AN ABSTRACT OF THE THESIS OF

JIMMY IVAN RUCKER for the MASTER OF SCIENCE
(Name) (Degree)

in COMPUTER SCIENCE presented on November 19, 1973
(Major) (Date)

Title: A COMPILER-COMPILER FOR THE CDC 3300

Abstract approved: Redacted for privacy
Curtis R. Cook and Joel Davis

This thesis describes a syntax-directed compiler-compiler called COMCOM which has been implemented by the author on the CDC 3300 under the OS-3 operating system. The theory and terminology of the parsing method and compiler-compilers in general are briefly discussed. COMCOM uses Floyd's operator precedence bottom-up parsing technique which avoids backup and is very efficient. The syntactic metalanguage is similar to Backus-Naur Form. The source language semantics may be composed using a library of supplied routines or the user can expand the semantic capabilities by writing semantic routines in COMPASS or FORTRAN. System features available to the user and methods to bypass shortcomings of the theory are described. The system is briefly compared to the META/OS-3 compiler writing system which uses a top-down parsing technique. Simplified versions of the algorithms, and a simple compiler built using COMCOM are included.

A Compiler-Compiler for the CDC 3300

by

Jimmy Ivan Rucker

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed November 1973

Commencement June 1974

APPROVED:

Redacted for privacy

Assistant Professor of Computer Science
in charge of major

Redacted for privacy

Chairman of Department of Computer Science

Redacted for privacy

Dean of Graduate School

Date thesis is presented November 19, 1973

Typed by Clover Redfern for Jimmy Ivan Rucker

ACKNOWLEDGMENTS

The author is most grateful to Joel Davis for his assistance in the early stages of this thesis and to Curtis R. Cook for his supervision of the preparation of this paper. Thanks is also extended to G.A. Bachelor for the use of his various subsystems and to the members of the examination committee.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

A COMPILER-COMPILER FOR THE CDC 3300

I. INTRODUCTION

The purpose of this thesis is to present a compiler-compiler system called COMCOM which has been implemented by the author on the CDC 3300 under the OS-3 operating system.

A compiler-compiler accepts a description of the structure and meaning of a language and builds a translator for the language. Compiler-compilers can be used to build translators for a wide variety of languages. Plus, the implementer is relieved of many of the details present in the traditional methods of writing compilers. Evolution of compiler-compilers began as early as 1960 when Irons [13] was able to construct a compiler whose structural phase was independent of the source language being translated. Some compiler-compilers are the META systems [14, 16, 17], Feldman's Formal Semantic Language (FSL) [7], EULER [18, 19], and Compiler Compiler [5]. A version of META has been implemented at Oregon State University, META/OS-3 [2]. Some of the external features of COMCOM are similar to META/OS-3.

Feldman and Gries [8] give an excellent introduction and state of the art (1968) survey of translator writing systems (TWS, of which compiler-compilers are a subclass). It also includes an extensive bibliography.

The heart of a compiler is its method for recognizing or parsing the source language. The languages for which a compiler can be constructed depends heavily on the parsing algorithm selected. Two general categories of parsers are top-down and bottom-up. Most top-down processes require backup, a very undesirable feature. Bottom-up methods using precedence techniques eliminate backup by using tables of information about the language. Floyd was the first to formalize the idea of precedence. His operator precedence technique [9], is used by COMCOM. Other types of precedence are simple precedence [18, 19], weak precedence [12], and various higher order precedence methods [10].

Summaries of parsing techniques and compiler construction methods can be found in the book by Gries [10].

The next chapter defines some key terms, discusses bottom-up recognizers, and describes the operator precedence recognizer which is used by COMCOM. The third chapter gives a brief description of the COMCOM system. A more detailed description of some features can be found in the Appendices. Chapter IV compares COMCOM and META/OS-3.

II. THEORY

A. Terminology and Definitions

The first section of this chapter reviews some definitions and terminology used in this thesis. Most of them are from Gries [10] where the formal definitions can be found.

Translators are programs which translate a source program into an equivalent object program, for example, compilers, interpreters, and assemblers. A translator writing system (TWS) is a program or set of programs which aids in writing translators. The main purpose of a TWS is to simplify the implementation of translators. TWS's which are tailored towards writing compilers are called compiler-compilers. COMCOM is a compiler-compiler, but it is flexible enough to be used for other translating tasks such as format conversions and interpreters.

A vocabulary is a finite set of symbols. A string is a finite sequence of symbols from some vocabulary. The empty string is the string containing no symbols and is denoted ϵ. V^* denotes the set of all strings over vocabulary V and includes the empty string. For example if $V = \{a, b\}$, then $V^* = \{a, b\}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. V^+ denotes V^* minus the empty string.

A context free grammar is a four-tuple $G = (N, T, P, S)$ where

1. N is a finite set of nonterminals;

2. T is a finite set of terminals ($N \cap T = \emptyset$);

3. P is a finite set of productions or rewriting rules of the form $U \rightarrow x$ where U in N and x in $(N \cup T)^*$.

4. S in N is the distinguished start symbol.

Henceforth, the term grammar will mean a context free grammar.

If $U \rightarrow u$ is a production of G and x, y in $(N \cup T)^*$ then xUy directly produces xuy (denoted $xUy \Rightarrow xuy$). We also say xuy directly reduces to, or is a direct derivation of xUy. If there exist a sequence of direct derivations

$$u_0 \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_n$$

where $x = u_0$, $y = u_n$, and $n > 0$ then x produces y ($x \overset{+}{\Rightarrow} y$) or y reduces to x. The sequence is called a derivation of length n. If $n \geq 0$ then x generates y ($x \overset{*}{\Rightarrow} y$). Thus "$\Rightarrow$" is one, "$\overset{+}{\Rightarrow}$" is a sequence of one or more, and "$\overset{*}{\Rightarrow}$" is a sequence of zero or more direct derivations.

A string is called a sentential form if it can be generated from the distinguished start symbol. A sentential form which contains only terminal symbols is a sentence. The language defined by a grammar L(G) is the set of all sentences generated from the start symbol. A grammar which generates the unsigned integers is shown in Figure 1.

$G_1 = (N, T, P, \text{INTEGER})$

$N = \{\text{INTEGER}, \text{INT}, \text{DIGIT}\}$

$T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

The elements of P are:

1.	INTEGER	->	INT
2.	INT	->	INT DIGIT
3.	INT	->	DIGIT
4.	DIGIT	->	0
5.	DIGIT	->	1
6.	DIGIT	->	2
7.	DIGIT	->	3
8.	DIGIT	->	4
9.	DIGIT	->	5
10.	DIGIT	->	6
11.	DIGIT	->	7
12.	DIGIT	->	8
13.	DIGIT	->	9

Figure 1. Context free grammar G_1 for the unsigned integers.

A grammar is a description of a language's structure, or <u>syntax</u>, and conveys nothing about the meaning, or <u>semantics</u>, of the language. The derivation of a sentential form can be illustrated by drawing a <u>syntax tree</u> with the start symbol at the top, the branches extending downward, and the node labels are the terminals and nonterminals of the grammar. A syntax tree for the sentential form "INT 9 4" (generated from G_1) is shown in Figure 2. Readers not familiar with syntax trees are referred to Gries' book [10].

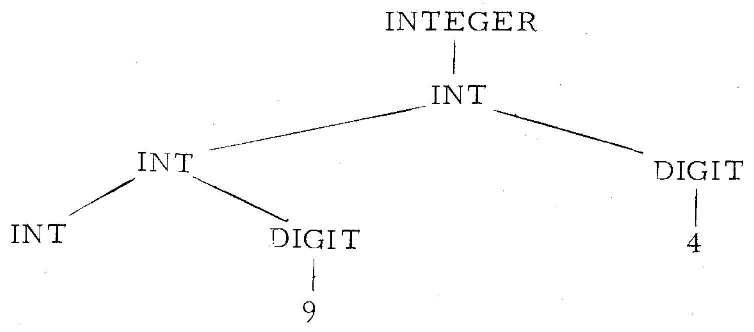


Figure 2. Syntax tree for the sentential form "INT 9 4".

A sentence of a grammar is _ambiguous_ if there exists two different syntax trees for it. A grammar is _ambiguous_ if it contains an ambiguous sentence; otherwise it is _unambiguous_. Sometimes an ambiguous grammar can be converted into an unambiguous grammar which generates the same language. Notice it is the defining grammar which is called ambiguous and not the language. However, there are languages for which no unambiguous grammar exist and these languages are called _inherently ambiguous_.

In most theoretical and practical situations the empty string is not allowed as a production right part. It is well known that given a context free grammar G and ϵ-free grammar G' can be constructed such that $L(G') = L(G) - \{\epsilon\}$. Moreover, if G is unambiguous then so is G' [10].

A language which is used to describe another language is called a _metalanguage_. A well known syntactic metalanguage is Backus-Naur

Form (BNF) which was first used in the ALGOL 60 Report [4]. A process which uses a syntactic metalanguage in an algorithm for recognizing sentential forms is called syntax-directed. COMCOM and most compiler-compilers are syntax-directed.

A parse of a sentential form is the construction of a derivation and possibly a syntax tree for it. A program which parses sentential forms is called a parser, or recognizer. The parsers in this thesis are called left-to-right because they examine the source language starting at the left and proceed to the right. Most parsers can also be classified as either top-down or bottom-up. This refers to the way in which the syntax tree is constructed. The top-down parsers start at the distinguished symbol, the root, and work towards the end nodes in a predictive manner testing alternative productions until the sentential form is parsed. Top-down recognizers can be programmed in many different ways such as recursive descent [10] or a single routine working on a stack [8], but the distinguishing feature is their predictive or goal-oriented nature.

The bottom-up recognizers search the input string for a substring which is the right part of a production and replace the substring with the left part. This reduction corresponds to building a syntax tree from the bottom up towards the root. When only the start symbol remains the parse is complete.

Let $w = xuy$ be a sentential form defined by a grammar G.

Then u is called a <u>phrase</u> of the sentential form w if $S \overset{*}{=>} xUy$ and $U \overset{+}{=>} u$ where S is the start symbol; u is a <u>simple phrase</u> if $S \overset{*}{=>} xUy$ and $U => u$. For example, in Figure 2 "INT 9 4" is a phrase for "INT" and "9" is a simple phrase for "DIGIT". One must be careful with the term <u>phrase</u>. The fact $U \overset{+}{=>} u$ does not necessarily mean that u is a phrase of a sentential form xuy; we must also have $S \overset{*}{=>} xUy$. To illustrate this, consider the sentential form "INT 9". The existence of a rule "INTEGER —> INT" in G_1 (Figure 1) does not mean that "INT" is a phrase because "INTEGER 9" can not be generated from the start symbol "INTEGER".

A <u>prime phrase</u> is a phrase which contains no other phrase but at least one terminal. For example, "9" in Figure 2 and the left or right "i" in Figure 3 are prime phrases. Most bottom-up parsers reduce the <u>handle</u> or leftmost simple phrase which is "FACT" in Figure 3. However, some recognizers (such as the one described in this thesis) reduce the leftmost prime phrase, the leftmost "i" in Figure 3. Notice the difference between the handle ("FACT") and the leftmost prime phrase (leftmost "i") in Figure 3. The prime phrase must contain a terminal whereas the handle could consist of only nonterminals. One would expect the parser which reduces the left-most prime phrase, instead of the handle to be faster since it does not have to make reductions such as "FACT" to "TERM" or "TERM" to

"EXPR" in Figure 3.

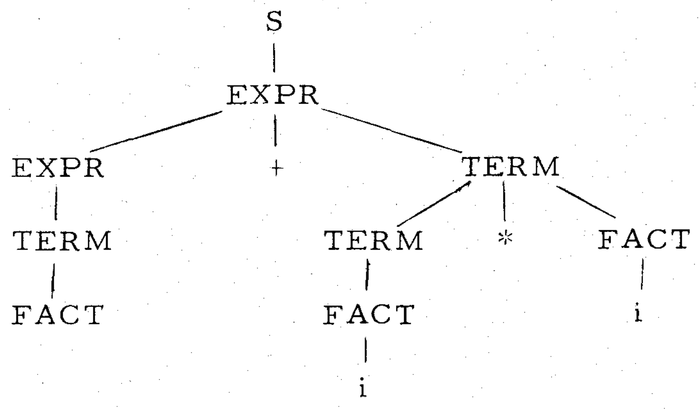


Figure 3. Syntax tree for "FACT + i * i".

