) *grammar* is a type 1 grammar in which all productions take the form $\varphi \rightarrow \psi$ where $\varphi \in V$, $\varphi \neq I$, $\psi$ is a string over V.

A *context-free language* is a language all of whose sentences are generated by a context-free grammar. (We note that aside from minor technicalities, a set of productions in Backus Normal Form form a type 2 or context-free grammar.)

In a type 1 grammar (or type 0) the neighboring strings $w_1$ and $w_2$ (which of course may either or both be null) determine whether the transformation $\varphi \rightarrow \psi$ is applicable for a given example. The restriction which defines a type 2 or context-free grammar essentially states that the rule $\varphi \rightarrow \psi$ will always be applicable no matter what the surroundings of $\varphi$ may be in any particular string. This distinction is useful for some discussions, but the classifications of languages made here is rather too broad to be useful in the all important matter of classifying recognizers for various grammars. In the past few years, several grammars and recognizers have come into being which have significant applications in language translation and particularly in translation of formal languages including algebraic computer languages, etc. These grammars vary in power and scope from Chomsky's type 0 to those more restricted than type 2; yet the distinction between them cannot be made simply in terms of the Chomsky classification.

We propose here to establish a classification of languages aimed more at clarifying the distinctions between and unifying common concepts of some existing and proposed recognition techniques, than at exhibiting mathematical properties of grammars. We shall attempt to do so by dealing with strings from a language and the parses of these strings according to some (unspecified) grammar rather than by concentrating on the form of the grammars themselves. To this end, we adopt the following notation: 1) for basic symbols, small roman letters, a; 2) for strings of basic symbols, capital roman letters, A; 3) for syntactic categories, bracketed words [A]. We state the following basic definitions:

1. A string A *contains* the string B ($A \supseteq B$) if the string B occurs in A or is A. (We also say B is a *substring* of A); e.g., abc $\supseteq$ abc $\supseteq$ ab.
2. A string B is a proper substring of A if $A \supseteq B$ but A is not identical to B.
3. If A and B are disjoint substrings of C A precedes B ($A < B$) if A occurs before B in C.
4. An *assignment* of string A to a syntactic category [A] ($A \in [A]$) is simply a statement that A belongs to category [A] in a given instance.
5. A *parse* of a string A is an ordered set of assignments of substrings of A subject to the following restrictions:

1) Where $\beta$ is the assignment, $B \in [B]$ and $\gamma$ is the assignment $C \in [C]$ for all assignments $\gamma$ and $\beta$ in the

* Presented at a Working Conference on Mechanical Language Structures, Princeton, N. J., August 1963, sponsored by the Association for Computing Machinery, the Institute for Defense Analyses, and the Business Equipment Manufacturers Association.

parse; if B and C are disjoint, if B < C, then $\beta < \gamma$ (read $\beta$ precedes $\gamma$ in the list of assignments); if B and C are not disjoint, if $B \subset C$ then $\beta < \gamma$, if $B \supset C$ then $\beta > \gamma$, and if B = C then $\beta > \gamma$ or $\gamma > \beta$ arbitrarily. No other assignments are allowed (specifically, where B and C are not disjoint as for ab and bc in abc, we may not have an assignment for both ab and bc).

2) The set of assignments must contain at least one assignment for A.

In defining a parse this way, we are essentially setting forth a precise notation for the commonly used "parse diagram." For example, the diagram in Figure 1 is equivalent to the parse

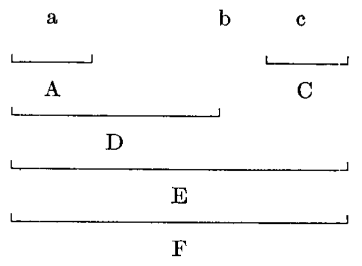


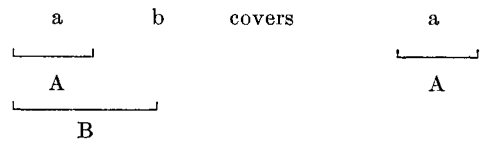
FIG. 1

NOTE. Since there is no ambiguity in the diagrams, [A] is represented by A.

$$a \in [A]$$
$$ab \in [D]$$
$$c \in [C]$$
$$abc \in [E]$$
$$abc \in [F]$$

6. We say a parse $\varphi$ is *equivalent* to a parse $\psi$ ($\varphi = \psi$) if the assignments comprising $\psi$ are exactly those comprising $\varphi$ and in the same order.

