A FORMAL SEMANTICS FOR COMPUTER-ORIENTED LANGUAGES

by

Jerome A. Feldman

Submitted to the Carnegie Institute of Technology
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Pittsburgh, Pennsylvania
1964
Pages i, ii, and iii are missing and will not be supplied.
PREFACE

In this thesis several aspects of the formalization of computer-oriented languages are considered. The emphasis has been placed on the practical results made possible by a formalization of the semantics of such languages. The most important of these results is the compiler-compiler which has been implemented using the formal systems described here as a base.

The author is deeply indebted to Professor Alan Perlis for his guidance in this work. He would also like to acknowledge the contributions made by the students of his undergraduate advanced programming course. Also of great value was the assistance of many members of the staff of the Carnegie Tech Computation Center, especially that provided by Messrs. D. Blocher and R. MacEwan and by Miss Nancy Lintelman.

The research reported here was supported by the Advanced Research Projects Agency of the Department of Defense under the Grant SD-146 to the Carnegie Institute of Technology.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter/Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface.</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>Abstract</td>
<td>vi</td>
</tr>
<tr>
<td><strong>CHAPTER I. INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>CHAPTER II. SYNTAX.</strong></td>
<td>9</td>
</tr>
<tr>
<td>A. The Production Language</td>
<td>9</td>
</tr>
<tr>
<td>B. Formal Properties</td>
<td>21</td>
</tr>
<tr>
<td><strong>CHAPTER III. SEMANTICS.</strong></td>
<td>33</td>
</tr>
<tr>
<td>A. Introduction</td>
<td>33</td>
</tr>
<tr>
<td>B. Storage Declarations</td>
<td>37</td>
</tr>
<tr>
<td>C. Storage Operations</td>
<td>42</td>
</tr>
<tr>
<td>D. Operands and Primaries</td>
<td>51</td>
</tr>
<tr>
<td>E. Expressions</td>
<td>63</td>
</tr>
<tr>
<td>F. Statements</td>
<td>66</td>
</tr>
<tr>
<td>G. Subroutines</td>
<td>73</td>
</tr>
<tr>
<td>H. Program Structure</td>
<td>78</td>
</tr>
<tr>
<td><strong>CHAPTER IV. IMPLEMENTATION.</strong></td>
<td>83</td>
</tr>
<tr>
<td><strong>CHAPTER V. CONCLUSION</strong></td>
<td>95</td>
</tr>
<tr>
<td>A. Review and Critique</td>
<td>95</td>
</tr>
<tr>
<td>B. Future Research</td>
<td>102</td>
</tr>
<tr>
<td><strong>APPENDICES</strong></td>
<td>109</td>
</tr>
<tr>
<td>Notes on the Appendices</td>
<td>109</td>
</tr>
<tr>
<td>A. Backus Normal Form Syntax of the Small Language</td>
<td>115</td>
</tr>
<tr>
<td>B. Production Language Syntax of the Small Language</td>
<td>117</td>
</tr>
<tr>
<td>C. Semantics of the Small Language</td>
<td>127</td>
</tr>
<tr>
<td>D. Backus Normal Form Syntax of FSL</td>
<td>133</td>
</tr>
<tr>
<td>E. Production Language Syntax of FSL</td>
<td>137</td>
</tr>
<tr>
<td>F. Sample Small Language Programs</td>
<td>147</td>
</tr>
<tr>
<td><strong>BIBLIOGRAPHY</strong></td>
<td>153</td>
</tr>
</tbody>
</table>
ABSTRACT

This dissertation presents a number of results attained in a study of the formalization of certain properties of computer-oriented languages. The most important result is a computer program which is capable of translating into machine language, any of a large class of source languages. Included in this class are all the usual high level problem-oriented languages.

The presentation of the results in the thesis is based on the structure of this program, called the compiler-compiler. Although there are several sections devoted to theoretical questions, these are set off from the main development. This organization was chosen so that the thesis could also serve as a user's guide to the compiler-compiler as a computer language.

A more detailed introduction to the paper is given in Chapter I, where we also consider some of the philosophical questions raised by formalizing semantics. Chapter II contains a discussion of a formal syntax language used in the compiler-compiler. In this chapter we establish relationships between this formalization of syntax and others appearing in the literature.

Chapter III is a complete discussion of the Formal Semantic Language which is the main contribution of this thesis. In Chapter IV we show how the two formal systems were combined to form the basis
for a useful computer technique. The final chapter contains a
discussion of the strengths and weaknesses of our system as well as
several suggestions for future research.

