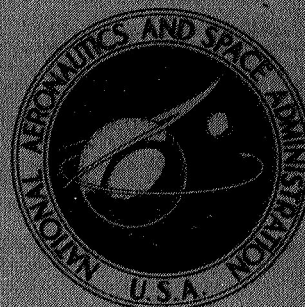


NASA TECHNICAL  
MEMORANDUM



NASA TM X-3490

NASA TM X-3490

CASE FILE  
COPY

A SURVEY OF COMPILER DEVELOPMENT AIDS

*B. P. Buckles, B. C. Hodges, and P. Hsia*

*George C. Marshall Space Flight Center*

*Marshall Space Flight Center, Ala. 35812*

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • FEBRUARY 1977



1. REPORT NO. NASA TM X-3490	2. GOVERNMENT ACCESSION NO.	3. RECIPIENT'S CATALOG NO.	
4. TITLE AND SUBTITLE  A Survey of Compiler Development Aids		5. REPORT DATE February 1977	
		6. PERFORMING ORGANIZATION CODE	
7. AUTHOR(S) B.P. Buckles*, B.C. Hodges, and P. Hsia**		8. PERFORMING ORGANIZATION REPORT # M-200	
9. PERFORMING ORGANIZATION NAME AND ADDRESS  George C. Marshall Space Flight Center Marshall Space Flight Center, Alabama 35812		10. WORK UNIT NO.	
		11. CONTRACT OR GRANT NO.	
		13. TYPE OF REPORT & PERIOD COVERED  Technical Memorandum	
12. SPONSORING AGENCY NAME AND ADDRESS  National Aeronautics and Space Administration Washington, D.C. 20546		14. SPONSORING AGENCY CODE	
15. SUPPLEMENTARY NOTES Prepared by Data Systems Laboratory, Science and Engineering *Science Applications, Inc. **University of Alabama in Huntsville			
16. ABSTRACT  This report establishes a theoretical background for the compilation process by dividing it into five phases and explaining the concepts and algorithms that underpin each. The five selected phases are lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. Several methods for both top-down and bottom-up syntax analysis are illustrated via examples. Graph theoretical optimization techniques are likewise presented, and approaches to code generation are described for both one-pass and multipass compilation environments. Following the initial tutorial sections, more than 20 tools that have been developed to aid in the process of writing compilers are surveyed. Care is taken to categorize each according to the theoretical framework just established. A uniform notation is used throughout this portion rather than resorting to that notation used by each individual system. Eight of the more recent compiler development aids are selected for special attention — SIMCMP/STAGE2, LANG-PAK, COGENT, XPL, AED, CWIC, LIS, and JOCIT. The concluding sections assess the impact of compiler development aids, describe some of their shortcomings, and inspect some of the areas of research currently in progress.			
17. KEY WORDS Compiler writing systems, translator writing systems, compiler-compilers, compiler generators, compilers, metacompilers, generators, context-free grammars, syntax analysis, parsers, semantic analysis, global optimization, code generation, interval analysis		18. DISTRIBUTION STATEMENT  STAR Category 61	
19. SECURITY CLASSIF. (of this report)  Unclassified	20. SECURITY CLASSIF. (of this page)  Unclassified	21. NO. OF PAGES  96	22. PRICE  \$5.00

\* For sale by the National Technical Information Service, Springfield, Virginia 22161

## ACKNOWLEDGMENTS

Assembling the material and organizing the text for this report required the cooperation of a number of people. The authors are grateful to Mr. Terry Dunbar of Computer Sciences Corporation and Mr. David Abt of Chi Corporation for their willingness to discuss the CWS projects in which they were involved. Much credit is due Sandra Austin and Pat Ryan of Science Applications, Inc., who were kind enough to carefully analyze the earlier versions and point out inconsistencies and shortcomings. We, the authors, are responsible for any that remain. Finally, we thank Lynda Suto, also of Science Applications, Inc., who exhibited almost limitless patience in the typing and preparation of numerous earlier drafts.

# TABLE OF CONTENTS

	Page
I. INTRODUCTION . . . . .	1
II. THE STRUCTURE OF A COMPILER WRITING SYSTEM . . . .	1
III. METHODOLOGIES CHARACTERISTIC OF A CWS . . . . .	5
A. Lexical Analysis . . . . .	6
B. Syntax Analysis . . . . .	6
C. Semantic Analysis . . . . .	23
D. Optimization . . . . .	30
E. Code Generation . . . . .	35
IV. EXPECTED PERFORMANCE RANGE OF A CWS . . . . .	41
V. A BRIEF HISTORY . . . . .	42
VI. CURRENT SYSTEMS . . . . .	46
A. SIMCMP/STAGE2 . . . . .	47
B. LANG-PAK . . . . .	50
C. COGENT . . . . .	54
D. XPL/XCOM . . . . .	57
E. CWIC . . . . .	59
F. AED . . . . .	62
G. LIS . . . . .	64
H. JOCIT . . . . .	66
I. Other Systems . . . . .	69
VII. THE USER'S PERSPECTIVE . . . . .	71
VIII. DEPARTURE POINTS FOR FUTURE DEVELOPMENTS . . . .	74
IX. CONCLUSION . . . . .	78
REFERENCES . . . . .	79



# LIST OF ILLUSTRATIONS

Figure	Title	Page
1.	Typical one-pass compiler . . . . .	2
2.	Typical multipass compiler . . . . .	3
3.	Ideal CWS structure . . . . .	4
4.	Realistic CWS structure . . . . .	4
5.	Example precedence matrix . . . . .	11
6.	Top-down parse tree construction . . . . .	14
7.	Bottom-up parse tree construction . . . . .	15
8.	Example parse machine . . . . .	17
9.	Example production language . . . . .	19
10.	Example grammar automaton . . . . .	21
11.	Example automaton parser tables . . . . .	22
12.	Directed graph with three strongly connected regions . . . . .	32
13.	Directed graph partitioned into intervals . . . . .	33
14.	Derived graph from Figure 13 . . . . .	34
15.	Decision table for code selection . . . . .	39
16.	CWS performance range spectrum . . . . .	42
17.	SIMCMP/STAGE2 organization . . . . .	47
18.	LANG-PAK organization . . . . .	53



## LIST OF ILLUSTRATIONS (Concluded)

Figure	Title	Page
19.	COGENT organization . . . . .	56
20.	XPL/XCOM organization . . . . .	58
21.	CWIC organization . . . . .	61
22.	AED organization . . . . .	63
23.	LIS organization . . . . .	65
24.	JOCIT organization . . . . .	68
25.	Comparison of parsing strategies . . . . .	73
26.	Generalized CWS . . . . .	76



# A SURVEY OF COMPILER DEVELOPMENT AIDS

## I. INTRODUCTION

As early as 1960, E. T. Irons was able to construct a compiler in which the syntax recognition phase was independent of the source language being translated [1]. This effort encouraged those who speculated that the entire compilation process could be automated. The immediate result was a period of frenzied activity in the area of programming language syntax analysis, the goal being to develop an algorithm applicable to the broadest possible class of grammars. Meanwhile, formal studies of semantics lagged behind. A large number of aids to the compiler writer emerged [2], employing many different techniques and designed to reduce the implementation effort of one or more phases of a compiler. The structure, performance range, methodologies employed, and other topics concerning these aids are discussed as follows.

## II. THE STRUCTURE OF A COMPILER WRITING SYSTEM

Within the computer science community, the generally accepted definition of a translator is a processor that automatically converts one language (the source language) to another language (the object language). It is also generally accepted that a compiler is a translator for which the source language is procedural and the object language is an assembly or machine language. Thus, a compiler writing system (CWS) is, strictly speaking, a software package that automates the production of compilers. However, the term has come to be used with those systems that automate only part of the task and provide a framework (and perhaps a philosophy) for the remainder [3]. Therefore, this report will treat the terms compiler writing system, translator writing system, compiler-compiler, compiler generator, and metacompiler as synonymous. The term host computer (or host) shall refer to the machine on



which either the CWS or generated compiler executes. Which host, either that of the CWS or generated compiler, will be stated only when it is not apparent from the context. The computer that executes the object language produced by the generated compiler will be called the target computer (or target).

Before launching into the structure of a CWS it is necessary to say something about the structure of compilers. Compilers are normally segmented into passes. A pass may be loosely defined as a complete examination of the source text or some intermediate form of it. A compiler that generates object language concurrently with recognition of phrases in the source language is called a one-pass compiler. A typical one-pass compiler configuration is shown in Figure 1. The semantic routines depicted are correlated to individual phrase types of the source language and have the responsibility of selecting the proper object code for each phrase type. A multipass compiler, for which a typical configuration is shown in Figure 2, usually converts the source language to an intermediate form on the first pass; subsequent passes examine the intermediate form in its entirety for purposes of optimization or code generation. A compilation phase may be even more loosely defined as one pass or some distinguishable subset.

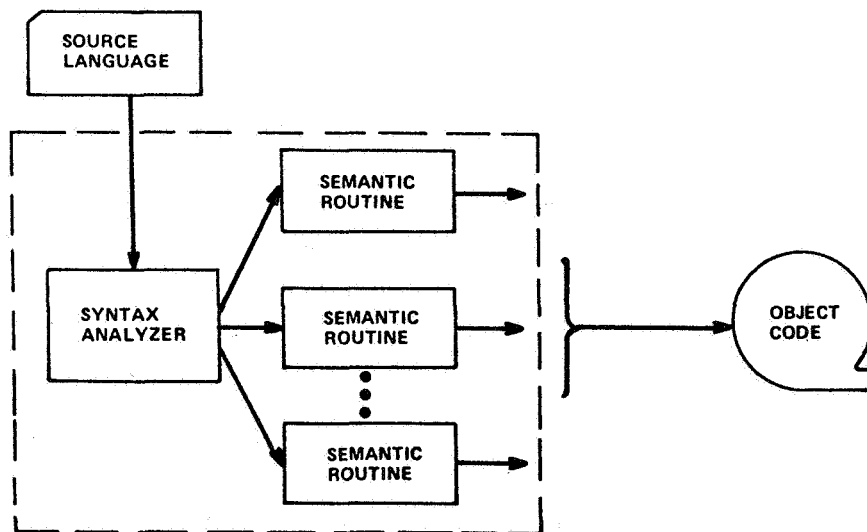


Figure 1. Typical one-pass compiler.

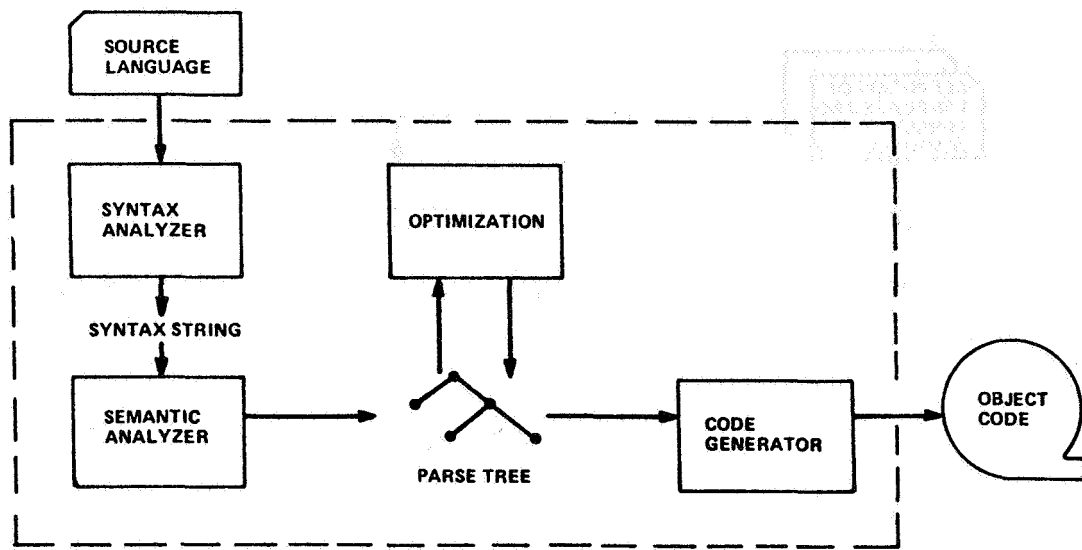


Figure 2. Typical multipass compiler.

To perform its task a CWS must have a description of the source language (both syntax and semantics) and the object language. A description of the characteristics of the target computer is also necessary if the object language is a machine or assembly language. The descriptive form used is referred to as metalanguages (literally, languages for describing other languages). Often, several metalanguages or sublanguages are required, one for each aspect of the translation.

An idealized CWS would exhibit the structure depicted in Figure 3. Under this conceptual design the compiler developer would input the descriptions of the source language grammar, the object language, and the target computer characteristics via a single uniform metalanguage. The CWS would then automatically generate tables from the descriptive inputs which would reconfigure the generalized compiler to accept the described source language programs and output object language for the described target computer. Realistically, however, neither uniformity nor completeness is characteristic of any single compiler writing system now available. Figure 4 illustrates the most common configuration in which separate definition methodologies are utilized to describe separate compilation phases (denoted as CWS 1, CWS 2, and CWS 3). The block labeled "Language/ Computer Dependent Support Procedures" represents functions unique to a specific language-computer combination that are beyond the descriptive capability of the definition methodologies and must be independently



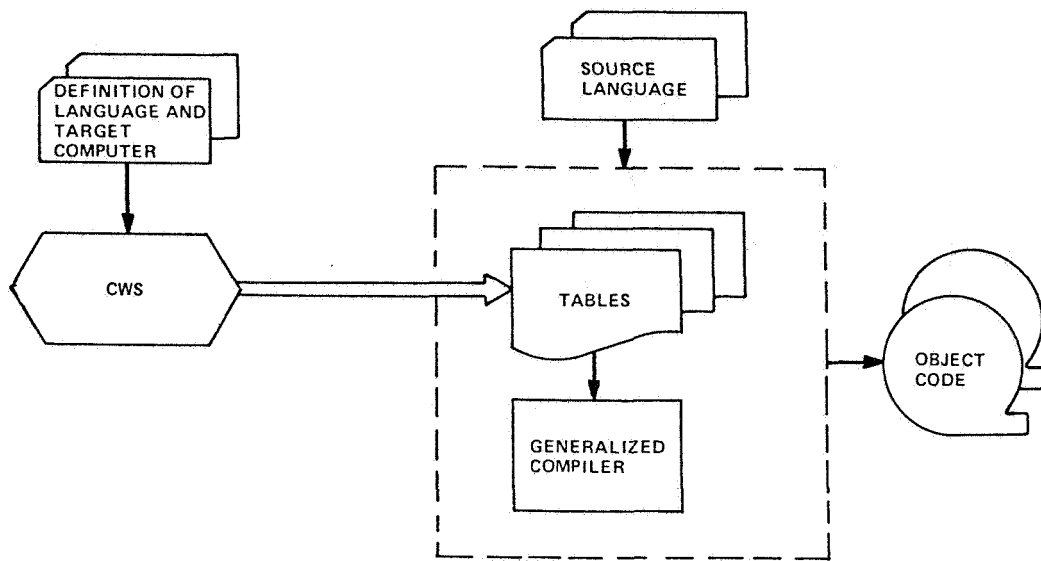


Figure 3. Ideal CWS structure.

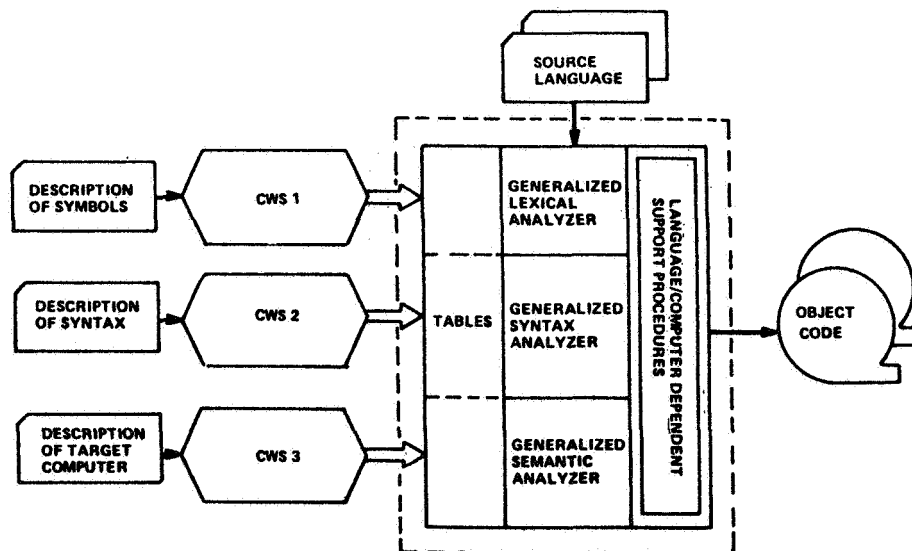


Figure 4. Realistic CWS structure.

produced for each new compiler. In the illustrations that follow, blocks with double-edged borders denote dependent support procedures and dashed borders encircle all components of a final compiler configuration. Components of the CWS that are not also part of the resultant compiler are enclosed in hexagons.

### III. METHODOLOGIES CHARACTERISTIC OF A CWS

A broad array of tactics has been employed by the CWS developer, based partly on the problem class for which the system was intended. Although it is necessary to avoid details in favor of presenting principal trends and alternatives, these efforts have been instrumental in shifting the emphasis in compiler construction from art to science.

The phases of compilation selected for examination with respect to tactics used by compiler writing systems are:

- Lexical Analysis — Scanning the character string comprising the input source language statement and collecting units of information (called tokens) including identifiers, operators, numbers, and reserved words.
- Syntax Analysis — Determining the grammatical structure of the input source statements.
- Semantic Analysis — Attaching meaning to the input source language statements in terms of data attributes (e.g., type and structure) and computer operations.
- Optimization — Seeking memory space and/or execution time improvements for the to-be-generated object language.
- Code Generation — Selecting and emitting code based on source language semantics, object language description, and target computer characteristics.

The latter three phases shall be referred to as postsyntactic analysis.

A specific CWS will often emphasize support in one or more areas while neglecting others. In most cases little help is provided in the lexical analysis phase, and resultant compilers are often inflexible in the form of input they will accept. It is in the area of syntax analysis that the CWS has been the most useful to the compiler writer. A general theory of semantics does not exist; thus, a semantic analyzer for the CWS system may not be provided or its use



may be optional. In this area there are two extremes. In the first extreme, minimum effort needs to be applied to semantic definition by the CWS user, but the resultant compiler will produce inefficient object code. In the second extreme, extensive effort needs to be applied to semantic definition up to and including coding parts by hand, but the resultant compiler will produce object code equal to or exceeding those that are "hand-tuned."

## A. Lexical Analysis

A common approach to lexical analysis is to attach attribute descriptors to single character symbols which may be subsequently interrogated by the syntax analyzer. Multicharacter symbols such as reserved words, identifiers, or numbers are often built into the syntax definitions where they cannot be handled efficiently. Methods have been developed, however, to describe symbols of the above type and direct their synthesis into a single unit prior to passing them to the syntax analysis phase [4, 5]. Another method utilizes a fixed basic set of multicharacter symbols such as integers and identifiers to which the language designer must adhere [3]. This is not as inflexible as it might first appear. For example, if the basic set includes integers, it is possible to define a real value as being two integers separated by a decimal point. As a final resort, the CWS may provide a framework in which the user inserts a customized lexical analyzer.<sup>1</sup>

## B. Syntax Analysis

Due to the incredible productivity in the area of formal grammar theory, the task of classifying compiler writing systems on the basis of syntax analysis methodologies is relatively straightforward. Assuming some background is required, the following discussion is divided into: (1) definition of the class of grammars employed in computer languages, (2) important subclasses of this class of grammars, and (3) the most prevalent syntax analysis (parsing) methodologies. Where possible, grammar subclass is related to parsing methodology. Neither the subclass categories nor parsing methodologies are exhaustive; rather, an attempt has been made to choose those having the greatest impact on CWS technology.

---

1. Language Implementation System. Chi Corporation, Cleveland, Ohio, Undated Report.

## 1. CONTEXT-FREE GRAMMARS

The grammatical class of programming languages is known as the context-free grammars (CFG). A CFG may be defined as a quadruple  $G = (N, T, P, S)$ , where

$N$  = A set of symbols called nonterminals, each of which stand for a string of symbols.

$T$  = A set of symbols called terminals, each of which may stand for no other symbol except itself.

$P$  = A set of formulas called productions which define each nonterminal as a string of symbols; there may be more than one definition for any nonterminal.

$S$  = A distinguished member of  $N$  called the goal symbol.

Subsequently, nonterminals will be enclosed within the special brackets, "<" and ">." Productions will be of the form

$$\langle a \rangle ::= B \quad ,$$

where  $\langle a \rangle$  is a nonterminal,  $::=$  is a symbol which means "is replaced by," and  $B$  is a string which may be empty or consist of one or more terminals and nonterminals in a definite order. There is one further CFG restriction, i. e., the goal symbol,  $S$ , may appear to the right of  $::=$  in no production except those in which it appears on the left. This notation conforms to the well known Backus-Naur Form (BNF) metalanguage [6]. BNF is by far the most prevalent method used to describe context-free grammars, but other methods do exist. Two of these, the macro [7] and analytic [8] approaches, will be discussed later, but attention will be devoted primarily to BNF here.

Because the concepts of grammatical definition appear esoteric when first encountered, they can be best explained with an example. Consider the following grammar with productions:



$\langle \text{sentence} \rangle :: = \langle \text{subject} \rangle \langle \text{predicate} \rangle$   
 $\langle \text{subject} \rangle :: = \langle \text{article} \rangle \langle \text{noun} \rangle$   
 $\langle \text{predicate} \rangle :: = \langle \text{verb} \rangle \langle \text{direct object} \rangle$   
 $\langle \text{direct object} \rangle :: = \langle \text{pronoun} \rangle$   
 $\langle \text{article} \rangle :: = \text{a}$   
 $\langle \text{article} \rangle :: = \text{the}$   
 $\langle \text{noun} \rangle :: = \text{boy}$   
 $\langle \text{noun} \rangle :: = \text{dog}$   
 $\langle \text{verb} \rangle :: = \text{chased}$   
 $\langle \text{verb} \rangle :: = \text{had}$   
 $\langle \text{pronoun} \rangle :: = \text{him}$   
 $\langle \text{pronoun} \rangle :: = \text{it} \quad .$

These generate sentences such as "a dog chased him" and "the boy had it."  
 In terms of these productions, the other members of the grammar are:

$N = \{ \langle \text{sentence} \rangle, \langle \text{subject} \rangle, \langle \text{predicate} \rangle, \langle \text{direct object} \rangle, \langle \text{article} \rangle, \langle \text{noun} \rangle, \langle \text{verb} \rangle, \langle \text{pronoun} \rangle \}$

$T = \{ \text{a, the, dog, boy, chased, had, him, it} \}$

$S = \langle \text{sentence} \rangle \quad .$

## 2. CFG SUBCLASSES

A context-free language is simply a language which can be generated by a context-free grammar. Generally, there exist more than one CFG for each language, which lends importance to the concept of CFG subclasses. The subclasses discussed in the following are not mutually exclusively, and none are capable of generating all the context-free languages. An attempt has been made to present each as informally as the subject matter permits. Sufficient references are included for investigation by those more theoretically inclined.

