A GENERAL PARSING ALGORITHM
FOR CONTEXT-FREE GRAMMARS

A Thesis

Presented to

the Faculty of Graduate Studies and Research

The University of Manitoba

In Partial Fulfilment

of the requirements for the Degree

Master of Science

in the Institute for Computer Studies

-----

by

Rainer Kossmann

February 1971

© Rainer Kossman 1972

ABSTRACT

The work presented in this thesis arose out of an investigation concerning the compilation of a new programming language, namely Algol-68. A radically new syntax representation used in the definition of this language makes the syntax analysis of this language a major problem. One promising method of handling this problem is the transformation of most of the Algol-68 syntax to a context-free grammar which is readily handled by well-known and existing methods. A brief description of this grammar transformation is given in chapter 5 of this thesis.

A further result of the preliminary investigation was that it was found to be extremely useful and possibly necessary to have available a general parsing algorithm for context-free grammars. This algorithm was developed by the writer and forms the major subject of this thesis. The theory of this algorithm is presented in chapters 1 through 4. A listing for a Pl/1 program, fully working but still under development, plus a few examples are appended.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. P. R. King, for the help and encouragement that he provided throughout the development of this work.

Further thanks go to the Department of Computing Science which made available the text-editing facilities used in the preparation of this thesis. This feature has saved a considerable amount of work which would have been required in the preparation of multiple drafts by manual means.

Finally, I would like to thank my wife, Maureen, for her patience with a somewhat temperamental machine and for many hours spent in typing the original draft of this thesis into the computer files.

TABLE OF CONTENTS

Chapter		Page
1	Introduction	1
	1.1 Basic definitions	4
	1.2 A tree structure for syntax	10
2	Floyd's Algorithm	14b
	2.1 The syntax table	14b
	2.2 The analysis record	22
	2.3 The parsing algorithm	28
3	Recursion	41
	3.1 Basic definitions	41
	3.2 An algorithm for handling recursion	60
4	A general recursion handling algorithm	80
	4.1 Handling type 2 and type 3 cycles	80
	4.2 Administration of the recursion list	85

Chapter		Page

5 . Parsing of Algol-68 92

5.1 Basic definitions 92

5.2 Reducing two-level syntax to one-level syntax 104

5.3 The context-problem 110

5.4 Relevance of the general parsing algorithm 115

6 . Conclusions and suggestions for future work 117

Chapter 1

INTRODUCTION

Many parsing algorithms for context-free grammars require the grammar to be restricted to some proper subset of context-free grammars. Indeed, general parsing algorithms for context-free grammars are rather scarce. This is not by accident since a general parsing algorithm tends to be of a rather inefficient nature. By placing a few restrictions on the type of context-free grammar that a parsing algorithm will work for, it is often possible to derive greatly increased efficiency in the parsing algorithm for such grammars.

One example which demonstrates this clearly consists of the Wirth-Weber precedence grammars (1). Efficient parsing algorithms for these grammars exist

and a recent paper by A. Learner and A. L. Lim demonstrates an algorithm that will transform any context-free grammar into an equivalent Wirth-Weber precedence grammar (2).

A similar state of affairs exists for other classes of context-free grammars which are subject to some restriction. It is often possible to find a grammar satisfying a given set of restrictions but which is at the same time equivalent to a given context-free grammar.

The obvious question that one must ask now is whether it is worthwhile to design a general parsing algorithm for context-free grammars, even though such parsing algorithms may prove too inefficient to be feasible as a parsing algorithm for a production compiler, or interpreter.

The answer is an emphatic "yes" for the following reason:

The language designer is not as concerned

initially with the type of grammar he must provide as with the type of problem the grammar must solve. The design of the grammar is thus his major problem and will require the bulk of his effort. He should thus be expected to have at his disposal the largest possible set of grammars. Furthermore, he should expect to do a minimal amount of rewriting of a grammar he has designed in order to test the grammar under some system. A general parsing algorithm for context-free grammars fulfills these requirements very nicely as the language designer is provided with a tool that will parse any context-free grammar. Once a grammar for a particular application has been designed, the language designer may then of course consider the problem of transforming his grammar into one which will conform to the restrictions imposed by a particular parsing algorithm which he wishes to use in a production compiler or interpreter.

The reason just cited, and the fact that no

general top-down, left-right parsing algorithms for context-free grammars were known to the writer, were considered sufficient justification for the development of such an algorithm.