The appendices form an integral part of the thesis. The examples
contained there include a record of the development of a translator for
one small language from a formal definition of the language to examples
of resultant machine code.
I. INTRODUCTION

This thesis is a study of some theoretical and practical consequences of an attempt to formalize the semantics of computer-oriented languages. Among the languages in this class are the existing problem-oriented languages such as ALGOL 60 [31] and COMIT [43]. In addition, we include formalisms such as Markov Algorithms [30] which specify computational processes, but are not usually considered to be computer languages. Since the semantics definitions are to be processor-independent, we will usually not consider a particular machine order code as a computer-oriented language. Much more will be said about this situation in later chapters.

The emphasis in this paper will be placed on the practical results attainable from a formalized semantics. The most interesting application is in a computer program which will construct, for any computer-oriented language which has been appropriately described, a translator (compiler) for that language. This program would be a compiler-compiler for the machine on which it is implemented. Formalized semantics, combined with one of the known techniques for formalizing syntax, provides the facility for the complete formal description of computer languages. We have constructed a computer program, based on the formalisms described below, and have used it in the compilation of compilers for several well-known languages.

The entire formal scheme depends on a particular definition of the distinction between syntax and semantics. The definitions we have chosen
differ from those used by most mathematical logicians, but do correspond to the intuitive notions of syntax and semantics.

We define an alphabet, $\mathcal{A}$, to be a finite set of symbols such that any string of symbols in $\mathcal{A}$ can be uniquely decomposed into its component symbols. For any such alphabet, $\mathcal{A}$, there is an infinite set, $\mathcal{U}$, of finite strings of symbols from $\mathcal{A}$. Any subset, $\mathcal{L}$, of $\mathcal{U}$ can be considered an uninterpreted language over the alphabet $\mathcal{A}$. A specification of a process which will select the subset $\mathcal{L}$ of $\mathcal{U}$ is called a formal syntax of the uninterpreted language $\mathcal{L}$. The process mentioned above need not be effective [10], but in this paper we will consider only languages with effective tests. The language in which the test is specified is called the syntactic meta-language used to specify $\mathcal{L}$. The selection process (syntax) of $\mathcal{L}$ may be either an algorithm for generating all the strings in $\mathcal{L}$ or an algorithm for deciding of a string in $\mathcal{U}$ whether or not it is in $\mathcal{L}$.

Any other well-defined operation on strings in $\mathcal{U}$ will be considered semantical. An uninterpreted language together with at least one semantic processor will be called an interpreted language. Although there are many interesting problems connected with uninterpreted languages, we will deal only with interpreted languages in this thesis, referring to them simply as languages. A formal system used to describe a semantic processor on some language, $\mathcal{L}$, is called a semantic meta-language of $\mathcal{L}$. The fact that it is possible to define uninteresting semantic operators will not concern us here.
In our formulation, the syntax of a formal language includes only a specification of which strings are permissible in that language. This is what logicians have called syntax in the narrow sense [9, p.58]. There are good reasons why logicians have chosen not to use our definition of syntax and why we have done so. As Church [9] points out, there is a sense in which the difference is not essential. In mathematical logic, it is usual to refer to any completely formalized operation within a system as syntactical. A simple example may help point out the difference in the two notions of syntax.

In any formulation of the propositional calculus (quantifier-free logic) there are two well-known algorithms for selecting strings from the set of all strings over the alphabet. One algorithm specifies which strings are well-formed according to the formation rules. A second algorithm describes a method for choosing those strings which are theorems. Both of these operations are considered syntactical in mathematical logic.

Using our definitions, one comes up with the following results. The operation which chooses the well-formed strings is once again purely syntactic. However, the nature of the theorem selection algorithm depends on what one considers to be the language, $\mathcal{L}$. If $\mathcal{L}$ is the set of well-formed formulas, then the theorem selector is a semantic operator. If we choose to consider the set of theorems in the predicate calculus as the language, $\mathcal{L}$, then the theorem selector is a formal syntax of that language.
A good discussion of why the wider definition of syntax is useful in logic may be found in Church [9, pp. 47-68]. The principle motivation behind our definitions is their close correspondence with the operation of compilers operating on digital computers. As we will show, the specification of the formal syntax and formal semantics of a computer-oriented languages contains all the information needed in a translator for that language.