In bottom-up parsing the main problem is finding the phrase to reduce. One solution is the precedence scheme. If someone is asked to evaluate "4 + 3 * 5" they will reply "19". The multiplication is performed before or has "precedence" over the addition. A formalization of the precedence idea is the operator precedence technique developed by Floyd [9]. An operator grammar is a context free grammar in which no production may be of the form $U \rightarrow xU_1U_2y$ for some strings x and y and nonterminals U_1 and U_2. Thus, no sentential form contains two adjacent nonterminal symbols [9].

In order to find the leftmost prime phrase to reduce, the following precedence relations are defined between terminal symbols T_1 and T_2 of an operator grammar.

1. $T_1 \doteq T_2$ if there is a production $U \to xT_1T_2y$ or $U \to xT_1U_1T_2y$ where U_1 is a nonterminal and x, y in $(N \cup T)^*$.

2. $T_1 > T_2$ if there is a production $U \to xU_1T_2y$ and a derivation $U_1 \overset{*}{\Rightarrow} z$ where T_1 is the rightmost terminal character of z.

3. $T_1 < T_2$ if there is a production $U \to xT_1U_1y$ and a derivation $U_1 \overset{*}{\Rightarrow} z$ where T_2 is the leftmost terminal character of z.

An operator precedence grammar is an operator grammar for which no more than one of the three relations hold between any ordered pair T_1, T_2 of terminal symbols. An operator precedence grammar G_2 for a subset of arithmetic expressions is shown in Figure 4. Languages generated by such grammars are called operator precedence languages. Any sentence of an operator precedence grammar has a syntax tree with a unique structure. However, the language may still contain ambiguous sentences since the name of the nodes could differ. This occurs when the right parts of productions are identical. For example, if production 1 of Figure 4 were replaced by productions 1a through 1e of Figure 5a the grammar would still be an operator precedence grammar. Figure 5b shows a syntax tree for "i + i" in which the third node from the top can be named "REALEXPR" or "INTEXPR" making the sentence ambiguous.

This will be discussed further in Section C of this chapter.

$G_2 = (N, T, P, S)$

$N = \{S, T, EXPR, TERM, FACT\}$

$T = \{+, *, (,), i\}$

The elements of P are:

1. S -> EXPR
2. EXPR -> EXPR + TERM
3. EXPR -> TERM
4. TERM -> TERM * FACT
5. TERM -> FACT
6. FACT -> (EXPR)
7. FACT -> i

Figure 4. Operator precedence grammar G_2 for a restricted subset of arithmetic expressions.

The parser described in this thesis reduces not the handle but the leftmost prime phrase. However, the parse is still left to right, bottom-up. The prime phrase of a sentential form is found using the precedence relations. These relations can be represented in an n x n precedence matrix where n is the number of terminals. Notice that the three precedence relations "<", "=", and ">" hold between ordered pairs of terminals and are not necessarily symmetric [10]. For example, "$T_1 = T_2$" does not mean that "$T_2 = T_1$".

1a. S —> E

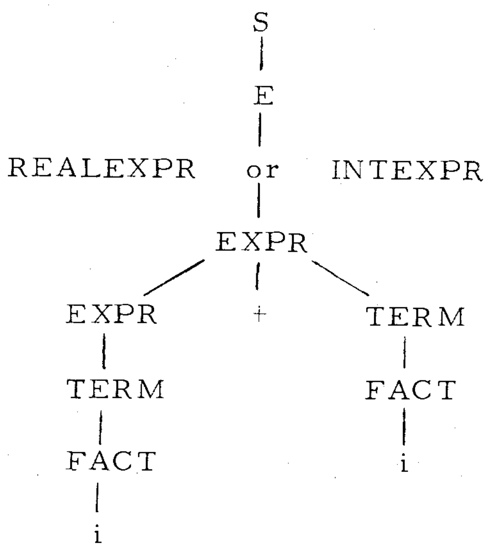
1b. E —> REALEXPR

1c. E —> INTEXPR

1d. REALEXPR —> EXPR

1e. INTEXPR —> EXPR

(a)



(b)

Figure 5. An ambiguous sentence "i + i" of an operator precedence grammar.

B. Operator Precedence Grammar Recognizer

This section describes Floyd's operator precedence grammar recognizer used by COMCOM, and briefly compares it to other bottom-up recognizers which can be constructed automatically from the grammar's productions.

The operator precedence grammar recognizer uses the precedence matrix to find a prime phrase. Figure 6 illustrates the precedence matrix for the grammar G_2 of Figure 4. Suppose $T_0 \times T$ is a substring of the sentential form $s = x_1 T_0 \times T x_2$ and that the terminal symbols in the substring x are, in order, $T_1, T_2, \ldots, T_n$ $(n \geq 1)$. If the following relations hold between

$T_0, T_1, \ldots, T_n,$ and T:

$$T_0 < T_1 \doteq T_2 \doteq \ldots \doteq T_n > T$$

then x is a prime phrase [10]. A nonterminal to the left of T_1 or to the right of T_n always belongs to the prime phrase. The first symbol in the phrase is the <u>head</u> and the last is the <u>tail</u>.

	+	*	(	)	i
+	>	<	<	>	<
*	>	>	<	>	<
(	<	<	<	=	<
)	>	>		>	
i	>	>		>	

Figure 6. Precedence matrix for the grammar G_2.

The algorithm for parsing a sentence is quite straightforward. Starting at the left of the sentence, the symbols are pushed onto a stack until $T_n > T$ holds between the top terminal on the stack and the next incoming symbol T, which is called the <u>window</u>. If the string is actually a sentence in the language, the top stack elements are the string T_0x as previously described. One searches back in the stack using the relations to find the head of x. The left most prime phrase is then x and can be reduced to some nonterminal which is placed on the top of the stack. This process repeats by comparing T_0 to T. A parse of "i + i * i" (delimited by "|-" and "-|") is illustrated in Figure 7. Note the nonterminals ("N") are just place holders.

Step	Stack	Relation	Window	Prime Phrase	Matching Production
1.	|-	<	i		
2.	|- i	>	+	i	7
3.	|- N	<	+		
4.	|- N +	<	i		
5.	|- N + i	>	*	i	7
6.	|- N + N	<	*		
7.	|- N + N *	<	i		
8.	|- N + N * i	>	-|	i	7
9.	|- N + N * N	>	-|	N * N	4
10.	|- N + N	>	-|	N + N	2
11.	|- N	STOP			

Figure 7. The parse of |- i + i * i -|.

If one was concerned only with syntactic correctness, the non-terminals could be completely eliminated and the parse would proceed essentially unchanged. However, languages have semantic information associated with the syntax. When a prime phrase is detected the productions are searched for a right hand side of the same form and the associated routine is called to do semantic processing. Thus, the nonterminals must be retained for at least place holding purposes.

Two other similar precedence schemes are the simple [18, 19] and weak [12]. In the simple precedence method there is no restriction of adjacent nonterminals and the precedence relations hold between both terminals and nonterminals. Thus, the precedence matrix is much larger than for the operator precedence method. Because of the way simple precedence grammars are defined and the restriction that right parts of productions be unique, all sentences of simple precedence languages are unambiguous [1]. The simple and operator precedence parsing algorithms are the same except the simple reduces the handle ("FACT" in Figure 3) instead of the leftmost prime phrase (leftmost "i" in Figure 3). The operator parser is faster since it does not have to make reductions such as "FACT" to "TERM" and "TERM" to "EXPR" in Figure 3. Also, it is harder to manipulate a programming language grammar into a simple precedence grammar [10]. Because of the above the operator precedence recognizer was chosen over the simple precedence scheme.

The weak precedence technique is a slight modification of the simple. The relations "<" and "=" are combined into one. The incoming symbols are stacked until the top stack item has the relation ">" to the window. The head of the handle is found by pattern matching the top stack symbols to the right side of productions, the longest match being the handle. It is easier to construct weak precedence grammars, but the size of the precedence matrix is again large, and this procedure is even slower than the simple precedence method because of the pattern matching required.

Higher order precedence methods [10] use more symbols in the sentential form to detect the head and tail of the handle and thus use large precedence matrices. Bounded context parsers [10] use tables consisting of stack configurations and incoming symbols to find the handle and determine the correct reduction. The higher order precedence and bounded context methods can be made practical but become complicated [10], especially for a compiler-compiler system.

C. Theory Advantages and Disadvantages

As mentioned in Section A of this chapter, an operator precedence language may contain ambiguous sentences. The structure of the syntax tree is unique for each sentence but the names of the nodes may be ambiguous. For a prime phrase x there may be more than one nonterminal to which it can be reduced, since there is no restriction that right parts of productions be unique. However, nonterminals are usually manipulated as operands by the semantic routines, and not so much by the syntax parser. The syntax defines the structure; whether a node is named say real expression or integer expression as in Figure 5b is a matter to be handled by the semantic routines.

The restriction of no two adjacent nonterminals and problems with precedence conflicts can complicate the construction of an operator precedence grammar for a language with the desired attributes. Operators in a left recursive rule are usually in an infix notation similar to the "+" and "*" in Figure 4. For example, Figure 8 shows an attempt to write a grammar for a language in which each statement is followed by a dollar sign. The statements of the program are represented by "PSTATE". The precedence conflicts can be eliminated using right recursion by replacing productions 2 through 4 (inclusive) with "BODY —> STATE $ BODY" and "BODY —> STATE $". But then the last statement and dollar sign of the program must be reduced before any other dollar signs and the parsing stack becomes very large. The author has been unable to produce an operator precedence grammar for operators in a postfix notation (operator appears after operand or operands) so that the stack does not proliferate. However, there is a method in COMCOM to implement postfix operators and it will be developed in Chapter III, Section B.

```
G₃ = (N, T, P, PROGRAM)
 N = {PROGRAM, BODY, STATEDS, STATE}
 T = {BEGIN, END, $, PSTATE}
The elements of P are:
  1. PROGRAM —> BEGIN  BODY  END
  2. BODY    —> BODY  $  STATEDS
  3. STATEDS —> STATE  $
  4. BODY    —> STATE
  5. STATE   —> PSTATE
Precedence conflict:    $ > $
                        $ < $
```

Figure 8. Operator precedence grammar with a precedence conflict.

The restriction of no two adjacent nonterminals is not serious. Many programming language's grammars are already in this form and those which are not can be manipulated into operator precedence grammars without essentially disturbing the structure of the language. For example, if productions 2 and 3 of Figure 8 are replaced with "BODY —> BODY $ STATE" the precedence conflicts are eliminated and the program statements are separated by, instead of followed by, a dollar sign. The source language is the same except the dollar sign after the last statement in the source program is omitted.

Precedence conflicts may also occur when the same terminal is used for multiple purposes, for example, the unary and binary minus sign. If the preceding operator is checked or if two distinct terminals are used, the unary and binary uses can be distinguished. Thus, in the operator precedence technique precedence conflicts and the restriction of no two adjacent nonterminals can require many terminals in the language.

The operator precedence technique yields a recognizer which is extremely efficient and can be constructed automatically from the (grammar's) productions. Compared to top-down methods, the operator precedence technique requires extra processing to produce the precedence matrix and other tables but is usually justified by the increased parsing speed and no backup. It is faster and uses less space than the simple, weak, and other precedence methods mentioned earlier. Floyd [9] has constructed a grammar for an algol like language for which the precedence matrix is only about 40 x 40. The other precedence methods would require a substantially larger matrix because the precedence relations hold between both the terminals and nonterminals.