7. We say a parse $\varphi$ *covers* a parse $\psi$ ($\varphi \supseteq \psi$) if the assignments of $\psi$ are among the assignments of $\varphi$ and occur consecutively in $\varphi$ in the same order as in $\psi$. Also, we write $\varphi \supset \psi$ if $\varphi \supseteq \psi$ and $\varphi \neq \psi$. For example, the parse



8. A *grammar* is a set of rules for producing parses on certain strings of basic symbols. A grammar may, for a certain string of basic symbols, produce no parse at all, or several parses.

9. A *complete parse* for a string A under a grammar G is a parse which is covered by no other parse of A produceable under the rules of G.

Having defined a parse, we now attempt to classify certain languages and their grammars according to the parses the grammars produce on strings in the alphabets of the languages. Roughly speaking we intend to classify languages according to the complexity of interaction between parses on disjoint substrings of a parsed string. The simplest language with which we will work has no such interactions. We would choose to call this type of language *context-free* and the others of the system *context-dependent.* However, since the grammars for these languages are much more restricted than Chomsky type 2 grammars we will hopefully avoid confusion by calling our simplest class of languages "structurally unconnected" languages and more complicated languages "structurally connected".

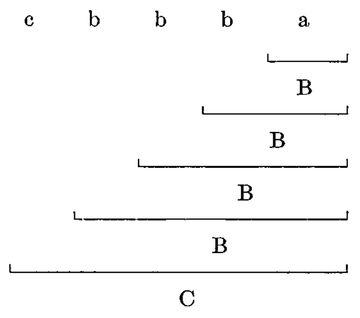
To give a precise meaning to these terms, we define

10. A *structurally unconnected* (SU) *language* L is one such that for every string A on the alphabet of L there exists not more than 1 complete parse of A; and if there exists 1 parse $\alpha$, then for every substring B of A there exists not more than 1 complete parse of B ($\beta$) and $\alpha \supseteq \beta$.

A structurally unconnected language is indeed a very simple one. It does have the property that its recognizer is very simple, since every assignment of a string is determined uniquely by the string itself. An example of a structurally unconnected language is one with the following (BNF) grammar over the alphabet abc:

$$[B] ::= a \mid b[B]$$
$$[C] ::= c[B].$$

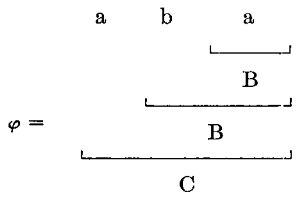
A string and its parse in this language are:



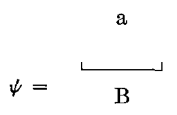
The grammar of this example is a Chomsky type 3 grammar [1] (one in which all productions take the form $A \rightarrow aB$ or $A \rightarrow a$ where a is a basic symbol and A and B are single "meta variables"). Not all type 3 grammars, however, are SU. For example (using BNF notation), the grammar

$$[B] ::= a \mid b[B]$$
$$[C] ::= a[B]$$

gives the parse



to the string S = aba, but the substring consisting of the first 'a' of S has the parse



and $\varphi$ does not cover $\psi$.

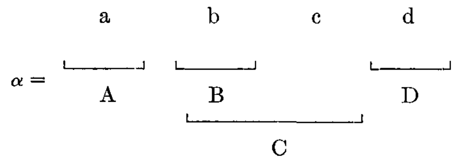
Therefore type 3 languages are in general *structurally connected* according to

11. Any language L for which a parse exists for every string considered to be "in" the language is *structurally connected* (SC) if L is not SU.