In Chapter II we will discuss two types of syntactic meta-languages, with the emphasis on a particular recognizer-oriented language which has proved useful in building translators. Section II-A contains a description of this language as a programming language. In Section II-B we consider the formal properties of this syntactic meta-language in comparison with some well-known formal systems.

Chapter III contains a complete description of the semantic meta-language, FSL (Formal Semantic Language), which is the principle contribution of this thesis. The discussion treats FSL as a programming language and contains frequent references to the semantic description of a small language given in Appendix C. In Chapter IV we describe a computer program which builds, from a formal description of a language, a translator for that language. Although this program is highly machine dependent, we require the formal descriptions themselves to be independent of the machine on which the translator is run. In setting up such a scheme we have followed the semiotic of C. S. Pierce, introduced into programming by Gorn [21].
The semiotic concept of language is that it consists of three parts: syntax, semantics, and pragmatics. Syntax is concerned with the selection of those strings which are members of the language. Semantics consists of the meanings of a construct in the language independent of the interpreter involved. Those aspects of language introduced by the consideration of what is interpreting the language are called pragmatic. The interesting point is that we have been able to devise a useful trichotomy between these three aspects of computer-oriented languages.

As we have mentioned, the syntax and semantics of computer languages are to be described in machine-independent formal systems. Machine dependence (pragmatics) occurs in the programs which utilize the formal descriptions. The following block diagram should help illustrate how formal syntax and semantics can be used in a compiler-compiler.

![Diagram](image-url)

Figure 1.
When a translator for some language, L, is required, the following steps are taken. First the formal syntax of L, written in the syntax meta-language, is fed into the Syntax Loader. This program builds tables which will control the recognition and parsing of programs in the language L. Then the semantics of L, written in the Formal Semantic Language, is fed to the Semantic Loader. This program builds another table which in turn contains a description of the semantics of L.

The Basic Compiler is a table driven translator based on a recognizer using a single pushdown stack. Each element in this stack consists of two machine words -- one for a syntactic construct and one holding the semantics of the construct. The syntax tables determine how the compiler will parse a source language program. When a construct is recognized, the semantic tables are used to determine what actions are to be taken.

The entire system is based on the formalization of syntax and semantics, but not on the particular representations chosen. The remainder of this thesis will be largely devoted to a discussion of the two meta-languages used in the compiler-compiler at Carnegie Institute of Technology. The system has been running since early in 1964 and translators for such languages as ALGOL, FORTRAN, LISP and MARKOV ALGORITHMS have been written.

Chapter II of this thesis discusses the syntax meta-language used in our system. This language, Production Language, is similar to systems used for different purposes by Floyd [15], Evans [13] and Graham [22].
The most interesting feature of these systems is the absence of the explicitly recursive processes which are found in most syntax directed compilers.

The Formal Semantic Language (FSL) is described in Chapter III. The discussion there is basically a user's guide to the FSL system. Part of the FSL description deals with the semantic loader of Figure 1. Each of the three basic parts of Figure 1; syntax loader, semantic loader, and basic compiler, is a complex computer program. In Chapter IV we describe how these were combined to form a compiler-compiler.

In Chapter V we review and criticize the first four chapters. Among the topics discussed are the strengths and weaknesses of our system, the quality of translators produced and opportunities for future research.

The appendices form an important part of the paper. In Appendices A-C we describe a subset of ALGOL 60 called the "small language". Appendix A consists of the Backus Normal Form syntax of the small language. Its syntax in Production Language is Appendix B and Appendix C contains its formal semantics written in FSL. Throughout Chapters II and III there are references to examples taken from these appendices. Appendices B and C were used just as they appear to build a compiler for the small language.

Appendices D and E contain a formal syntactic definition of the Formal Semantic Language itself. The Backus Normal Form syntax is
Appendix D and the Production Language syntax is Appendix E. The question of writing the FSL semantics in FSL is discussed in the section of Chapter V which deals with weaknesses of the system.

Appendix F contains several examples of small language programs and their machine language equivalents. Thus Appendices A, B, C, and F enable one to follow the operation of the compiler-compiler from abstract syntax to machine code for the small language.
II. SYNTAX

A. The Production Language

For most computer scientists, formal syntax is synonymous with Backus Normal Form or BNF. The use of BNF to represent the formal syntax of ALGOL 60 [31] had a marked effect on the field. Its simple and precise specification of ALGOL syntax was a great improvement over the unwieldy English descriptions associated with previous programming languages. Besides improving human communication, the use of formal syntax has led to new formal developments in the theory and practice of programming.