Thus, the operator precedence method is very efficient in its use of time and space, it is easy to understand and is relatively easy to use. For these reasons it is the recognizer used by COMCOM. As of 1968 the authors of [8] were not aware of a compiler containing a mechanically constructed operator precedence recognizer, but indicated the precedence technique itself has been used in quite a few ALGOL, MAD, and FORTRAN compilers. In a search of the literature since then the author has been unable to find a TWS using the operator precedence method. Hence COMCOM appears to be the only TWS using this scheme.

III. THE COMCOM SYSTEM

A. Description of COMCOM

The following is only a brief description of the COMCOM system. Additional information may be found in the Appendices.

The components of the COMCOM system are the constructor, parser, scanner, overlay writer, and semantic routines. A diagram of how these components interact is in Appendix A. Simplified versions of the algorithms for some of the components are also in the Appendices.

The constructor accepts the syntactic metalanguage productions building the precedence matrix and other tables required by the parser. Any precedence conflicts are indicated and the precedence matrix may be printed out if desired. The printed matrix and the productions enable the users to determine how the parse will proceed. Using a method described by Floyd in [9] the precedence matrix can often be reduced to two arrays reducing storage requirements. However, this is not done by COMCOM because the algorithm fails for some operator precedence grammars and the matrix cannot be reduced. Also, in a parse syntax errors would go undetected longer using the arrays because the relation "no relation" between two terminals is not detected immediately.

The syntactic metalanguage input to the constructor resembles BNF and an example is shown in Figure 9. ".SYNTAX" and ".END" are control statements signifying the beginning and end of the program. The two lines following ".SYNTAX" are declarations of the maximum number of rules, terminals, and semantic routines, and declare the name of an error routine. In the productions (which may appear in any order) "=" is used for "—>", the terminals are enclosed in " ' ", and the nonterminals are variables such as "PROGRAM" or "BODY". Production right sides cannot be the empty string. Semantic routine calls may be associated with each production and are preceded by a ".", for example ".OUTEND". Each production is ended with ".,". The syntax specification in Figure 9 describes a language consisting of a "BEGIN" followed by a body and an "END". The body consists of statements in the form "x =" an arithmetic expression, where the arithmetic expressions involve the variables "A" and "B", the operators "+" and "*", and each statement is separated by a "$". Notice in the sample source program of Figure 10 there is no dollar sign after the last statement. The program of Figure 9 specifies that each statement is separated by, not followed by, the terminal "$".

Thus, to build a translator using COMCOM one writes the syntax specification in an operator precedence grammar. Production right parts containing terminals represent prime phrases. If the user wants control to do semantic processing when a prime phrase is detected (by the parser) he includes a semantic routine call after the production right part. Right parts consisting of a single nonterminal will never be a prime phrase, by definition, and are forbidden to have semantic routine calls. If several productions have right parts of the same form, that is, the terminals match and they have nonterminals in the same positions, then only one of them is allowed to have an associated semantic routine call. The called routine must either take the same action for all productions or determine which production is the correct one. This will be discussed further following the description of terminal classes. Appendix G shows a simple compiler implemented using COMCOM, complete with semantic routines and sample runs.

```
.SYNTAX ARITHMETIC EXPRESSIONS

RULES(15) TERMINALS(15) ROUTINES(10)
ERROR(ERRORSR)

PROGRAM = 'BEGIN' BODY 'END'                   .OUTEND .,
BODY = BODY '$' STATEMENT .,
BODY = STATEMENT .,
STATEMENT = 'X' '=' EXPR                       .ASSIGN .,
EXPR = EXPR '+' TERM                           .PLUS .,
EXPR = TERM .,
TERM = TERM '*' OPERAND                        .MULT .,
TERM = OPERAND .,
OPERAND = 'A'                                  .AOPERAND .,
OPERAND = 'B'                                  .BOPERAND .,

.END
```

Figure 9. COMCOM program for a restricted subset of arithmetic expressions.

```
BEGIN
X = A * B     $
X = A + B * A     $
X = B * B * B
END
```

Figure 10. Sample source program for compiler of Figure 9.

The scanner reads the source language and classifies each item as a particular terminal. There is no provision for backup in COMCOM so this classification is final unless the user changes it with semantic routines. The properties of the scanner can be changed by declaring certain terminal classes. For example, an unsigned integer can either be built from scratch

```
 INTEGER = '0' .,
 INTEGER = '1' .,
 INTEGER = '2' .,
           .
           .
 INTEGER = '9' .,
 INTEGER = INTEGER '0' .,
 INTEGER = INTEGER '1' .,
 INTEGER = INTEGER '2' .,
           .
           .
 INTEGER = INTEGER '9' .,
OPERAND = INTEGER .,
```

or the terminal class .INTEGER (indicated by a period before the class name) can be declared with the other declarations (line after .SYNTAX in Figure 9) as follows

CLASSES (. INTEGER)

and writing the production

OPERAND = . INTEGER . ,

This will cause the scanner to automatically classify strings of digits as the terminal class . INTEGER. When the scanner classifies a string as a terminal class it copies the actual characters into a block of storage and hands the parser a pointer to the block.

A special terminal class is the end of line (. EOL). When it is declared the scanner supplies the terminal . EOL at the end of each source record. Figure 11 gives the syntax specification of a simple compiler which uses . EOL between each statement. Notice that the last source line will contain a statement, the terminal "END", and a supplied . EOL. The available terminal classes are . ID (identifier), . INTEGER, . SPECIALC (special character), . LETTER, . NUMBER, . DIGIT, . STRING, . EOL.

The parser is the basis of the constructed translator. The parser calls the scanner to get terminals pushing them onto a syntax stack until a prime phrase is detected. The prime phrase's form is matched to right parts of the productions and control is transferred to the semantic routine or routines associated with the matching production. Upon return, the parser performs the reduction by popping the phrase from the stack and placing a nonterminal on top. As an example from the simple compiler in Appendix G

```
READ = 'READ' LIST .,
LIST = LIST ',' .ID        .READWT .,
WRITE = 'WRITE' WLIST .,
WLIST = WLIST ',' .ID .,
```

When the stack holds a prime phrase of the form "N , .ID" (where "N" means nonterminal, "," is a terminal, and ".ID" is the identifier terminal class) control is transferred to the subroutine named "READWT". Notice the nonterminals are merely place holders on the stack and an ambiguity problem arises with the write statement. The prime phrase matches both the "LIST" and "WLIST" productions thus, the phrase may be part of a read or a write statement. When a number of right parts are of the same form COMCOM allows only one to have semantic routine calls. In ambiguous situations, like "LIST" and "WLIST" or when production right parts are identical, it is the semantic routine's responsibility to determine the correct production and take appropriate action. In this example the user's semantic routine named "READWT" resolves the ambiguity by a context check. The top item on the stack is the ".ID" and the item below the "N" of the prime phrase is either the terminal "READ" or "WRITE". If it is "READ"("WRITE") the routine "READWT" produces the code to read (write) the value of the identifier represented by the terminal class .ID.

There is a library of semantic routines available such as TPLUS, TMINUS, XLPLUS, or XLMINUS. These routines aid in temporary location and label generation. Another option for semantic processing is the assembler *SAM [3]. This feature enables one to translate a source language into COMPASS assembly language instructions and call *SAM to assemble them one line at a time avoiding intermediate output. Semantic routines which call *SAM are OUT and PUT.

Semantic routines can also be written by the user in COMPASS or FORTRAN making COMCOM readily expandable and extremely flexible. The user's routines may call numerous supplied primitives performing such tasks as to call *SAM, set up labels and temporary storage locations in an output buffer (to which the user also has direct access) for interfacing with *SAM, scan ahead in the source input string, or do context checks on the stack and input string. Primitives can also be used to access or change the syntax stack, precedence matrix, window item, or manipulate a supplied operand stack. These routines are instrumental in translating, generating object code, performing syntactic error recovery, and other tasks. A list of the primitives and supplied semantic routines is in Appendix F.

Syntax errors often occur in source programs. A practical translator should recover and continue parsing the remaining source program. Translators for widely differing languages can have different error recovery requirements. COMCOM allows the user to write his own error recovery routine so that he may apply the recovery method most appropriate to his language. When COMCOM's parser detects structural errors or illegal characters in the source program it transfers control to the user's error routine passing a code indicating the exact nature of the error. Using the primitives the error routine can alter the state of the parse allowing it to continue.

Another feature of the COMCOM system is the implementation of comments. According to the theory, once the recognizer starts parsing the source language, everything in the program must consist of terminals arranged structurally as defined in the productions. However, comments are usually in a natural language different from the source language. Both the symbols and their arrangement differ. Unless the comment's structure and symbols are defined in the productions the parser will reject them as illegal. In COMCOM the user can declare the terminals which constitute the beginning and end of a comment in his source language. When the scanner encounters the beginning terminal it skips the entire comment and the parse proceeds as if the comment never occurred. A similar problem arises with the terminal class .STRING. The structure and characters within the string are usually illegal in the source language. Again the user can declare what terminal characters constitute the beginning and end of a string. In this case the productions must indicate how the terminal class .STRING fits in the structure of the source language because when the scanner encounters the beginning of a string, it scans to the end and hands the parser the terminal class .STRING with a pointer to the actual terminals constituting the string. The .EOL terminal class can also be declared as the terminal ending the string or comment. A sample program illustrating these features is given in Figure 11. The third line declares the terminal classes used in the compiler. The fourth line declares the beginning of a string is the terminal "'" and the end is the end of line terminal class. The fifth declares the beginning of a comment is the terminal "COMMENT" and the end is again the end of line terminal class.

```
.SYNTAX SIMPLE COMPILER

RULES(20) TERMINALS(20) ROUTINES(15) ERROR(ERR)
CLASSES(.ID,.NUMBER,.INTEGER,.STRING,.EOL)

STRING(''', EOL)
COMMENT('COMMENT', EOL)

.OUTSTART
PROGRAM = 'BEGIN' BODY 'END' .EOL          .PROG .,
BODY = BODY .EOL STATEMENT                 .BODY .,
BODY = STATEMENT .,
STATEMENT = ASSIGN .,
STATEMENT = STRING .,
STRING = .STRING                           .STRI .,
ASSIGN = VARB '=' E                        .OUTAS .,
E = E '+' T          .PLUS .,
E = T .,
T = T '*' F          .STAR .,
T = F .,
F = VARB .,
VARB = .ID           .VARB .,
F = .NUMBER          .NUM .,
F = .INTEGER         .INTEG .,
.OUTEND

.END
```

Figure 11. A simple compiler.

The semantic routine call ".OUTSTART" before the productions is executed before the parse starts, and the call ".OUTEND" after the productions is executed after the parse is complete. Each semantic routine in this program just outputs its own name when called. Figure 12 shows a sample run of the compiler in Figure 11. The inputs are the indented lines and the others are the outputs of the semantic routines. Thus, Figure 12 in conjunction with Figure 11 illustrates how the parse proceeds by showing when each semantic routine is called.

```
OUTSTART

     BEGIN  A  =  10

VARB
INTEG
OUTAS

     COMMENT THIS IS A COMMENT

     ' THIS IS A STRING

STRI
BODY

     A = S    END

VARB
VARB
OUTAS
BODY
PROG
OUTEND
```

Figure 12. A sample run of the compiler of Figure 11.

B. Operators in a Postfix Notation

As discussed in Chapter II Section C the author has been unable to produce an operator precedence grammar in which left recursive rules define operators in a postfix notation. The term postfix means the operators appear after the operand or operands. For example, the operator "+" in "A + B" is in infix and in "AB +" is in postfix notation.