Our main interest is in classifying SC languages, since most of the interesting languages are SC. To this end we give:

12. A *bracketed substring* A of a string S (with complete parse $\alpha$) to the {left/right} of a substring B of S is (1) the longest string {preceding/following} B in S which has a complete parse $\varphi$ such that $\alpha \supseteq \varphi$; (2) if no such string exists, then A is the symbol {preceding/following} B.

We will say that A is the first bracketed substring of S after B and that if C is the bracketed string after A then C is the *second* bracketed string after B, and so on. For example, in



bc is the first bracketed string after a and d is the second.

13. For every parse $\alpha$ of every string S on the alphabet of language L, (1) let B be any substring of S (with parse $\beta$ occurring in $\alpha$); (2) let R be the M {symbols/bracketed strings} to the left of B; (3) let L be the N {symbols/bracketed strings} to the left of B; (4) let $\xi$ and $\varphi$ be the parses of L and R (respectively) which occur in $\alpha$; (5) let $\psi$ be any parse of a string containing LBR such that $\psi \supseteq \xi$ and $\psi \supseteq \varphi$. Then if B has no parse other than $\beta$ contained in $\psi$, L is said to be SC N {symbols/brackets} left and M {symbols/brackets} right, which we abbreviate SC N {S/B}LM{*S/B*}R.

The essential point in 13 is to define the extent to which symbols surrounding a string determine its parse. For a language which is (for example) SC 5SL 5SR we are guaranteed that we can give the complete parse for any string knowing only the string and the 5 symbols on either side of it.

As an example of one such language, the type 3 language given by

[B] ::= a | b[B]

[C] ::= a[B]

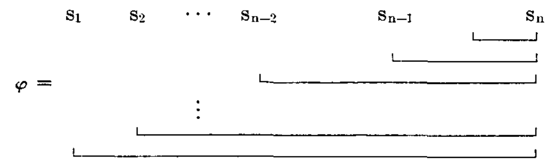
from our earlier example is SC 1 SR 0SL or SC 1 symbol to the right. Since all parsable strings for [C] take the form abbb···bba, (1) there exists no parse for any string ending in b; (2) there exists at most one parse for any string beginning with b; and (3) there exists no parse of the initial a except $\varphi$ = a, B, and $\varphi$ is not contained in any parse of ab or any string containing ab.

In fact we can state generally that: *A type 3 language* (at least as we have presented it on page 68) *is SC 1 SR.*

*Proof.* The only allowable productions of a type 3 language are of the form [A] ::= a[B] or [A] ::= a where 'a' represents a basic symbol and [A] and [B] meta variables. Therefore, the only parse for any string S = $s_1s_2 \cdots s_n$ is

$$\varphi = \begin{array}{l} s_1 \cdots s_n \rightarrow [A_i] \\ s_1 \cdots s_{n-1} \rightarrow [A_j] \\ \quad \vdots \\ s_1 \rightarrow [A_k] \end{array}$$

or diagramatically



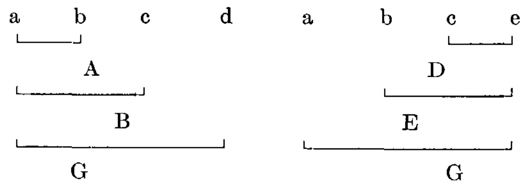
Every substring A = $S_i \cdots S_j$ ($j \geqq i$) of S must have a parse $\alpha$ of the same form. But the string B = $S_i \cdots S_jS_{j+1}$ has no parse for j < n; therefore B determines the parse of A.

A class of grammars of considerable practical interest today is a restricted BNF grammar described by Paul [3, 4]. A language which can be described by such a grammar (e.g., most of ALGOL) can be parsed using the Bauer-Samuelson technique on a table constructed automatically from productions in Paul's grammar. Such a grammar must evidently be classed as SC 1BL 1SR. This can be seen by examining the Bauer-Samuelson technique itself. In this technique, the assignments of the parse are made according to the top two elements of a stack of previous assignments and the next unparsed symbol (left to right) of the string being parsed. The N assignments of the stack are, in fact, the N bracketed strings to the left of the unparsed string. Since assignments are made over the string covered by the top one or two assignments of the stack, any language parsable by this technique must be SC 1BL 1SR at most.