The first subclass is the LL(k) grammars described by Lewis and Stearns [9]. LL(k) stands for left-to-right scan, leftmost reduction with k symbol look-ahead. Informally, the LL(k) grammars may be described as those for which, given a sentential form (i.e., sentence which may have some phrases replaced by nonterminals), it is possible to predict the next production to be applied by scanning the sentential form from left to right to a length of, at most, k symbols into the phrase to be replaced. An interesting subset of the LL(k) grammars is those that may be represented in Greibach Normal Form [10], a form in which the right part of every production begins with a terminal symbol. A still more restrictive subset, the LL(1) grammars, requires that each production right part begin with a unique terminal. This subset is characterized by Korenjak and Hopcroft [11].

An excellent analysis of the LL(k) grammars is given by Rosenkrantz and Stearns [12]. Among other things, they give three necessary and sufficient conditions for testing a grammar to be of subclass LL(k) for a given value of k. They also prove that, in general, whether or not a grammar is LL(k) is undecidable unless k is given a priori and that all LL(k) grammars are unambiguous. (A grammar is unambiguous if, for every sentence of the language it describes, there exists only one syntactical interpretation.)

The next CFG subclass is the simple precedence grammars [13]. Although the authors know of no CWS that directly implements them, the concept is central to at least one such system [3], and several parsing methods implicitly use the idea that different symbol pairs bind to each other with different strengths [14,15].

A CFG is simple precedence if it contains no productions having empty or equivalent right parts and for each pair of symbols, A and B, at most one of the following relations hold:

$A \dot{=} B$  A and B are adjacent in some production.

$A <\cdot B$  B is the start symbol of some production and there exists at least one sentential form of which AB is a substring.

$A \cdot > B$  A is the tail symbol of some production and there exists at least one sentential form of which AB is a substring.

A more rigorous definition is given by Aho and Ullman [16] among others [2,17-19].



The significance of the precedence grammars is that they exhibit simple criteria for deciding when and how much of a sentential form to reduce when scanning from left to right, namely, the portion between the last  $<\cdot$  and the first  $\cdot>$ . (Henceforth, this substring of the sentential form will be called the "handle.") For example, the following steps would reduce "the dog chased him" using the precedence relations in Figure 5:

<u>Already Scanned</u>	<u>Relation</u>	<u>Next Symbol</u>
	$<\cdot$	the
$<\cdot$ the	$\cdot>$	dog
$<\cdot$ <article>	$<\cdot$	dog
$<\cdot$ <article> $<\cdot$ dog	$\cdot>$	chased
$<\cdot$ <article> $\dot{=}$ <noun>	$\cdot>$	chased
$<\cdot$ <subject>	$<\cdot$	chased
$<\cdot$ <subject> $<\cdot$ chased	$\cdot>$	him
$<\cdot$ <subject> $<\cdot$ <verb>	$<\cdot$	him
$<\cdot$ <subject> $<\cdot$ <verb> $<\cdot$ him	$\cdot>$	
$<\cdot$ <subject> $<\cdot$ <verb> $\dot{=}$ <pronoun>	$\cdot>$	
$<\cdot$ <subject> $\dot{=}$ <predicate>	$\cdot>$	
$<\cdot$ <sentence> $\cdot>$		

A mathematical technique for deriving precedence matrices can be found in Reference 18.

The concepts embodied in the simple precedence grammars can be easily extended to grammars which have had a more immediate impact on CWS technology. The paragraphs that follow introduce the (m,n) precedence, weak precedence, operator precedence, and bounded-context grammars using the ideas developed for the simple precedence subclass.

A grammar is said to be (m,n) precedence if the three relations  $<\cdot$ ,  $\dot{=}$ , and  $\cdot>$  are defined unambiguously when comparing the last m symbols scanned to the following n symbols. Thus, the simple precedence grammars are also

	< sentence >	< subject >	< predicate >	< direct object >	< article >	< noun >	< verb >	< pronoun >	a	the	boy	dog	chased	had	him	it
< sentence >																
< subject >			≡				<.						<.	<.		
< predicate >																
< direct object >																
< article >						≡					<.	<.				
< noun >			•>				•>						•>	•>		
< verb >				≡			<.								<.	<.
< pronoun >																
a						•>					•>	•>				
the						•>					•>	•>				
boy			•>				•>						•>	•>		
dog			•>				•>						•>	•>		
chased				•>			•>								•>	•>
had				•>			•>								•>	•>
him																
it																

Figure 5. Example precedence matrix.

(1,1) precedence, because, for them, the last symbol is compared to the next when scanning left to right. Precedence grammars may have no productions with empty or equivalent right parts.

The weak precedence grammars [20,21] require only the  $\cdot >$  relation to be uniquely defined for each pair of symbols. Ambiguity is permitted in the  $< \cdot$ ,  $\dot{=}$  relations. Thus, in scanning a sentential form from left to right, it is possible to determine the right boundary of the handle but not the left. The latter decision is made by comparing the tail portion of the scanned part to entries in the production list. The longest production that matches the tail of the scanned portion is accepted. Naturally,  $(m,n)$  weak precedence is analogous to  $(m,n)$  precedence.

The operator precedence grammars [22] require that the  $< \cdot$ ,  $\dot{=}$ , and  $\cdot >$  relations be defined uniquely only between terminal symbols and that no production has two adjacent nonterminal symbols. During scanning, nonterminal symbols in the sentential form are, in essence, ignored. This is an increase in efficiency, but at the cost of the generation of an incomplete parse tree. Frequently, this is acceptable because not all parse steps are semantically important to the language.

All precedence grammars require that productions have unique right parts. This restriction is relaxed for the bounded-context grammars [23]. A grammar is said to be bounded-context of degree  $(m,n)$ , if, upon determining a candidate for the handle, decisions can be made concerning whether or not it is the handle and which production to apply by looking no more than  $m$  symbols to the left and  $n$  symbols to the right of the candidate. It may appear contradictory to state that the bounded-context grammars are a subclass of the context-free grammars. This is because the word "context" is not being used uniformly in the two terms. A context-free grammar is one for which it is possible to state the definition of each nonterminal irrespective of its left or right context; that is, the left part of each production is a single nonterminal. A bounded-context grammar is one in which, given a substring, it is possible to determine which, if any, nonterminal it defines (in a sense, running the production backward) by examining a fixed number of symbols to the left and right.

The final subclass, the  $LR(k)$  grammars [24], include as subsets all the previous subclasses. Although not capable of generating all context-free languages, the  $LR(k)$  grammars are able to generate practically all the programming languages in use today [25].

$LR(k)$  stands for left-to-right scan, rightmost reduction with  $k$  symbol look-ahead. [Compare this definition to that of the  $LL(k)$  grammars.] A rightmost reduction is one that, at each step, replaces the rightmost nonterminal with its derivation. The following example depicts both a leftmost and rightmost parse for "the dog chased him:"

### LL Derivation

<sentence>  
<subject> <predicate>  
<article> <noun> <predicate>  
the <noun> <predicate>  
the dog <predicate>  
the dog <verb> <pronoun>  
the dog chased <pronoun>  
the dog chased him

### LR Derivation

<sentence>  
<subject> <predicate>  
<subject> <verb> <pronoun>  
<subject> <verb> him  
<subject> chased him  
<article> <noun> chased him  
<article> dog chased him  
the dog chased him

Every LL(k) grammar is also an LR(k) grammar [12]. For an extensive comparison of the two, consult Reference 9. A parser that performs the derivation steps in the order enumerated previously is called a top-down parser. One that performs the steps in the reverse order is bottom-up.

In practical terms, an LR(k) grammar is one for which the decisions associated with left-to-right parsing (i.e., reducing or scanning) can always be made correctly by considering everything scanned thus far plus the k leading symbols of the string to the right of the scan. Thus, it is evident that the LR(k) grammars include all of the precedence grammars. For example, an (m,n) weak precedence grammar is also an LR(n) grammar. By inspection, it can be shown that the sample grammar presented earlier is LR(0).

In concluding the discussion of CFG subclasses, it should be noted that more detailed descriptions of all of those mentioned (plus others) can be found in the standard texts [16,18,19,26,27]. In addition, the paper by Ullman [17] describes the LL(k), LR(k), and simple precedence grammars.

### 3. PARSING ALGORITHMS

The objective of formulating a rigid grammatical structure and dividing that structure into subclasses is the construction of algorithms, called recognizers or parsers, capable of accepting a particular subclass. These algorithms fall into two general categories, top-down and bottom-up. Although a particular parser may have aspects of both, those generated by a CWS generally are in only one of the two categories.



Distinction between the two can best be explained by the manner in which each constructs a parse tree. Figure 6 illustrates intermediate steps in top-down parse tree construction. The parser sets a goal (initially the distinguished symbol) then expands that goal via one of the alternative productions which define it. Nonterminals in each such expansion constitute subgoals. Eventually, a subgoal is set that requires comparison to the input statement. A bottom-up parser (Fig. 7) performs tree construction in the reverse sequence. It first reads the input string, then finds a production that matches both the symbol read and the tree already partially constructed. Note that the top-down parse produces an LL-derivation and the bottom-up parse produces an LR-derivation, but the final trees are equivalent. A thorough description of top-down parsing (under the name *syntax-directed analysis*) and bottom-up parsing (under the name *syntax-controlled analysis*) is given in Reference 28.

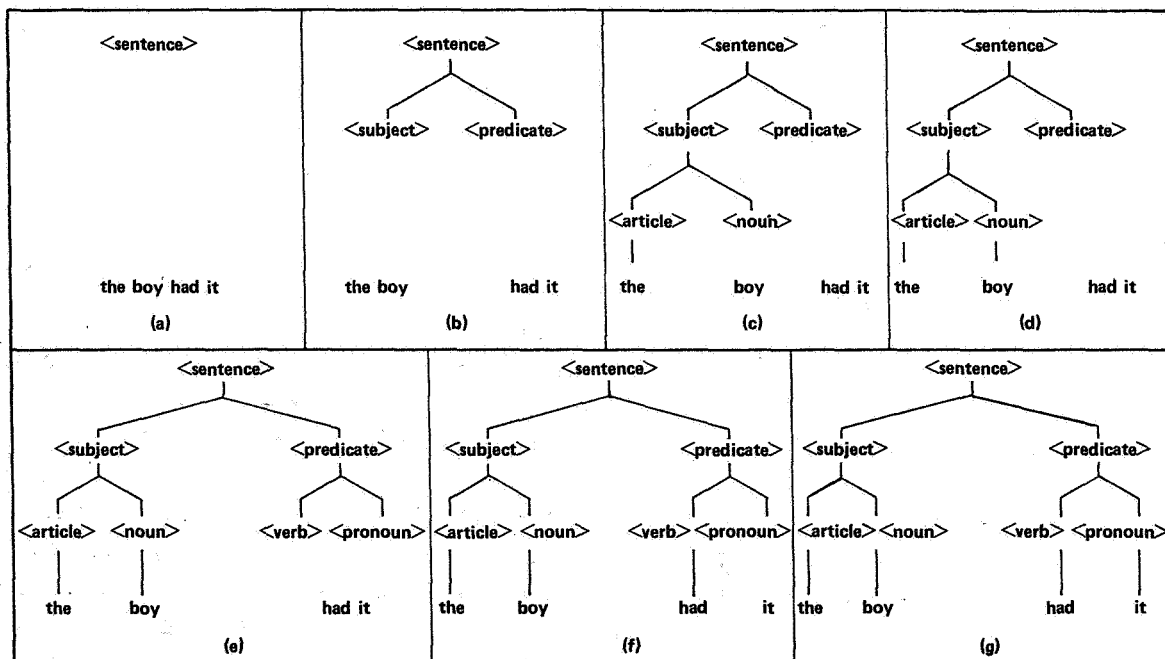


Figure 6. Top-down parse tree construction.

Occasionally, a top-down parser will select an incorrect production to define a previous subgoal. When the mistake is discovered, it must undo the parse and try a different alternative. This action is known as backtracking and can lead to a degradation in performance. However, if the parser is

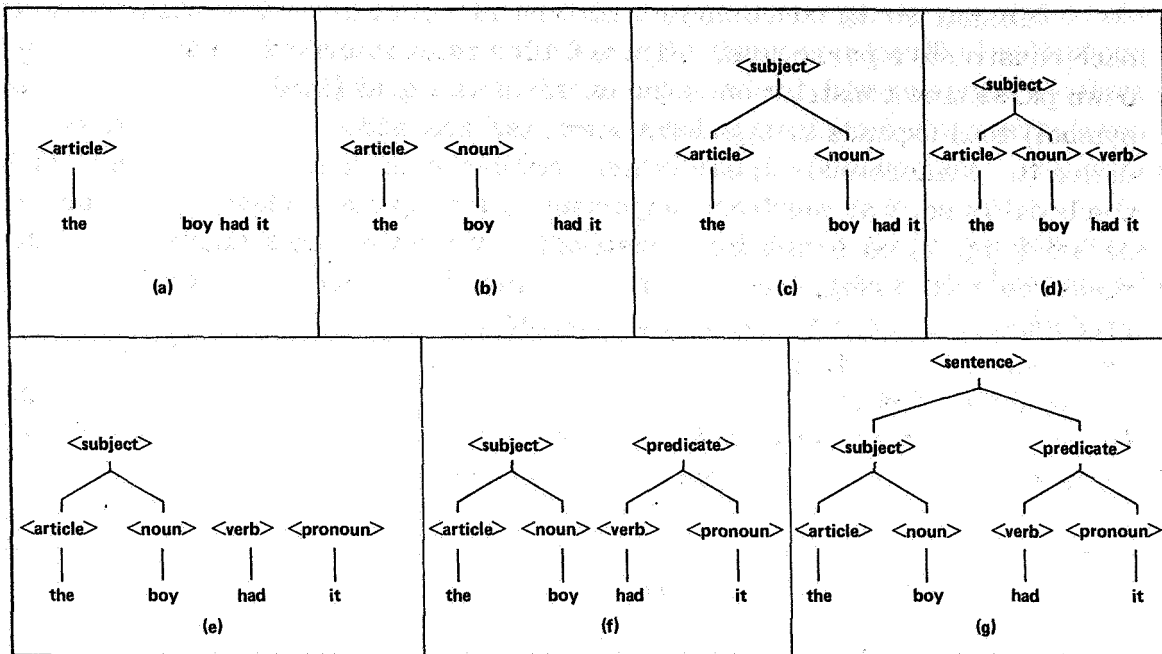


Figure 7. Bottom-up parse tree construction.

permitted to look ahead a few symbols before setting a subgoal and the next few symbols will 'predict' the proper production [recall LL(k) grammar definition], this inefficiency can be eliminated. This is not to say, however, that a top-down recognizer is strictly limited to the LL(k) grammars. Even if the grammar is not LL(k), look-ahead may still be used to eliminate productions that could not begin with the next symbols [29]. Given backtracking, a top-down algorithm will cover a very large set of grammars, generally larger than any specific bottom-up technique. Exactly which subclass applies to which top-down algorithm is still an unsolved problem. Reference 30 may be consulted for an extensive analysis. There is, nonetheless, an exceptional circumstance affecting top-down analysis which will be discussed next.

Left recursion is the bane of the top-down parser. A production is directly left recursive if it is of the form

$$\langle a \rangle ::= \langle a \rangle B \quad ,$$

where B is any string containing terminals and/or nonterminals. That is, a production is directly recursive if it defines a nonterminal as being a string of symbols beginning with the same nonterminal. In the above example, a top-down parser would continue to set <a> as a subgoal at the next tree level ad infinitum. Fortunately, simple transformations exist to convert any grammar with left recursive productions to an equivalent one without them [31]. However, the most frequent device relied upon is a metalanguage extension, as will be presently demonstrated. Incidentally, an LL(k) grammar is never left recursive [32].

There are three basic approaches to the top-down algorithm. The first is to translate the grammar to a hierarchical data structure in which there is a one-to-one correspondence between nodes in the structure and nonterminals in the language [33]. Each node in the structure enumerates the possible alternative definitions for its nonterminal, using pointers to other nodes to designate other nonterminals within the definition. A generalized algorithm is then provided which sets analysis goals, beginning with the root node, and attempts to find the subnetwork within the data structure that is the parse tree for the input sentence. When all alternatives are exhausted without success, the algorithm reports an error.

The second approach is recursive descent [34], which has been appraised as the most widely employed method of syntax analysis [35]. The basic idea behind recursive descent is that there is a one-to-one correspondence between nonterminals in the grammar and procedures in the parser. Each procedure is responsible for identifying, within the input sentence, the strings that comprise its nonterminal. It reports the success or failure of its endeavor to the higher level (i.e., evoking) procedures. A CWS that generates a recursive descent parser translates the metalanguage describing the grammar into executable procedural statements.

The third approach, having much in common with recursive descent, is the parse machine [36]. The basic idea is to design a set of recursively called interpretive procedures, each identified by a nonterminal and responsible for determining if that nonterminal can be applied to the sentence. Figure 8 depicts the parse machine for the previous sample grammar. In Figure 8, there are two operations: CALL evokes another interpretive procedure and has the effect of setting a subgoal; SCAN tries to match the next input text symbol with a specific terminal. Either operation may succeed or fail and may return a success or failure flag to an evoking procedure. A second option,

LOCATION	OPERATION [OPERAND]	SUCCESS	FAILURE
<sentence>	CALL [<subject>] CALL [<predicate>]	CONTINUE EXIT	ERROR ERROR
<subject>	CALL [<article>] CALL [<noun>]	CONTINUE RETURN [ yes ]	RETURN [ no ] RETURN [ no ]
<predicate>	CALL [<verb>] CALL [<direct object>]	CONTINUE RETURN [ yes ]	RETURN [ no ] RETURN [ no ]
<direct object>	CALL [<pronoun>]	RETURN [ yes ]	RETURN [ no ]
<article>	SCAN [ a ] SCAN [ the ]	RETURN [ yes ] RETURN [ yes ]	CONTINUE RETURN [ no ]
<noun>	SCAN [ boy ] SCAN [ dog ]	RETURN [ yes ] RETURN [ yes ]	CONTINUE RETURN [ no ]
<verb>	SCAN [ chased ] SCAN [ had ]	RETURN [ yes ] RETURN [ yes ]	CONTINUE RETURN [ no ]
<pronoun>	SCAN [ him ] SCAN [ it ]	RETURN [ yes ] RETURN [ yes ]	CONTINUE RETURN [ no ]

Figure 8. Example parse machine.

other than returning, is continuing interpretation at the next line of interpretive code. A more realistic example of a parse machine for recognizing a language with a grammar expressed in extended BNF would involve from 10 to 15 operation types.

A more varied group of bottom-up algorithms exist. Those that have had the greatest impact on CWS technology are the precedence, production language, and LR(k) automata techniques. Thus, they are the ones that have been chosen for expansion here. Each one is predicated on a single left to right scan of the input text, and each is designed to resolve two questions at each parse step: (1) What substring of the sentential form should be reduced next? and (2) Which production should be used in the reduction? The former question may be rephrased as: What is the handle?

An understanding of the precedence relations is tantamount to understanding the precedence parser. The input sentence is scanned from left to right, and each symbol is placed on a pushdown stack. When the relation between the top symbol on the stack and the next symbol in the input string is  $\cdot >$ , a reduction is made by comparing the top symbols on the stack to a production list.



One further refinement is possible over the technique implied during the explanation of precedence grammars. Storing the precedence matrix can consume considerable memory space, on the order of  $2 \times m \times n$  bits where there are  $m$  terminal and nonterminal symbols and  $n$  terminals. It is unnecessary to store the columns corresponding to nonterminals since the symbols to the right of the scan, being unreduced, will contain only terminals. Memory space requirements may be lessened if functions  $f(x)$  and  $g(x)$  can be found such that for symbols  $a$  and  $b$ :

$$\begin{aligned} f(a) < g(b) & \text{ if } a < \cdot b \\ f(a) = g(b) & \text{ if } a \dot{=} b \\ f(a) > g(b) & \text{ if } a \cdot > b \end{aligned}$$

Such functions are called linear precedence functions [22, 37-39] and, unfortunately, do not exist for all precedence grammars.

Production language parsers [14, 40] will be discussed here in terms of bottom-up techniques. The technique itself is quite flexible, however, and can be applied to top-down algorithms. It employs a table containing fixed form statements (the production language) and a generalized algorithm that interprets the table while scanning the input sentence.

The method is most easily explained in terms of an example such as that for the sample grammar depicted in Figure 9. Interpretation begins at line 1 of the production language table. As the statement is scanned, the symbols are placed on a pushdown stack and the top of the stack is compared to the first field of the current production language statement. If the compare succeeds, several actions may take place: reduction, continue of scan, and selection of next production language statement. The ability to evoke an applicable semantic routine is not shown. If the compare fails, an error may be reported or a new production language statement tried.

In bottom-up analysis, production languages are particularly well suited, but not limited, to bounded-context grammars. This strength, which is not particularly well illustrated by the  $LR(0)$  sample grammar, is due to the inherent ease in scanning past (that is, looking at) several terminals before

LINE NO.	IF TOP OF STACK IS:	SUCCESS			FAILURE	
		REPLACE WITH:	SCAN?	GO TO LINE:	ERROR?	GO TO LINE:
1	(empty)	---	yes	2	---	---
2	a	< article >	yes	4	no	3
3	the	< article >	yes	4	yes	---
4	boy	< noun >	no	6	no	5
5	dog	< noun >	no	6	yes	---
6	< article > < noun >	< subject >	yes	7	yes	---
7	chased	< verb >	yes	9	no	8
8	had	< verb >	yes	9	yes	---
9	him	< pronoun >	yes	11	no	10
10	it	< pronoun >	yes	11	yes	---
11	< verb > < pronoun >	< predicate >	no	12	yes	---
12	< subject > < predicate >	< sentence >	no	exit	yes	---

Figure 9. Example production language.

reducing a substring deeper in the stack. However, the interpretive mode of operation causes the method to be somewhat less efficient than precedence techniques.

The last selected bottom-up method is that of LR(k) automata, which have only recently become practical. The technique was first suggested by Knuth [24], and the underlying theory was eloquently expounded in a book by Hopcroft and Ullman [41]. However, the first practical algorithms were developed independently by DeRemer [15,42] and Earley [43]. Other similar methods exist [11,44,45], but the ones having the greatest impact on CWS technology have been those by Aho [46] and DeRemer. It is the latter that has been chosen for development here.

The underlying concept of the LR(k) automata methods is that as a parser scans a sentence, it moves through a series of states for which the next state is always uniquely determined by the current state and next symbol. This concept is informally implemented in many ad hoc parsers. The formal implementation is to convert the metalanguage description of the source language into an automaton represented as a series of tables which, in turn, are interpreted by a generalized algorithm. For the sake of brevity, the automaton construction techniques are not given here but may be found in the cited references. It is sufficient to say that the techniques are amenable to automation [47-49].

The DeRemer approach divides the automation into states of four basic types:

READ — Scans the next symbol and transitions to a new state depending on the symbol value.

LA — Divides the set of possible next symbols into two subsets. If the next symbol in a sentential form is a member of the first subset, LA acts just like a READ state; if the next symbol is a member of the second subset, transition is made based on symbol value without scanning past the symbol (i.e., LA is a look-ahead state).