The problem developed in Chapter II was a programming language in which each statement was followed by a dollar sign. Here the operand is a program statement and the operator is the dollar sign. A right recursive scheme was presented like that of Figure 13a where "PSTATE" of production 4 represents the program statements. In Figure 13b each terminal is assigned an integer code and Figure 13c is the precedence matrix defined by the productions. Figure 15 is a parse of the sample source program of Figure 14. Notice in Figure 15 the form "N $" (N means nonterminal) repeats for each program statement causing the stack to proliferate. When the last statement and dollar sign are encountered the stack size can finally decrease as the prime phrase "N $" is reduced using production 3 of Figure 13a. Then the rest of the dollar signs are reduced in the prime phrases of the form "N $ N" using production 2. This stack proliferation and order of reduction is what one would expect from the right recursive production 2 in Figure 13a.

```
     .SYNTAX POSTFIX DOLLAR SIGNS
     RULES(5) TERMINALS(5) ROUTINES(3)
     ERROR(ERRORSR)
     .BEFORE
(1)  PROGRAM = 'BEGIN' BODY 'END' .,
(2)  BODY = STATEMENT '$' BODY .,
(3)  BODY = STATEMENT '$' .,
(4)  STATEMENT = 'PSTATE' .,
     .END
```

(a)

```
000 BEGIN
001 END
002 $
003 PSTATE
```

(b)

```
     000 001 002 003
000       =   <   <
001
002       >   <   <
003           >
```

(c)

Figure 13. Program and precedence matrix for a language with "$" in postfix notation.

```
BEGIN

PSTATE  $
PSTATE  $
PSTATE  $
   .
   .
   .
PSTATE  $
PSTATE  $

END
```

Figure 14. Source program for compiler of Figure 13.

Stack	Relation	Window	Prime Phrase	Matching Production
⊢	<	BEGIN		
⊢BEGIN	<	PSTATE		
⊢BEGIN PSTATE	>	$	PSTATE	4
⊢BEGIN N	<	$		
⊢BEGIN N $	<	PSTATE		
⊢BEGIN N $ PSTATE	>	$	PSTATE	4
⊢BEGIN N $ N	<	$		
⊢BEGIN N $ N	<	PSTATE		
⊢BEGIN N $ N $ PSTATE	>	$	PSTATE	4
⋮		⋮	⋮	
⊢BEGIN N $ N $...$ N $ N	<	$		
⊢BEGIN N $ N $...$ N $ N $	>	END	N $	3
⊢BEGIN N $ N $...$ N $ N	>	END	N $ N	2
⊢BEGIN N $ N $...$ N	>	END	N $ N	2
⋮	⋮	⋮	⋮	
⊢BEGIN N $ N	>	END	N $ N	2
⊢BEGIN N	=	END		
⊢BEGIN N END	>	⊣	BEGIN N END	1
⊢N	STOP			

Figure 15. Parse of program in Figure 14.

The order in which the productions are used in a parse can be seen by generating a program from the start symbol "PROGRAM". The order is reversed for the parse or reduction of the program. One can also simulate the parse by hand as shown in Figure 15. Notice the stack becomes very large because of the precedence relation "$ < $". Ordinarily one would change the grammar to a left recursive form to keep the stack small but this produces the precedence conflicts shown in Figure 8.

In this example the postfix operator parsing problem can be solved by the user writing the "BEFORE" semantic routine of Figure 13 (executed before the parse starts) to change "$ < $" to "$ >$". This can be done by calling the primitive subroutine "STOREREL". Figure 16 shows how the parse would procede using the amended precedence matrix. Notice the nonterminals are just place holders on the stack and production 2 of Figure 13a is then used to reduce the prime phrases of the form "N $ N" keeping the stack small. Notice also, the prime phrases and matching productions occur in a different order than that of Figure 15. Since there are no semantic routine calls with productions 2 and 3 of Figure 13a it does not matter that the changed relation ("$ >$") causes the changed order of the matching productions. Even if there were routine calls with these productions the routines could be written to take the proper semantic action on the basis of how the amended parse of Figure 16 proceeds. This method of changing the matrix and amending the semantic routines (if necessary) solves the problem for this example and could easily be incorporated into the compiler of Appendix G. However, it departs from the present operator precedence theory. This is not a general solution and could become awkward in other postfix operator applications.

Stack	Relation	Window	Prime Phrase	Matching Production
⊢	<	BEGIN		
⊢ BEGIN	<	PSTATE		
⊢BEGIN PSTATE	>	$	PSTATE	4
⊢BEGIN N	<	$		
⊢BEGIN N $	<	PSTATE		
⊢BEGIN N $ PSTATE	>	$	PSTATE	4
⊢BEGIN N $ N	>	$	N $ N	2
⊢BEGIN N	<	$		
⊢BEGIN N $	<	PSTATE		
⊢BEGIN N $ PSTATE	>	$	PSTATE	4
⊢BEGIN N $ N	>	$	N $ N	2
⊢BEGIN N	<	$		
⋮	⋮	⋮	⋮	
⊢BEGIN N $ PSTATE	>	$	PSTATE	4
⊢BEGIN N $ N	>	$	N $ N	2
⊢BEGIN N	<	$		
⊢BEGIN N $	>	END	N $	3
⊢BEGIN N	=	END		
⊢BEGIN N END	>	⊣	BEGIN N END	1
⊢N	STOP			

Figure 16. Parse of program in Figure 14 using an amended precedence matrix.

C. COMCOM System Disadvantages

Whenever a theoretical concept is implemented certain restrictions and modifications are usually necessary. A compiler-compiler should not be biased toward any traits of a particular language. COMCOM departs from the ideal in some instances, for example, most terminals are restricted to eight characters. An end of file or end of data is required at the end of each source program. This could have been avoided by a declaration method where the user declares what terminates his source program similar to the way one can declare what begins and ends a comment or string in COMCOM. The concept of end of line (.EOL) as a terminal is implemented in a limited fashion. When declared it is supplied at the end of every source record which may cause problems.

As discussed earlier ambiguity problems may arise when productions have right parts of the same form requiring context checks to resolve them. However, this is an inadequacy of the theory not of COMCOM's implementation.

D. COMCOM System Advantages

This section gives some of the general advantages of COMCOM over other systems.

COMCOM uses a bottom-up precedence parsing method which avoids backup and is very fast. The metalanguage resembles BNF making it easy to learn. COMCOM readily accepts left and right recursive productions and requires no factoring. Most top-down schemes do not allow left recursive productions and require factoring of right parts. There is a wide class of languages which can be generated by operator precedence grammars and the user can supply the error recovery technique most suitable to his particular language.

The terminal class features may be turned on or off and make implementation of translators much simpler. Also, the grammatical productions may be input in any order. The supplied semantic routines and primitives reduce the number of routines the user must write, perform the more common tasks required to generate object programs, and relieve the user of many details. The routines also give the user access to the parser's internal features. For example, as discussed in Section B of this chapter, the user could even change the precedence matrix. Comments in the source program may be implemented by declaring the terminals which begin and end each comment.

The most important advantage of COMCOM is its extreme flexibility, owing to the fact the user can supply his own semantic routines. Thus, the system semantic features are readily expandable by the user in that he can write his semantic routines in the COMPASS and FORTRAN languages. For example, a symbol table feature could be added and since the user would write it himself it would fit his exact needs. COMCOM could be used to implement the first pass of a multiple pass compiler and even the later passes if the intermediate output's structure could be expressed in an operator precedence grammar. COMCOM can also be used to implement other types of translators such as an interpreter. The difference between compilers and interpreters is that a compiler produces an object program equivalent to the source program, whereas an interpreter executes the source program.

IV. THE META/OS-3 SYSTEM

A. Brief Description of META/OS-3

The following is an extremely sketchy description of META/OS-3 [2] which only outlines the main characteristics so it can be compared to COMCOM.

META/OS-3 is a very good special-purpose, syntax-directed compiler writing system. It uses a single pass, top-down parsing method with no backup, and includes provisions for using the *SAM assembler. The metalanguage resembles BNF with two additional syntactic features. One is the use of parentheses for grouping and for factoring to avoid backup and left recursion which causes endless loops in top-down methods. The other is a "repeat" feature which causes the next item (or group of items) to be iterated replacing left recursion. See Appendix H for a sample program. The metalanguage productions, or rules, in META/OS-3 are each converted into a subroutine which returns a true status if the source input matches one of its ordered alternatives and false otherwise. Starting with the main rule each nonterminal encountered calls another rule in a recursive descent scheme. The system's semantic routine calls are embedded in the syntax rules and are executed as the rule recognizes the source langage. When the descent procedure completes the main rule the parse is complete.

B. Comparison of COMCOM to META/OS-3

META/OS-3 and COMCOM are both syntax-directed compiler-compilers which run under the OS-3 operating system, but their parsing techniques differ sharply. COMCOM is bottom-up and META/OS-3 is top-down. Bottom-up parsing is usually publicized as faster [11]. Since META/OS-3 does not allow backup these two implementations are competitive in their use of computer time. Appendix H shows a time comparison of the parsing algorithms of COMCOM and META/OS-3. In this trial the constructed compilers required almost an identical amount of time to parse the same source program. The META/OS-3 parsing algorithm becomes less efficient when it must try many incorrect alternatives in rules and descend extensively. This corresponds to building the syntax tree from the top down and having many choices for a node name each of which has many possible node names below it and so on. In cases such as this COMCOM would most likely be faster.

The differing parsing methods also have an effect on the meta-languages. META/OS-3 users must factor and carefully order the alternatives in their rules avoiding backup and left recursion. COMCOM never backs up and uses left recursion as a key feature. In META/OS-3 the semantic routine calls and parameter strings are embedded in the syntax specification whereas in COMCOM they follow the syntactic portion of each rule.

Some of the supplied semantic routines and terminal classes of COMCOM resemble those of META/OS-3. Classes which COMCOM has but META/OS-3 does not are .DIGIT, .LETTER, and .SPECIALC. The end of line (.EOL) terminal class more closely resembles an actual terminal in COMCOM making possible compilers for which the end of a source line separates statements.

Both systems include *SAM as an optional feature. However, META/OS-3 is a closed system in which it is often rather awkward to write the semantic processing and code generation aspects of a compiler. This is because the semantic routine calls and parameters are embedded in the rules and the semantic routines lack certain features. For example, there is no provision for doing arithmetic at compile-time (other than incrementing and decrementing a temporary storage counter) for such purposes as computing the storage needed for a multi-dimensional array. COMCOM users have the arithmetic plus other COMPASS and FORTRAN features available for compile-time semantic routines since they can write the routines in these languages. This expandable feature can make COMCOM harder to use, however the flexibility gained appears to be well worth the effort.

One of the major limitations of META/OS-3 is its lack of a symbol-table facility. This means one can not record the attributes of an identifier and therefore one can not efficiently generate code whose form depends on the types of variables involved. However, COMCOM users can implement a symbol-table feature to fit his exact needs since he will supply it himself. The only limitation in adding these extra features is the user's COMPASS and FORTRAN programming abilities.

META/OS-3 can be used to implement compilers for a wider class of languages than COMCOM. However, COMCOM can be used to implement interpreters whereas META/OS-3 is not designed for this purpose.

V. SUMMARY AND CONCLUSIONS

This thesis has presented a potentially useful syntax-directed compiler-compiler called COMCOM. The theory and terminology of the parsing method and compiler-compilers in general were briefly discussed. COMCOM uses Floyd's operator precedence bottom-up parsing technique which avoids backup and is very efficient. The syntactic metalanguage is similar to BNF. The source language semantics may be composed using a library of supplied routines or the user can expand the semantic capabilities by writing semantic routines in COMPASS or FORTRAN. The user may supply an error recovery routine applying the methods most suited to his source language. Optional features such as terminal classes, a source language comment facility, and the *SAM assembler also aid in implementing translators. Many implementation techniques were not discussed because of their detailed nature.

A comparison of COMCOM to META/OS-3 revealed that META/OS-3 may be able to implement compilers for a wider class of languages and is probably easier to use. However, COMCOM users can expand the system's semantic features, hence it is a more flexible system. Also, COMCOM can be used to implement interpreters whereas META/OS-3 was not designed for this purpose.

Possible additions to this system or others built like it could include a more flexible scanner. A "scanner-compiler" could be used in conjunction with the compiler-compiler. Its input could be declarations of characters to serve as delimiters, digits, special characters, and parts of identifiers. One could also declare characters he wanted completely ignored by the constructed scanner or declare other character uses. Another modification which would save storage and increase parsing speed is to reduce the precedence matrix to two arrays when possible and use the entire matrix only when the reduction algorithm fails. COMCOM is expandable by the user in COMPASS and FORTRAN but could be made expandable in other assembly language compatible langauges such as PL/1.