Theoreticians were able to show that BNF grammars were capable of generating the same class of languages as those generated by several other formalisms. From the theory of natural languages came the Context Free Grammars of Chomsky [6] and the Simple Phase Structure Grammars of Bar Hillel [1], both equivalent to BNF. A mathematician, Ginsburg showed that BNF was equivalent to a formalism he called Definable Sets [20]. In addition, many results on the formal properties of such systems were attained.

The most interesting practical development was the syntax-directed compiler. If the syntax of programming languages can be represented formally, it should be possible to construct a language-independent syntax processor for a translator. As early as 1960, Irons [25] was able to write a translator whose syntax phase was independent of the source
language being translated. This work and that of other early contributors such as Brooker and Morris [2] led to speculation that the entire translation process can be automated. This dissertation can be viewed as an effort in this direction.

With this weight of accomplishment behind BNF one might well question the usefulness of developing other syntax languages. Unfortunately, for several reasons BNF is inadequate as a universal syntax meta-language. We will present the theoretical weakness first and then discuss some practical difficulties and how they may be overcome.

As Floyd [18] has shown, Backus Normal Form is not capable of representing the language ALGOL 60 as defined by the English description in the report [31]. That is, not only is the BNF syntax in the ALGOL report inadequate to represent that language, but there is no BNF grammar that can do so. The concepts not expressible in BNF include relations imposed by the data types of declared variables and the number of parameters of subroutines. Since at least one of these constructs occurs in almost all programming languages, BNF is in some sense too weak to represent programming languages.

To make matters worse, there is a sense in which BNF grammars are too powerful. The languages generated by BNF grammars are sufficiently complex to have recursive undecidability play a significant role.
A problem in artificial language theory is undecidable if there is no single algorithm which will produce a solution in a finite time for any language in the class [10]. Bar Hillel [1] has shown that the following problems are undecidable for BNF grammars:

1) Do two grammars generate the same language?

2) Is the language generated by one grammar contained in the language generated by a second grammar?

3) Does a BNF grammar generate all (or all but a finite number) of the possible strings over its alphabet?

In addition, it is undecidable whether a single BNF grammar is ambiguous, i.e., if it generates the same sentence in more than one way [44]. These are questions one would like to be able to answer about programming languages and the results above indicate that, in general, this cannot be done for languages with BNF grammars.

The practical limitations on BNF grammars stem from their inherently recursive nature. The elegance of the recursive notation indicates implied processing, which on computers means slow processing. Although there are no proofs of such things, it is generally agreed (cf. [5], [22], [42]) that recursive recognizers are slower processors than the type described below. Intuitively, the speed difference arises from the housekeeping details incurred in recursive processing. There are many applications where greater human convenience compensates for this difference, but translator design does not seem to be one of these. Furthermore, error
detection and recovery is more difficult in recursive recognizers [5]. A survey of the relation of formal syntax to programming may be found in an article by Floyd [17].

It was also Floyd who introduced an alternative notational technique, that of productions, into language analysis [16]. Like BNF, this production notation was originally devised to improve communication between people. It was so useful in representing symbol manipulation algorithms that it was natural to make a programming language from it. When an effort to implement ALGOL 60 was undertaken at Carnegie Tech, this notation was chosen for the coding of the syntax phase of the translator [13].

A good deal is now known about the theoretical and practical aspects of Production Language and other recognizer-oriented grammars. The theory [15] and practice [22] of bounded context grammars has received the greatest attention. Our Production Language system differs from those described elsewhere and thus will be presented in some detail below. The remainder of this section deals with the use of our system while its formal properties are discussed in Section II-B.

The Production Language, PL, is based on the model of a recognizer with a single pushdown stack. The syntax of a source language, when written in PL, will specify the behavior of the recognizer as it scans an input text in that source language. As a character is scanned it is brought into the stack of the recognizer. The symbol configuration at the top of this
stack is compared with the specification of the source language being translated. When the characters in the stack match one of the specified configurations for the source language, certain actions are performed on the stack. If a match is not attained the stack is compared with the next PL statement. This process is repeated until a match is found or until the stack is discovered to be in an error state. The functioning of the recognizer will be discussed in connection with the example at the end of this section.

