Note on ordering of grammar rules in syntax-analyzers

By J. Cohen and X. Nguyen-Dinh*

The ordering of grammar rules in the syntactical analysis of context-free languages (as utilized in syntax-directed compilers) plays an important role in the parsing efficiency. No theoretical treatment of the optimum ordering is currently available. This paper presents a practical approach to this problem whereby reordering of rules is adjusted to optimize the analysis of input string samples.

During the past few years considerable interest has been given to the implementation of syntax-directed compilers. Although the analysis algorithm itself represents a minor part of the entire compilation process, it presents a great interest from a practical point of view. This interest stems from the fact that the efficiency of the compiler depends considerably on the efficiency of the analyzer. On the other hand the latter depends on the ordering of syntactical rules: syntactical analysis of general context-free languages involves "trial-and-error", and parsing time is obviously reduced if the rules most likely to succeed are the first ones to be tried.

Irons (1961) remarks that the problem of optimum ordering of rules is non-trivial and suggests that it be left "for contemplation on rainy days". As far as we know no theoretical solution to this problem is currently available. Our approach is a practical one and consists in developing a program that keeps track of the number of times a rule is successfully utilized. The reordering is done in a subsequent phase so that the most frequently used rules will be the first ones to be tried in the analysis phase.

In this presentation we assume that the reader is already familiar with some of the algorithms for the analysis of context-free languages. A concise and clear presentation on this subject is given by Floyd (1964), who also gives an excellent bibliography.

The reordering program which we shall describe was specifically developed in connection with the Kuno and Oettinger (1962) analyzer; however, similar schemes could be devised for other types of analyzers.

Syntactic ambiguities are common in natural languages; Kuno and Oettinger were interested in their detection by determining all parses associated with a given input string. Programming languages are designed to be unambiguous, but often only actual practice with a given context-free language could "confirm" the inexistence of ambiguity. Most analyzers utilized in current syntax-directed compilers take for granted the inexistence of ambiguity and search only for the first successful parse. The Kuno–Oettinger algorithm can be easily modified to search only for a first possible parse. In this case ordering of syntactical rules becomes critical. The main reasons for selecting this type of analyzer to investigate the advantages of the reordering technique were:

(1) Optimum ordering is critical when only one parse is required.
(2) This analyzer is closer to the selective top-to-bottom classification of Griffiths and Petrick (1965) and it would parse faster than the normal top-to-bottom algorithms.
(3) The Kuno–Oettinger algorithm is relatively simple mainly because it requires that grammar rules be given in a special form. This makes the proposed reordering scheme particularly easy to implement and to describe.

The syntactical rules utilized by the Kuno–Oettinger analyzer are necessarily of the form

$$X \rightarrow a Y_1 Y_2 \ldots Y_m \qquad (1)$$

or

$$X \rightarrow a. \qquad (2)$$

X, Y_1, Y_2, . . ., Y_m are elements of the auxiliary vocabulary (metavariables) and a is an element of the terminal vocabulary (basic symbols). It is convenient to express the above rules in the following form:

$$(X, a) \rightarrow Y_1 Y_2 \ldots Y_m \qquad (1a)$$

and

$$(X, a) \rightarrow \lambda \qquad (2a)$$

where λ is the null string.

The analysis algorithm utilizes push-down-stores (PDS's or stacks) and can be briefly described as follows:

(*a*) Consider the couple formed by the metavariable X at the top of a PDS and the next basic symbol a in the input string. Three cases may occur:

1. There exists only one rule $(X, a) \rightarrow Y_1 Y_2 \ldots Y_m$ in the grammar under consideration. In this case X is replaced by Y_m; Y_{m-1}, . . ., Y_2, Y_1 are pushed into the store, Y_1 becoming the top element of this PDS. The process described in (*a*) is then repeated. When the (X, a) rule is of the type (2*a*) this operation is equivalent to the elimination of X from the top of the store. In both cases the pointer to the next basic symbol in the input string is advanced one position to the right.

* *Institut de Mathématiques Appliquées, Université de Grenoble, Grenoble, France.*

2. There are several rules (X, a). New PDS's are then created for every rule by recopying the contents of the original one. Step (a) is repeated for each newly created PDS.
3. There is no rule (X, a) and the above-described manipulations with this PDS are abandoned. The algorithm considers the next PDS that might have been formed while in 2.

(b) The iteration described in (a) terminates successfully with one parse when all elements of the input string have been considered and the current PDS is empty. (If all parses are to be found an additional terminating condition is that no other PDS remains to be considered.)