The remainder of this chapter will be dedicated to providing the basic framework of definitions within which the thesis will be presented.

1.1 BASIC DEFINITIONS

A convention adopted throughout the thesis is that single, underlined, capital letters represent sets whereas single capital letters, indexed or unindexed, represent single elements of a set. Thus, "$\underline{B}$" represents a set and both "B" and "B_1" represent single elements of some set. Additionally the notation "$\underline{A}$*" is meant to represent the set of all strings that can be formed with elements of the set $\underline{A}$.

The syntax of a context-free phrase structure grammar is expressed with the aid of a finite alphabet $\underline{A}$. in this thesis $\underline{A} = \underline{S} \cup \underline{D}$ where

$\underline{S} = \{a, b, c, \ldots, x, y, z\}$ is the set of "syntactic marks", except where otherwise stated

$\underline{D} = \{:, ;, ., ,\}$ is the set of "other syntactic marks".

Define $\underline{S}^*$ as the set of (nonempty) strings over $\underline{S}$. The elements of $\underline{S}^*$ are termed "protonotions." Define a context-free phrase structure grammar (c.f.p.s.g.) as a 4-tuple

($\underline{V}$, $\underline{R}$, symbol, Z) where

$\underline{V} \subset \underline{S}^*$ is a finite set, known as the set of "notions",

$Z \in \underline{V}$ is a particular notion known as the "head",

$\underline{R}$ is a finite set of rules of the form

1) A: . called the empty rule, or

2) A: B_1 , B_2 , ..., B_n . $n \geq 1$

where $B_i \in \underline{V}$, i=1,2, ..., n and $A \in \underline{N}$ where $\underline{N}$ = $\underline{V}$ - $\underline{T}$. $\underline{T}$ is the subset of $\underline{V}$ consisting of those elements of $\underline{V}$ which end with the particular protonotion "symbol". Elements of $\underline{T}$ are called (terminal) symbols. Elements of $\underline{N}$ are called non-terminals.

Additionally, any set of rules of $\underline{R}$ with identical left hand sides, say "A: B., A:C., ..., A:Z." are rewritten as "A: B; C; ...; Z.". Furthermore, there must not be any rules A: B. $\in \underline{R}$ such that $A \in \underline{T}$.

A few further definitions are required. Let members of the set $\{u,v,w,x,y,z\}$ be elements of $\underline{V}^*$. We say that "w directly produces v (written '$w \Rightarrow v$') by application of the rule U:u." if there are (possibly empty) strings "x" and "y" such that "w=xUy" and "v=xuy".

We say that "w produces v" (written $w \overset{*}{\Rightarrow} v$) if

1. $w = v$ or

2. $w \overset{*}{\Longrightarrow} u$ and $u \Longrightarrow v$, where $u \in \underline{V}^*$.

The "language generated by a grammar G" (written L(G)) is defined as

$L(G) = \{z \mid Z \overset{*}{\Longrightarrow} z \text{ and } z \in \underline{T}^*\}$ where, as before, "Z" is the head of the grammar G and "$\underline{T}$" is the set of "symbols" of the grammar G.

To explain these definitions a little more clearly, a short example is now in order. Consider the context-free, phrase structure grammar defined by the 4-tuple ($\underline{V}$, $\underline{R}$, symbol, Z) where

$\underline{V}$= {identifier, letter, digit, asymbol, bsymbol, onesymbol} ,

$\underline{R}$ is the set of rules

{identifier: letter; identifier, letter; identifier, digit.

letter: asymbol; bsymbol.

digit: onesymbol.}

"symbol" is the protonotion "symbol", and Z is the notion "identifier".

Then the protonotions such as "identifier", "letter", and "asymbol" are also notions by virtue of the fact that they are in V.

The set of rules R consists of three rules. To aid in the reading of these rules, the other syntactic marks {:, ;, ., ,} may be read as follows:

"is defined to be" for ":",

"or" for ";", and

"followed by" for ",".

The other syntactic mark "." is not read but serves merely to indicate the end of a rule.

Thus, the rules in R may now be read as

"(an) identifier is defined to be (a) letter or (an) identifier followed by (a) letter or (an) identifier followed by (a) digit."

"(a) letter is defined to be (an) asymbol or (a) bsymbol."

"(a) digit is defined to be (a) onesymbol."

Quantities in brackets ( ) have been added to improve readability.