POP — Reduces the rightmost symbols in the scanned portion of the sentential form.

EXIT — Signifies completion of parse tree construction.

These four state types are sufficient if the scan is restarted at the beginning of the sentential form after each reduction. However, it is desirable to always restart the parse at the current scan point, and this can be done if the automaton maintains a record of the states it has thus far occupied. This is done by allowing READ and LA states to place their names (i.e., state numbers) onto a pushdown stack at the time they are entered. POP states then remove the  $n$  top names from the stack if there are  $n$  symbols in the rightmost reduction. The top name of those remaining on the stack can then be used to determine at which state to continue the parse. This requires additional states to be added to the automaton. These new states are of a fifth type:

LB — Transitions to a new state based on the value at the top of the name stack (i.e., LB is a look-back state).

The LR(0) sample grammar used heretofore is inadequate for illustrating the method. Thus, the following LR(1) grammar for arithmetic expressions is provided in its stead:

#1.  $\langle s \rangle : : = | \langle \text{exp} \rangle |$

#2.  $\langle \text{exp} \rangle : : = \langle \text{exp} \rangle + \langle \text{fact} \rangle$

#3.  $\langle \text{exp} \rangle : : = \langle \text{fact} \rangle$

#4.  $\langle \text{fact} \rangle ::= \langle \text{fact} \rangle * \langle \text{pri} \rangle$

#5.  $\langle \text{fact} \rangle ::= \langle \text{pri} \rangle$

#6.  $\langle \text{pri} \rangle ::= (\langle \text{exp} \rangle)$

#7.  $\langle \text{pri} \rangle ::= i$

The productions are numbered for reference, and the pad symbols,  $\mid$   $\mid$ , are added to provide unique start and end symbols. The automaton for this grammar is shown in Figure 10. Braces,  $\{ \}$ , are used to indicate the look-ahead subset; production numbers and the name stack remove count are associated with each POP state. A LA or READ state recognizes an error when the next symbol of the sentential form does not correspond to one of its transition symbols.

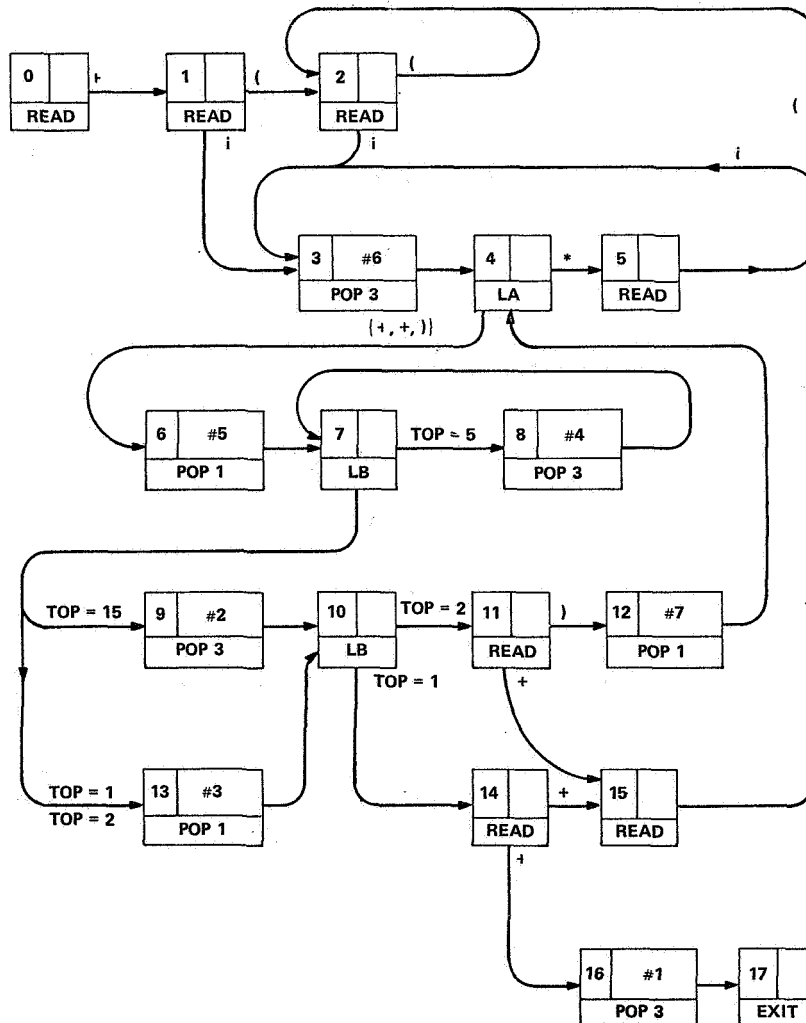


Figure 10. Example grammar automaton.

The tabular representation of this automation is illustrated in Figure 11. An actual implementation would require the information be packed much more densely with each column serving multiple purposes. The state table (ST) is a linear list of all states with attached characteristics. The ST references both the transition table (TT) and look-ahead table (LAT) to designate lists of legal transition symbols. The ST references the TT by means of a beginning line number and a count of the applicable consecutive lines. The TT associates with each transition symbol the state to which to go if that symbol is encountered. The LAT is a boolean matrix having one row for each look-ahead subset. Each column corresponds to a terminal symbol, and each element is true if the corresponding column symbol is a member of the look-ahead subset.

STATE TABLE (ST)							TRANSITION TABLE (TT)				
STATE NAME	STATE TYPE	TT LINE	TT COUNT	POP COUNT	LAT LINE	TT LINE FOR LA	REDUCTION	LINE NO.	SCAN SYM	TOP SYM	GOTO ST
0	READ	0	1					0	t		1
1	READ	1	2					1	(		2
2	READ	1	2					2	i		3
3	POP	3		3			<pri> :: = (<exp>)	3			4
4	LA	4	1		1	6		4	*		5
5	READ	1	2					5			6
6	POP	6		1			<fact> :: = <pri>	6			7
7	LB	7	4					7		5	8
8	POP	6		3			<fact> :: = <fact>*<pri>	8		15	9
9	POP	11		3			<exp> :: = <exp> + <fact>	9		1	13
10	LB	12	2					10		2	13
11	READ	14	2					11			10
12	POP	3		1			<pri> :: = i	12		2	11
13	POP	11		1			<exp> :: = <fact>	13		1	14
14	READ	15	2					14	)		12
15	READ	1	2					15	+		15
16	POP	17		3			<t> :: = t<exp> +	16	+		16
17	EXIT							17			17

LOOK AHEAD TABLE (LAT)

	t	+	+	*	i	(	)
LINE 1	f	t	t	f	f	f	t

Figure 11. Example automaton parser tables.

The method just described is applicable to most LR(1) languages. The exact set is called Simple LR(1) or SLR(1), which may be described informally as those that may be parsed left to right with one symbol look-ahead and "some" left context. They include as a subset the weak precedence grammars. Practical methods exist for extension to the SLR(k) grammars where k is small but greater than one [42].



In summary the LR(k) automata methods are quite general, and most languages are indeed LR(1) or almost LR(1). For those that are not, techniques exist for transforming LR(k) grammars to LR(1) [50]. Such transformations may lead to grammars so large as to be intractable, however. Studies exist [42, 43, 51, 52] indicating execution and storage advantages for automata parsers, although these frequently make worst case assumptions when estimating storage requirements for the precedence relations. Further increases in execution efficiency have been obtained by eliminating semantically irrelevant reductions [25, 53].

The trend in CWS technology has been from top-down to bottom-up parsers as more general bottom-up techniques have been developed and refined. However, not all earlier systems were top-down (for example, see Reference 54), nor are all current systems bottom-up (for example, see Reference 5). Undoubtedly, this stems from the general belief that bottom-up recognizers are more efficient, as was first substantiated by Griffiths and Petrick [55]. In any event, such trust in efficiency may not be entirely justified since Griffiths and Petrick also indicated that an equivalent grammar existed for most languages that could be parsed top-down nearly as efficiently. Furthermore, in a large scale compiler, the time required to construct the parse tree is relatively small compared to overall compilation time.

It may appear that an inordinate amount of space and time has been expended here on a compilation phase estimated at 5 to 10 percent of the code for a compiler [56]. The space allotted reflects the efforts of researchers. As noted earlier, the application of formal mathematical methods to the symbol manipulation aspects of syntax analysis has been instrumental in transforming software technology from an art to a science.

## C. Semantic Analysis

Once the syntactic structure of a source input statement is determined, an operational interpretation is necessary. This process is frequently called semantic analysis. One might suspect, correctly, that if several grammatical definitions were possible for the same language, one form might facilitate semantic analysis to a greater extent than the rest. That is, the syntax and semantics of a language are not entirely independent.

An example of interdependence is the concurrent assignment statement [57]. A statement of the form

$$A, B := e_1, e_2$$

means A is to be assigned the value of expression e1 and B is to be assigned the value of expression e2. In general, a variable in the list to the left of the assignment operator is assigned the value of the expression in the corresponding list position on the right. One way to describe the syntax of the concurrent assignment statement is

$$\langle \text{assignment} \rangle ::= \langle \text{variable} \rangle := \langle \text{expression} \rangle$$

$$\langle \text{assignment} \rangle ::= \langle \text{variable} \rangle, \langle \text{assignment} \rangle, \langle \text{expression} \rangle$$

This is adequate for exposing syntax anomalies, but if the parse tree is drawn for this example, one finds B and e1 combined in one branch of the tree while A and e2 are combined at a higher branch. This complicates semantic interpretation.

A second way to represent the syntax might be

$$\langle \text{assignment} \rangle ::= \langle \text{variable list} \rangle := \langle \text{expression list} \rangle$$

$$\langle \text{variable list} \rangle ::= \langle \text{variable} \rangle$$

$$\langle \text{variable list} \rangle ::= \langle \text{variable list} \rangle, \langle \text{variable} \rangle$$

$$\langle \text{expression list} \rangle ::= \langle \text{expression} \rangle$$

$$\langle \text{expression list} \rangle ::= \langle \text{expression list} \rangle, \langle \text{expression} \rangle$$

Constructing the parse tree of the example using the second grammar would show that all variables are in one subtree and all expressions are in a second. This is more tractable semantically, however the syntax is no longer able to determine whether the variable list and expression list are balanced. It is probable

that this latter grammar would be accepted with the stipulation that determination of balanced lists be placed in the realm of semantic error analysis.

Questions of importance concerning both semantic analysis and CWS organization include: (1) When is semantic analysis performed? (2) How are control and data passed from the syntax analyzer to the semantic analyzer? and (3) How is the semantic interpretation of a source language statement represented? These will be addressed in the following paragraphs.

With respect to when semantic analysis is performed, there are two alternatives. The first is after the parse tree is complete. This is advantageous if a parsing algorithm employing backtracking is utilized, because semantic analysis is more difficult to undo than syntactic analysis. The second alternative is that of interleaving syntax analysis and semantic actions. This approach allows an execution cycle for the semantic analyzer each time the sentential form is changed. When coupled with top-down methods, the actual execution of the semantic actions may be delayed until the parser is certain the correct production has been applied. When interleaved semantic action is employed, the parse tree may never be explicitly constructed because the semantic routines "consume" it at the same rate it is generated.

There exists an unlimited number of variations in the manner in which information is passed from the syntax analyzer to the semantic analyzer. There are, however, several basic methods. The first, providing the entire parse tree to the semantic procedures, is an option available only if it is completed prior to the first semantic call. Otherwise, the approaches are not limited by when semantic interpretation is performed, and it should not be inferred that each is mutually exclusive of the others.

The second and third approaches are similar in that both generally assume a single control entry to the semantic analyzer. In the second approach, the syntax analyzer passes through this control point the identity of the production applied plus the current sentential form or part thereof [58]. The third approach requires an extension to the syntax definition metalanguage. A code (possibly null) is associated with each production denoting a specific semantic action [59]. In addition, the metalanguage might also provide a method of expressing data arguments to accompany the action codes. Such arguments may refer to symbols in the source input sentence or to results of previous semantic actions.

The fourth approach is somewhat simpler in concept. It requires that the specification of the name of an applicable semantic routine be attached to each production [60]. Data information may consist of the sentential form itself or an argument referencing capability similar to that of the third approach. With respect to CWS technology, the semantic routines thus referenced may be either provided by the CWS user or drawn from a standard set provided by the CWS.

The fifth approach, transduction grammars [9], is not dissimilar to the third, except perhaps in formality. It is more formal in that its structure is based on the grammar and not the software comprising the semantic analyzer. With each production is associated a "transduction." The symbols permitted in transductions include all those of the original grammar plus any new terminal symbols that are necessary to convey semantics. The new grammar may be written as follows (with the transductions in braces):

$$\langle a \rangle :: = \langle b \rangle \text{ op } \langle c \rangle \quad \{ \langle b \rangle \langle c \rangle \text{ op} \} \quad .$$

If, for example, the expression grammar previously presented is rewritten as a transduction grammar using the rules,

1. Nonterminals in each transduction will be sequenced in the same manner as the corresponding production
2. Terminals in each transduction will be the same both in sequence and value as those of the corresponding production except parentheses and pad symbols will be omitted
3. All nonterminals in a transduction will precede the terminals

the result is

- |     |                               |        |  |  |
|-----|-------------------------------|--------|--|--|
| #1. | $\langle s \rangle$           | $:: =$ | $\mid \langle \text{exp} \rangle \mid$                     | $\{ \langle \text{exp} \rangle \}$                               |
| #2. | $\langle \text{exp} \rangle$  | $:: =$ | $\langle \text{exp} \rangle + \langle \text{fact} \rangle$ | $\{ \langle \text{exp} \rangle \langle \text{fact} \rangle + \}$ |
| #3. | $\langle \text{exp} \rangle$  | $:: =$ | $\langle \text{fact} \rangle$                              | $\{ \langle \text{fact} \rangle \}$                              |
| #4. | $\langle \text{fact} \rangle$ | $:: =$ | $\langle \text{fact} \rangle * \langle \text{pri} \rangle$ | $\{ \langle \text{fact} \rangle \langle \text{pri} \rangle * \}$ |

- #5.  $\langle \text{fact} \rangle :: = \langle \text{pri} \rangle \quad \{ \langle \text{pri} \rangle \}$
- #6.  $\langle \text{pri} \rangle :: = ( \langle \text{exp} \rangle ) \quad \{ \langle \text{exp} \rangle \}$
- #7.  $\langle \text{pri} \rangle :: = i \quad \{ i \}$

The significance of the grammar produced by these particular rules will be demonstrated presently. When a transduction grammar is employed, the syntax analyzer constructs the parse tree by any of the methods previously mentioned. As reductions are made on the sentential form (or alternatives applied if top-down), the parser presents the transductions associated with productions to the semantic analyzer.

Once information is acquired from the syntax analyzer, it must be represented in a standard form for further processing. One standard form might be the object code of the target computer, in which case the resultant compiler would correspond to the one-pass translator depicted in Figure 1. Languages such as FORTRAN and JOVIAL are amenable to one-pass compilation, but others such as recursive Algol are not. Thus, attention here will be directed toward semantic representation for additional internal manipulation in a multipass environment.

Such internal representations are often called intermediate languages (IL). Those frequently used include Polish notation, tree graphs, and tuples. Advantages accrue from using a standard representation in that a wealth of theory and methods exist for their manipulation. One variant of Polish notation is called Polish postfix. It is a manner of representing the order in which operations are to be performed as an expression without using parentheses. The name Polish notation may be credited to its development by the Polish logician J. Lukasiewicz. Examine the expression obtained from the language generated by the previous grammar:  $i*(i+i)*i$ . The Polish postfix for this expression is  $i \ i \ i \ + \ * \ i \ *$ . To "execute" the Polish representation, scan it from left to right; each time an operator (i.e.,  $*$  or  $+$ ) is encountered, perform it on the previous two operands and replace them with the result.

In the transduction grammar recently described, by applying the productions in a manner corresponding to a rightmost derivation of the expression just given, one obtains



#7. <pri> :: = i	{i}
#5. <fact> :: = <pri>	{<pri>}
#7. <pri> :: = i	{i}
#5. <fact> :: = <pri>	{<pri>}
#3. <exp> :: = <fact>	{<fact>}
#7. <pri> :: = i	{i}
#5. <fact> :: = <pri>	{<pri>}
#2. <exp> :: = <exp> + <fact>	{<exp> <fact> +}
#6. <pri> :: = (<exp>)	{<exp>}
#4. <fact> :: = <fact> * <pri>	{<fact> <pri> *}
#7. <pri> :: = i	{i}
#4. <fact> :: = <fact> * <pri>	{<fact> <pri> *}
#3. <exp> :: = <fact>	{<fact>}
#1. <s> :: =  —<exp>—	{<exp>}

Now, reading the transductions, arrange the terminal symbols in the order generated: i i i + \* i \* .

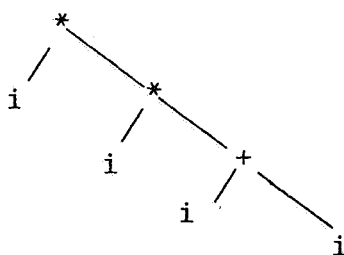
Polish notation itself has been used as the IL of several industrial compilers (e.g., DOS/360 Fortran IV [61]). More importantly, the underlying concept — that operations be performed in the order encountered — is also the basis of tree graphs and tuples. Therefore, a semantic analyzer that does not use Polish as the ultimate IL might very well first arrange expressions in Polish to assist in IL generation.

Tree graphs represented as list structures are a second and frequently used [4,62] form of intermediate language. Trees may be generated directly from Polish postfix by scanning from left to right and upon encountering an operator:

1. Creating a new node containing the operator
2. Attaching the nearest operand to the left as the right subtree

3. Attaching the next nearest operand to the left as the left subtree
4. Replacing the operator and operands in the Polish string with a descriptor of the new node.

The tree graph for the expression  $i*(i+i)*i$  is:



Note that commutative operators permit different but semantically equivalent tree graphs for the same expression. This can be advantageous during optimization.

The last form of IL to be discussed is that of tuples, of which there are several forms. Three-tuples (or triples) admit one operator and two operands per element [63]. Four-tuples (or quadruples) admit one operator and three operands [64]; the third operand designates the result of the operation. Free-tuples admit one operator and as many operands as apply [58]. Representing the expression  $i*(i+i)*i$  in each, one gets:

<u>Triples</u>	<u>Quadruples</u>	<u>Free-Tuples</u>
#1. + i i	#1. + i i Temp1	#1. + i i
#2. * i (#1)	#2. * i Temp1 Temp2	#2. * i (#1) i
#3. * (#2) i	#3. * Temp2 i Temp 3	

Strictly speaking, the forms shown which permit one line to explicitly reference another are called indirect forms. However, no distinction between the direct and indirect forms will be made in the following discussions.

It should be noted that the objective of translating the source grammar to an intermediate language may not be compilation but direct execution by an interpreter. Another aspect to consider is that the semantic analyzer is also responsible for things other than IL generation. It must maintain the internal storage tables and ascertain that all operations and symbols are used consistently (i.e., it must diagnose semantic errors). Indeed, the latter is both more difficult to perform and more difficult to describe in a generalized fashion than the translation itself. A discussion of these issues, as well as a more thorough discussion of IL, is contained in Reference 18.

## D. Optimization

There are two categories of optimization. The first, local optimization, consists of transformations performed upon the intermediate language that require only limited knowledge of the context of each operation. Examples are folding (performing a designated operation on adjacent constants at compilation time) and unary complement analysis (combining a unary operation with an adjacent operation). If local optimization is performed by a CWS generated compiler, it may be during (1) semantic analysis, (2) a separate pass over the IL, or (3) after code generation. Of course, a combination of these is possible. Specifics of local code transformations may be found in References 65 and 66.

In global optimization, the second category, larger program contexts are examined prior to making transformations. This category legitimately includes target dependent improvements such as register allocation, but only those that are independent of the target architecture will be considered here. Examples of transformations that are not constrained by an architecture are:

- Constant Propagation — Replacing variables with their known constant values; this may lead to additional opportunities for folding.
- Dead Definition Elimination — Eliminating assignments to variables which are not used again prior to being reassigned values.
- Subexpression Factoring — Saving via temporary variables the values of common subexpressions and substituting these variables at subsequent points where the subexpressions appear.

- Invariant Expression Factoring — Moving to the outside of a loop an expression whose value does not depend on variables computed inside the loop.
- Operator Strength Reduction — Changing a multiplication operation within a loop to a faster cumulative addition; theoretically the same transformation could be made from exponentiation to multiplication.

It should be noted that (1) each of the above operations involves consideration for safety factors and (2) most global optimizations make heuristic, not absolute, improvements. With respect to safety, operator strength reduction over type real operands may change the resultant code by causing roundoff errors not otherwise present. With respect to heuristic, it is possible to devise a case of invariant expression factoring in which memory space is not decreased and execution time is increased. For a more detailed discussion of global optimization transformations, see References 26 and 67 through 70. For an excellent discussion of the safety factors involved, see Reference 71.

Given a set of global optimization strategies, there yet remains the decision as to how to divide the program being compiled into sections within which each strategy will be applied. That is, how may the "optimization windows" be found? There are two approaches: (1) sectioning the program based on explicit semantic characteristics of the language and (2) graph theoretical techniques.

Two techniques of the first approach are block analysis and loop analysis. In block analysis, the program being compiled is divided into basic blocks (linear code segments that have the property that if the first instruction is executed, all others are also executed exactly once). Afterwards, each strategy is performed over each basic block with little, if any, optimization over multi-block segments.

Loop analysis is an extension of block analysis whereby those program loops defined by explicit language constructs (e.g., "DO" and "FOR" loops) are examined as multiblock segments. Blocks not within loops are optimized individually. Several compiler writing systems and compilers use the loop analysis approach [63, 69, 72].

One graph theoretical technique is strongly connected region (SCR) analysis. An SCR consists of a subset of nodes within a directed graph for which there is at least one path between each node pair. This technique involves treating the basic blocks as nodes from which a directed graph (digraph) of the program being compiled is constructed. Figure 12 illustrates a digraph of a program with three SCR's. Next, the nodes within each SCR are ordered and optimization is performed on the innermost region first. Optimization over nodes not within an SCR is performed on a basic block basis. The authors know of at least one compiler [64] using SCR analysis, but know of no compiler writing systems.

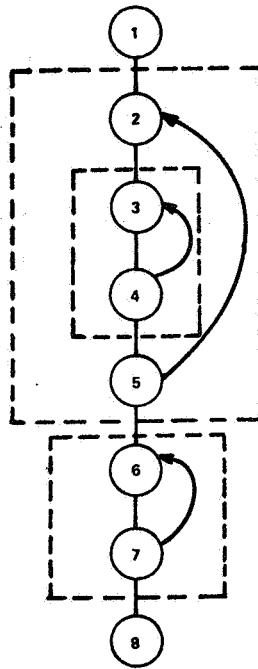


Figure 12. Directed graph with three strongly connected regions.

There are two weaknesses in the SCR approach: (1) SCR's do not completely partition a program digraph, and (2) there is no systematic manner in which to carry optimizations across SCR boundaries unless the regions are nested. A second graph theoretical technique, interval analysis [61, 71, 73, 74], overcomes these problems.