Initialization consists in placing the metavariable ⟨program⟩ in a PDS prior to following step (a). It should be pointed out that the above description is only pedagogical and at most two PDS's are needed in actual implementation of this algorithm. The detailed ALGOL program describing the analyzer is given by Nguyen-Dinh (1965).

Before proceeding to the description of the adopted memory allocation scheme, let us make a few remarks about the syntactical rules utilized in the Kuno–Oettinger analyzer. Greibach (1965) has proved that every context-free grammar is weakly equivalent to another context-free grammar whose rules are of the type (1) or (2). Greibach also indicates how the transformation into these rules could be accomplished. Nguyen-Dinh (1965) gives an ALGOL program that performs this transformation, and prepares the data needed by the analyzer and the subsequent reordering program.

POINTER $[X, a]$ is a two-dimensional array storing the pointers $p^1_{X,a}$. The entries X and a correspond to the elements of a rule written as $(1a)$ or $(2a)$. The $p^1_{X,a}$'s are actually indices to a one-dimensional array *TABLE* where the right-hand-part of the rules are stored. The following information is stored in *TABLE*:

$TABLE\ [p^i_{X,a}]$	contains	$p^{i+1}_{X,a}$
$TABLE\ [p^i_{X,a} + 1]$	contains	m^i
$TABLE\ [p^i_{X,a} + 2]$	contains	$\Sigma^i_{X,a}$
$TABLE\ [p^i_{X,a} + 3]$	contains	Y^i_1
$TABLE\ [p^i_{X,a} + 4]$	contains	Y^i_2
.....	.....	
$TABLE\ [p^i_{X,a} + m^i + 2]$	contains	$Y^i_{m^i}$

The superscript i indicates the ith rule of the type $(1a)$ or $(2a)$ for a given couple (X, a). $\Sigma^i_{X,a}$ represents the sum box indicating the frequency of use for the ith (X, a) rule. When no syntactic rule exists for a couple (X, a), *POINTER* $[X, a]$ contains a special symbol whose presence can be tested by the analyzer. Similarly when $p^{i+1}_{X,a}$ is a special symbol, the information that follows it in *TABLE* represents the last of the (X, a) rules. An m^i equal to zero indicates that the corresponding rule is of the type $(2a)$.

The operation modes of the program may now be described. First in an analysis mode the parsing of input string samples is performed using an arbitrary configuration of the $p^i_{X,a}$'s. After each successful parse is completed a second mode takes care of bookkeeping the $\Sigma^i_{X,a}$'s. Finally, in a third mode, the program examines these sum boxes and reorders the $p^i_{X,a}$'s so that the one corresponding to $\max \Sigma^i_{X,a}$ is given by *POINTER* $[X, a]$ and the others, in decreasing order of their $\Sigma^i_{X,a}$'s, are specified by the chain-links stored in *TABLE*.

The switching between the above-described modes of operation is controlled by the user. He might, for example, decide that after the most typical string samples have been analyzed and rules reordered, only mode one would be operational.

The above reordering scheme has been programmed in ALGOL and tested at the Computing Centre of the Université de Grenoble. Detailed description of programs and discussions of the results are presented by Nguyen-Dinh (1965).

A part of the ALGOL syntax comprising arithmetic and Boolean expressions was selected to test the advantages of reordering. For some sample strings the analysis performed after reordering took 1/2 to 1/20 of the time spent in the previous analyses. These figures are mentioned only to give an idea of possible gains in parsing speed. Actual gains are highly dependent on the grammar and on the structural similarity between sample strings utilized in the reordering and those to be analyzed after it. It was also observed that for certain grammars, several of the computed $\Sigma^i_{X,a}$'s had the same value, thus indicating that for these special cases a second level of optimization would be required to increase further the parsing efficiency.

References

FLOYD, R. W. (1964). "The Syntax of Programming Languages—A Survey", *IEEE Transactions on Electronic Computers*, Vol. EC-13, p. 346.

IRONS, E. T. (1961). *Maintenance Manual for Psyco*, Part One, Princeton University.

KUNO, S., and OETTINGER, A. G. (1962). "Multiple Path Syntactic Analyzer", *Proc. IFIP Congress*, Amsterdam: North Holland, p. 306.

GREIBACH, S. (1965). "A New Normal-Form Theorem for Context-Free Phrase Structure Grammar", *J. Assoc. Comp. Mach.*, Vol. 12, p. 42.

NGUYEN-DINH, X. (1965). "Méthodes d'analyse descendante pour les langages 'Context-Free' ", Thèse de Troisième Cycle, Université de Grenoble.

GRIFFITHS, T. V., and PETRICK, S. R. (1965). "On the Relative Efficiencies of Context-Free Grammar Recognizers", *Comm. Assoc. Comp. Mach.*, Vol. 8, p. 289.