In order for TWS's to be more practical there must be an adequate formalization of metalanguages for syntax and semantics. There has been much progress in the syntax but very little in semantics. There is a need for more developments in formal language theory. In particular, as brought out in Chapter II Section C, can an operator precedence grammar left recursively generate a language which has operators in postfix notation? Every context free language has an operator grammar [1], but does every context free language have an operator precedence grammar? In general, the properties and limitations of a class of languages generated from a particular type of grammars requires more research.

BIBLIOGRAPHY

1. Aho, Alfred V., and Ullman, Jeffery D. The Theory of Parsing, Translation, and Compiling. Englewood Cliffs, N.J.:Prentice-Hall, Inc., 1972.

2. Bachelor, G.A. META/OS-3 Reference Manual. Unpublished manual, Oregon State University, February 28, 1973.

3. ________________. *SAM. Unpublished class notes Mth457/8/9, Oregon State University, January 26, 1972.

4. Backus, J.W. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." Proceedings of the International Conference on Information Processing. UNESCO, Paris, June 1959, 125-132.

5. Brooker, R.A., MacCallurn, I.R., Morris, D., and Rohl, J.S. "The Compiler Compiler." Annual Review in Automatic Programming, New York: The MacMillan Co., 1963, 229-275.

6. Colmerauer, Alain. "Total Precedence Relations." Journal of the ACM, 17 (January, 1970), 14-30.

7. Feldman, Jerome A. "A Formal Semantics for Computer Language and its Application in a Compiler-Compiler." Communications of the ACM, 9 (January, 1966), 3-9.

8. ________________, and Gries, David. "Translator Writing Systems." Communications of the ACM, 11 (February, 1968), 77-112.

9. Floyd, Robert W. "Syntactic Analysis and Operator Precedence." Journal of the ACM, 10 (July, 1963), 316-333.

10. Gries, David. Compiler Construction for Digital Computers. New York: John Wiley & Sons, Inc., 1971.

11. Griffiths, T.V., and Petrick, S.R. "On the Relative Efficiencies of Context-Free Grammar Recognizers." Communications of the ACM, 8 (May, 1965), 289-299.

12. Ichbiah, J. D., and Morse, S. P. "A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars." Communications of the ACM, 13 (August, 1973), 501-508.

13. Irons, E. T. "A Syntax Directed Compiler for Algol 60." Communications of the ACM, 4 (January, 1961), 51-55.

14. Oppenheim, David K., and Haggerty, Daniel P. "META5: A Tool to Manipulate Strings of Data." Proceedings of the ACM 21st National Conference. 1966, 465-468.

15. Rosen, Saul. "A Compiler-Building System Developed by Brooker and Morris." Communications of the ACM, 7 (July, 1964), 403-414.

16. Schneider, F. W., and Johnson, G. D. "META-3: A Syntax-Directed Compiler Writing Compiler to Generate Efficient Code." Proceedings of the ACM 19th National Conference. 1964, p. D1.5-1.

17. Schorre, D. V. "META-II: A Syntax-Oriented Compiler Writing Language." Proceedings of the ACM 19th National Conference. 1964, p. D1.3.

18. Wirth, Nicklaus, and Weber, Helmut. "EULER: A Generalization of ALGOL, and its Formal Definition: Part 1." Communications of the ACM, 9 (January, 1966), 13-25.

19. ________________. "EULER: A Generalization of ALGOL, and its Formal Definition: Part 2." Communications of the ACM, 9 (February, 1966), 89-99.

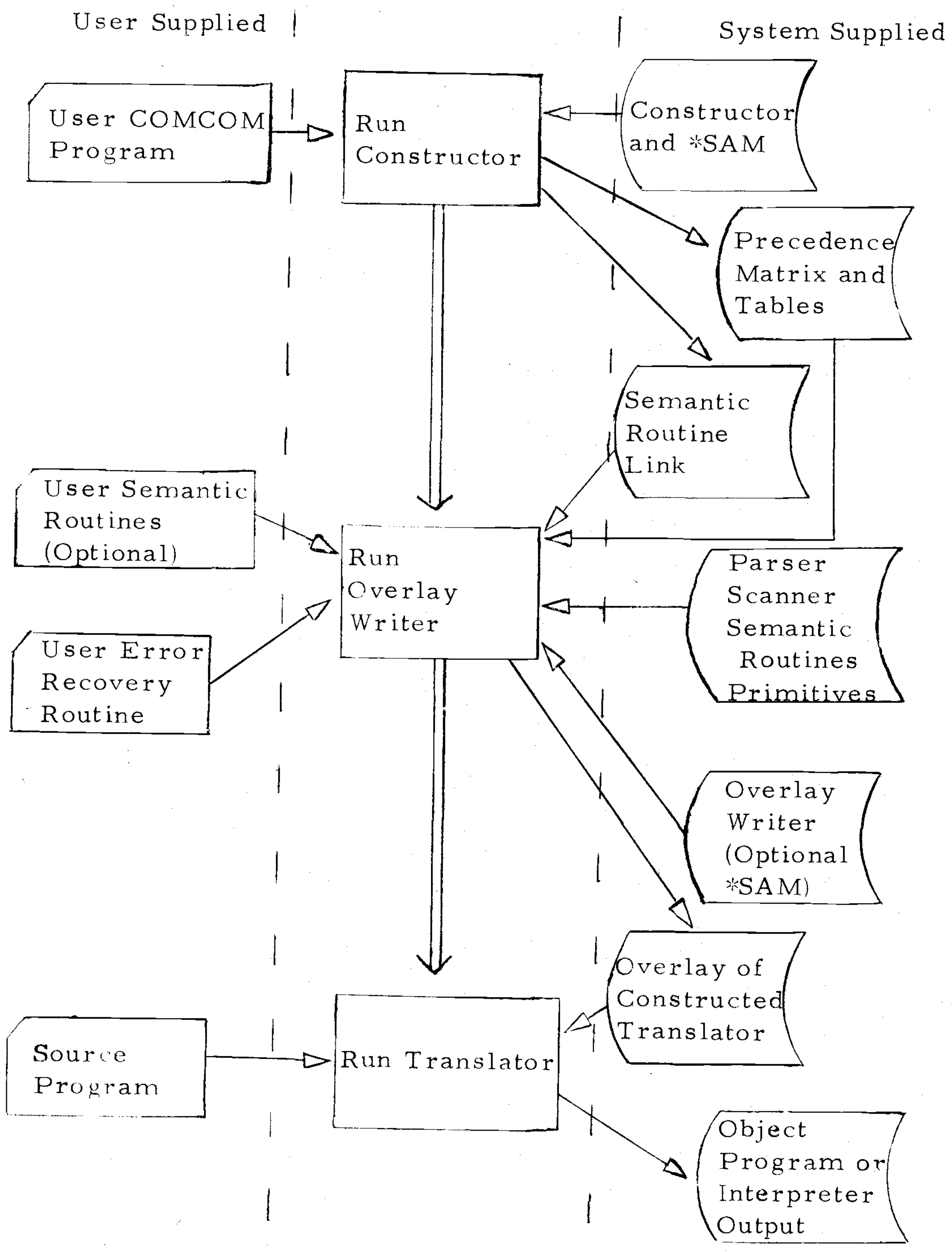
APPENDICES

APPENDIX A

COMCOM System Flowchart

The diagram on the next page is a COMCOM system flowchart. Inputs and outputs are signified by "$\longrightarrow\!\triangleright$" and time flow is signified by "$\Longrightarrow$". The programs on the left are user supplied and the files and programs on the right are system supplied.

The constructor accepts the syntactic metalanguage productions and builds the precedence matrix, tables, and semantic routine link. The overlay writer writes out an absolute copy of memory or overlay of the constructed translator. When the overlay writer gets control the semantic routine link has been loaded forming a jump table linking the parser to the user's and system's semantic routines. The parser is the controlling program of the constructed translator. When the translator overlay is copied back into memory and run its output could be a compiler's object program. This program would be loaded and run with its own inputs and outputs but is not shown in the flowchart on the next page.


User Supplied
System Supplied
User COMCOM Program
Run Constructor
Constructor and *SAM
Precedence Matrix and Tables
Semantic Routine Link
User Semantic Routines (Optional)
Run Overlay Writer
User Error Recovery Routine
Parser Scanner Semantic Routines Primitives
Overlay Writer (Optional *SAM)
Overlay of Constructed Translator
Source Program
Run Translator
Object Program or Interpreter Output

COMCOM System Flowchart

APPENDIX B

Constructor Algorithm

This appendix contains a simplified version of the constructor algorithm. The constructor accepts the syntactic metalanguage productions and builds the precedence matrix and other tables. It stores the productions by assigning an integer code to each terminal and nonterminal. Semantic routine names appearing with a production are also coded and stored with the rule. If a semantic routine has a parameter string, the string is not assigned codes by the constructor, but it is stored character for character including its ending parenthesis in BCD codes. When the parser calls a semantic routine the parameters, if any, are supplied in the BCD codes in COMMON (PARA in semantic routine listings in Appendix G).

The constructor orders the terminal table so that if two terminals match character for character from the left the longest is first. For example, say "**" is used for the power operator and "*" is used for multiplication, "**" will appear first in the terminal table. This is done so that when the scanner is searching the terminal table for a match to a source string, the longest match is found first.

Constructor Algorithm

1. Read source records until get ".SYNTAX".
2. Process the declarations of the number of rules, terminals, semantic routines, and put the name of the user's error routine in the semantic routine name table. Process the declaration of the terminal classes if any. Process the declaration of the beginning and ending of the comment and string if they are to be implemented.
3. If there are any semantic routine calls before the syntax rules put the names in the semantic routine name table and store the parameters if any.
4. Read a syntax rule (terminated by ". ,").
5. Place the terminals of the rule in the terminal table and nonterminals in the nonterminal table (if not already present).
6. If the production is of the form $U_1 \rightarrow T_1 x$ or $U_1 \rightarrow U_2 T_1 x$ (where T_1 is a terminal, U_1 and U_2 are nonterminals, and x is a string of terminals or nonterminals, or both, or empty), enter T_1 as a leftmost terminal character of the derivatives of U_1 (LTCD of U_1) and each LTCD of U_2 as a LTCD of U_1.
7. If the production is of the form $U_1 \rightarrow xT_1$ or $U_1 \rightarrow xT_1U_2$ enter T_1 as a rightmost terminal character of the derivatives of U_1 (RTCD of U_1) and each RTCD of U_2 as a RTCD of U_1.

8. If the production contains terminals T_1, T_2 in the form $U_1 \rightarrow xT_1T_2y$ or $U_1 \rightarrow xT_1U_2T_2y$ enter T_1 "=" T_2 in the precedence matrix. If the two terminals already have a relation different from T_1 "=" T_2, then there is a precedence relation conflict and the grammar is not an operator precedence grammar. Print the conflicting relations and terminals, set an error flag, and go to 4.

9. If the rule has two adjacent nonterminals set an error flag and go to 4 (not an operator precedence grammar).

10. Store the rule (in coded form) in the rule table. If there are any semantic routine calls with the rule, place the names in the semantic routine name table (if not already there), store each call and parameter string with the rule in the rule table.

11. If the rule is of the form $U_1 \rightarrow U_2$ and it has semantic routine calls, set an error flag and go to 4 (a right part consisting of a single nonterminal U_2 will never become the prime phrase by definition).

12. If the next item is not a semantic routine call or ".END" go to 4.

13. Done inputing productions. If there are any semantic routine calls after the productions put the names in the table (if not already there) and store the calls and parameter strings (if any).