An interval is a digraph construct consisting of an initial node (called the head node or h-node) and the set of all nodes that can be reached from it subject to the following restrictions: (1) if the h-node is removed, the set contains no strongly connected regions, and (2) every path containing one of the nodes and leading from the entry node to the exit node of the digraph first enters the h-node. Figure 13 illustrates a program digraph divided into intervals. (The nodes of this figure are labeled arbitrarily.) An algorithm for determining intervals is:

1. Enter the program entry node as the first member of a set H.
2. Remove a node from H and enter it as the first member of an interval set  $I(h)$ .

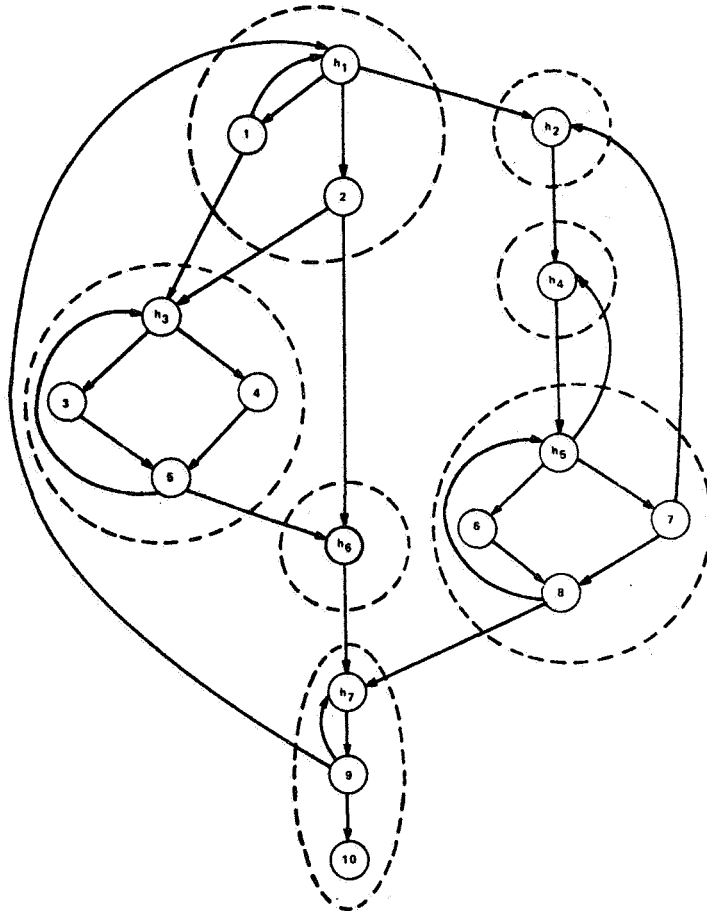


Figure 13. Directed graph partitioned into intervals.

3. For each node,  $n$ , not a member of  $H$  or previously assigned to an interval set:
  - a. Enter  $n$  in  $I(h)$  if all of the immediate predecessors of  $n$  are members of  $I(h)$ .
  - b. Enter  $n$  in  $H$  if some (but not all) of its immediate predecessors are in  $I(h)$ .
4. Repeat steps 2 and 3 until  $H$  is empty.

Each interval will have a single entry and the set of intervals determined by this algorithm will uniquely partition the program digraph.

Once the intervals are established, the optimization strategies are performed over each interval separately. Once done, the program digraph is collapsed by treating each interval as a single node and establishing the intervals for the new (derived) graph. Figure 14(a) depicts the first derived graph of the primary graph in Figure 13. Optimizing and collapsing continues until the final derived graph consists of a single node as shown in Figures 14(b), 14(c), and 14(d). Thus, intervals provide a systematic manner of propagating each optimization strategy throughout the entire program.

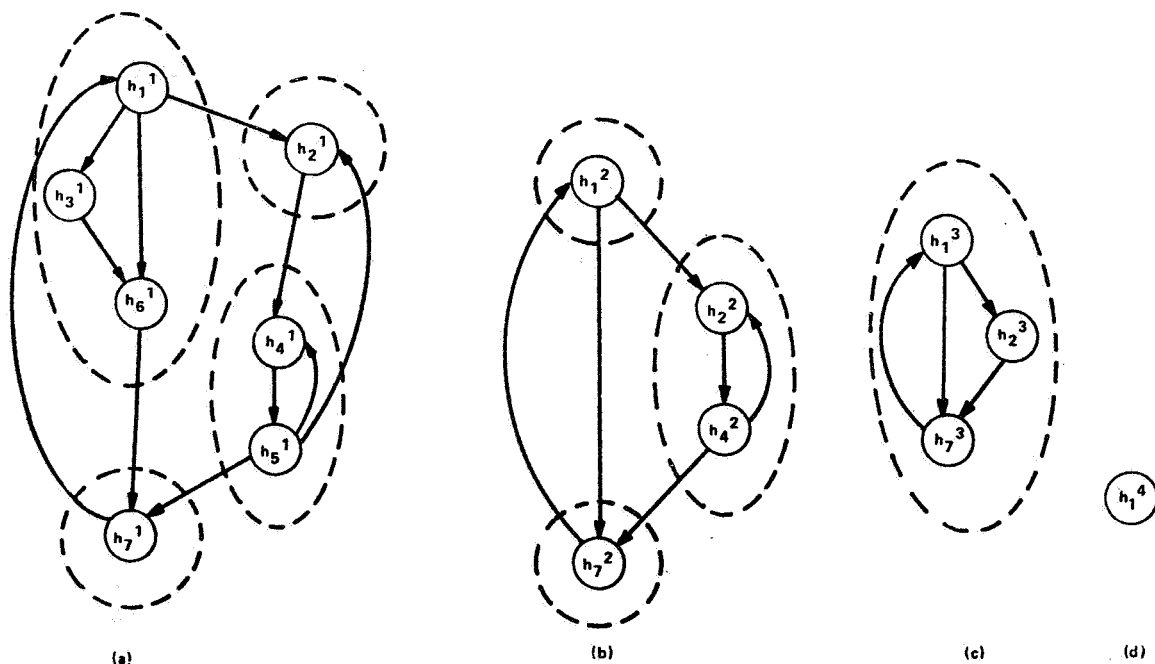


Figure 14. Derived graph from Figure 13.

It is not possible to leave interval analysis without injecting a comment about irreducibility [75]. Certain digraph constructs, notably the program loop having two entries, will not collapse to a single node. A digraph with this condition is said to be irreducible. Although this phenomenon is relatively rare [76], allowances must be made for it in a generalized system. It is possible to either transform the digraph into a semantically equivalent one by a method called node-splitting [71] or altering the mathematical procedures used in performing the optimization strategies [77].

Other approaches to optimization via graph theoretical analysis exist. For example, Kildall [78] suggests an iterative approach in which all paths to a given node are searched in consecutive order until an information pool "converges." By converge, it is meant that the information remains unchanged through an iteration. Miller et al. [79], for purposes other than optimization, embodied the concept in a method of enumerating decision-to-decision paths. Paige [80] formalized the purely graph theory aspects of the approach. Graham and Wegman [81] combined it in an algorithm that also included the interval notion. Although the iterative method has attracted considerable attention from other sources as well [82,83], it has not yet impacted CWS technology.

Yet another approach is suggested by Wulf et al. in the design of a compiler for the language BLISS [84]. The method is akin to the iterative approach but with a difference. Because BLISS is a language containing only the sequential, alternative, and repeat control structures permitted in the structured programming philosophy, all necessary data flow relationships are explicit in the semantic interpretation of the language. Of course, the method lacks the generality required of a CWS algorithm.

Currently, only the more recent and ambitious compiler writing systems employ self-contained graph theoretical optimizers. Others, however, provide languages and/or components through which postsyntactic analysis may be performed, and the user must choose which, if any, of the above techniques to incorporate.

## E. Code Generation

Code selection and editing may consume 50 percent or more of the implementation effort for a new compiler. Techniques and knowledge in this area are less systematic and thus are difficult to classify and analyze. It is the intent of this section to provide a general background from which to expand in the following sections.

In analyzing techniques used in code generation, it is helpful to distinguish between systems for building one-pass translators and those for building multi-pass translators. The distinction does not, however, separate the alternatives as distinctly as one might wish. For instance, the fusion of semantic analysis and code generation that occurs in a one-pass translator permits those methods previously discussed in the semantic analysis section to be applicable. Conversely, some methods discussed here are also applicable to semantic analysis. Furthermore, a one-pass CWS generated compiler may be converted to a multi-pass configuration by generating "standard" assembly code and adding subsequent hand-coded passes.

Nearly all one-pass generators are designed to emit assembly language statements. One method allows the attachment of assembly language statements to productions of the grammar for emission when the production is applied. This requires the solution of two problems: (1) how to reference operands defined in the source language and (2) how to generate and reference internal labels used as targets of branch instructions. To understand the solution to either, it is necessary to realize that not all calls to code generation or semantic routines actually generate code; some perform ancillary functions. With respect to the operands referenced by emitted assembly language instructions, there may be a number of pushdown stacks defined. Semantic routines are then evoked to push the identities of variables, numbers, and other symbols recognized onto the stacks. Attached to each assembly instruction is the name of the stack from which the operand must be fetched. Generation of internal labels has a similar solution. Attached to each assembly instruction requiring a forward reference is a request to generate and save a unique internal label. The conditions under which the label will be subsequently emitted may be attached to the generation request, or there may be a subsequent explicit emit label command.

This technique may be extended by allowing the compiler writer to reference routines that will subsequently be hand-coded and included within the resultant compiler. Continuing along this line, the CWS may provide a fixed set of built-in code generation functions which the compiler writer may delete or augment as necessary. This method often necessitates major alterations in the built-in functions if a new target computer is designated.

Another method (not necessarily limited to one-pass generators) is to provide the compiler writer with a language tailored to code generation. Once the code generator is represented in the provided language, the CWS may

translate it to executable statements or to a tabular form for interpretation by the resultant compiler. Actual code selection may be imbedded within the statements of the special language or (again not limiting the discussion to one-pass translators) separately provided via a set of primitive user-provided functions.

Ignoring, for the moment, that a one-pass translator may form the basis for a large compiler, there are several limitations to the above approaches. First, extensive optimization is prohibited. This excludes not only the global techniques of the previous section, but also optimal assignment of registers for targets that have an array of registers from which to select. A second limitation is an inability to handle directly memory addressing that is less than straightforward, such as either fixed-page or floating-page relative addressing. Of course, this may be overcome by a powerful postcompilation assembler or the inclusion of a special hand-coded postprocessor in the resultant compiler.

A CWS designed to produce multipass translators tends to be more flexible at the cost of increased effort in generating the compiler. Here, it is even more difficult to be exhaustive. The techniques discussed next are major strategies and should be considered neither complete nor mutually disjoint in application.

An assumption applied here to multipass compilers is that the first pass includes a distinct semantic analysis phase that generates a well understood intermediate language. The function of subsequent passes is to manipulate the IL (for purposes of optimization) and to generate object code. It has already been alluded that two methods for accomplishing this are: (1) provision of a tailored language for IL inspection, or (2) provision of "off-the-shelf" functions that the compiler writer may utilize or augment by adding functions coded in a compatible language. The following methods may serve either as the conceptual framework within which to apply one of the above two methods or may be incorporated more automatically by the CWS during compiler generation.

One general method is that of pattern matching followed by action sequences. Here, there are similarities with the method of production languages for syntax analysis. The semantic interpretation of the source program must be through an intermediate language such as triples, Polish notation, or others that were mentioned. The code generation metalanguage (or a tailored procedural language) provides for the representation of IL patterns and for the expression of actions to be performed when a pattern match occurs during code generation.

Patterns are expressed in terms of operations and associated operands. The operations expressed are chosen from those representable in the IL, and the operands are denoted by references to internal compiler tables and include the operand attributes (type, length, etc.). Thus, some knowledge is required on the part of the CWS user of both the IL and the internal tables. Alternatives available for representing actions to be performed upon the recognition of a pattern are similar to those described for semantic analysis. This includes both explicit calls to subordinate procedures and the direct emission of object code. Actions need not lead directly to code generation. For example, an action may result in a rearrangement of IL in a manner achieving local optimization.

Another technique, not necessarily incompatible with those previously mentioned, is a distinct separation of the target dependent and target independent aspects of code generation. It may be surprising to learn that a large portion of the processing in code generation is not particularly constrained by the target architecture. Most notable is the analysis required to determine the order in which code is generated. For example, in the emission of data definitions, it is usually safe to emit first the variables requiring the greatest storage space (e.g., double precision prior to single precision). Instruction ordering generally requires more complex algorithms.

One algorithm for instruction ordering is that of Sethi and Ullman [85]. Conceptually, the expressions represented by the IL must be in binary tree form. (Note that triples and other forms are actually alternate representations of a tree structure.) Nodes of the tree are augmented with resource numbers. A resource number corresponds to the number of registers it will require to generate a value for the subexpression subtended by a node. Resource numbers are assigned under the assumption that at least one operand, specifically the left one if an operation is not commutative, must be in a register prior to an operation. Once resource numbers are assigned, the algorithm traces the path through the tree delineated by greatest resource numbers and emits the last operation on the path. It then deletes the operation from the tree and regenerates the resource numbers for the remaining nodes.

The idea behind the algorithm just described is that by emitting the operations requiring the most resources first, some of the registers utilized will be released and thus become reusable in following operations. There is another aspect of the algorithm which guarantees that the result of a left sub-expression of a noncommutative operation will be available before that of the

right subexpression. This is accomplished via an artificial manipulation of resource numbers. It can be proved that the algorithm gives optimum instruction sequences under ideal conditions; namely, (1) there are no external calls, and (2) no operation requires dedicated registers. In other circumstances, the algorithm gives near-optimal sequences.

Target dependencies assert themselves at the point where assembly or machine instructions must be chosen to reflect an IL operation. Such dependencies may be imbedded within a rather large set of primitives which the CWS user frequently must write and insert within the code generator of the resultant compiler. It is possible, however, to represent the mapping of IL operations to machine instructions within a data structure that is largely independent of the generating algorithm. One such representation is the decision table method reported by Lowry and Medlock [64].

The decision table method actually consists of a series of small decision tables, each called a skeletal code block (SCB). There is roughly one SCB per IL operation, but if the operation has several quite different candidate instruction sequences based on data type or other conditions, it may be represented by several SCB's.

Each SCB consists of a small decision table matrix with rows labeled by machine instructions and the columns simply numbered beginning with zero (Fig. 15). The code selection procedures generate an internal status vector consisting of a series of true/false values. The vector is collected to form a

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LA	A1, U1, X1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
LA	A3, U1, X1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
LA	A2, U2, X2	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
LA	A3, U2, X2	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
LA	A3, A1	0	0	0	0	1	1	0	1	0	0	0	0	0	1	0	1
FA	A3, A1	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	0
FA	A3, A2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
FA	A3, U1, X1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
FA	A3, U2, X2	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
SA	A3, U3, X3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

LA = LOAD ACCUMULATOR

FA = FLOATING POINT ADD

SA = STORE ACCUMULATOR

BIT MASK				
1	0	0	1	1

SELECTS 9th COLUMN

Figure 15. Decision table for code selection.



binary number (with true represented by one) called the bit mask. The most significant bits of the mask constitute an integer value used to select a column from the decision table. Accessing the elements of the selected column in ascending order, instructions (i.e., row labels) are selected in the following manner:

1. If an element is zero, then the corresponding instruction is skipped.
2. If an element is one, then the corresponding instruction is emitted.
3. If an element is greater than one, then the element value is used to select a bit from the bit mask (always from the least significant portion) whose value determines whether the instruction is to be skipped or emitted.

(The above procedure differs slightly from that of Lowry and Medlock but is equivalent in concept.)

The role of the bit mask can be illustrated easily by an example. Assume that the conditions constituting the mask, beginning with the most significant bit, are as follows:

- Bit 0 — The first operand is already in a register.
- Bit 1 — The first operand must be left in a register after the operation.
- Bit 2 — The second operand is already in a register.
- Bit 3 — The second operand must be left in a register after the operation.
- Bit 4 — The result must be stored.

Figure 15 depicts the selected instructions (designated by circled decision table elements) for a floating point addition on a target machine similar to a Univac 1108. The conditions leading to the selected instructions of Figure 15 are: first operand is already in a register, the second operand must be left in a register following the operation, and the result must be stored.

In Figure 15, the codes used for the instruction operands are: A1, A2, A3 — the first, second, and result operand accumulators, respectively; U1, U2, U3 — the base addresses for the first, second, and result operands, respectively; and X1, X2, X3 — the index registers for the first, second, and result operands, respectively. It falls upon the code selection procedures to maintain content

descriptors for all registers in order to make proper substitutions for the A1, A2, A3, X1, X2, and X3 codes when emitting an instruction. Register attributes (types of operations permitted, etc.) are maintained in parameter tables that may be reconfigured using the code generation metalanguage.

There is another wrinkle to the decision table method that is worthy of notice. Rather than grouping instructions into SCB's based on IL operations, they may be grouped based on function. For example, register loads may be in one SCB, and addition instructions may be placed in a second. Using this approach, one of the functions of the bit mask would be selection of the proper SCB's. If there are a large number of conditions comprising the bit mask, the previous approach of grouping by IL operation may result in sparse decision table matrices. Thus, grouping by function can result in increased storage efficiency.

In concluding this discussion of code generation, it is again appropriate to state that it is almost impossible to assemble an exhaustive catalog of techniques. It is hoped that the descriptions of actual systems presented in some of the following sections will indicate the variations and combinations of methods possible.

#### IV. EXPECTED PERFORMANCE RANGE OF A CWS

A particular CWS system will exhibit performance boundaries in two contexts: (1) the range of languages for which it is suited and (2) the range of target architectures for which it is suited. With respect to language range, CWS systems may be used to generate processors for the scope extending from the simple macrolanguages to those which dynamically induce semantic transformations (such as LISP [86]). A macrolanguage is a source language/object language pair for which it is possible to map one onto the other by direct string substitution, irrespective of semantic context. A semantics altering language is one that can dynamically change the attributes of a defined object during execution. A specific CWS will support only a subset of this spectrum, as depicted in Figure 16.

The architectural range supported by a particular CWS is generally inversely proportional to the level of support it provides. Pure translators (i.e., those providing little postsyntactic support) are effectively unbounded

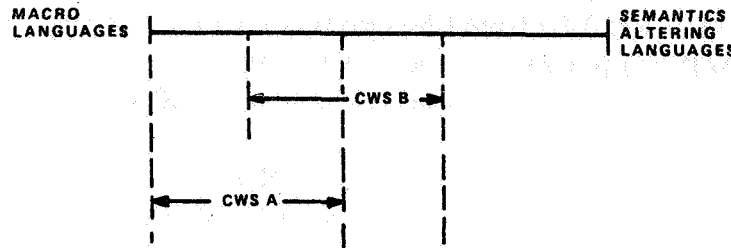


Figure 16. CWS performance range spectrum.

in range. Meanwhile, those that provide extensive aid for semantic analysis may be utilized to the fullest extent for only one architectural family, usually the one on which the CWS itself executes. However, perhaps at the cost of customized coding for each language, some CWS systems have achieved a broader architectural applicability [87,88].<sup>2</sup>

## V. A BRIEF HISTORY

The evolution of CWS technology has not witnessed the quantum jumps that enable one to differentiate between different generations of systems. New techniques have arisen steadily, with the more recent advances concentrated in postsyntactic analysis. Yet, these have augmented rather than replaced existing techniques; thus, an understanding of past efforts can lead to better comprehension of current trends. A more thorough description of most of the systems discussed is available elsewhere [2].

One of the earliest successful compiler writing systems was BMCC developed at Manchester University [62,89-91]. The syntax metalanguage for BMCC is similar to BNF. The syntax analyzer for compilers generated by BMCC utilizes the top-down approach and constructs a tree in list structure form for each input source language sentence. The tree is passed to the semantic analyzer at intermediate construction steps designated by the user within the syntax metalanguage.

---

2. Language Implementation System. Chi Corporation, Cleveland, Ohio, Undated Report.

The semantic analyzer has a single control entry and is designed to interpret semantic routines supplied by the user in a very high level language. Each user-supplied semantic routine is identified by a heading containing the production forming the root of all syntax subtrees for which the routine is applicable. The interpretive semantic language is procedural (i.e., allows control flow). Control flow choices are expressed in terms of conditional expressions for which the truth value is based on whether or not a given production was applied at the level being examined. For example,

IF <factor> : : = <primary>\*<primary> THEN GOTO L1

will execute the statement at L1 if, at the current tree node, <factor> was expanded by the given production. (No attempt has been made in the above example or will be made in others to replicate the notation of a given system. Rather, notational constructs have been selected which have the greatest intuitive meaning.) Attributes of input symbols may be referenced within the semantic language by using the nonterminal producing the symbol. For example,

ACC : = <variable>

means load an accumulator with the identifier symbol appearing in the source input sentence. Actual selection of target instructions is performed by independently written selection primitives that are retrofitted to the semantic analyzer. Thus, BMCC is basically a one-pass generator.

A second very significant early effort was the TGS-II system [92, 93]. It is remarkable for several reasons: (1) a uniform metalanguage, called TRANDIR, was defined in which the user describes all phases of compilation; (2) TGS-II is an application of itself; that is, the difference between TGS-II and a compiler generated by TGS-II is that one replaces internal tables describing TRANDIR with TGS-II generated tables describing the source language; (3) TGS-II was one of the first projects to grapple with the difficulty of allowing complex and composite data structures to be defined in source languages; and (4) TGS-II was one of the earliest systems to incorporate the IL concept.

Lexical analysis for TGS-II generated compilers is partly built-in, but does allow the user to incorporate a table of reserved words. Syntax analysis is via the production language method. There is a single control entry to the semantic analyzer through which are passed action codes and arguments. Arguments are specified via direct references to the syntax reduction stack, symbol table, and other internal data structures. The purpose of the semantic analyzer is to generate IL in the form of triples.

Like BMCC, TGS-II is normally used to generate one-pass translators. However, the presence of IL permits greater freedom in deciding when code should be generated. Several triples may be accumulated and then analyzed for optimizations such as the elimination of common subexpressions. Analysis for both optimization and code generation is performed via a pattern matching facility in TRANDIR. For example,

```
IF BRANCH $LABEL . . THEN EMIT (BRA $LABEL)
```

means if the triple operator is 'branch' and the operands are, respectively, a reference to the label table and null, then generate an instruction (in assembly language) to branch to the symbolic label referenced. Actual code selection is contained in user-written primitives that are retrofitted to the semantic analyzer.

The FSL system [54] can best be described by enumerating the similarities and differences it shares with TGS-II. Similarities include: (1) FSL is basically a one-pass generator; (2) syntax analysis utilizes the production language approach; (3) the semantic analyzer has a single control entry; and (4) actual code selection is performed by user-written routines retrofitted to the semantic analyzer. A further description of the differences is contained in the following.