For example, an SC 1BL 2SR language which cannot be parsed by the Bauer-Samuelson technique is given by

| | |
|---|---|
| [A] ::= ab | [D] ::= ce |
| [B] ::= [A]c | [E] ::= b[D] |
| [G] ::= [B]d | [G] ::= a[E] |

The parses of two strings of this language are:



The parse of ab is affected by the second symbol to the right.

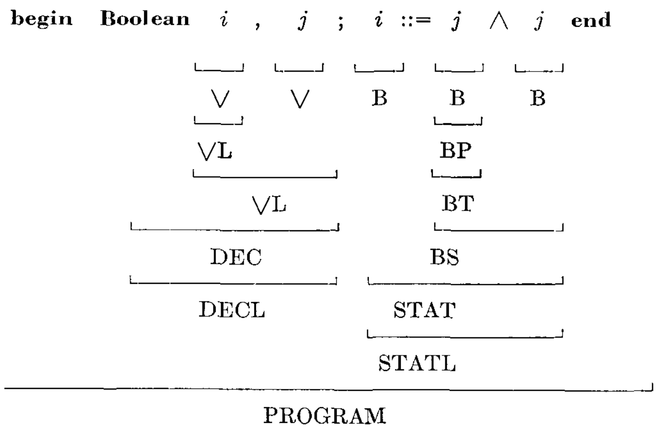
The most general BNF grammars are structurally connected in such a way that one cannot even fix the degree of connectedness for a specific grammar. Consider, for example, the grammar

| | |
|---|---|
| [S] ::= b | [T] ::= b |
| [S] ::= x[S]x | [T] ::= x[T]x |
| [R] ::= p[S] | |
| [R] ::= q[T] | |

In the string px···xbx···x, the substring b has two parses ([S] or [T]) inside all the strings x···xbx···x which are composed of the N bracketed strings to the left and right where N is the number of x's in the string. Since we may have any number of x's we cannot fix the degree of connectedness for this language.

The nature of the difference between BNF languages and yet "more complicated" languages must lie then not in the degree of connectedness, as we have defined it, but in the nature of the connections.

It is characteristic of all BNF grammars that the information needed to parse a substring is contained entirely in the "names" of a certain number of brackets to the left and right of the substring. No information is needed about the structure of the parses associated with these brackets. In languages "more complex" than BNF, it will in general be true that the structures of parses outside of a substring will affect the parse of the substring. It is precisely this distinction that separates "context free" from "non-context free" languages in most informal definitions of the terms. Consider, for example, a string from ALGOL-60 (whose parse cannot be generated from a BNF grammar)



(The parse given here is not intended to conform even partially to the official ALGOL syntax, but is given to illustrate the point at hand as simply as possible.)

The implication in this parse is that the occurrences of *i* and *j* are given the parse [B] because the declaration Boolean *i*, *j* has occurred in a declaration list and that they would be given another parse (say [I] for integer) if the declaration had been different.

The information necessary to give the assignment [B] to *i* is, in fact, contained in the second bracket to the left ([DECL]) but the "name" of this bracket is not all the information needed to parse the *i*. Rather the parse of *i* depends on the detailed structure of the **parse** of [DECL] (namely, that *i* occurs in a "variable list" after the declarator **Boolean**. (One may say that the occurrence of '∧' suffices to determine that *i* and *j* must be Boolean; however, one must clearly be able to distinguish the types of variables independently of the local context to be able to observe that e.g., if the '∧' had been a '+' the program is not parsable.)

Let us informally define languages such as those of the last example to be languages "structurally connected in depth" or SCD languages. There are apparently additional classifications of complexity in languages involving perhaps the specific nature of structural connections. Again there are languages which are not even parsable in the sense we have used here (for example, the Theorems of the Prepositional Calculus, etc.). The languages thus far discussed here, however, cover most of those for which automatic recognition techniques exist.