14. Read the ".END" record. Complete the LTCD and RTCD tables as follows.

15. Go through the stored syntax rules from the last to the first (algorithm usually converges faster this way). If a rule is of the form $U_1 \rightarrow U_2 x$ enter each LTCD of U_2 as a LTCD of U_1.
16. Repeat 15 until the process converges.
17. Go through the stored syntax rules from the last to the first. If a rule is of the form $U_1 \rightarrow xU_2$ enter each RTCD of U_2 as a RTCD of U_1.
18. Repeat 17 until process converges.
19. Go through the stored rules once. For every occurrence of TU in a right part (where T is a terminal and U is a nonterminal) enter T "<" each LTCD of U into the precedence matrix. For every occurrence of UT enter each RTCD of U as ">"T into the precedence matrix. When each new relation is being stored if the old relation is not the "no relation" and is different from the new relation then there is a precedence conflict. Print the conflicting terminals and relations, set an error flag and continue.
20. Print precedence matrix if requested.
21. If any errors occurred print errors and <u>STOP</u>.
22. Order the terminal table so that if two terminals match character for character (from the left) the longest is first.
23. Produce a jump table from the semantic routine name table and call *SAM to assemble it into a binary deck. This deck is called the <u>link</u> and is used by the parser to convert the semantic routine

name's code (in the rule table) into the address of the semantic routine to jump to.

24. Write out the tables required by the parser (terminal table, precedence matrix, coded rules, and others) and the link binary deck.

25. If no errors print "NO COMCOM DIAGNOSTICS" and STOP.

APPENDIX C

Parser Algorithm

This appendix contains a simplified version of the parser algorithm used by COMCOM. The parser is the controlling program of the constructed translator. It calls the scanner to read and classify the source characters as particular terminals. When the parser detects a prime phrase it uses the semantic routine link to call the appropriate semantic routine. When it detects errors in the source program it calls the user's error routine. When the parse is complete it terminates the run.

Parser Algorithm

1. Call user semantic routines which are to be executed before parse starts, if any.
2. Call scanner to get first terminal; if end of file print error and STOP.
3. Push terminal onto top of stack.
4. Call scanner to get next terminal (the window); if end of file go to 7.
5. If top stack terminal ">" window go to 7; if no relation call the user's error routine and go to 4.
6. Push terminal onto stack; go to 4.

7. Look down in stack until find "<" between two terminals or find bottom of stack.

8. Prime phrase is now the terminals and nonterminals above the "<" relation on the stack. Match the phrase's form to the production right parts. If find a match call associated semantic routines passing parameter characters if any. If no match call user's error routine (illegal stack configuration) and go to 4.

9. Pop prime phrase off stack and place a nonterminal on top. If window is end of file go to 10 else go to 5.

10. Window is end of file. If there is a single nonterminal on stack the parse is done, else go to 7.

11. Call user semantic routines which are to be executed after parse, if any. STOP.

APPENDIX D

Scanner Algorithm

This appendix contains a simplified version of the scanner algorithm. The scanner is called as a subroutine by the parser. The scanner reads the source records into an input buffer eighty characters long. The input pointer, denoted "I" in the algorithm, keeps track of the current scanner position in the buffer. The scanner classifies strings of characters as a particular terminal in the terminal table and returns the terminal's integer code. When the scanner classifies a string as a declared terminal class it copies the actual characters into a block of storage and passes the parser the terminal class code and a pointer to the block. The terminal classes .DIGIT and .INTEGER can not be declared at the same time. An end of file is required at the end of each source program.

Scanner Algorithm

1. If input pointer (I) is less than 80 go to 3. If $80 \leq I < 100$ (i.e., first time have end of line) and .EOL is a declared class set I to 120 and <u>RETURN</u> with terminal class .EOL. Else, read a source record and set I to zero.
2. If record is end of file <u>RETURN</u> with end of file.

3. Increase input pointer to first nonblank character (if record all blanks go to 1 with I = 80). If first character is a special character go to 4; if a letter go to 15; if a digit go to 24; else have a system error.
4. First character is a special character. If it is a period followed by a digit and .NUMBER is declared RETURN with the period and digits as the terminal class .NUMBER.
5. Match the first character to each first character of the terminals in the terminal table. If found go to 6. If .SPECIALC is a declared class set input pointer to second character and RETURN with the first as the terminal class .SPECIALC. Else, call the user's error routine (illegal character), set input pointer to second character and go to 1.
6. The terminal table is ordered so that the first match will be the longest match (see constructor explanation Appendix B). Match the terminals character by character. If the entire terminal in the table matches the source input this is the correct terminal, go to 7. Else, the table item is longer or they do not match, go to 5 and resume the first character search.
7. If terminal is the comment beginning terminal go to 10.
8. If terminal is the .STRING class beginning terminal go to 12.
9. RETURN with terminal.

10. Have a comment. If comment ending terminal is .EOL (end of line) terminal class read a source record and set input point to zero; else go to 11. If record is end of file <u>RETURN</u> with end of file. If record not end of file go to 1 to get a new terminal (ignoring the comment).

11. Scan to end of comment terminal and go to 1 to get a new terminal.

12. Have a string. If string ending terminal is .EOL terminal class copy rest of source line into a block of storage, set input pointer to eighty, and go to 14.

13. Copy string and end of string terminal into a block of storage.

14. <u>RETURN</u> with terminal class .STRING (and a pointer to block).

15. First character is a letter. Scan to end of consecutive letters and digits and search the terminal table for a matching terminal. If found go to 16. If not found return input pointer to first letter and go to 19.

16. If terminal is the comment beginning terminal go to 10.

17. If terminal is the .STRING class beginning terminal go to 12.

18. <u>RETURN</u> with the terminal.

19. String not found in the terminal table. If the .ID class is not declared go to 23. If .ID is declared but .LETTER is not declared go to 21 to make the terminal an .ID.

20. Both .ID and .LETTER are declared terminal classes. If there are one or more consecutive letters or digits after the first letter

go to 21 else go to 22.

21. Copy the consecutive letters and digits into a block of storage (skipping remaining if more than eight), and RETURN with .ID terminal class.

22. Copy first letter into a block of storage. Set input pointer to character after letter, and RETURN with .LETTER terminal class.

23. If .LETTER is a declared class go to 22. Since the terminal is not in the terminal table and .LETTER and .ID are not declared it is an illegal character. Call user's error routine, skip letter, and go to 1.

24. The first character is a digit. If digits have an associated period go to 30.

25. If .INTEGER is declared copy digits and RETURN with terminal class .INTEGER (note that .INTEGER and .DIGIT can not be declared at the same time).

26. If .NUMBER and .DIGIT are both declared but there is more than one digit, copy consecutive digits to a block of storage supplying a decimal point at the end and RETURN with .NUMBER terminal class.

27. .INTEGER is not a declared class. If .DIGIT is declared set input pointer to next character and RETURN with terminal class .DIGIT.

28. .INTEGER, .DIGIT, and .NUMBER are not declared. Match the first digit to the first characters of the terminals in the terminal table. If found go to 29. If not, call user's error routine (illegal character), set input pointer to second character, and go to 1.

29. Match the terminals character by character. If the entire terminal in the table matches the source input this is the correct terminal; go to 7. Else, the table item is longer or they do not match. Go to 28 and resume the first character search. (Note that the terminal table is ordered so the first match is the longest and so on.)

30. There is a period adjacent to digits. If .NUMBER is declared scan to end of number and <u>RETURN</u> with .NUMBER terminal class.

31. If .INTEGER is declared set input pointer to period and <u>RETURN</u> with the digits before the period as the terminal class .INTEGER (.INTEGER and .DIGIT can not be declared at the same time).

32. If .DIGIT is declared set input pointer to second character and <u>RETURN</u> with the first as the terminal class .DIGIT; else go to 28 to try the terminal table.

APPENDIX E

Terminal Classes

This appendix lists the terminal classes and briefly describes the strings which can be each class. When the scanner classifies a string as a class it copies the characters into a block of storage (see scanner, Appendix D).

The classes .DIGIT and .INTEGER can not be declared at the same time. If .DIGIT and .NUMBER are declared and there is no decimal point associated with a string of digits they will be classified as .NUMBER if more than one digit in string or as .DIGIT if the string is just one digit. If .NUMBER and .INTEGER are both declared, a string of digits will be classified as a .NUMBER if there is a decimal point associated with the digits and .INTEGER if there is no decimal point. If .LETTER and .ID are declared at the same time a string will be classified as .LETTER if the string is only one letter long and as .ID if longer.

The following is a list of the terminal classes and a brief description of the properties of the strings which can be classified as that class.

.DIGIT	- a single digit.
.EOL	- a special terminal class. When declared it is supplied by the scanner at the end of each source record.

.ID	- a string of letters or digits or both, beginning with a letter. If the string which is classified as an .ID is longer than eight letters and digits the remaining characters are ignored.
.INTEGER	- a string of digits.
.LETTER	- a single letter.
.NUMBER	- a string of digits. If the scanner classifies a string as .NUMBER and there is no decimal point in the string one will be supplied in the storage block to which the string is copied.
.SPECIALC	- a single special character.
.STRING	- a string of characters beginning and ending with the declared beginning and ending .STRING terminals. The string is copied into the block including the ending terminal.

APPENDIX F

Semantic and Primitive Routines

This appendix contains information about the semantic and primitive routines supplied to the user. Table 1 lists the semantic and primitive routines and shows what they are usually used for. The routines can either be called by including them in a syntax production or by calling them from a user supplied semantic routine. As illustrated by Table 1, those which can be called in a production are semantic routines, those called in a user routine are primitives, and those which can be called from either serve as both semantic and primitive routines.

The items on the parser's syntax stack and in the window are coded. Thus, primitives are required to code and uncode the terminals for context checks and error recovery. Some of the routines work with an output buffer eighty characters in length. If *SAM is called it assembles the COMPASS code in the buffer, and blank fills the buffer. The user has direct access to the output buffer, and to the parser's window, input buffer, and input buffer pointer in COMMON. When the user includes a semantic routine call with parameters in a production, the parameters are supplied character for character in BCD codes in COMMON (PARA in listings of Appendix G).

Table 1. Semantic and primitive routines.

Routine Name	Can be Called in: Syntax Specification	or	Can be Called in: User Semantic Routine	Usual Routine Use
OUTSAM			x	Call *SAM
TERMINAL			x	Context check
COPYSTK			x	Context check
STACK	x		x	User operand stack
UNSTACK			x	User operand stack
TPLUS	x		x	Temporary storage generation
TMINUS	x		x	Temporary storage generation
TBUF			x	Temporary storage generation
RESET	x		x	Temporary storage generation
XLPLUS	x		x	Label generation
XLMINUS	x		x	Label generation
XLBUF			x	Label generation
PUT	x		x	Generate strings in output buffer
OUT	x		x	Generate strings in output buffer and call *SAM
STOREREL			x	Change precedence matrix
READ			x	Read source records and error recovery
POP			x	Error recovery
SCAN			x	Error recovery
PUSH			x	Error recovery
STOREWIN			x	Error recovery
GETWIN	x		x	Error recovery
ICODE			x	Error recovery
IGETREL			x	Error recovery

The routines are listed in the following pages in alphabetic order and include a brief explanation of the action each performs. Primitives are listed with formal parameters if they are required. Most of the routines are FORTRAN subroutines except IGETREL, TERMINAL, and ICODE which are FORTRAN functions. OUT and PUT do not have formal parameters but operate on BCD character strings put in PARA in COMMON. The primitives can be called from a FORTRAN or COMPASS subroutine.

Semantic and Primitive Routines

COPYSTK (IPOS, CHARAD) - copy the terminal (in BCD character codes) at stack position IPOS (where IPOS = 1 is top, IPOS = 2 is next to top, etc.) placing the first character at the character address CHARAD.

GETWIN - gets the next terminal in the source string and stores it in the window; updates the input pointer.

ICODE (TERM, IFCLASS) - FORTRAN integer function returns the integer code of the terminal TERM (BCD codes) in ICODE. Returns IFCLASS = 1 if TERM is a terminal class.

IGETREL (ICODE1, ICODE2) - FORTRAN integer function returns the precedence relation between the two terminals whose codes are ICODE1 and ICODE2. IGETREL = 0 if no relation, 1 if greater than, 2 if less than, and 3 if equal.