The first difference is that FSL generated compilers have a built-in lexical analyzer. However, it may be replaced by one written by the user. A second difference is the manner in which information is passed from the syntax analyzer to the semantic analyzer. In FSL, each line of the production language program for which semantic action is necessary has an associated semantic routine number (written, for example, as EXEC 12). The semantic routines are written in an interpretive language. As arguments, the semantic routines

receive the contents of the current sentential form both before and after the most recent reduction. An IL does not exist as such, since code is usually generated based directly on the reduction applied and emitted by calling one or more of the user-written code selection primitives. A final difference is that FSL limits the source language data types to boolean, integer, single, and double precision. Of course, the language designer may combine these into composite types such as arrays and complex variables by making appropriate definitions in the grammar and semantic routines. Iturriaga et al. describe a sample FSL application [94].

The META systems [95-97] offer quite a different pattern. Lexical analysis in META generated compilers consists of recognizing several built-in symbols. Syntax analysis is performed via the recursive descent method. Semantic analysis/code generation is performed by attaching symbolic instructions to productions (which are emitted when the production is applied) and certain predefined functions (such as emit a symbolic label). When a symbol is read from the input stream, it is placed on a principal internal pushdown stack. From this stack, copies may be placed on up to four auxiliary stacks. Operands (and labels) are specified by referencing one of these. Actually, members other than the top element may be referenced or erased; therefore, strictly speaking, they are not true stacks. It is possible to add handwritten semantic routines and specify when during syntax analysis they are to be called. Later versions include a more complete set of built-in functions, particularly in the area of input/output handling.

There were a number of other successful systems developed during the 1960's that merit attention. CGS [63] preceded TGS-II and was a product of the same research team. Compilers generated by it employ top-down parsing strategies to construct a syntax tree. A semantic language is used to generate IL (in triple form) from which code (described by a different language) is generated. Gargoyle [98] is a language in which one writes a syntax analyzer using the top-down parse machine approach. Within the action portions of the parse machine it is possible to emit assembly instructions, save input stream symbols for future reference, or set and reset various flags and conditions. Additional tests may be imbedded within the action portion to select a particular action from the set appropriate to the semantic context. COGENT produced compilers [99] utilize a grammar representation in list structure form but are not strictly top-down parsers. A form of backtracking is necessary, however, but it is not performed in the manner of erasing the parse tree and restarting as was previously described. Rather, it constructs all alternatives in parallel,

dropping ones that eventually prove infeasible. A list structure of the parse tree is constructed and semantic analysis is interleaved with syntax analysis except when several parse alternatives exist, in which case semantic analysis is delayed until only a single alternative remains. Semantic analysis/code generation is performed using a tailored procedural language incorporating list processing and pattern matching concepts. TMG produced compilers [100] utilize the parse machine approach to syntax analysis. Semantic analysis/code generation is performed both by references to built-in functions (to which the user may add) and explicit emission of character strings and labels. AMOS [60] is more total system oriented but accomplishes this by resorting to the generation of translators which produce a common IL for which an interpreter is provided. The syntax analysis algorithm is that of the top-down parse machine. Semantic routines are written in a tailored language for which execution is interleaved with parsing. The output of translation is essentially Polish notation. The META PI system [101] borrows much in the way of notational methods from the META systems from which it derives its name. The differences are: (1) it is implemented in extended recursive FORTRAN, (2) it is designed to be used in an interactive mode, and (3) it generates compilers that are both interactive and incremental. META PI produced compilers utilize recursive descent parsing techniques and, like the META systems, apply semantic commands attached to productions. Unlike the META systems, immediately executable machine code is produced and there are several facilities available in the metalanguage for handling general purpose registers.

## VI. CURRENT SYSTEMS

In many cases, the compiler writing systems in use today have been developed by teams composed of, or in communication with, developers of the earlier systems. Recent systems surveyed here range from those produced in academic environments with a primary purpose of instruction to those produced by or for industrial users. If these differ from those of the last decade, it is a propensity to provide multipass generators and greater post-syntactical support with fewer constraints. However, the simple parser generator has far from disappeared. Systems chosen for review were selected on the basis of availability and, to a lesser extent, diversity. Judging from past developments, it is reasonable to assume that others are extensions, refinements, or variations of the basic patterns presented.



## A. SIMCMP/STAGE2

Simple Compiler (SIMCMP) [102] and STAGE2 [103, 104] are the least complex of those systems which may be used to generate language processors. It is a macro-based generative system used by its authors (who do not acclaim it a CWS) as a research tool for architecture independent programming techniques. Its operation is shown diagrammatically in Figure 17.

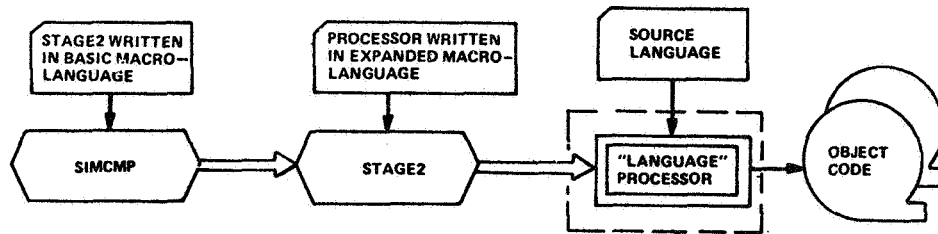


Figure 17. SIMCMP/STAGE2 organization.

The underlying idea is to use a simple macroprocessor to generate a more complex macro-based system which may be used to generate yet another more powerful processor. The approach of evolving systems by basing their development on simpler systems of the same generic type is called bootstrapping. (Bootstrapping is frequently employed in the absence of CWS tools for developing, from scratch, a compiler for a new language or an assembler for a new computer.) The SIMCMP/STAGE2 system is highly transportable, since both are based on FORTRAN. However, if efficiency so dictates, it is easy to generate an assembly language version of STAGE2. The systems, being generative rather than analytical, do not possess the syntax, semantic, and code generation phases emphasized earlier. SIMCMP will be described by illustrating a macro and then outlining its algorithm.

A SIMCMP macro definition consists of a heading called the template by which it is identified and a body representing a character string to be emitted when the macro is evoked. Taking some liberties with the macro parameter specification notation in order to simplify the explanation, a SIMCMP macro definition might appear as:

```

TO ab IF CAR cd = CDR ef                                (1)

    I = CDR (cd)                                          (2)

    J = CDR (ef)                                          (3)

    IF (CAR(I) - CDR(J)) '$1' , ab , '$1'              (4)

    '$1' CONTINUE                                        (5)

END                                                        (6)

```

Line (1) is the template, lines (2) through (5) are the body, and line (6) denotes end of definition. The parameters are the lower case letters a, b, . . . , f; that is, each character is a separate parameter. The notation '\$1' causes an internal reusable numeric label to be generated; up to 10 such labels are permitted per macro definition. Should SIMCMP receive the string

```
TO 13 IF CAR 17 = CDR 32 ,
```

which matches the template of line (1) with parameters being compared only for equivalent substring length, the following code would be generated:

```

    I = CDR(17)

    J = CDR(32)

    IF (CAR(I) - CDR(J)) 100,13,100

100 CONTINUE .

```

This example illustrates internal label generation and the direct character by character transposition of arguments into the generated code. Not shown is the capability to make simple predefined numeric substitutions for actual arguments and to specify dissimilar input and output character codes.

In using SIMCMP, one first initializes all macro definitions, which are stored internally in tabular form and preceded by a template. Each source language statement is then compared to the templates in sequential order. The first match causes the corresponding macro to be selected and expanded.

SIMCMP is used to generate STAGE2. First Language Under Bootstrap (FLUB) is defined, then STAGE2 is programmed using FLUB. Usually, FLUB macros (of which there are only 30) are specified in FORTRAN. Once STAGE2 exists, it is possible to regenerate it in more efficient form by rewriting the FLUB macros in the more powerful STAGE2 capabilities. Frequently, the second translation is used to create STAGE2 in assembly code form.

STAGE2 capabilities can best be described by comparison to those of SIMCMP. STAGE2 templates are stored in an internal tree rather than in tabular form. Source statements are compared to the templates using a series of rules that have the effect of selecting the one that maximizes the number of literal characters matched. Although templates and macros are specified in the same general manner for STAGE2 as for SIMCMP, the tree matching algorithm makes it unnecessary to assume a fixed string length (namely, one) for each parameter as does SIMCMP. STAGE2 allows a portion of a macro body to be iteratively emitted by a count that is either prespecified or controlled by the source statement. While SIMCMP allows only the literal translation of parameters or substitution by predefined numeric values, STAGE2 additionally permits predefined character strings, translation time expressions, and insertion of the numeric value for parameter string length among other options.

The SIMCMP/STAGE2 combination has been used to generate text processors, line editors, and page layout applications. More significantly, it has been used to build processors for the intermediate language JANUS [105]. A program written in JANUS consists of a series of statements, each having one operator and one operand plus attached attributes such as arithmetic type. The operators are generally recognizable as the directives and instructions available in most assembly languages. JANUS code can easily be translated in macro fashion to zero-address, one-address, or general register targets. The general idea is to construct compilers that translate source language to JANUS, then utilize a STAGE2/JANUS processor to obtain object code.

## B. LANG-PAK

LANG-PAK [106] is a parser generator designed for utilization in interactive environments. A compiler writer using LANG-PAK may design a grammar incrementally during one or more sessions at a terminal. During design, the grammar or any embedded subgrammar may be tested by entering and tracing trial statements of the new language. The LANG-PAK metalanguage is an extended BNF. Since it incorporates several features common to meta-languages of other top-down parser generators, this is an ideal place to introduce these extensions.

Since it will be necessary to add new metasymbols (that is, symbols other than the  $\langle$ ,  $\rangle$ ,  $::=$  used thus far), the following BNF extension examples will use quotes (") to bracket and distinguish terminal symbols. The first such extension is factoring. In BNF, if the nonterminal  $\langle a \rangle$  could be composed of either substring  $\langle b \rangle \langle c \rangle$  or substring  $\langle b \rangle x$ , it would be expressed as

$$\langle a \rangle ::= \langle b \rangle \langle c \rangle \mid \langle b \rangle x \quad ,$$

where the symbol,  $\mid$ , means "or." In the LANG-PAK metalanguage, this would be expressed as

$$\langle a \rangle ::= \langle b \rangle (\langle c \rangle \mid 'x') \quad ,$$

with  $\langle b \rangle$  factored from each alternative. This tends to reduce the size of the grammar, but, more importantly, it tends to reduce backtracking. For example, in the first representation (strict BNF), assume the parser expanded  $\langle a \rangle$  with  $\langle b \rangle \langle c \rangle$ , recognized  $\langle b \rangle$ , then failed to find  $\langle c \rangle$ . It would then erase the subtree below  $\langle a \rangle$ , expand with  $\langle b \rangle x$ , then proceed to find  $\langle b \rangle$  again. Using the second representation (extended BNF),  $\langle b \rangle$  needs to be recognized only once in attempting both alternatives for  $\langle a \rangle$ .

As has been noted, left recursive definitions cannot be properly handled by top-down analyzers. Although any grammar that can be expressed in BNF with left recursion can also be expressed without left recursion, the final representation may be larger and more awkward. Thus, a BNF extension is a

more palatable solution. Since recursive algorithms can be converted to semantically equivalent nonrecursive ones using iteration, it is natural to seek a BNF extension that utilizes repetition to eliminate grammatical recursion. Consider the recursive definition used to specify that an integer is a string of digits:

$$\begin{aligned}\langle \text{integer} \rangle &::= \langle \text{digit} \rangle \mid \langle \text{integer} \rangle \langle \text{digit} \rangle \\ \langle \text{digit} \rangle &::= 0 \mid 1 \mid 2 \mid \dots \mid 9\end{aligned}$$

Using repetition in a manner similar to that of LANG-PAK, one obtains

$$\begin{aligned}\langle \text{integer} \rangle &::= (\langle \text{digit} \rangle) \{ \underline{\text{rep}} \ 1 \ 100 \} \\ \langle \text{digit} \rangle &::= ('0' \mid '1' \mid '2' \mid \dots \mid '9')\end{aligned},$$

or even better,

$$\langle \text{integer} \rangle ::= ('0' \mid '1' \mid '2' \mid \dots \mid '9') \{ \underline{\text{rep}} \ 1 \ 100 \}$$

The rep construction indicates the preceding definition which is enclosed within parentheses may be repeated. The first number following rep is the minimum repeat count; the second number is the maximum. By setting the minimum count to zero, it is possible to denote an optional symbol. For example,

$$\langle \text{number} \rangle ::= ('+' \mid '-') \{ \underline{\text{rep}} \ 0 \ 1 \} \langle \text{integer} \rangle$$

indicates that  $\langle \text{number} \rangle$  may be either a signed or unsigned integer value. Other metalanguage extensions are related to lexical and semantic analysis and will be discussed in due course.

The LANG-PAK lexical analyzer automatically recognizes numbers. Other terminal symbols are collected into a look-up table; identifiers are recognized on a character by character basis by incorporating the definition directly into the grammar. Optionally, a user may specify within the grammar points where a user-supplied lexical analyzer is to be called to test for the next input symbol. The user-supplier analyzer may return a failure flag to the parser if the expected symbol is not found.

The syntax analyzer is of the parse machine type and generates an output stream of control codes defined by the parser and semantic codes specified by the compiler writer. There are several ways by which semantic codes may be added to the output stream:

1. Direct insertion by the lexical analyzer upon symbol recognition
2. Character strings from the source input statement (identifiers, etc.) inserted by the parser upon both recognition and request within the grammar definition
3. Semantic attributes attached to productions by the compiler writer.

There are two variations of the third method. Assume the production definition,

$$\langle \text{gotostatement} \rangle : : = \text{"goto"} \{ \underline{\text{sem}} \ 1 \ 5 \} \ \langle \text{label} \rangle \ \{ \underline{\text{sem}} \ \text{'abc'} \}$$

The notation  $\{ \underline{\text{sem}} \ 1 \ 5 \}$  will cause the integers "1" and "5" to be inserted into the output stream whenever "goto" is recognized. Handling of the second semantic specification,  $\{ \underline{\text{sem}} \ \text{'abc'} \}$ , is more complex. First the user supplies a package called the semantic compiler which is used at the time the parsing tables are generated (Fig. 18). During translation of the metalanguage, a semantic specification of the form  $\{ \underline{\text{sem}} \ \text{'...'} \}$  causes the semantic compiler to be evoked to process the character string between the apostrophes. The semantic compiler returns an array of semantic codes which become part of the parse tables generated. Semantic specifications thus translated may be arbitrarily complex, depending on the effort the user wishes to expend on the semantic compiler.

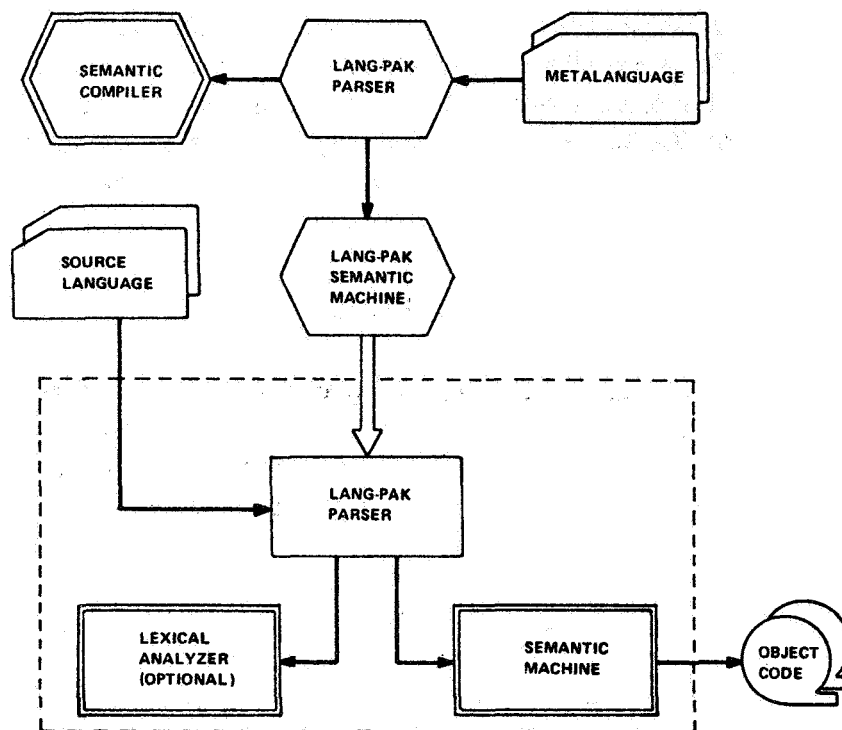


Figure 18. LANG-PAK organization.

In the resultant compiler, semantic analysis/ code generation is performed by a user-written package with a single control entry. This package is called the semantic machine to distinguish it from the semantic compiler and is driven by the output stream produced by the parser. An alternative organization allows the parser to return the output stream to an evoking program which may then proceed with semantic analysis.

The first organization is depicted in Figure 18. It is of interest because the metalanguage itself is an application of the system. (Recall that TGS-II was similar in this respect.) In Figure 18, the LANG-PAK parser is shown both outside and within the resultant compiler. The only difference between the two is that the one outside contains predefined tables for parsing the metalanguage. The one within contains the parsing tables produced for the new language.

The authors of LANG-PAK warn that its use should be confined to the generation of processors for "small" grammars such as are typical for interactive query languages. The supposed limitation is due not to theoretical



considerations of language construction but to design philosophy. To maintain the broadest possible range of host computers (it is coded in both FORTRAN and PL/I), certain inefficiencies in storage and speed were introduced. Though parsers generated by LANG-PAK are relatively large, they are not unreasonably so. LANG-PAK can be used to generate parsers for the larger languages.

## C. COGENT

Compiler Generator (COGENT) is a proprietary product of Virtual Systems, Inc., [107] and is not related to a previous compiler writing tool of the same name [99]. A discussion of it is included at this point because it is an excellent vehicle on which to continue the discussion of BNF extensions begun under LANG-PAK.

The LANG-PAK metalanguage contains extensions (factoring and iteration) that tend to optimize the top-down parsing algorithm and other extensions for embedding semantics. In LANG-PAK, only simple backtracking is performed. All productions beginning with identical nonterminal symbols must be grouped and factored in a manner that allows parsing of any production to proceed from left to right without erasing a previously recognized nonterminal directly contained within it. The COGENT metalanguage also contains the factoring and iteration extensions, but the parsing strategy permits more elaborate backtracking. To reduce the amount of redundant information saved during the parse, the language designer specifies through the metalanguage the points to which backtracking may be necessary. Consider the productions,

```
<goto> :: = mark <uncongoto> | <assigngoto>
<uncongoto> :: = "goto" <label>
<assigngoto> :: = "goto" <id> ", "'(" <label list> ")"
```

The mark metasymbol indicates to the parser that if it is unsuccessful in recognizing <uncongoto>, it is to reset the scan and parse tree before attempting the next alternative. Of course, this simple example could probably be handled more effectively with factoring.

More exciting from a language theoretical viewpoint is the capability to select production alternatives based on the next several terminal symbols in the input stream. [Recall the definition for LL(k) grammars.] To illustrate this facility, consider a programming language for which all statements other than assignment begin with a keyword from the set — IF, DO, CALL, or END. Using the look-ahead technique, the syntax for the assignment statement may be represented as follows:

```
< assignment > :: = ifnot ('IF'|"DO"|"CALL"|"END")
                        <id> '=' <expression>
```

The domain of the ifnot metasymbol is the alternatives immediately following and enclosed within the parentheses. If all the alternatives fail to match the next input symbol, then the parser attempts to recognize a sequence corresponding to <id> '=' <expression>.

The BNF extensions to embed semantics are similar to those described earlier for the META systems. This includes: (1) the capability to insert literal character strings (usually assembly language statements) into the output stream, (2) access to fields from the symbol table, (3) generation and explicit emission of labels, (4) references to procedures written by the user, and (5) references to an internal stack on which input symbols are stored. Access to the internal stack is less restrictive than in the META systems. Identities may be attached to input symbols as they enter the stack to facilitate subsequent referencing. In addition, arbitrary character strings (such as instructions to be generated at a later point) may be placed on the stack. However the innovation that probably adds the greatest power to the technique of utilizing an internal stack to control output generation is the concept of attributes. Any stack or symbol table element may be assigned an integer value called an attribute. (A simple example of an attribute is an integer from one to five denoting data type.) Attributes may be combined in arithmetic expressions, compared against constants or other attributes, and tested for range. True/false results of attribute tests are then used to select among candidate productions or alternative output streams. In the event of backtracking, both the stack and output stream are restored to their original state along with the scan and parse tree.

There are a number of other BNF extensions in the COGENT metalanguage. Some of these enable:

1. Embedding of syntax error messages in the grammatical description; a message that is reused needs to be defined only once.
2. Skipping a portion of the input stream delimited by a given character sequence; this is useful in ignoring comments and in skipping to a "safe" place to restart after a syntax error.
3. Definition of patterns against which the input stream may be tested; this is useful in performing local optimization.
4. Stepping through the symbol table and applying a rule for each entry that meets a given attribute constraint; this is useful in storage allocation.
5. Column positioning for fixed format languages and statement rescanning.

There is currently insufficient public information to detail the internal components of a COGENT generated compiler; thus, Figure 19 is a simple generic diagram. The built-in lexical analyzer collects decimal, octal, and binary numbers. It also recognizes identifiers for which the only restriction is that they must begin with an alphabetic character. The set of permissible identifier characters following the initial one is specified by the user. The syntax analyzer employs the recursive descent method. The COGENT system is in FORTRAN and, thus, is quite portable. Since it has been described in its own metalanguage, rehosting it in assembly language is not difficult and several assembly language versions exist. Subsequent releases are expected to possess multipass generation capability.

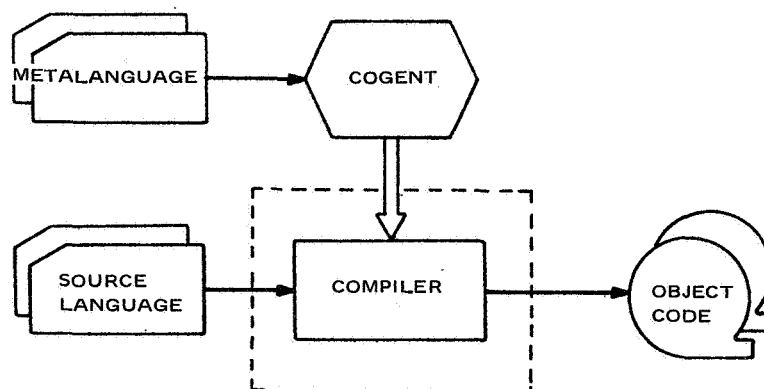


Figure 19. COGENT organization.

## D. XPL/XCOM

The XPL language and its compiler XCOM were developed by McKeeman, Horning, and Wortman [3] primarily as a research/teaching tool for syntax analysis. In this role it has met resounding success, having become widely distributed in a very short time. The XPL language, a derivative of PL/I, has reached beyond pedagogical applications as is attested by its use in the development of an HAL translator [108] by Intermetrics, Inc., for NASA. (HAL is a procedural language very similar to Algol. It contains vector operations and permits a two-dimensional coding format.) XPL differs from PL/I in that it is not recursive, permits only integer arithmetic, and has more operations for string input/output and manipulation.