Rather than continuing the attempt to classify more and more complex language structures, we choose to devote the remainder of the paper to a brief discussion of the implication for recognizers of the classification thus far established. In particular, it is evident that the complexity and efficiency of recognition techniques is closely related to the complexity of the languages in the sense we have used here. A recognizer for a SU language is little more than a table lookup algorithm. Languages which are SC a limited number of symbols or brackets to the left or right will tend to have efficient recognizers. The apparent great efficiency of the Bauer-Samuelson technique lies in the fact that the restrictions imposed on the grammar allow each assignment to be made once and for all (i.e. once an assignment is made, it is never rescinded), for a given string. Less restricted grammars such as BNF, in particular (for a left-to-right processor) those which are SC several symbols or brackets to the right, require recognizers which make tentative assignments while parsing a string and await further developments in the parsing to determine whether a given assignment is kept or rejected for the complete parse. Languages which are SCD to the left may require dynamic modification of grammar specifications or complicated intermediate tabling procedures in their recognizers (for a left to right recognizer). Languages which are SCD to the right may require multiple-pass recognizers or in fact may not even have a recognizer (for example a programming language which is "self defining" in the sense that statements in the language may define the syntax and semantics of other statements which may occur either before or after the defining statement).

In conclusion, I believe that the invention and continued refinement of automatic recognizers for more and more complex grammars, will have a profound effect on future programming languages particularly in the area of self-defining languages, and that investigations into these areas may generate substantial contributions to sophisticated activities in natural language manipulations, probabilistic recognizers, and the like.

## REFERENCES

1. Chomsky, N. On certain formal properties of grammars. *Informat. Contr. 2*, 137, 167.
2. ——. A note on phrase structure grammars. *Informat. Contr. 2*, 393–395.
3. Paul, M. *A General Processor for Certain Formal Languages. Gymbalie Languages in Data Processing*, Gordon and Breach, London 1962, pp. 65–74.
4. Eickel, J., Paul, M., Bauer, F. L., Samuelson, K. "A Syntax Controlled Generator of Formal Language Processors." Inst. fur Ang. Math. Univ. Mainz. (September, 1962).
5. Backus, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. Proc. Internat. Conference on Information Processing, UNESCO, June, 1959, pp. 125–132.
6. Naur, Peter (Ed.) Report on the algorithmic language ALGOL 60. *Comm. ACM 3* (1960), 299–314.
7. Irons, E. T. A syntax-directed compiler for ALGOL 60. *Comm. ACM 4* (1961), 51–55.

## DISCUSSION

· · · Ross commented on both the Floyd and Irons papers with a prepared note entitled "On Context and Ambiguity in Parsing." This note is included in the papers of session 6 · · ·

*Graham:* The talk about context is very misleading. People are talking about context of characters in strings, context of syntactic types in strings, and mixtures of these.

*Newell:* I have an argument with the goals of your work. Indeed it seems to me that you are trying to make a connection between a simple theory and simple recognizers. This doesn't make any sense because we have no real standard for a recognizer in the first place. For example, what would happen if we changed the underlying memory structure to an associative memory? Then the kinds of things that it would be economic to search for would be completely different.

*Irons:* An associative memory could certainly affect the kinds of searching which would be reasonable. However, I think we would still have the same level of complexity.

*Gorn:* The fact that Ross does not keep syntactic types in his trees but keeps only the operators and pointers to symbols might increase the speed of the parsing. How are $n$-ary operations handled.

*Ross:* We can always represent an $n$-ary operation by a string of binary operations.

*Merner:* Behind what you, and Irons in the preceding paper, have presented seems to lie a tacit assumption that it is necessary or at least desirable that practical languages be syntactically unambiguous. Algol 60 unrevised has an interesting but semantically well-defined syntactic ambiguity in the source language:

$$\textbf{if } B \textbf{ then } X \textbf{ else } Y < Z$$

If $X$ was a formal, call-by-name parameter the following two bracketings would occur depending on the type of the actual parameter.

$$X \textbf{ real} \text{ or } \textbf{integer: } (\textbf{if } B \textbf{ then } X \textbf{ else } Y) < Z$$
$$X \textbf{ Boolean: } \quad \textbf{if } B \textbf{ then } X \textbf{ else } (Y < Z)$$

This obviously can be implemented by inserting both meanings in the target language and selecting the appropriate meaning on the basis of the type of the actual parameter. Are such language features really a priori undesirable or do they add power to the language?