It should now be noted that the members of T are not further defined by R. The implication is that the definition of symbols is to be provided by the implementor of a language defined by the afore mentioned syntax. The implementor is thus free to choose his own representation for a symbol, be it an English alphabetic character, a Greek one, or possibly several different characters for the same symbol.

The symbols may thus be regarded as undefined notions and it may be considered that an implementor of a language will provide his own définitions such as for example:

asymbol : a.

asymbol : A.,

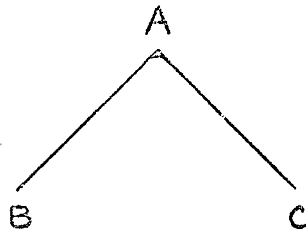
or else will provide a routine that will be used in recognizing symbols.

Members of T may thus be regarded just as any other notions and their use effectively removes from the syntax definitions any terminal symbols.

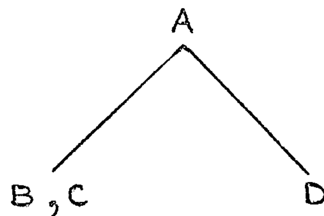
1.2 THE TREE STRUCTURE FORM OF SYNTAX DEFINITIONS

Occasionally, a particular example will be more easily understood if the syntax definitions are represented in the form of a tree structure. The convention that will be adopted here is that the

lefthand side of a rule will be a node of the tree and there will be one branch emanating downward from this node for each alternative of the rule. Thus, the rule A : B; C. would have the following appearance in a tree structure:



If an alternative consists of more than one notion, then these are represented on the tree as nodes separated by a comma. Thus, A : B, C; D. would have the appearance



In the latter case, there will be a set of branches emanating from each such node of an alternative with multiple nodes. Thus, the set of rules

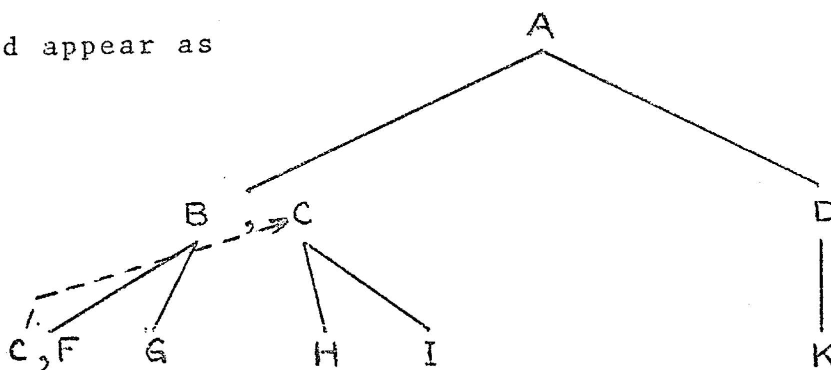
A : B, C; D.

B : C, F; G.

C : H; I.

D : K.

would appear as



It may now happen that a node such as C in the rule B : C, F; G. has been previously defined in the tree. In that case, it will not be necessary to generate a new subtree for this node C. Instead, a dotted line to the previously defined C will indicate where the definition for node C may be found in the tree.

If the syntax tree for a context-free, phrase structure grammar is drawn up, there will be two types of nodes in the tree:

a) those which have branches emanating downward from the node

b) and those which do not have branches emanating downward from them.

The nodes that do not have downward emanating branches may be called end nodes. There exist two types of end nodes:

a) those that refer back into the tree because a previous definition for the node exists

b) those nodes which are symbols and for which no definition exists.

To conclude the discussion on the definition tree for context-free grammars, one important point must be stressed. This point is the distinction that exists between notions and nodes. A "notion" is a member of $\underline{V}$ which may however occur as more than one "node" of the definition tree for a context-free grammar. These "nodes" are quite distinct from each

other.

In further discussions, the concept of "node" will be extended to cover the "node of a syntax rule". Thus, in the syntax rule

A: B, C; B, D.,

the "notions" A, B, C, and D are "nodes" within that syntax rule. Note further that the first node "B" of the syntax rule is quite distinct from the second node "B" of that syntax rule. It is evident that, if a definition tree is drawn up for a syntax rule, each node of the syntax rule will be represented as a node on the definition tree and to each node of the definition there corresponds a node of the syntax rule.