Lexical analysis is an XCOM built-in function, but the operators and reserved words are contained in a replaceable table. Syntax analysis is performed by a method developed by its authors called mixed strategy precedence (MSP) and may be described in terms of the weak precedence and bounded-context methods described earlier. While scanning the input sentence from left to right, the syntax algorithm behaves as a (1,1) weak precedence parser until it encounters a conflict. At that point, a (2,1) weak precedence check is performed to resolve the ambiguity. Eventually a point in the scan is reached at which a decision is made to reduce the sentential form. There the algorithm ceases to behave as a weak precedence parser and begins behaving as a bounded-context parser. However, bounded-context analysis is required only to distinguish between two or more productions having the same right part. In these cases, the MSP algorithm selects the proper production based on a (1,1) bounded-context check.

The semantic analyzer/code generator must be independently written for each language/target combination. The semantic analyzer has a single control entry and has access to the following data items:

1. The number of the next production to be applied
2. The portion of the sentential form already scanned
3. The left and right limits of the portion of the sentential form to be reduced
4. A symbol sequence ordered in the same manner that identifiers occur in the sentential form.

A typical semantic analyzer maintains several descriptor stacks that are manipulated in parallel to the one in which the sentential form is stored.

In using XCOM, one first describes the source language using BNF. This description is the input to the provided program, called ANALYZER, for generating MSP tables (Fig. 20). The produced tables are physically inserted into XCOM. The lexical analyzer must then be modified to account for the comment conventions, reserved words, and other eccentricities of the new language. Last, the semantic analyzer and code emitters must be rewritten.

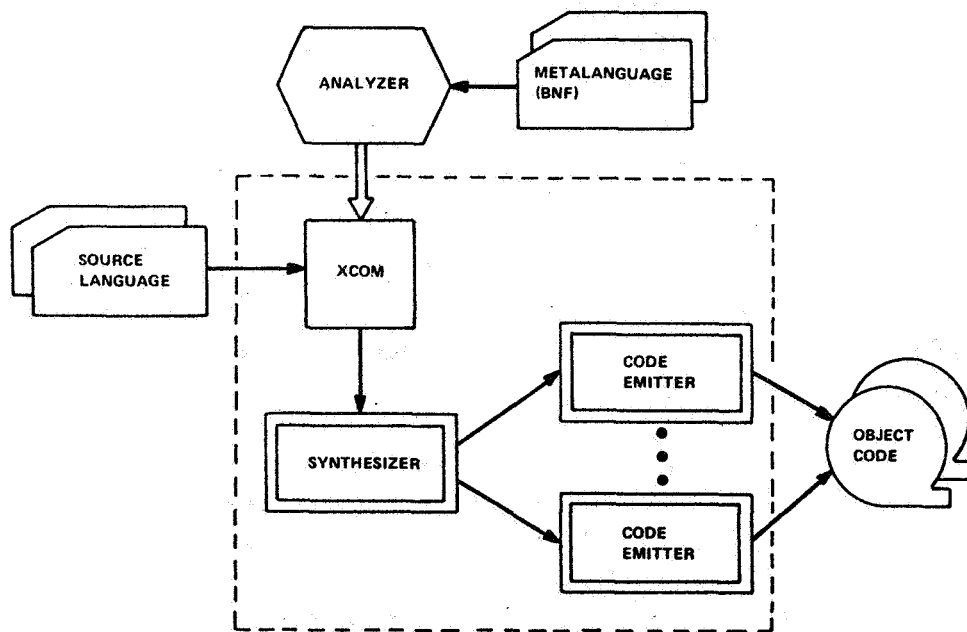


Figure 20. XPL/XCOM organization.

Leach and Golde [109] describe in detail the history of a project to transport the system from its original host, the IBM S/360, to the Honeywell Sigma 5. Kamnitzer [110] aided by Murry and Mohr performed a similar conversion to the Univac 1100 series. The MSP algorithm in the Sigma version was subsequently replaced by one utilizing LR(k) automata techniques. Bahler [111] reports on plans to utilize XCOM as a basis from which to build a multi-pass compiler. The first pass generates IL in the form of triples and a subsequent handwritten pass selects and generates code. Reference 112 describes

an effort utilizing a Sigma 9 version to add real arithmetic and formatted input/output to the XPL language. Due to the availability of ANALYZER, less than 10 percent of the total required effort was placed on syntax analysis. Approximately 90 percent of the work was dedicated to altering the semantic analyzer/code generator. Some modification to the lexical analyzer was also necessary.

## E. CWIC

Compiler for Writing and Implementing Compilers (CWIC) is a proprietary product of System Development Corporation [113,114]. It descended directly from the META systems previously described; in fact, several individuals participated in the development of both. The most significant difference is that CWIC is a multipass generator.

The CWIC generated parser is top-down, and the CWIC metalanguage [5] is extended accordingly. Since the most useful extensions are probably becoming quite familiar by now, major ones will be enumerated without further comment: (1) iteration specification, (2) symbol factoring, (3) backtrack marking, and (4) look-ahead specification. Metalanguage features that deal with lexical and semantic analysis will be discussed more fully.

The CWIC metalanguage permits three levels of productions; the first and second levels specify information for lexical analysis, and the third is the normal grammatical specification as used heretofore. The first level, called class definitions, is used to group the characters into subsets with similar attributes. This production type is distinguished from the others by the use of the metasymbol ":", rather than ":-=". Two examples are

```
<letter> : "A"|"B"|. . .|"Z"
```

```
<digit> : "0"|"1"|. . .|"9" .
```

Right side alternatives for a class definition must be either a single character or another class definition nonterminal as in

```
<hexdigit> : "A"|"B"|"C"|"D"|"E"|"F"|<digit> .
```

The second level, token-making equations, is distinguished by use of the meta-symbol "...", rather than ":",

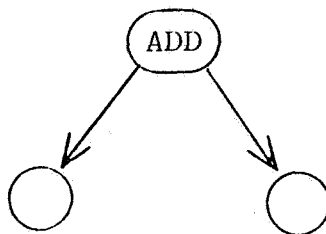
`<identifier> ... <letter> (<letter>|<digit>) {rep 0 5}`

Alternatives in token-making equations must be expressed in terms of terminal symbols and class definition nonterminals. This production level specifies sequences of characters to be treated as a unit by the syntax productions. The CWIC system generates a more efficient algorithm for their recognition than that used for the syntax productions. (It should be mentioned that numbers may be converted from character string to value via reference to a built-in function.)

Semantic extensions to the metalanguage carry forward the concept of operand referencing from an internal stack as in the META systems. However, the objective is now to produce IL (in tree-form) rather than assembly language. The extensions are part of the third level (syntax) productions. For example,

`<exp> : : = <term> "+" <term> {sem ADD #2}`

will cause a node to be created with the operation code "ADD". The top two stack elements (designated by #2) will be popped and attached to the node as shown in the following diagram.



A descriptor of the ADD node will then be returned to the stack. Nodes may have an arbitrary number of descendents, and provisions are available for creating nodes for which the number of descendents is not immediately known. An example in which descendent count uncertainty arises is the parsing of a subroutine call followed by a yet unscanned parameter list.

A typical operational configuration for a CWIC generated compiler is shown in Figure 21. The metalanguage translator produces machine language packages comprising the lexical and syntax analyzers. The syntax analyzer utilizes recursive descent to translate the source language to tree-form IL. At that point the compiler writer is provided a tailored language, GENERATOR [115], to complete the process. A few remarks about the language are in order.

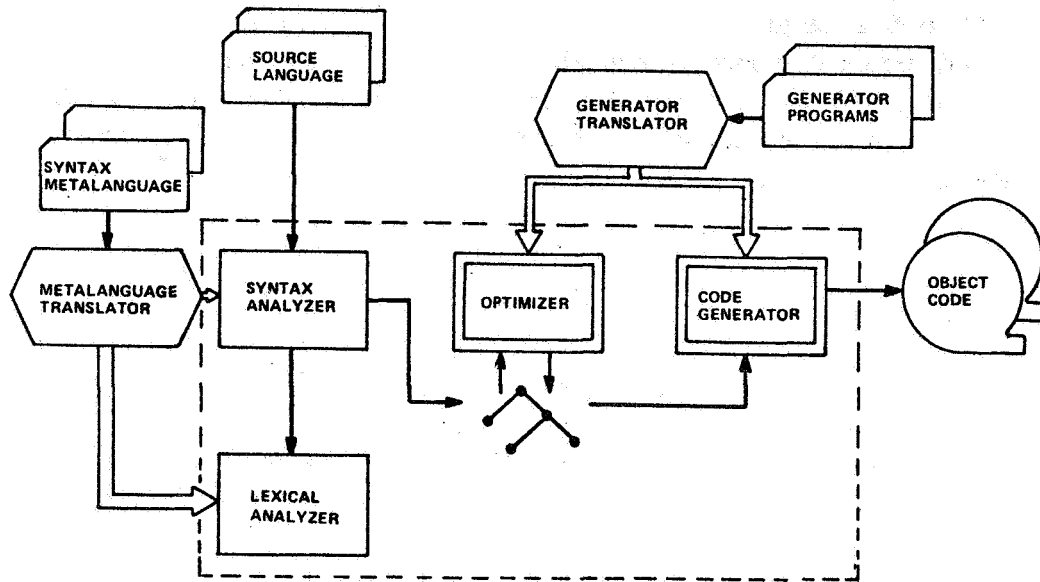


Figure 21. CWIC organization.

GENERATOR is a very high level language that is based on LISP and still bears some LISP characteristics; for example, a variable does not have a static type but may contain an integer at one time during execution and a list at another. Machine registers may be referenced symbolically, and GENERATOR provides some automatic facilities for their allocation and deallocation. The typical global strategy for a code generator using GENERATOR is to partition the object module into separate sections (data, instruction, COMMON, etc.) into which information is placed and "flushed" (output) periodically. Specific strategies include providing a single GENERATOR procedure for each operation. These procedures employ a powerful pattern-matching facility to determine the structure of the input parameters in order to select code or call other procedures for suboperations.



CWIC generated compilers, like others, require operating system support and communication. To accomplish this, a machine oriented language with Algol-like syntax called MOL [116] is provided. The support package for the CWIC system itself is in MOL, and the entire system was originally implemented on the IBM S/360 and has since been transported to the CDC 6000/7000 series.

Finally, there exists a CWIC-like compiler writing system hosted by the CDC 6000 series computers called SPLIT, SPL Implementation Tool [117]. SPLIT is designed to generate compilers for various subsets of Space Programming Language (SPL), a language tailored to flight applications. SPLIT incorporates the syntax metalanguage and GENERATOR, as well as a support package corresponding to MOL.

## F. AED

Automated Engineering Design (AED) is a proprietary product of Softech, Inc., containing compiler writing tools [88] which are constituents of a larger Algol-like language and system [118]. The present version, which utilizes automata and decision table methods, replaces a prior package [4] which used the operator precedence technique.

The metalanguage for the lexical/syntax phase is partitioned into the three levels previously discussed: character class definition, token-making equations, and syntax productions. Within the second level, token-making equations, one may instruct the system to ignore a string (useful in skipping comments), employ a simple form of repetition, or designate character class exclusion rather than class inclusion. Syntax productions are expressed in a manner practically equivalent to BNF, as is frequently the case when bottom-up parsing methods are employed. Discussion of the second metalanguage (used in code generation) is contained within the following operational description.

The various processing components available within the AED system permit flexibility in determining the resultant compiler configuration. Figure 22 illustrates a possible organization. A finite state machine (FSM) generator (similar to the one described by Johnson, et al. [119]) utilizes the first and second level productions to create the tables used by the lexical analyzer. The user may augment the lexical analyzer with procedures to perform such functions as string conversion and symbol table construction.

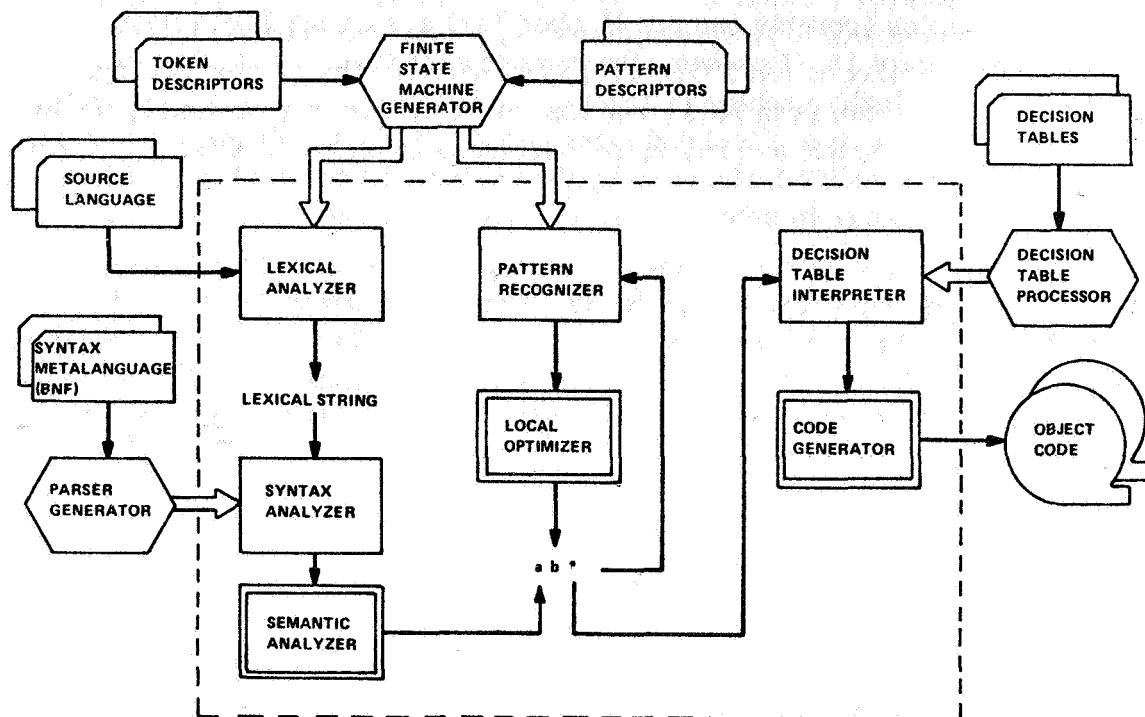


Figure 22. AED organization.

The syntax productions are employed to construct LR(k) automata tables for the parser. The technique employed is similar to the automata method previously discussed, but is more nearly equivalent to that of Aho and Johnson [46]. The specific subset of context-free grammars recognized is the LALR(1) grammars, a superset of the SLR(1) group. With the grammar, the user supplies a dispatch table of semantic procedures (written by the user) indexed by production number. It is the responsibility of the parser to call the appropriate semantic procedure whenever a production is applied to reduce the current sentential form. It is the responsibility of the semantic procedures to generate IL.

An optimization phase may optionally be incorporated into a generated compiler. The FSM generator is used to construct a pattern recognizer which accepts IL as input. Recognition of a pattern results in the evocation of a user-written procedure to perform local optimization over the elements of the pattern.

Code generation is accomplished partially by the decision table method. Decision tables (roughly one per IL operator) are encoded in a manner similar to that of Figure 15. However, the rows of each table are labeled with generic, rather than actual, machine instructions. The decision tables are preprocessed via a provided software tool and incorporated within an interpreter also provided. Selection of a generic instruction by the interpreter results in a call to a user-provided procedure to select the actual target machine instructions.

There are a few additional facets of the implementation that bear mentioning. More than 90 percent of the system is coded in the AED language, and the user-written components can easily be added if similarly coded. Since the system (counting prior versions) has been in existence for a number of years, many modular components for user-written sections are available "off the shelf."

## G. LIS

The Language Implementation System (LIS) is a proprietary product of Chi Corporation.<sup>3</sup> LIS and compilers produced by it have several significant characteristics:

1. Code generation is almost fully automated.
2. Global program optimization utilizes advanced graph theoretical concepts.
3. Operating system interfaces are effectively isolated.

LIS itself is written mostly in the systems implementation language CHILI [120] with some assembly language procedures. CHILI, in turn, is implemented on the UNIVAC 1100 series.

Figure 23 depicts the operational configuration. The syntax rules (in BNF) are used to generate LR(k) automata tables [of type LALR(1)] which are then incorporated in an LIS supplied syntax analyzer. The lexical analyzer which reads the source statements is written by the user, but work is underway to automate this stage and some aids exist already. The semantic procedures

---

3. Language Implementation System.- Chi Corporation, Cleveland, Ohio, Undated Report.

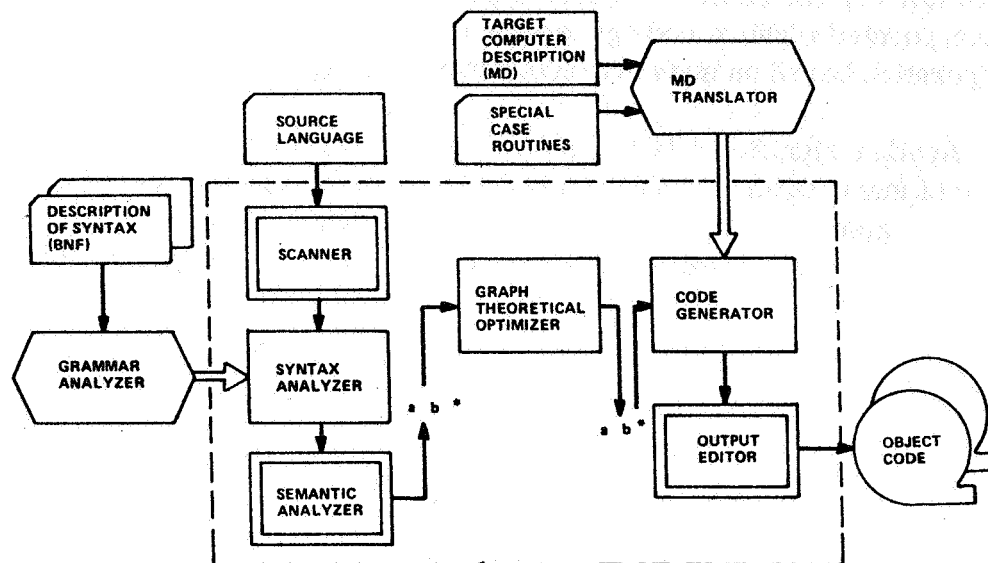


Figure 23. LIS organization.

must also be handwritten. One is called each time a syntax reduction is made and the current sentential form is provided by the syntax analyzer. It is the responsibility of the semantic procedures to check for semantic misuse and generate IL in the form of Polish notation.

The optimization phase is self-contained within the resultant compiler. It performs several passes over the IL but only the first examines the entire text. The first separates the code into basic blocks and partitions them into intervals; subsequent ones perform the optimization strategies. Among others, these strategies include subexpression factoring, invariant expression factoring, and dead definition elimination. If desired, the optimization phase can be omitted from a compiler by permitting the semantic procedures to format the IL in a manner compatible with the code generation phase.

A second metalanguage (machine description or MD) is employed in code generation. Encoded within the machine description are the target instructions plus the registers with characteristics of each. The MD translator groups the instructions into decision tables which are quite different from the one depicted in Figure 15. Instructions are grouped hierarchically by function. For example, all add instructions are placed in the same class then further

differentiated on the basis of operand type. Tables produced by the MD translator are incorporated within a code generator which selects the "best" instruction for each operation based on such characteristics as length and cycle time.

Another significant MD translator input is special case routines. One purpose of such routines is the generation of code for operations that require dedicated registers (e.g., subroutine linkage). The code generator automatically references these procedures when defined in lieu of an instruction description for an operation. The final constituent of code generation, output editing, is user written. It is the function of this component to select an output medium and produce a file which conforms to the conventions prescribed by the link editor of the target machine.

As mentioned at the beginning of this description, interfaces to the operating system are effectively isolated in resultant compilers. Two functions of this category, reading source programs and producing an object file, have been alluded to. The others are producing the source listing and intermediate (temporary) file storage access. All four of these are written by the user and retrofitted to the compiler produced.

## H. JOCIT

Jovial Compiler Implementation Tool (JOCIT) was developed by Computer Sciences Corporation for the Air Force [87]. A previously developed CWS, GENESIS, was used to develop part of the front end. JOCIT is unique in two respects: first, it is language specific but computer architecture reconfigurable; second, it employs the third major method of syntax definition, analytic grammars. Limiting the design to a single language achieved several objectives:

1. By reducing the amount of generalization, it is possible to produce compact and efficient compilers to be hosted by smaller computers.
2. By stabilizing the compiler's front end, the user can ensure that consistent language sets are implemented on different computers.
3. By restricting the scope to the JOVIAL language, it was feasible to allocate more resources to the development of debugging aids and diagnostics.

The concepts are applicable to other programming languages.

Like BNF, analytic grammars are replacement rules but emphasize string context to a greater extent. Analytic grammar rules are of the form

$$\psi \longrightarrow \phi \quad ,$$

where  $\psi$  is a substring whose length is at least as great as that of  $\phi$ . The meaning of the rule is: if  $\psi$  occurs in the sentential form, it is to be replaced by  $\phi$ . An (only slightly contrived) example is

$$\langle \text{id} \rangle * \langle \text{number} \rangle + \langle \text{id} \rangle \longrightarrow \langle \text{factor} \rangle + \langle \text{id} \rangle \quad .$$

The analytic algorithm does not reduce the sentential form in a strict left-to-right manner. The essential steps are:

1. Find the leftmost substring within the current sentential form that matches the left part of a rule.
2. Use the rule to reduce the sentential form.

Separate implementations of the algorithm are discussed in the previously cited reference by Dunbar [88] and an article by Hext and Roberts [8].

The JOCIT front end consists of syntax and semantic analysis. The syntax analyzer is the analytic system produced by the GENESIS CWS. Semantic analysis is performed by a set of procedures called Pragmatic Functions (Fig. 24), generally organized around the recognition of syntactic entities. However, certain actions transcend the context of specific source language phrases. Communication with the succeeding compilation phases is through IL (Polish notation) and various dictionaries such as that for symbols.

A graph theoretical optimizer is included in each generated compiler but in a manner that permits optional evocation for individual compilations. The specific flow analysis algorithm employed is linear nested region analysis (LNRA). LNRA [61] is intermediate in sophistication, falling between strongly connected region analysis and interval analysis. The general idea behind the method is to divide the flow graph into basic constructs — strongly connected

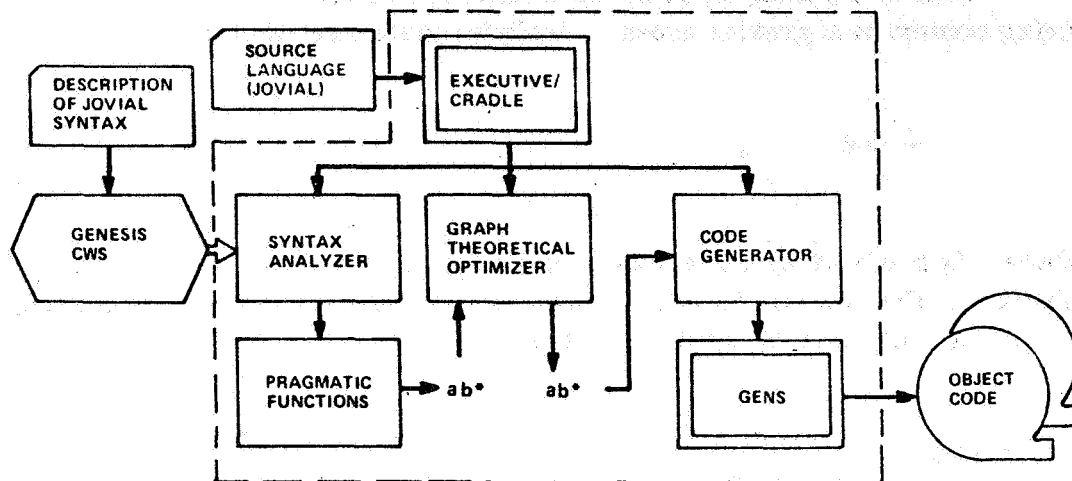


Figure 24. JOCIT organization.

region, conditional forward branch, if..then..else, and others — and to apply specialized techniques to each. Constructs not within the selected basic set are ignored except for basic block analysis. Optimization is performed in exactly two passes over the IL and flow graph. (This contrasts with interval analysis which makes as many passes as necessary to collapse the graph to a single node.) The first pass by the LNRA optimizer constructs the flow graph and collects the definition points for all variables. The second pass performs the optimization strategies which include constant propagation, subexpression factoring, operator strength reduction, and invariant expression factoring. Local optimization strategies are performed simultaneously.