*Irons:* I think that they add power.

*Ingerman:* What is a definition of ambiguity, in the sense that Merners example is clearly *unambiguous* at run-time? If both interpretations are compiled and the correct one selected at run-time, then there is no ambiguity at compile time.

*Gorn:* They constitute, however, an ambiguous selection.

*Warshall:* Does, then, one have to have an infinite number of meanings in order to admit that the statement is really ambiguous?

*Wegstein:* Doesn't the parse depend on how the statement in the language is to be used? For example, the parse of the string "317" would be different if the intent was to generate the string "three hundred and seventeen" than if the intent was to generate the binary equivalent.

*Irons:* Yes, I agree.

*Cheatham:* Incidentally, I think Wegstein's comment provides a good indication of why the Uncol concept is a poor one. That is, it is often difficult to specify even a parse unless one knows the use to which the result will be put. The use of a standard "intermediate language" requires, of course, that one is committed to a fixed form of language from which to generate machine code in addition to a fixed parse.

*Kirsch:* It is desirable to measure the complexity of a language. It is difficult to measure, however, in terms of processors for the language and whether or not the language is hard to process. There are some priori measures which could be defined by canonical questions like: Does there exist a decision procedure for determining whether or not a string belongs to the language, and so on?

*Irons:* I think that it would be very useful to be able to classify languages on the basis of a measurable complexity, but the measure should be the recognizer.

*Kogon:* From what you have said it appears that you consider ambiguities of the type $\textbf{if } B \textbf{ then } X \textbf{ else } Y < Z$ to be useful and desirable; I therefore assume that you would also welcome the possibility to make the substitution of

$$\textbf{if } B' \textbf{ then } X' \textbf{ else } X'' \quad \text{for} \quad B.$$

The problem seems to be not so much whether these ambiguities can be handled but rather whether they are desirable.

The question of desirability seems to be linked to the question of how much should a person using a language be expected or required to know and understand about the procedure for the solution of his problem. If you do not insist on precise and detailed knowledge on the part of the programmer, how do you justify the desirability or provide a language for a programmer at all?

*Dijkstra:* I should like to correct one of the speaker's statements: it has been proved that a one-pass load-and-go translator for Algol 60 can be made which poses no restriction whatsoever on the relative positioning of declaration and use of identifiers.

*Irons:* This must lead to some real inefficiencies at run time.

*Perlis:* What do people mean by one pass?

*Wilkes:* With a big enough memory almost anything can appear one-pass.

*Dijkstra:* I am asking myself whether the speaker is willing to consider two languages equivalent if between texts from the different languages a one-to-one correspondence can be established in both directions by a mechanical process.

For the purpose of illustration, some examples are: (1) Writing with black ink on a white paper versus writing with white chalk on a black board. (2) Writing the characters in the reverse order. (3) Replacing in an Algol program each capital letter by a point followed by the corresponding small letter. (4) Introducing separate characters for unary plus and minus. (5) Permuting all expressions to reverse polish.

*Holt:* One can extend this list and ask if the statement of Fermats Last Theorem is equivalent to its proof.

*Gorn:* One should not ask the question, "Is language 1 equivalent to language 2?". Rather, one should ask, "Is language 1 plus translator 1 equivalent to language 2 plus translator 2?".

*Backus:* I would like to hear some discussion on how we can measure the efficiency of some of these processors.

*Perlis:* One measure is the ratio of the overage number of instructions the compiler executes per instruction the compiler produces. I understand that the ratio for 220 BALGOL was 700.

*Bauer:* We have found ratios of 46 to 100 depending on the machine used.

*Dijkstra:* We are running 1000.