The basic definitions within which the thesis will be presented have now been covered. The following chapter will be devoted to presenting a parsing algorithm due to Floyd which will form the major part of the general algorithm presented in this thesis.

CHAPTER 2

FLOYD'S ALGORITHM

The parsing algorithm presented in this chapter is a modified version of an algorithm due to Floyd (3). It is a top-down, left-right parsing algorithm which includes backtracking.

In this chapter, a short section on terminology will be followed by a description of the two major data areas used by the parsing algorithm; the syntax table and the analysis record. Following this, the operation of the parsing algorithm will be discussed along with an example that demonstrates its operation.

2.1 THE SYNTAX TABLE

In this section, the structure of the syntax table will be explained. However, some of the terminology used must first be defined.

2.1.1 TERMINOLOGY

Suppose the following is a rule of a context-free grammar.

A:B;C,D.

Then B is called a "son" of A, A is called a "father" of B, and D is called the "next component" or simply "component" of C. Infrequently, C may be referred to as the "previous component" of D. Note that A is also the father of C and D and that C and D are sons of A. C is further called the "brother" of B.

2.1.2 THE TABLE

The approach used here is for the syntax definitions to be stored in a list structure form instead of a string of syntax definitions as used by Floyd (3). The advantage of using a list structure derives from the fact that fast parsing algorithms may be constructed for grammars specified in this form, especially if the parsing algorithm is written in a low

level language such as Assembler. At the same time, the parsing algorithm remains simple for grammars specified in list structure form(see also 8).

The list structure form of the syntax definitions as used here consists of one four element line in a syntax table for each definition. This line has the following configuration:

brother	component	son	code

The fields for the brother, component, and the son contain pointers to the respective rule within the syntax table for these quantities. For example, in the rule A: B; C, D., there will be a line in the syntax table for the notion A. The "son" field for the notion A will contain a pointer which points to the line in the syntax table in which the definition for B is given.

However, there is a brother to the notion B and

this brother consists of the notion C followed by the notion D. To note this fact, there will be a pointer in the "brother" field of B that points to the line in the syntax table in which the notion C is defined. The "component" field of the notion C will contain a pointer to the line in which the notion D is defined.

Absence of a brother or a component is noted by a zero in the corresponding field.

As yet, no reason has been given for the existence of the code field and this situation will now be rectified. It was mentioned previously that no terminals exist within the syntax definitions but there are "symbols" for which no further definitions exist. The parser must be able to recognize the occurrence of symbols and this is done by means of placing a terminal symbol code in the code field of a notion which is a symbol.

Recognition of terminals will be done by a recognizer routine which is activated whenever a code field contains a terminal symbol code. The subordinate

field of the line in the syntax table for a symbol will contain a pointer which points into a terminal symbol list and which will be used by the recognizer in trying to recognize a symbol.

Employing the scheme of a terminal symbol list for the recognition of terminals, it is relatively easy to construct a recognizer which not only recognizes single characters but also syntactic units which consist of the concatenation of two or more characters or which have several different character representations.

The parser must now test the code of every notion considered during a parse to determine whether or not the notion is a symbol. Great gains in generality can now be realized by making the code field a pointer which will point to a routine; the recognizer in the case of symbols.

As with many other topics, theory and practice do not always go hand in hand in the parsing of grammars. The difficulty in parsing of grammars of

course is that it is possible to construct context-free grammars in theory but that in practice the programming languages are generally based on syntax definitions which are not entirely context-free. It is thus necessary to be able to carry out special actions for some notions other than symbols. This is facilitated in the approach used above at no extra cost to the parsing algorithm by simply placing pointers to a special handling routine into the code field of notions requiring such special handling. The need for recognizing symbols of course is the reason why other special handling routines can be incorporated at no extra cost to the parsing algorithm.

As an example, the following grammar previously used in Chapter 1 will be encoded into a syntax table. Let R be the set of rules

{identifier: letter; identifier, letter;
identifier, digit.
letter: asymbol; bsymbol.
digit: onesymbol.}

Then, the following table is the syntax table for this grammar:

SYNTAX TABLE

#	Notion	Broth	Comp	Son	Code
1.	identifier.	0	0	2	0
2.	letter;	3	0	7	0
3.	identifier,	5	4	2	0
4.	letter;	5	0	7	0
5.	identifier,	0	6	2	0
6.	digit.	0	0	9	0
7.	asymbol;	8	0	1	t
8.	bsymbol.	0	0	2	t
9.	onesymbol.	0	0	3	t

Thus, line 1 states that the son for "identifier" is to be found in line 2; i.e. the syntax definition for "identifier" begins in line 2. Line 2 states that the "identifier" of line 1 may be possibly a letter which is defined in line 7 (pointed at by "son") or alternatively may be defined in line 3 (pointed at by "brother").