The code generator is divided into a machine independent part provided by JOCIT and a machine dependent part, called GENS, provided by the compiler implementer. The machine independent part converts the IL to tree-form and sequences the nodes for code emission via the tree weighting algorithm described previously. It presents to GENS one operation at a time for which code is to be emitted. Typical strategies employed by GENS are selection via decision tables and conditional code emission. The latter capability is supported by JOCIT via an output editor which is not shown in Figure 24. The source document contains an excellent catalog of difficulties to be surmounted in generalizing code selection [87].

To bind the separate phases of the produced compiler, the implementer must write a control procedure called the "Executive/Cradle." The responsibilities of this procedure are phase sequencing and loading plus input/output for the compiler. The interfaces with the generated constituents are standardized to permit a straightforward implementation of the control and service routines.

## I. Other Systems

The fact that all recent systems discussed thus far emanated from industrial sources, with the exceptions of SIMCMP/STAGE2 and XPL/XCOM, is an interesting but unplanned phenomenon. Perhaps it reflects success on the part of previous researchers who were largely members of academe. Of course, some of the systems described were direct extensions of previous academic efforts and others received university support. However, to correct the impression that academic research in this area has ceased, the remaining systems selected for description all began as university research projects.

The Computer System Research Group at the University of Toronto was the first to develop an LR(k) automata parser generator [47] following the groundbreaking research by DeRemer. Their work led to a generator that accepted BNF input and produced LALR(1) syntax analysis tables. The design of this system was the basis for the current version of the parser generator in AED. Several optimizations were effected, some being more technical than theoretical, and comparisons of size and speed were performed between it, MSP, and (1,1) precedence.

A parser generator employing the production language approach has been developed at Brandeis University [121]. It is based largely on more fundamental research by Ichbiah and Morse [122] and has been used to develop processors for languages of syntax complexity approaching that of Algol. Like LANG-PAK, the Brandeis system is designed to operate in an interactive environment. It is constructed on a somewhat larger scale than LANG-PAK, however, consisting of five separate phases.

The phase that constructs the parser accepts BNF input, performs various ambiguity checks, and produces production language tables for incorporation into the standard syntax analyzer. The analyzer, of which there are both FORTRAN and Algol versions, calls a user-written semantic procedure,



providing it with an action code associated with the production being applied. The user, of course, provides the action code through the metalanguage. The second phase constructs the lexical analyzer, allowing the user to attach codes to identifiers, numbers, and other symbols at the time they are recognized. The third phase is the error routine generator. Its input consists of a set of triples, each one correlated to a production language step. Each triple consists of: (1) an error message, (2) description of the state in which the parsing stack is to be left, and (3) description of the next "safe" point at which the scan can continue. The fourth phase accepts BNF input and generates a sufficient number of typical sentences of the language to use each BNF production at least once. This phase is primarily a grammar consistency check and is usually executed first. The last phase generates a main program to bind the syntax analyzer, lexical analyzer, and error routine into a single system.

The Parser Generating System (PGS) is a (1,1) bounded-context analyzer developed at Purdue University [123]. Recall that during the discussion of syntax analysis it was stated that each step in a bottom-up parse presented two problems that required solution: (1) what part of the sentential form to reduce and (2) which production to apply. Bounded-context analysis is adept at solving the latter once the former is given. Resolution of what to reduce has led the authors of PGS to a rather unique approach.

Accepting BNF input, PGS introduces new nonterminals to transform the grammar into a normal form. This normal form is capable of representing any (1,1) bounded-context grammar and has production rules of only four types,

$$\begin{aligned}
 A :: &= BC \text{ (type 1)} \\
 A :: &= Bx \text{ (type 2)} \\
 A :: &= B \text{ (type 3)} \\
 A :: &= x \text{ (type 4)} \quad ,
 \end{aligned}$$

where A, B, and C are nonterminals and x is a terminal. This reduces the problem of what to reduce to exactly five cases at each step. That is,

1. No reduction can be made (i.e., ERROR)
2. The top two nonterminals on the parse stack

3. The top nonterminal on the parse stack and the next input string terminal
4. The top nonterminal on the parse stack
5. The next input string terminal.

Bounded-context analysis can now be used very effectively in determining which case applies, although a look-ahead scan may be necessary to ascertain right context.

In using PGS, one associates a semantics routine with each production in the original grammar. It is the belief of subsequent investigators at Purdue that for a fixed target machine, one set of generalized semantic primitives can be developed to service a broad range of languages [124]. Their approach is to construct a profile of the semantic characteristics of the range of languages to be supported, then design the minimum number of primitives necessary to handle all facets of the profile. Examples of semantic characteristics are explicit versus implicit variable type declaration, explicit versus implicit type conversion in expressions, fixed versus varying array subscript ranges, and the names of library routines.

## VII. THE USER'S PERSPECTIVE

The goal of the potential CWS user might be stated as "implement compilers in shorter time and at reduced cost." For both cost and time, it is significant that the authors of both JOCIT and LIS estimate a 50 percent reduction during compiler implementation. Most of the recently developed systems are placed directly in the hands of the users. However, each requires programming support, primarily that of systems oriented personnel. Table 1 illustrates the type and magnitude of auxiliary support necessary for several of the systems surveyed.

To clarify further the relative power of several of the parsing strategies with respect to the entire domain of context-free languages, consult Figure 25. Be warned that the Venn diagram shown is an approximation and can be interpreted too literally. Also, be aware that a continuum of top-down, backtracking

TABLE 1. CONFIGURATION SUMMARY

	SIMCMP/STAGE2	LANG-PAK	COGENT	XPL/XCOM	CWIC	AED	LIS	JOCIT
Lexical Analysis Technique	N/A	Optionally written by compiler implementer	Parameterized built-in function	Built-in functions	Generated token-making functions	Generated token-making functions using FSM techniques	Written by compiler implementer	Built-in functions
Syntax Analysis Technique	String matching	Top-down parse machine	Top-down recursive descent	Mixed strategy precedence and LR(k) automata (two versions)	Top-down recursive descent	LR(k) automata	LR(k) automata	Analytic grammar algorithm
Grammar Subclass Supported	MACRO	Subset of CF grammars	Subset of CF grammars with special facilities for LL(k) grammars	MSP and SLR(1)	Subset of CF grammars with special facilities for LL(k) grammars	LALR(1)	LALR(1)	JOVIAL only
Intermediate Language Form	N/A	N/A	N/A	N/A	Tree-form	Either polish notation or triples are compatible with framework	Polish notation	Polish notation
Local Optimization	N/A	N/A	Pattern matching during syntax analysis	N/A	Possible using GENERATOR language	Separate pass using FSM pattern-matching techniques	Possible in user-written semantic analyzer	Combined with global optimization
Global Optimization	N/A	N/A	N/A	N/A	Possible using GENERATOR language	N/A	Interval analysis	Linear nested region analysis
Code Generation	N/A	Combined with semantic analysis and written by compiler implementer	Specified via extensions to syntax meta-language and internal stack referencing	Combined with semantic analysis and written by compiler implementer	Written by compiler implementer using GENERATOR language tailored for purpose	Decision table construction aids and interpreter available	Decision tables augmented with user-written procedures	Code sequencing algorithm provided; code selection algorithm is user written
Availability	Any FORTRAN machine	Any FORTRAN machine	Any FORTRAN machine with special versions for PDP-10 and PDP-11	At least: IBM 360/370, Univac 1100 series, Honeywell sigma series	IBM 360/370	At least: IBM 360/370, Univac 1100 series, CDC 6000 series	Univac 1100 series	Univac 1100 series, Honeywell 600

SAI-0727

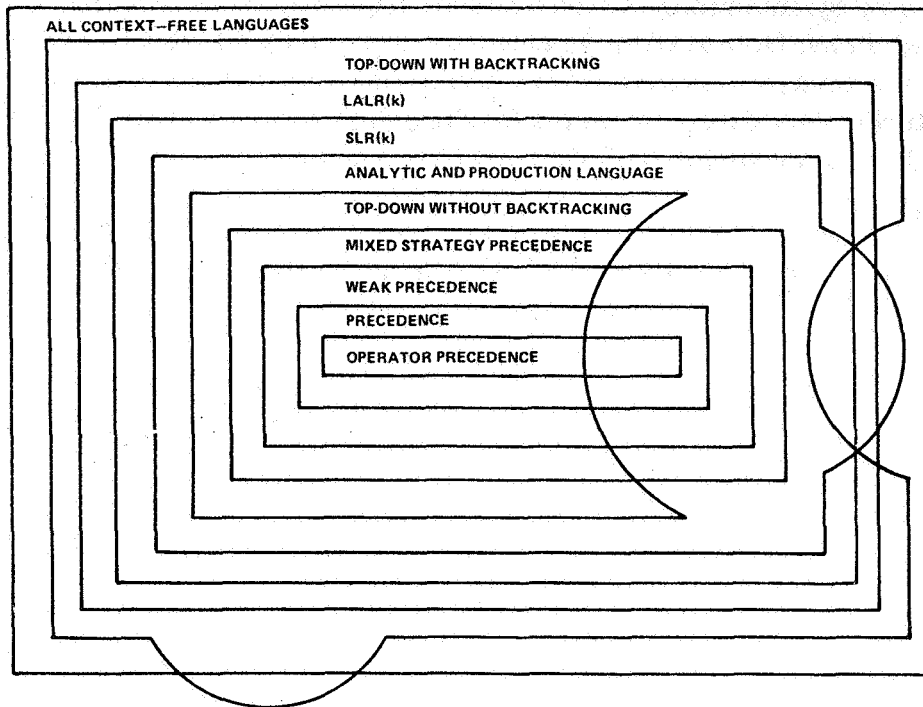


Figure 25. Comparison of parsing strategies.

algorithms exist, extending from those having a domain only slightly larger than the  $LL(k)$  subclass to those for which the domain is almost all context-free languages [30,125]. In fact, the top-down strategy has been applied to languages outside the context-free class [126].

Apart from generating production compilers, the use of compiler writing aids can be an invaluable education asset. The volume of information pertaining to grammatical subclasses and parsing strategies is indeed impressive. However, a frequent objective of a university introductory course in compiler construction is to have the student fully implement a minilanguage during a single semester. Providing sufficient information on the nuances of any one classical parsing strategy and requiring it to be implemented before continuing can consume the better part of a term. Further, this must be done to the detriment of other strategies and, more regrettably, other compilation phases. Given at least a parser generator, a term project can be well underway shortly after introducing the rudiments of grammar and syntax analysis. This more than anything else accounts for the broad distribution of XPL/XCOM.

The utility of compiler writing aids does not end with implementing compilers. Any symbol manipulation problem requiring the production of one well formatted string from another is amenable to the methods discussed here. Some of the areas in which these tools have been employed are text editing, restructuring data files, converting engineering units, and translating the names of organic compounds into two-dimensional formats.

## VIII. DEPARTURE POINTS FOR FUTURE DEVELOPMENTS

The peak of the creativity explosion in the area of syntax definition and analysis has probably passed. The parsing strategies that have evolved are sufficiently general to encompass all of the popular procedural languages in use today, and these strategies have firm theoretical foundations. More importantly, it can be demonstrated that a language for which the grammar can be expressed in precedence form or recognized via a deterministic automation is unambiguous. So, barring the introduction of much higher level languages, research here is far less critical today. Similar, though more recent, strides have been made toward establishing a mathematical foundation for program optimization. However, two major problems that have plagued CWS implementors continue to exist: generalized approaches to semantic analysis and translation of data structures. The two are not entirely disjoint.

With respect to data structures, most general purpose computers have several built-in, or concrete, data types: vector (or array), a subrange of integers, a subset of real numbers, and an extended subset of real numbers (i.e., extended precision). To this concrete set, even the most elementary high order languages add several abstract data types such as boolean (created by mapping two symbols onto two integer elements) and character (created by mapping a small number of symbols onto a subrange of integers). Many languages, most notably PASCAL [127] go much further and allow the dynamic creation of new abstract data types which may be arbitrarily complex. Composite variables called records may themselves be elements of a fixed length structure called an array or a variable length structure called a file. Symbols may be mapped onto a subrange of integers and henceforward used as scalars. To write a specific compiler to, first, capture all the data element inter-relationships in an internal symbol table and, second, to allocate storage and generate correct access code is not difficult. To generalize the mechanism in a manner that does not preclude other abstractions is difficult. Other

abstractions include the conventions of the language BLISS [128] which allow direct reference to the hardware registers and differentiate between the name of a variable and the contents of the location denoted by the variable. The dynamically defined data types permitted in LISP have been noted earlier.

With respect to semantic analysis, it may help to recall the example of the concurrent assignment statement which demonstrated how syntax and semantics were sometimes interlocked. However, the point of the example was that one manner of syntax definition facilitated semantic analysis to a greater degree than did another; it made no mention of how semantic analysis was to be performed. Compiler writing aids contribute most to the phases of translation that are concerned with the transformation of one string into another based primarily on form and structure. It is when the output string produced is also dependent on the meaning (semantics) of the input string that the theory begins to fail. It is significant that it is this area, interpretation of the parse tree, that three of the four multipass generators described — namely, CWIC, AED, and LIS — leave to the compiler implementer. The fourth, JOCIT, avoids the problem only by limiting itself to the JOVIAL language. Translation of intermediate language to target machine code suffers from the same theory gap.

Measures that may be taken to solve these problems can be divided into near-term and long-term approaches. For the near-term approach, allow the generic diagram of the "ideal CWS" of Figure 3 to be replaced by the "generalized CWS" depicted in Figure 26. In Figure 26, an arbitrary source language is first translated to a standard source language. The standard source language must exhibit all the semantic, but not necessarily syntactic, characteristics of the domain of languages for which the CWS is intended. If there are several conflicting semantic characteristics, then only one may be chosen to the diminishment of the source language domain. Configuring the standard source language would be an exercise in enumerative logic (not the soundest mathematical foundation) and the first stage of translation might be patterned after the extensions to Purdue's PGS. The second stage of translation would be from the standard source to a standard object language. The standard object language would be only slightly more complex in structure and semantics than an assembly language. Thus, translation to the machine languages of the selected range of targets could utilize methods similar to those of the STAGE2/JANUS system.

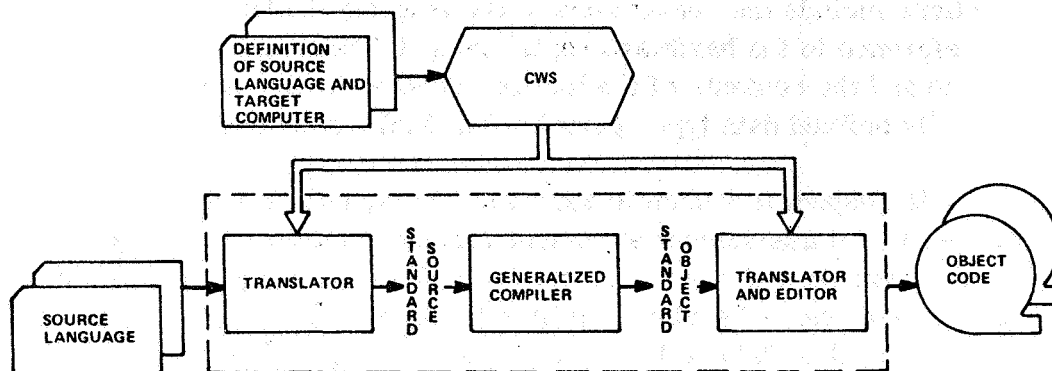


Figure 26. Generalized CWS.

Long-term solutions must await the development of more powerful theories. Currently, the outlook is much brighter in the area of data structures than in semantic interpretation. The reason for this is the current emphasis on data-base systems and recognition of the need to insulate users of such systems from the intricacies of internal data representations. That is, it is desirable to inform the user of all logical data element interrelationships but undesirable to require an understanding of how these interrelationships are physically realized. Analogously, it would be desirable for a CWS user to enumerate the data attributes for each language — type, name, access method, semantic dependencies — but undesirable to require the details of internal (compile-time) or external (execution-time) representation. One promising area of research that could conceivably produce side effects beneficial to CWS technology is relational data models [129,130].

In a relational data model, there is assumed to exist a finite number of not necessarily distinct sets,  $S_1, S_2, \dots, S_n$ . A relation is defined to be a list of ordered  $n$ -tuples where each  $n$ -tuple is unique and contains one element from each set.  $N$ -tuple lists may contain much redundant information, but they refer only to the logical structure and not to the internal data representation. By allowing the user to define new relations and utilize several primitive set operations, this simple conceptual framework becomes a powerful data description facility. It would be presumptuous to claim this is a solution to the data structure problem in the compilation process for, indeed, it is not. However, it might be a useful illustration to couch the compilation problem in terms of a relational model. Assume that a CWS permitted the definition of the sets,  $S_i$ ,

and the domain of each. Further, via semantic extensions to the metalanguage, assume the user provides interpretation rules to direct the construction of new tuples and test existing ones to select among candidate output strings. If these two tasks could be isolated from the internal and external tuple representation, the problem would be solved.

As stated, however, generalizing semantic interpretation is far more difficult. Again, the ongoing research is not directed specifically toward CWS technology. Two diverse systems will be described here to illustrate the directions in which these studies are proceeding. The first of these is Vienna Definition Language (VDL). VDL [131,132] was originally developed as a formal method for describing the semantics of PL/I. In VDL, one divides a program into a control (instruction) component and an environment (data) component. The two components are represented within separate tree structures. The leaves (called elementary objects) of the environment tree are data elements, and the leaves of the control tree are instructions. Two primitive operations are defined over the structures; select can access any elementary object or subtree, and assign can add or delete subtrees or change the value of an elementary object. Using these two primitives, instructions of two basic types are defined: (1) macroinstructions that, when executed, replace themselves with a subtree whose leaves are additional instructions; and (2) value-returning instructions that pass a computed value to the next higher control tree node with possible side effects on the environment tree.

The rather simple VDL concepts form a system of considerable subtlety and power that has been employed to describe the semantics of languages and algorithms [133]. However, it has two drawbacks limiting the applicability to compiler writing. First, the primitive operations are too basic to enable representation of all possible substrings within a reasonable space. A subsequent version of the system, BASIS/I [134], loosens the restrictions but at the cost of complete formality. The second and more theoretically significant drawback is that VDL is an interpretive system. This means that semantics are defined in terms of actions performed upon a given case. The semantic problem in compilers is quite different. It is to find a string in one language, the target, with semantics equivalent to a string in another language, the source.

Interpreting one string by transforming it into a second string containing only well understood terms that are defined via axioms is sometimes called the axiomatic approach. The primary difference between an axiomatic and the VDL



approach is that in the axiomatic approach the effect of the current state of the environment (values of variables, etc.) is treated more abstractly. One example of research in this area is that of Scott [135,136]. Scott provides a set of primitive functions (read, write, and update, among others) and argument domains for each. The domain sets consist of locations, storable values, storage states, and truth values. (Storable values and storage states are further divided into subdomains.) Once the mappings between elements of the domains by the primitive functions and several binary operations are understood, it is possible to construct formulas representing the semantics of a language construct.

It should be noted that an axiomatic system strictly for the description of machine instruction sets has been in use for several years [137,138].

## IX. CONCLUSION

This report has partitioned the compilation problem into five phases: lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. Representative techniques applicable to each phase were described. The development of compiler writing aids was traced from the earliest efforts to the present. Particular emphasis was placed on the more recent tools in deference to the monumental survey previously published by Feldman and Gries [2]. The final sections concentrated on pending technological developments which could impact the manner in which compilers are developed. Perhaps an appropriate way to end this survey is to ask: If current research in data abstraction and semantic description concludes in the most fortuitous manner, what will have been accomplished? The answer is not a compiler-compiler.

Assume that it is possible to state formally the semantics of both source and target languages. It would still be necessary to transform one to the other by finding two strings having equivalent semantics. This is not a simple exercise. The situation would be analogous to stating a conjecture in mathematics without being capable of proving it. However, such conjectures become well defined problems and well defined problems often attract solutions.

## REFERENCES

1. Irons, E. T.: A Syntax Directed Compiler for ALGOL 60. Comm. ACM, vol. 9, no. 1, January 1961, pp. 51-55.
2. Feldman, J. and Gries, D.: Translator Writing Systems. Comm. ACM, vol. 11, no. 2, February 1968, pp. 77-113.
3. McKeeman, W. M., Horning, J. J., Wortman, D. B.: A Compiler Generator. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1970.
4. Programmer's Guide for Building Language Processors with the AEDJR System. Softech, Inc., Softech-R-1, March 1970.
5. Book, E., Schorre, D. V., Sherman, S. J.: CWIC User's Guide the Syntax Language. System Development Corporation, TM-L-4185/002/01, June 19, 1970.
6. Naur, P. (ed.): Revised Report on the Algorithmic Language ALGOL 60. Comm. ACM, vol. 6, no. 1, January 1963, pp. 1-17.
7. Tanenbaum, A. S.: A General-Purpose Macro Processor as a Poor Man's Compiler-Compiler. IEEE Transactions on Software Engineering, vol. SE-2, no. 2, June 1976, pp. 121-125.
8. Hext, J. B. and Roberts, P. S.: Syntax Analysis by Domolki's Algorithm. Computer J., vol. 13, no. 3, March 1970, pp. 263-271.
9. Lewis, P. M. II and Stearns, R. E.: Syntax-Directed Transductions. J. ACM, vol. 15, no. 3, September 1968, pp. 464-488.
10. Greibach, S.: A Normal-Form Theorem for Context-Free Phrase Structure Grammars. J. ACM, vol. 12, no. 1, January 1965, pp. 42-52.
11. Korenjak, A. J. and Hopcroft, J. E.: Simple Deterministic Languages. Seventh Annual Symposium on Switching and Automata Theory, vol. 7, 1966, pp. 36-46.