OUT(. . .) - sets up strings in the output buffer at the current output buffer pointer character position. The output pointer is initially equal to ten.

Parameters: .OP - sets output pointer to one.

.LABEL - sets output pointer to ten.

.SXX - copies terminal at stack position XX to current pointer position in output buffer (XX = 1 is top, XX = 2 is next to top, etc).

'----' - the string in the quotes is copied to current pointer position in output buffer.

/ - call *SAM with current output buffer configuration.

The parameters are executed left to right and *SAM is called when done. *SAM blanks output buffer and sets output buffer pointer to ten before returning.

OUTSAM (INUM) - calls *SAM where INUM is the maximum number of nonblank words in the output buffer (four characters per word).

POP(I) - pop I items off the syntax stack.

PUSH(X) - pushes the coded item X onto the syntax stack.

PUT(· · ·) - puts items into the output buffer at the current output pointer position the same as OUT, but it does not call *SAM. The parameter "/" is not allowed.

READ(I) - reads one source record and sets the input pointer to I. If the record is end of file or end of data I is returned greater than 200, else I is returned as zero.

RESET - empties the user operand stack and sets the temporary storage location counter to zero.

SCAN(I, X) - scans input buffer starting at position I (positions are numbered 0-79 for this subroutine) and classifies the first terminal encountered. If want .EOL set $80 < I < 100$. If want first item on next source record set $100 < I < 200$. Returns with terminal coded in X ready to be put on stack or in window and I is character position after terminal. If item was end of file or end of data then $I > 200$. (Note. If want I to be new scanner input pointer position, user must set I equal to the input pointer in COMMON.)

STACK - pushes contents of output buffer onto user operand stack and blank fills the output buffer. Resets output pointer to ten.

STOREREL (ICODE1, ICODE2, IREL) - stores the precedence relation IREL (where IREL = BCD codes NR, GT, LT, or EQ) between the two terminals whose integer codes are ICODE1 and ICODE2.

STOREWIN(X) - stores the coded terminal in the window.

TBUF(CHARAD) - stores the string T.XX at the character address CHARAD where XX is the current decimal value of the temporary storage location counter.

TERMINAL(I) - FORTRAN real function searches the terminal table and returns the BCD codes of the terminal whose integer code is I. If terminal is a class the first character will be a period.

TMINUS - decreases the temporary storage location counter one.

TPLUS - increases the temporary storage location counter one.

UNSTACK(CHARAD) - pops the user operand stack and copies the string at the character address CHARAD.

XLBUF(CHARAD) - copies the characters L.XXX at the character address CHARAD where XXX is the current decimal value of the label generation counter.

XLMINUS - decreases the label generation counter one.

XLPLUS - increases the label generation counter one.

APPENDIX G

Complete Simple Compiler

This appendix contains a complete listing of a compiler built using COMCOM. The COMCOM program is shown on the next page. A source program accepted by this compiler begins with "BEGIN", the program name, and a " . , ", followed by a body of statements separated by "$", and ended with "END". The statements can be labeled or unlabeled assignment, go to, read, write, if then, or if then else statements. Table 2 shows the terminals and their integer codes and Table 3 is the precedence matrix. The constructed compiler produces assembly language code and uses *SAM to assemble one line at a time. Input and output of floating point numbers is done using "FIN", "FOUT", and "ENDOUT" routines from the *SYSLIB library under the OS-3 operating system.

Table 2. Terminals and their codes.

0 00	.ID	0 13	)
0 01	.NUMBER	0 14	GO
0 02	END	0 15	TO
0 03	.,	0 16	READ
0 04	BEGIN	0 17	,
0 05	$	0 18	WRITE
0 06	.	0 19	ELSE
0 07	=	0 20	THEN
0 08	+	0 21	LT
0 09	-	0 22	GE
0 10	*	0 23	EQ
0 11	/	0 24	NE
0 12	(	0 25	IF

```
.SYNTAX COMPILER
ROUTINES(25) RULES(50) TERMINALS(35)
CLASSES(.ID,.NUMBER)
ERROR(ERR)
PROGRAM = BODY1 ≠END≠        .OUT(≠RTJ ENDOUT≠/≠SBJP≠/≠DEF≠/
                             ≠BSS 2≠/≠END START≠) .,
BODY1 = BEG ≠.,≠ BODY .,
BEG = ≠BEGIN≠ .ID            .OUT(≠IDENT ≠ .S1/≠ENTRY START≠/
                             ≠EXT FIN,FOUT,ENDOUT≠/
                             .LABEL ≠START≠ .OP ≠NOF 00≠ ).,
BODY = BODY ≠$≠ LSTATE .,
BODY = LSTATE .,
LSTATE = LABEL ≠.≠ STATEMENT .,
LSTATE = STATEMENT .,
LABEL = .ID .,
STATEMENT = ASSIGN .,
STATEMENT = GOTO .,
STATEMENT = READ .,
STATEMENT = WRITE .,
STATEMENT = IFTHENELSE .,
ASSIGN = .ID ≠=≠ EXPR           .ASSIGN .,
EXPR = EXPR ≠+≠ TERM            .PLUS .,
EXPR = EXPR ≠-≠ TERM            .XMINUS .,
EXPR = TERM .,
TERM = TERM ≠*≠ OP              .XMULT .,
TERM = TERM ≠/≠ OP              .DIVIDE .,
TERM = OP .,
OP = .ID                        .XIDENTF .,
OP = .NUMBER                    .XNUM .,
OP = ≠(≠ EXPR ≠)≠ .,
GOTO = ≠GO≠ ≠TO≠ .ID            .OUT(≠UJP ≠ .S1) .,
READ = ≠READ≠ LIST .,
LIST = LIST ≠,≠ .ID             .READWT .,
LIST = .ID .,
WRITE = ≠WRITE≠ WLIST .,
WLIST = WLIST ≠,≠ .ID .,
WLIST = .ID .,
NONIFST = ASSIGN .,
NONIFST = GOTO .,
NONIFST = READ .,
NONIFST = WRITE .,
IFTHENELSE = IFTHEN ≠ELSE≠ NONIFST      .ELSE .,
IFTHENELSE = IFTHEN .,
IFTHEN = IFOL ≠THEN≠ NONIFST       .THEN .,
IFOL = REST ≠LT≠ EXPR           .XLTSR .,
IFOL = REST ≠GE≠ EXPR           .GESR .,
IFOL = REST ≠EQ≠ EXPR           .EQSR .,
IFOL = REST ≠NE≠ EXPR           .XNESR .,
REST = ≠IF≠ EXPR                .XIFEXP .,
.END
```

Table 3. Precedence matrix.

	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015	016	017	018	019	020	021	022	023	024	025
000			>	>		>	>	=	>	>	>	>		>				>		>	>	>	>	>	>	
001			>			>			>	>	>	>		>						>	>	>	>	>	>	
002																										
003	<		>			<	<								<		<		<	<	<	<	<	<	<	<
004	=																									
005	<		>			>	<								<		<		<	<	<	<	<	<	<	<
006	<		>			>									<		<		<	<	<	<	<	<	<	<
007	<	<	>			>			<	<	<	<	<							>						
008	<	<	>			>			>	>	<	<	<	>						>	>	>	>	>	>	
009	<	<	>			>			>	>	<	<	<	>						>	>	>	>	>	>	
010	<	<	>			>			>	>	>	>	<	>						>	>	>	>	>	>	
011	<	<	>			>			>	>	>	>	<	>						>	>	>	>	>	>	
012	<	<							<	<	<	<	<	=												
013			>			>			>	>	>	>		>						>	>	>	>	>	>	
014																=										
015	=																									
016	<		>			>												<		>						
017	=																									
018	<		>			>												<		>						
019	<		>			>									<		<		<							
020	<		>			>									<		<		<	>						
021	<	<							<	<	<	<	<								>					
022	<	<							<	<	<	<	<								>					
023	<	<							<	<	<	<	<								>					
024	<	<							<	<	<	<	<								>					
025	<	<							<	<	<	<	<									>	>	>	>	

The following is a sample source program for the compiler and a run's inputs and outputs. The next page is the assembly language code (COMPASS) of this program which was produced by the constructed compiler. This code was put in the output buffer and passed to the *SAM assembler one line at a time avoiding intermediate output. The code "DEF" and "BSS 2" appearing just before "END START" causes *SAM to allocate storage and define all previously undefined identifiers as using two words of storage.

Thus, this constructed compiler using the *SAM assembler reads in a source program and writes out the equivalent binary object deck.

```
BEGIN FIRST .,
READ A,B $
WRITE A,B $
IF A EQ B THEN WRITE A $
C = A + 2.0 * B $
WRITE C
END
```

```
INPUT IS

6.0      5.0
```

```
OUTPUT IS

6.000000000         5.000000000         16.00000000
```

```
          IDENT FIRST
          ENTRY START
          EXT FIN,FOUT,ENDOUT
START     NOP 00
          RTJ FIN
          STAQ        A
          RTJ FIN
          STAQ        B
          LDAQ        A
          RTJ     FOUT
          LDAQ        B
          RTJ     FOUT
          LDAQ        A
          XOA,S -0
          XOQ,S -0
          FAD         B
          AZJ,NE      L.001
          LDAQ        A
          RTJ     FOUT
L.001     EQU *
          LDAQ         =202.0
          FMU         B
          STAQ        T.01
          LDAQ        A
          FAD         T.01
          STAQ        T.02
          LDAQ        T.02
          STAQ        C
          LDAQ        C
          RTJ     FOUT
          RTJ ENDOUT
          SBJP
          DEF
          BSS 2
          END START
```