It may therefore be said that an alternative definition for "identifier" is to be found in line 3. This alternative definition consists of the two components "identifier, letter".

The "4" in the "component" field of line 3 points out the fact that the fourth line, a definition for letter, is the "next component" of line 3. Thus, lines 3 and 4 are just one of the definitions of the "identifier" of line 1.

Of interest now is the "son" field of line 3. Since line 3 must define an "identifier", the fact that "identifier" has been previously defined in line 1 is noted. In line 1, one finds that "identifier" is defined starting at line 2 (see the "son" field of line 1) and thus, this 2 is merely copied in to the "son" field of line 3. Line 3 therefore states that "a definition for identifier may be found starting at line 2".

Only the "symbols" of lines 7,8, and 9 remain unexplained. The "t" in the code field of these lines

is a code which indicates that these are terminals and require a recognizer. If any of these lines is made a goal of the parsing algorithm, the recognizer will use the "son" of these lines as an indication of what "symbol" must be recognized. For example, in this case, the 1, 2, or 3 might indicate the first, second, or third symbol in some terminal symbol list which is used by the recognizer.

2.2 THE ANALYSIS RECORD

The analysis record is an area in which details of a particular parse are recorded. This analysis record will normally be stored and passed to a semantic routine at a later stage for the purpose of code generation. Before going into details about the structure of the analysis record, it is again helpful to first define a few terms.

2.2.1 TERMINOLOGY

Each line in the analysis record will be of form

pointer	superior	subordinate	predecessor

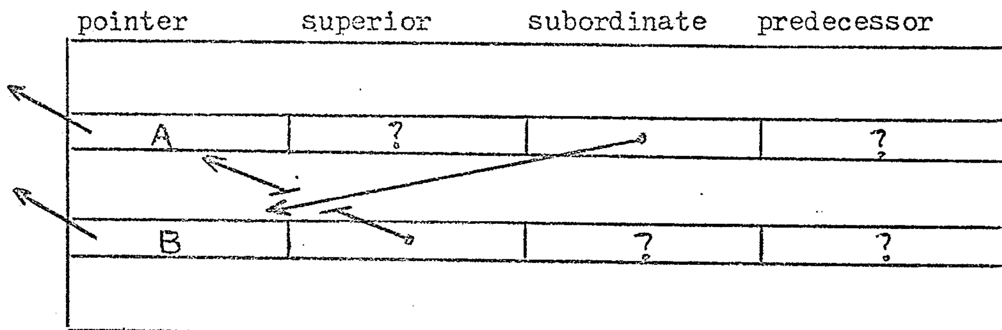
and will have a line number denoting its position within the analysis record.

At any stage during the parse, the parsing algorithm will be considering a goal and there will be one line in the analysis record corresponding to this goal. "Pointer" in the above analysis record serves as a link to the syntax definitions which ties the line in the analysis record to the specific goal in the syntax tables. This goal under consideration will in general have a father, a son, and a previous component and there will generally be lines in the analysis record for each of these quantities. In the above diagram, superior, subordinate, and predecessor are links which hold the line number within the analysis record of an

output line for the father, son, and the previous component respectively. Absence of one of these quantities is denoted by a zero in the respective field. The terms "pointer", "superior", "subordinate", and "predecessor" have now been defined and the operations carried out with the analysis record are discussed next.