## REFERENCES (Continued)

12. Rosenkrantz, D. J. and Stearns, R. E.: Properties of Deterministic Top-Down Grammars. *Information and Control*, vol. 17, 1970, pp. 226-256.
13. Writh, N. and Weber, H.: EULER — A Generalization of ALGOL and its Formal Definition, Part I. *Comm. ACM*, vol. 9, no. 1, January 1966, pp. 13-25.
14. Floyd, R. W.: A Descriptive Language for Symbol Manipulation. *J. ACM*, vol. 8, no. 4, October 1961, pp. 579-584.
15. DeRemer, F. L.: Simple LR(k) Grammars. *Comm. ACM*, vol. 14, no. 7, July 1971, pp. 453-460.
16. Aho, A. V. and Ullman, J. D.: The Theory of Parsing, Translation and Compiling, Vol. I. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1972.
17. Ullman, J. D.: Application of Language Theory to Compiler Design. *Proc. AFIPS Spring Jt. Computer Conf.*, vol. 40, 1972, pp. 235-242.
18. Gries, D.: Compiler Construction for Digital Computers. John Wiley and Sons, New York, 1971.
19. Lewis, P. M. II, Rosenkrantz, D. J., and Stearns, R. E.: Compiler Design Theory. Addison-Wesley, Reading, Mass., 1976.
20. Zimmer, R.: Weak Precedence. *Proc. Intern. Computing Symposium 1970*, 1970, pp. 576-587.
21. Aho, A. V., Denning, P. J., and Ullman, J. D.: Weak and Mixed Strategy Parsing. *J. ACM*, vol. 19, no. 2, April 1972, pp. 225-243.
22. Floyd, R. W.: Syntactic Analysis and Operator Precedence. *J. ACM*, vol. 10, no. 3, July 1963, pp. 316-333.
23. Floyd, R. W.: Bounded Context Syntactic Analysis. *Comm. ACM*, vol. 7, no. 2, February 1964, pp. 62-67.

## REFERENCES (Continued)

24. Knuth, D. E.: On the Translation of Languages from Left to Right. *Information and Control*, vol. 8, no. 6, 1965, pp. 607-639.
25. Aho, A. V. and Ullman, J. D.: A Technique for Speeding Up LR(k) Parsers. *SIAM J. Computing*, vol. 1, no. 2, June 1973, pp. 106-127.
26. Hopgood, F. R. A.: *Compiling Techniques*. American Elsevier, Inc., New York, 1969.
27. Aho, A. V. and Ullman, J. D.: *The Theory of Parsing, Translation and Compiling, Vol. II*. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1973.
28. Floyd, R. W.: The Syntax of Programming Languages-A Survey. *IEEE Trans. on Electronic Computers*, vol. EC-13, no. 4, August 1964, pp. 346-353.
29. Unger, S. H.: A Global Parser for Context-Free Phrase Structure Grammars. *Comm. ACM*, vol. 11, no. 4, April 1968, pp. 240-247.
30. Birman, A. and Ullman, J. D.: Parsing Algorithms with Backtrack. *Conf. Record of 11th Annual Symposium on Switching and Automata Theory*, 1970, pp. 153-174.
31. Foster, J. M.: *Automatic Syntactic Analysis*. American Elsevier, Inc., New York, 1970.
32. Kurki-Suonio, R.: Notes on Top-Down Languages. *BIT*, vol. 9, 1969, pp. 225-238.
33. Cohen, Doren J. and Gotlieb, C. C.: A List Structure Form of Grammars for Syntactic Analysis. *Computing Surveys*, vol. 2, no. 1, March 1970, pp. 65-82.
34. Kanner, H., Kosinski, P., and Robinson, C. L.: The Structure of Yet Another ALGOL Compiler. *Comm. ACM*, vol. 8, no. 7, July 1965, pp. 427-438.

## REFERENCES (Continued)

35. McClure, R. M.: An Appraisal of Compiler Technology. Proc. AFIPS Spring Jt. Computer Conf., vol. 40, 1972, pp. 1-9.
36. Knuth, D. E.: Top-Down Syntax Analysis. Acta Information, vol. 1, 1971, pp. 79-110.
37. Bell, J. R.: A New Method for Determining Linear Precedence Grammars. Comm. ACM, vol. 12, no. 10, October 1969, pp. 567-569.
38. Aho, A. V. and Ullman, J. D.: Linear Precedence Functions for Weak Precedence Grammars. Intern. J. Computer Math., vol. 3, no. 2, September 1972, pp. 149-155.
39. Martin, D. F.: A Boolean Matrix Method for Computation of Precedence Functions. Comm. ACM, vol. 15, no. 6, June 1972, pp. 448-454.
40. Evans, A., Jr.: An ALGOL 60 Compiler. Annual Review in Automatic Programming, vol. 4, 1964, pp. 87-124.
41. Hopcroft, J. E. and Ullman, J. D.: Formal Languages and Their Relation to Automata. Addison-Wesley, Reading, Mass., 1969.
42. DeRemer, Franklin Lewis: Practical Translators for LR(k) Languages. PhD. Thesis, National Technical Information Service, AD 699 501, October 24, 1969.
43. Early, Jay: An Efficient Context-Free Parsing Algorithm. Comm. ACM, vol. 13, no. 2, February 1970, pp. 94-102.
44. Vere, S.: Translation Equations. Comm. ACM, vol. 13, no. 2, February 1970, pp. 83-89.
45. Schkolnick, M.: Labelled Precedence Parsing. ACM Symposium on Principles of Programming Languages, October 1973, pp. 33-40.
46. Aho, A. V. and Johnson, S. C.: LR Parsing. Computing Surveys, vol. 6, no. 2, June 1974, pp. 99-124.

## REFERENCES (Continued)

47. Lalonde, W. R., Lee, E. S., and Horning, J. J.: An LALR(k) Parser Generator. Proc. IFIP Congress 71, 1971, pp. 513-518.
48. Anderson, T., Eve, J. and Horning, J. J.: Efficient LR(1) Parsers. Acta Information, vol. 2, 1973, pp. 12-39.
49. Shapiro, B.: SLR(1) Parser Generator. National Technical Information Service, PB-249 127/2WC, February 4, 1976.
50. Mickunas, M. D.: On the Complete Covering Problem for LR(k) Grammars. J. ACM, vol. 23, no. 1, January 1976, pp. 17-30.
51. Horning, J. J.: Empirical Comparison of LR(k) and Precedence Parsers. University of Toronto, August 1970.
52. Jolliat, M. L.: On the Reduced Matrix Representation of LR(k) Parser Tables. Ph. D. Thesis, University of Toronto, 1973.
53. Jolliat, M. L.: A Simple Technique for Partial Elimination of Unit Productions from LR(k) Parsers. IEEE Trans. on Computers, vol. C-25, no. 7, July 1976, pp. 763-764.
54. Feldman, J. A.: A Formal Semantics for Computer Languages and Its Application In a Compiler-Compiler. Comm. ACM, vol. 9, no. 1, January 1966, pp. 3-9.
55. Griffiths, T. V. and Petrick, S. R.: On the Relative Efficiencies of Context-Free Grammar Recognizers. Comm. ACM, vol. 8, no. 5, May 1965, pp. 289-300.
56. Schorr, H.: Compiler Writing Techniques and Problems. Proc. NATO Conference on Software Engineering, 1969, pp. 114-122.
57. Dijkstra, E. W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N. J., 1976.

## REFERENCES (Continued)

58. Liu, C. L., Chang, G. D., and Marks, R. E.: The Design and Implementation of a Table Driven Compiler System. Proc. AFIPS Spring Jt. Computer Conf., vol. 30, 1976, pp. 691-697.
59. Cheatham, T. E., Jr. and Sattley, K.: Syntax-Directed Compiling. Proc. AFIPS Spring Jt. Computer Conf., vol. 25, 1964, pp. 31-57.
60. Pratt, T. W. and Lindsay, R. K.: A Processor-Building System For Experimental Programming Languages. Proc AFIPS Fall Jt. Computer Conf., 1966, pp. 613-621.
61. Cocke, J. and Schwartz, J. T.: Programming Languages and Their Compilers, Preliminary Notes. Second Revised Edition, Courant Institute of Mathematical Science, New York University, April 1970.
62. Brooker, R. A., MacCallum, I. R., Morris, D., and Rohl, J. S.: The Compiler Compiler. Annual Review in Automatic Programming, vol. 3, 1963, pp. 229-275.
63. Warshall, S. and Shapiro, R. M.: A General-Purpose Table-Driven Compiler. Proc. AFIPS Spring Jt. Computer Conf., vol. 25, 1964, pp. 59-65.
64. Lowry, E. and Medlock, C.: Object Code Optimization. Comm. ACM, vol. 12, no. 1, January 1969, pp. 12-22.
65. McKeeman, W. M.: Peephole Optimization. Comm. ACM, vol. 8, no. 7, July 1965, pp. 443-444.
66. Bagwell, J. T., Jr.: Local Optimizations. SIGPLAN Notices, vol. 5, no. 7, July 1970, pp. 52-66.
67. Allen, F. E.: Program Optimization. In Annual Review in Automatic Programming, vol. 5, Pergamon Press, New York, 1969, pp. 165-212.
68. Allen, F. E. and Cocke, J.: A Catalogue of Optimizing Transformations. Design and Optimization of Compilers, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs, N. J., 1972, pp. 1-30.

## REFERENCES (Continued)

69. Gear, C. W.: High Speed Compilation of Efficient Object Code. Comm. ACM, vol. 8, no. 8, August 1965, pp. 483-488.
70. Geschke, M. C.: Global Program Optimizations. Ph.D. Thesis, National Technical Information Service, AD 762 621, October 1972.
71. Schaefer, M.: A Mathematical Theory of Global Program Optimization. Prentice-Hall Inc., Englewood Cliffs, N. J., 1973.
72. Busam, V. A. and Englund, D. E.: Optimization of Expressions in FORTRAN. Comm. ACM, vol. 12, no. 12, December 1969, pp. 666-674.
73. Allen, F. E.: Control Flow Analysis. SIGPLAN Notices, vol. 5, no. 7, July 1970, pp. 1-19.
74. Allen, F. E. and Cocke, J.: A Program Data End Flow Analysis Procedure. Comm. ACM, vol. 19, no. 3, March 1976, pp. 137-147.
75. Tarjan, R.: Testing Flow Graph Reducibility. Proc. 5th Annual ACM Symposium on Theory of Computing, May 1973, pp. 96-107.
76. Knuth, D. E.: An Empirical Study of FORTRAN Programs. In Software — Practice and Experience, vol. 1, no. 1, January-March 1971, pp. 105-133.
77. Earnest, C.: Some Topics in Code Optimization. J. ACM, vol. 21, no. 1, January 1974, pp. 76-102.
78. Kildall, G. A.: A Unified Approach to Global Program Optimization. ACM Symposium on Principles of Programming Languages, October 1973, pp. 194-206.
79. Miller, E. F., Jr., Bardens, J. A., Benson, J. P., Melton, R. A., Urban, R. J. and Wisehart, W. R.: Structurally Based Automatic Testing. Proc. EASCON'74, October 1974, pp. 134-139.



## REFERENCES (Continued)

80. Paige, M. R.: Program Graphs, An Algebra, and Their Implications for Programming. IEEE Trans. Software Engineering, vol. SE-1, no. 3, September 1975, pp. 286-291.
81. Graham, S. L. and Wegman, M.: A Fast and Usually Linear Algorithm for Global Flow Analysis. J. ACM, vol. 23, no. 1, January 1976, pp. 172-202.
82. Hecht, M. S. and Ullman, J. D.: Analysis of a Simple Algorithm for Global Data Flow Problems. Conf. Record of ACM Symposium on Principles of Programming Languages, October 1973, pp. 207-217.
83. Kam, John B. and Ullman, J. D.: Global Data Flow Analysis and Interactive Algorithms. J. ACM, vol. 23, no. 1, January 1976, pp. 158-171.
84. Wulf, W. A., Johnsson, R. K., Weinstock, C. B., and Hobbs, S. O.: The Design of an Optimizing Compiler. National Technical Information Service, AD-773 838, December 1973.
85. Sethi, R. and Ullman, J. D.: The Generation of Optimal Code for Arithmetic Expressions. J. ACM, vol. 17, no. 4, October 1970, pp. 715-728.
86. McCarthy, J., Abrahams, P. W., Edwards, J. J., Hart, T. P., and Levin, M. I.: LISP 1.5 Programmer's Manual. M.I.T. Press, Cambridge, Mass., 1962.
87. Dunbar, T. L.: JOCIT Jovial Compiler Implementation Tool. National Technical Information Service, AD/A-005 307, January 1975.
88. The Compiler Framework. Softech, Inc., vol. 7089, April 1976.
89. Brooker, R. A., Morris, D., and Rohl, J. S.: Compiler Compiler Facilities in Atlas Autocode. Computer J., vol. 9, no. 2, February 1967, pp. 350-352.

## REFERENCES (Continued)

90. Brooker, R. A., Morris, D., and Rohl, J. S.: Experience with the Compiler Compiler. *Computer J.*, vol. 9, no. 2, February 1967, pp. 345-352.
91. Rosen, Saul: A Compiler-Building System Developed by Brooker and Morris. *Comm. ACM*, vol. 7, no. 7, July 1964, pp. 403-414.
92. Dean, A. L., Jr.: Some Results in the Area of Syntax Directed Compilers. *Computer Associates, Inc.*, CA-64-5-R, December 1964.
93. Cheatham, T. E., Jr.: The TGS-II Translator-Generator System. *Proc. of the IFIP Congress*, 1965, pp. 592-593.
94. Iturriaga, R., Standish, T. A., Krutar, R. A., and Earley, J. C.: Techniques and Advantages of Using the Formal Compiler Writing System FSL to Implement a Formula Algol Compiler. *Proc. AFIPS Spring Jt. Computer Conf.*, vol. 28, 1966, pp. 241-252.
95. Schorre, D. V.: META II A Syntax-Oriented Compiler Writing Language. *Proc. 19th National ACM Conf.*, vol. 19, 1964, p. D1.3-1.
96. Schneider, F. W. and Johnson, G. D.: META-3 A Syntax-Directed Compiler Writing Compiler to Generate Efficient Code. *Proc. 19th National ACM Conf.*, vol. 19, 1964, pp. D1.5-1-D1.5-8.
97. Oppenheim, D. K.: META5: A Tool to Manipulate Strings of Data. *Proc. 21st National ACM Conf.*, vol. 21, 1966, pp. 465-468.
98. Garwick, J. V.: GARGOYLE, A Language for Compiler Writing. *Comm. ACM*, vol. 7, no. 1, January 1964, pp. 16-20.
99. Reynolds, J. C.: An Introduction to the COGENT Programming System. *Proc. 20th National ACM Conf.*, vol. 20, August 1965, pp. 422-436.
100. McClure, R. M.: TMG-A Syntax Directed Compiler. *Proc. 20th National ACM Conf.*, vol. 20, August 1965, pp. 262-274.

## REFERENCES (Continued)

101. O'Neil, J. T., Jr.: META PI — An On-Line Interactive Compiler-Compiler. Proc. AFIPS Fall Jt. Computer Conf., vol. 33, December 1968, pp. 201-218.
102. Orgass, R. J. and Waite, W. M.: A Base for a Mobile Programming System. Comm. ACM, vol. 12, no. 9, September 1969, pp. 507-510.
103. Waite, W. M.: Building a Mobile Programming System. Computer J., vol. 13, no. 2, February 1970, pp. 28-31.
104. Waite, W. M.: The Mobile Programming System: STAGE2. Comm. ACM, vol. 13, no. 7, July 1970, pp. 415-421.
105. Coleman, S. S., Poole, P. C., and Waite, W. M.: The Mobile Programming System, JANUS. Software-Practice and Experience, vol. 4, no. 1, January-March 1974, pp. 5-23.
106. Heindel, L. E. and Roberto, J. T.: LANG-PAK-An Interactive Language Design System, American Elsevier, New York, 1975.
107. COGENT Compiler Generator User's Guide. Virtual Systems, Inc. 1976.
108. Kole, R. E., Helmer, P. H. and Holtz, R. L.: HAL/S-360 User's Manual. National Technical Information Service, N74-25728, February 18, 1974.
109. Leach, G. and Golde, H.: Bootstrapping XPL to an XDS Sigma 5 Computer. Software-Practice and Experience, vol. 3, no. 3, July-September 1973, pp. 235-244.
110. Kamnitzer, S. H.: Bootstrapping XPL from IBM/360 to Univac 1100. SIGPLAN Notices, vol. 10, no. 5, May 1975, pp. 14-20.
111. Bahler, R. C.: Steps Toward a Compiler for BLISS-360. National Technical Information Service, AD 747 530, June 1972.

## REFERENCES (Continued)

112. Storm, Mark W. and Polk, J. A.: Usage of an XPL Based Compiler Generator System. Proc. 14th Annual Southeast Regional ACM Conf., 1976, pp. 19-26.
113. Book, E., Schorre, D. V., and Sherman, S. J.: CWIC User's Guide: General Description. System Development Corporation, TM-L-4185/001/00, May 14, 1969.
114. Book, E., Sherman, S. J., and Schorre, D. V.: CWIC/360 User's Guide. System Development Corporation, TM-4185/000/03, June 30, 1971.
115. Book, E., Sherman, S. J., and Schorre, D. V.: CWIC User's Guide: The Generator Language. System Development Corporation, TM-L-4185/003/01, March 9, 1971.
116. Book, E., Schorre, D. V., and Sherman, S. J.: A User's Manual for MOL-360. System Development Corporation, TM-3086/003/01, April 22, 1969.
117. SPLIT User's Manual. System Development Corporation, TM4765/609/01, June 5, 1972.
118. An Introduction to the Features and Uses of AED. Softech, Inc., vol. 6080, January 1975.
119. Johnson, W. L., Porter, J. H., Ackley, S. I., and Ross, D. T.: Automatic Generation of Efficient Lexical Processors Using Finite State Techniques. Comm. ACM, vol. 11, no. 12, December 1968, pp. 805-813.
120. Lynch, W. C., Langner, J. W., and Schwartz, M. S.: Reliability Experience with Chi/OS. Proc. Intern. Conf. on Reliable Software, April 1975, pp. 252-259.
121. Cohen, Jacques: Experience With A Conversational Parser Generating System. Software-Practice and Experience, vol. 5, no. 2, April-June 1975, pp. 169-180.

## REFERENCES (Continued)

122. Ichbiah, J. D. and Morse, S. P.: A Technique for Generating Almost Optimal Floyd-Evan Productions for Precedence Grammars. Comm. ACM, vol. 13, no. 8, August 1970, pp. 501-508.
123. Mickunas, M. D. and Schneider, V. B.: A Parser Generating System for Constructing Compressed Compilers. Comm. ACM, vol. 16, no. 11, November 1973, pp. 669-676.
124. Lancaster, R. H. and Schneider, F. B.: Quick Compiler Construction Using Uniform Code Generators. Software-Practice and Experience, vol. 6, no. 1, January-March 1976, pp. 83-91.
125. Chartres, B. A. and Florentin, J. J.: A Universal Syntax-Directed Top-Down Analyzer. J. ACM, vol. 15, no. 3, July 1968, pp. 447-464.
126. Kuno, S. and Oettinger, A. G.: Multiple-path Syntactic Analyzer. Information Processing 62, North Holland Publishing Co., Amsterdam, 1962, pp. 306-311.
127. Jensen, K. and Wirth, N.: PASCAL User Manual and Report. Springer-Verlag, New York, 1975.
128. Wulf, W. A., Russell, D. B., and Habermann, A. N.: BLISS: A Language for Systems Programming. Comm. ACM, vol. 14, no. 12, December 1971, pp. 780-790.
129. Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. Comm. ACM, vol. 13, no. 6, June 1970, pp. 377-387.
130. Date, C. J.: Relational Data Base Concepts. Datamation, vol. 22, no. 4, April 1976, pp. 50-53.
131. Lucas, P. and Walk, K.: On the Formal Description of PL/I. Annual Review in Automatic Programming, vol. 6, no. 3, 1969.
132. Wegner, P.: The Vienna Definition Language. Computing Surveys, vol. 4, no. 1, March 1972, pp. 5-63.

## REFERENCES (Concluded)

- 133. Lee, J. A. N.: Computer Semantics. Van Nostrand, Reinhold, New York, 1972.
- 134. Beech, D.: On the Definitional Method of Standard PL/I. ACM Symposium on Principles of Programming Languages, October 1973, pp. 87-94.
- 135. Scott, D.: Outline of a Mathematical Theory of Computation. Proc. Fourth Annual Princeton Conf. on Info. Science and Systems, 1970, pp. 169-176.
- 136. Scott, D.: Mathematical Concepts in Programming Language Semantics. Proc. AFIPS Spring Jt. Computer Conf., vol. 40, 1972, pp. 225-234.
- 137. Bell, C. G. and Newell, A.: Computer Structures: Readings and Examples, McGraw Hill Book Co., New York, 1971.
- 138. Barbacci, M.: A User's Guide to the ISPL Compiler. Carnegie-Mellon University, May 17, 1975.

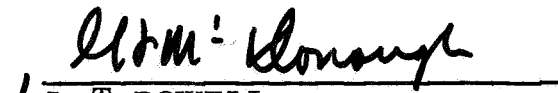
## APPROVAL

### A SURVEY OF COMPILER DEVELOPMENT AIDS

By B. P. Buckles, B. C. Hodges, and P. Hsia

The information in this report has been reviewed for security classification. Review of any information concerning Department of Defense or Atomic Energy Commission programs has been made by the MSFC Security Classification Officer. This report, in its entirety, has been determined to be unclassified.

This document has also been reviewed and approved for technical accuracy.

  
J. T. POWELL  
Director, Data Systems Laboratory



POSTMASTER :

If Undeliverable (Section 158  
Postal Manual) Do Not Return

*"The aeronautical and space activities of the United States shall be conducted so as to contribute . . . to the expansion of human knowledge of phenomena in the atmosphere and space. The Administration shall provide for the widest practicable and appropriate dissemination of information concerning its activities and the results thereof."*

—NATIONAL AERONAUTICS AND SPACE ACT OF 1958

## NASA SCIENTIFIC AND TECHNICAL PUBLICATIONS

**TECHNICAL REPORTS:** Scientific and technical information considered important, complete, and a lasting contribution to existing knowledge.

**TECHNICAL NOTES:** Information less broad in scope but nevertheless of importance as a contribution to existing knowledge.

**TECHNICAL MEMORANDUMS:** Information receiving limited distribution because of preliminary data, security classification, or other reasons. Also includes conference proceedings with either limited or unlimited distribution.

**CONTRACTOR REPORTS:** Scientific and technical information generated under a NASA contract or grant and considered an important contribution to existing knowledge.

**TECHNICAL TRANSLATIONS:** Information published in a foreign language considered to merit NASA distribution in English.

**SPECIAL PUBLICATIONS:** Information derived from or of value to NASA activities. Publications include final reports of major projects, monographs, data compilations, handbooks, sourcebooks, and special bibliographies.

**TECHNOLOGY UTILIZATION PUBLICATIONS:** Information on technology used by NASA that may be of particular interest in commercial and other non-aerospace applications. Publications include Tech Briefs, Technology Utilization Reports and Technology Surveys.

*Details on the availability of these publications may be obtained from:*

**SCIENTIFIC AND TECHNICAL INFORMATION OFFICE**

**NATIONAL AERONAUTICS AND SPACE ADMINISTRATION**

**Washington, D.C. 20546**