The following seven pages are FORTRAN semantic routines supplied by the user. The first is the error recovery routine. In case of a parsing error, this simple compiler outputs the value of the current input buffer scanner position, outputs the nature of the error, and stops. The remaining six pages are the other FORTRAN semantic routines. When the parser detects a prime phrase, the phrase's form is matched to the production right hand sides. The semantic routine associated with the matching production is called. Each FORTRAN subroutine may contain several semantic routines. These inner routines begin with "ENTRY" followed by the routine name, and are ended with "RETURN". Some of the following semantic routines call other subroutines which are also listed.

```
      SUBROUTINE ERR
C     PARSING ERROR SUBROUTINE
C
      INTEGER OUTBUF,PARA
      COMMON IDUM(23),OUTBUF(20),XDUM(8450),WINDOW
      COMMON INPOINT,PARA(50)
C
100   FORMAT(' INPUT POINTER = ', I4)
      WRITE(61,100) INPOINT
      I = PARA( 1 )
      GO TO ( 1, 2, 3, 4, 5 ) , I
C
101   FORMAT(' LOOKED DOWN STACK AND NO < RELATION')
1     WRITE (61,101)
      GO TO 6
C
102   FORMAT(' TWO NONTERMINALS TOGETHER ON STACK')
2     WRITE(61,102)
      GO TO 6
C
103   FORMAT(' NO RULE MATCHES PRIME PHRASE')
3     WRITE(61,103)
      GO TO 6
C
104   FORMAT(' ILLEGAL CHARACTER IN SOURCE PROGRAM')
4     WRITE(61,104)
      GO TO 6
C
105   FORMAT(' NO RELATION BETWEEN TOP STACK AND WINDOW')
5     WRITE(61,105)
      GO TO 6
C
106   FORMAT(' COMPILATION TERMINATED')
6     WRITE(61,106)
      STOP
      END
      SUBROUTINE ASSIGN
      INTEGER OUTBUF,PARA
      COMMON IDUM(23),OUTBUF(20),XDUM(8450),WINDOW
      COMMON INPOINT,PARA(50)
      EQUIVALENCE (X,XCHARAD),(XREAD,IREAD),(XBCD,IBCD)
      EQUIVALENCE (XWRITE,IWRITE)
      DIMENSION IREAD(2),IWRITE(2),IBCD(2),IPERIOD(2)
      CHARACTER CPOS1,CPOS21,XCHARAD,CPOS5
      EQUIVALENCE (OUTBUF(6),DDD)
      EQUIVALENCE (DDD,CPOS21)
      EQUIVALENCE (OUTBUF,CPOS1)
      EQUIVALENCE (OUTBUF(2),XXX)
      EQUIVALENCE (XXX,CPOS5)
      EQUIVALENCE (XPERIOD,IPERIOD)
C
      OUTBUF(3)=4H LDA
      OUTBUF(4) =4HQ
      CALL UNSTACK(CPOS21)
      CALL OUTSAM(10)
      OUTBUF(3) = 4H STA
      OUTBUF(4) = 4HQ
      CALL COPYSTK(3,CPOS21)
      CALL OUTSAM(10)
      CALL RESET
      RETURN
C
      ENTRY PLUS
      IOP=4H FAD
      CALL COM(IOP)
      RETURN
C
      ENTRY XMINUS
      IOP=4H FSB
      CALL COM(IOP)
      RETURN
C
      ENTRY XMULT
      IOP=4H FMU
      CALL COM(IOP)
      RETURN
C
      ENTRY DIVIDE
      IOP=4H FDV
      CALL COM(IOP)
      RETURN
      ENTRY XIDENTF
C   TOP OF STACK IS .ID, IS NEXT ITEM DOWN A READ OR WRITE
      IF(IFLAG .EQ. 178) GO TO 40
C   FIRST TIME XIDENTF IS CALLED
      IFLAG = 178
      XBCD=0.0
      IPERIOD(1) = 4H.
      IPERIOD(2) = 4H
      IREAD(1) = 4HREAD
      IREAD(2) = 4H
      IWRITE(1) = 4HWRIT
      IWRITE(2) = 4HE
40    X = TERMINAL(WINDOW)
      IF(X .EQ. XPERIOD) GO TO 84
C   XCHARAD IS EQUIVALENT TO X
      CALL COPYSTK(2,XCHARAD)
      IF (X .EQ. XREAD) GOTO 71
      IF (X .EQ. XWRITE) GOTO 81
C   MUST BE IN AN EXPRESSION
      CALL COPYSTK(1,CPOS1)
      CALL STACK
      RETURN
71    CALL READSUB
      RETURN
81    CALL WRITE
      RETURN
C   HAVE A LABEL
84    CALL COPYSTK(1,CPOS1)
      OUTBUF(3) = 4H EQU
      OUTBUF(4) = 4H *
      CALL OUTSAM(10)
      RETURN
C
      ENTRY READWT
C   HAVE FORM T NT ≠,≠ .ID ON STACK, IS T=READ OR WRITE
      CALL COPYSTK(4,XCHARAD)
      IF ( X .EQ. XREAD ) GOTO 31
      IF ( X .EQ. XWRITE ) GOTO 32
C   HAVE A SYSTEM ERROR
100   FORMAT(≠ RD WT ERROR≠)
      WRITE(61,100)
      STOP
C   READ STATEMENT
31    CALL READSUB
      RETURN
C   WRITE STATEMENT
32    CALL WRITE
      RETURN
      ENTRY XNUM
      OUTBUF(1)=  4H =20
      CALL COPYSTK(1,CPOS5)
      CALL STACK
      RETURN
C
      ENTRY ELSE
      CALL XLPLUS
      CALL XLBUF(CPOS1)
      OUTBUF(3)  =  4H EQU
      OUTBUF(4)  =  4H *
      CALL OUTSAM(10)
      RETURN
C
      ENTRY THEN
C  IS THERE AN ELSE TO THE IF STATEMENT
      IBCD(1)=  4HELSE
      IBCD(2)  =  4H
      X  =  TERMINAL(WINDOW)
      IF( XBCD .NE. X) GO TO 82
C  HAVE ELSE
      CALL XLPLUS
      OUTBUF(3)  =  4H UJP
      CALL XLBUF(CPOS21)
      CALL XLMINUS
      CALL OUTSAM(10)
82    CALL XLBUF(CPOS1)
      OUTBUF(3)  =  4H EQU
      OUTBUF(4)  =  4H *
      CALL OUTSAM(10)
      RETURN
C
      ENTRY XLTSR
      CALL COMPARE
      OUTBUF(4)  =  4H,GE
      CALL XLPLUS
      CALL XLBUF(CPOS21)
      CALL OUTSAM(10)
      RETURN
C
      ENTRY GESR
      CALL COMPARE
      OUTBUF(4)  =  4H,LT
      CALL XLPLUS
      CALL XLBUF(CPOS21)
      CALL OUTSAM(10)
      RETURN
      ENTRY EQSR
      CALL COMPARE
      OUTBUF(4) = 4H,NE
      CALL XLPLUS
      CALL XLBUF(CPOS21)
      CALL OUTSAM(10)
      RETURN
C
      ENTRY XNESR
      CALL COMPARE
      OUTBUF(4) = 4H,EQ
      CALL XLPLUS
      CALL XLBUF(CPOS21)
      CALL OUTSAM(10)
      RETURN
C
      ENTRY XIFEXP
      OUTBUF(3) = 4H LDA
      OUTBUF(4) = 4HQ
      CALL UNSTACK(CPOS21)
      CALL OUTSAM(10)
      RETURN
      END
      SUBROUTINE COM(IOP)
      INTEGER OUTBUF,PARA
      COMMON IDUM(23),OUTBUF(20),XDUM(8450),WINDOW
      COMMON INPOINT,PARA(50)
      EQUIVALENCE (X,XCHARAD),(XREAD,IREAD),(XWRITE,IWRITE)
      DIMENSION IREAD(2),IWRITE(2)
      CHARACTER CPOS1,CPOS21,XCHARAD
      EQUIVALENCE (OUTBUF(6),DDD)
      EQUIVALENCE (DDD,CPOS21)
      EQUIVALENCE (OUTBUF,CPOS1)
      DIMENSION ITEMP(10)
C
      OUTBUF(3)=IOP
      CALL UNSTACK(CPOS21)
      DO 6 I=1,10
6     ITEMP(I)=OUTBUF(I)
      DO 65 I=1,10
65    OUTBUF(I)=4H
      OUTBUF(3)=4H LDA
      OUTBUF(4)=1HQ
      CALL UNSTACK(CPOS21)
      CALL OUTSAM(10)
      DO 66 I=1,10
66    OUTBUF(I)=ITEMP(I)
      CALL OUTSAM(10)
      OUTBUF(3) = 4H STA
      OUTBUF(4) = 4HQ
      CALL TPLUS
      CALL TBUF(CPOS21)
      CALL OUTSAM(20)
C  STACK THE TEMP
      CALL TBUF(CPOS1)
      CALL STACK
      RETURN
      END
      SUBROUTINE COMPARE
      INTEGER OUTBUF,PARA
      COMMON IDUM(23),OUTBUF(20),XDUM(8450),WINDOW
      COMMON INPOINT,PARA(50)
      EQUIVALENCE (X,XCHARAD),(XREAD,IREAD),(XWRITE,IWRITE)
      DIMENSION IREAD(2),IWRITE(2)
      CHARACTER CPOS1,CPOS21,XCHARAD
      EQUIVALENCE (OUTBUF(6),DDD)
      EQUIVALENCE (DDD,CPOS21)
      EQUIVALENCE (OUTBUF,CPOS1)
C
      OUTBUF(3) = 4H XOA
      OUTBUF(4) = 4H,S -
      OUTBUF(5) = 4H0
      CALL OUTSAM(10)
      OUTBUF(3) = 4H XOQ
      OUTBUF(4) = 4H,S -
      OUTBUF(5) = 4H0
      CALL OUTSAM(10)
      OUTBUF(3) = 4H FAD
      CALL UNSTACK(CPOS21)
      CALL OUTSAM(10)
      OUTBUF(3) = 4H AZJ
      RETURN
C
      ENTRY READSUB
C  HAVE A READ STATEMENT
      OUTBUF(3) =4H RTJ
      OUTBUF(4) = 4H FIN
      CALL OUTSAM(10)
      OUTBUF(3) = 4H STA
      OUTBUF(4) = 4HQ
      CALL COPYSTK(1,CPOS21)
      CALL OUTSAM(10)
      RETURN
C
      ENTRY WRITE
C  HAVE A WRITE STATEMENT
      OUTBUF(3) = 4H LDA
      OUTBUF(4) = 4HQ
      CALL COPYSTK(1,CPOS21)
      CALL OUTSAM(20)
      OUTBUF(3) = 4H RTJ
      OUTBUF(5) = 4HFOUT
      CALL OUTSAM(10)
      RETURN
      END
```

APPENDIX H

META/OS-3 and COMCOM Parsing Time Comparison

This appendix contains a parsing time comparison of META/OS-3 to COMCOM. The following page is the syntax specification for two translators. The first is a COMCOM program, and the second a META/OS-3 program. The constructed translators accept the same source language except COMCOM requires an end of file after the source program. The only semantic processing the translators perform is to print out "DONE" when the parse is complete.

THE FOLLOWING IS A COMCOM PROGRAM

```
.SYNTAX COMPARE
CLASSES(.ID,.NUMBER)
TERMINALS(20) ROUTINES(4) ERROR(ERR) RULES(15)
PROGRAM = ≠BEGIN≠ BODY ≠END≠        .OUT( ≠DONE≠ ) .,
BODY = BODY ≠$≠ STATEMENT .,
BODY = STATEMENT .,
STATEMENT = ASSIGN .,
ASSIGN = .ID ≠=≠ EXPR .,
EXPR = EXPR ≠+≠ TERM .,
EXPR = EXPR ≠-≠ TERM .,
EXPR = TERM .,
TERM = TERM ≠*≠ OP .,
TERM = TERM ≠/≠ OP .,
TERM = OP .,
OP = .ID .,
OP = .NUMBER .,
OP = ≠(≠ EXPR ≠)≠ .,
.END
```

THE FOLLOWING IS A META PROGRAM

```
.SYNTAX PROGRAM
PROGRAM = ≠BEGIN≠ STATEMENT $( ≠$≠ STATEMENT ) ≠END≠
                                 .OUT( ≠DONE≠ ) .,
STATEMENT = ASSIGN .,
ASSIGN = .ID+≠=≠ EXPR .,
EXPR = TERM $( ≠+≠ TERM / ≠-≠ TERM ) .,
TERM = OP $( ≠*≠ OP / ≠/≠ OP ) .,
OP = .ID / .NUMBER / ≠(≠ EXPR ≠)≠ .,
.END
```

The following is the sample source program used in this parsing time comparison. Table 4 on the next page is the results of the sample run. The two sets of programs were run consecutively under the OS-3 operating system. This operating system is a time sharing system, thus the cost and time figures are not extremely reliable. However, the programs were run several times and the figures appear to be representative. The important comparison is the parsing times since this indicates the efficiency of the constructed translator's recognizer. One must keep in mind that these are two specific implementations run on one specific set of inputs and that no real conclusions other than estimates can be drawn from this trial.

```
BEGIN
HDB = 5426 + HFGFG -(HG - .452 )           $
A = ( 0.676 + 8.3 * A ) / (10) + C $
DAS = E + D * FER + 6.8 $
DFDE = JHG *(DF+DFE* FGB )      $
AS = AS+ HG - JH - HGFV -HGFV*GVC -(FDD*GY) /YT $
AASF = KJD*JH + ( JH *( KH*( KJGB*( KJ+KG)/KGB) ))    $
DFDF Z=        DFF*(DF*(DFGT*(DF+SF)/RG))          $
SDEF =542.45 +ASW *(DF+65*78.52+SED*(DF+SX))          $
     A   = 3 + C    $
LHGK JGJ =  JHG*( H-6 ) - GGF          $
DF = NGGB * (EDFR+ HG) $
SD=  DFSDFG +  KJH-(SD-JH )
END
```

Table 4. Time comparison results.

	COMCOM		META/OS-3	
	Cost ($)	Time (sec)	Cost ($)	Time (sec)
Compile-compile time	.17	1.3	.12	1.3
Load and write overlay	.20	2.4	.09	1.1
Parse time	.10	.8	.10	.7
Total	.47	4.5	.31	3.1