2.2.2 THE ANALYSIS RECORD ORGANIZATION

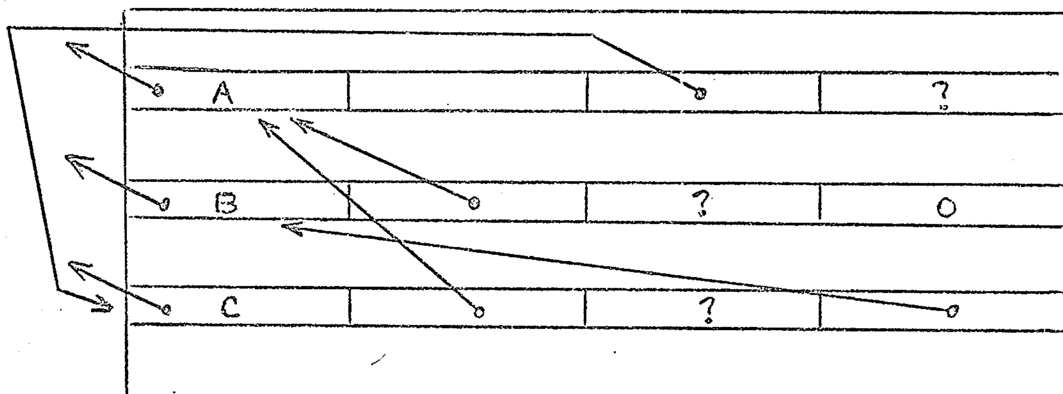
Since a goal may have more than one subordinate, the subordinate in the analysis record will contain a pointer to the last subordinate considered. Consider for example the rule A: B, C. During the parse, a line will exist in the analysis record for the goal A. When the goal B is considered as a subgoal, the subordinate field of the line for A will contain a pointer to the line for goal B as shown below:



Once B has been recognized, C must be made the next subordinate of A. This is done by generating a line in the analysis record for C with the following rearrangement of pointers:

1) the subordinate of A is placed in the predecessor field of C

2) a pointer pointing to the new line for C is placed in the subordinate field of A.



Thus, it is possible to reconstitute an analysis by locating the last subordinate of a goal as indicated by the subordinate pointer for that goal and then chaining back through the predecessors to locate all

subordinates of the goal.

The following analysis record constitutes the analysis of the identifier "ab1". The grammar used is the same grammar defining "identifier" that was used previously. The syntax table for this grammar was previously given on page 20.

ANALYSIS RECORD

#	type	point	sup	sub	pred
1	identifier	1	0	8	0
2	identifier	5	1	6	0
3	identifier	3	2	4	0
4	letter	4	3	5	0
5	asymbol	7	4	0	0
6	letter	4	2	7	3
7	bsymbol	8	6	0	0
8	digit	6	1	9	2
9	onesymbol	9	8	0	0

The reader may have noticed in the discussion of the syntax table and the analysis record that the quantities "father" and "son" of the syntax table appear to be identical to the quantities "superior" and

"subordinate" of the analysis record. This is not quite correct. The quantities "father" and "son" are links which connect the syntax definitions within the syntax table. Quantities "superior", "subordinate", and "predecessor" on the other hand are links within the analysis record which connect the various components of an analysis. To emphasize this distinction between these two sets of pointers and to avoid possible confusion between them, they have purposely been given different names. Thus, whenever reference is made to the "father", "son", "previous component", or "next component", it is immediately obvious that a line of the syntax tables is intended. Reference to "superior", "subordinate", or "predecessor" on the other hand immediately makes it known that a line of the analysis record is being referred to.

The connection between these two tables is of course via the "pointer" of the analysis record. It indicates the line of the syntax table to which the line of the analysis record corresponds.

2.3 THE PARSING ALGORITHM

To explain the logic of the parsing algorithm, a metaphor due to Floyd is reproduced here(3). Only the statement of syntactic rules has been changed to conform to the convention adopted in this thesis.

"Suppose a man is assigned the goal of analyzing a sentence in a phrase structure language 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 notion 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:G1., S must appoint a subordinate S1 with the command, 'try to find a G1'. S repeats S1's report to his supperior, firing S1 on a report of failure. If told to try again, S must tell S1 to try again, again transmitting the report to his superior and firing S1 on failure.

3. If G:G1, G2, ..., Gn., S must appoint successively one subordinate Si for each Gi, with the command, 'try to find a Gi.' If Si succeeds, i is increased by one, a new subordinate hired and the process repeated until i is greater than n, when S reports success. If Si fails Si is fired, i is decreased by one and if i is greater than zero,

the new Si (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 Si to try again, and proceeds as before on Si's report.

4. If G:G1;G2; ... Gn., S must appoint successively one subordinate Si for each Gi, with the command, ' try to find a Gi.' If S fails he is fired, i is increased by one, a new subordinate hired, and the process repeated until i is greater than n, when S reports failure. If Si succeeds, S reports success. If after success S is told to try again, he tells Si (who succeeded) to try again, and proceeds as before on Si's report.

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

It is evident that the backtracking feature of