*Greiback:* Concerning pp. 6–7ff. in IDA-CRD Working Paper No. 93, "An Error-correcting Parse Algorithm," the conjecture is correct—one can always find a BNF specification with desired properties. The theorem states: *For every context-free psg one can find context-free psg whose rules are of form:* $Z \rightarrow aY_1, \cdots, Y_n$ *where a is a terminal symbol, Z and* $Y_1$ *are nonterminal.* Here one can link $(B, a)$, eliminate the pointers, and the algorithm is almost the multiple-path analyzer of Kuno-Oettinger. Details can be found in my thesis, "Inverse of Phrase Structure Generators" [Harvard Report NSF-11] and an unpublished paper, "A New Normalform Theorem for Phrase Structure Generators." The bracket ({ }) device you use is indeed an intermediate.

Any BNF system (context-free psg) can be mechanically placed in this special form (which I call standard form), preserving ambiguities (or lack thereof).

# FORTRAN IV as a Syntax Language*

B. M. Leavenworth
*IBM Corporation, White Plains, New York*†

## 1. Introduction

It is a generally known fact that an algorithmic (source) language is defined by its processor in the sense that meanings of statements in that language are defined in terms of a target language which is produced by the processor [1]. The processor does not exhibit explicitly the syntax of this source language but rather hides the syntax in the details of its construction.

There is a trend toward specifying the syntax of context-free programming languages by using generators or production schema represented in some formal symbolism such as Backus normal form. We believe that it will be more convenient to specify the syntax of languages *behaviorally* [2] so that a specification of this type can be easily converted to recognition algorithms by a suitable processor.

The purpose of this paper is to show how FORTRAN IV can be used as a syntax language, that is, to specify the syntax of a source language in a suitable form, then to compile these specifications as recognizers together with generators to synthesize a given target language. In order to transform an input string of basic symbols into a target string, let us define a *processor* (which solves this problem) as a set of recognizers and generators $P = \{R_1, R_2, \cdots, R_n, G_1, G_2, \cdots, G_n\}$ together with some control mechanism which governs the sequencing of the recognizers. With each $R_i$ is associated a corresponding $G_i$ (which may be null). There are two types of recognizer: (1) *basic recognizer* (recognizes basic symbols): this type is a recognizer either for single symbols of the input string or a class of symbols (such as the class of letters); (2) *string recognizer*: this type is a recognizer of syntactic types (class names), which encompass subsets of the input string, as well as temporary strings which are formed in the course of translation. These temporary strings are usually created to "remember" information previously encountered on the input string.

If $\alpha_0$ denotes the input string and $\alpha_T$ the target string then: $P\alpha_0$ implies $\alpha_T \leftarrow G_r G_q \cdots G_p\alpha_T$. That is, the application of the processor P to $\alpha_0$ results in a sequence of generator transformations on $\alpha_T$ (initially null). Each recognizer except one, called the language recognizer (corresponding to the root of the tree describing the source-language) has a unique successor determined at run time. This successor is a function of the truth value of the recognizer and the control mechanism.

The syntax specification of a language implicitly determines the control flow between recognizers, and therefore *is* the control mechanism of a syntax processor. To allow recognizers to call themselves recursively, a control pushdown must be provided which becomes part of the control mechanism. This type of organization allows a considerable amount of flexibility. For example, we can operate on more than one input string, shift attention from one tree to another, have generators call on recognizers for additional information, and so forth.

## 2. Language Properties Required for Syntax Translation

If an algorithmic language can be used to specify the syntax of a source language, then this specification can be converted (compiled) into an algorithmic recognizer. Some of the properties required for this type of conversion are now discussed.

*Sequencing of Boolean Expressions.* Assuming that the language under consideration contains Boolean expressions, the type of sequencing we have in mind is the "optimization" described by Huskey and Wattenburg [3].

* Presented at a Working Conference on Mechanical Language Structures, Princeton, N. J., August 1963, sponsored by the Association for Computing Machinery, the Institute for Defense Analyses, and the Business Equipment Manufacturers Association.

† Present Address: Advanced Systems Development Division, International Business Machines Corp., Yorktown Heights, N.Y.