The format of a statement (production) in PL is given below.

```
Label L5 L4 L3 L2 L1 | → R3 R2 R1 | Action Next
```

In this format the first vertical bar from the left represents the top of the stack in the recognizer. The symbols L5...L1 in a given production will be characters which might be encountered in translating the source language. The symbol in position L1 is at the top of the stack. Only the top \( n \) characters (\( 1 \leq n \leq 5 \)) need be present, the others being blank. If only three symbols are specified in some production, it will be compared with the three top positions of the stack during translation. The symbols L5...L1 constitute the "specified configuration" mentioned in the preceding paragraph. Any of L5---L1 may be the character '0' which represents any symbol and is always matched.

A match occurs when the actual characters in the stack during translation are the same as L5...L1 in the production being compared. At this
time the remainder of the information in that production is used. The '→' following the bar specifies that the stack is to be changed. The symbols R₃...R₇ specify the symbols which must occupy the top of the stack after the change. If no '→' occurs then the stack will not be changed and R₃...R₇ must be blank. Thus L₅...L₁ refer to the top of the stack before and R₃...R₇ to the top of the stack after a production is executed which changes the stack.

An example of a production containing a '→' is found on line 11 of the fragment example at the end of this section. If this production is matched the characters on top of the stack will be

\[ T * P \]

The execution of line 11 will leave the top of stack in the configuration

\[ T \]

Such a production would be used in a language such as ALGOL to recognize a multiplication.

In the 'action' field of the production on line 11 there occurs 'EXEC 2'. All actions are performed after adjusting the stack as described above. The execution of EXEC 2 initiates the call of a semantic routine. In this case the semantic routine would cause the multiplication of the fourth and second elements in the stack. The value of the result would then be put in the second position in the stack and translation continued.
The semantic routines themselves are written in a special purpose language discussed in Section III.

The last part of the production in line 11 is the symbol 'T2' in the next field. This means that the next production to be compared with the stack is the one labeled 'T2'. (It is on line 13.)

The production on line 13 is an example of a production in which the stack is not changed. If this line is matched the symbol at the top of the stack is '*' denoting multiplication. At this point the translator requires another operand before the multiplication can be performed. Since no action can be taken, the production on line 13 does not change the stack and its action field is blank. In the label field of this production is '*Pl'. The asterisk is this position is a special symbol of the Production Language and specifies that a new character is to be brought into the stack before transferring to the production labeled Pl. The asterisk is equivalent to the command "SCAN" which may occur in the action field of a production. These are the only ways that a new character can be introduced into the stack.

Among the other commands which can appear in the action field of a production are EXEC, ERROR, and HALT. The action EXEC is the main link between syntax and semantics in our system and will be discussed in detail below. The action ERROR causes its parameter to be used in printing an error message for the program being translated. An execution of
HALT causes translation to end and, if there have been no errors, the running of the compiled code.

There is one feature of Production Language which does not appear in the syntax of the language fragment. It is possible to specify classes of symbols and use the class name in a production. This technique allows the user to replace many similar productions with one production. In Appendix B the class <TP> consists of the reserved words REAL, BOOLEAN, and LABEL. In the productions under D3, the class name <TP> is used in processing declarations in the small language, each production using <TP> replacing three productions.

The remainder of this section will be a discussion of the example on page 20. The language fragment description in the example is a subset of the syntax of arithmetic expressions in a language such as ALGOL 60.

The mnemonic significance of the letters in the example is related to the ALGOL syntax and is given in the following table.

<table>
<thead>
<tr>
<th>Letter</th>
<th>May be Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Identifier</td>
</tr>
<tr>
<td>P</td>
<td>Primary</td>
</tr>
<tr>
<td>T</td>
<td>Term</td>
</tr>
<tr>
<td>E</td>
<td>Expression</td>
</tr>
</tbody>
</table>
The parentheses, asterisk and minus sign have their usual meanings. There are no productions to recognize identifiers because this is done in a pre-scan phase. We will assume that the production syntax of the rest of the language is compatible with the fragment.

The productions under the label P1 (lines 7-10) are used when a primary is expected. Line 7 is matched if the primary is preceded by a unary minus sign. In this case the minus is left in the stack where it will be processed by line 15. Similarly, the parenthesis in line 8 is left in the stack to be processed by line 18 when its mate is scanned.

In line 9 an identifier is recognized and changed to a primary. Then EXEC 1 will initiate a call of a semantic routine which will see if the identifier is properly defined and, if so, will assign the value associated with that identifier to the primary.

Any other character appearing in the stack at this point is a syntactic error. This is expressed in line 10 where the action field contains an error message.

Line 11 recognizes a multiplication and has been discussed earlier. Line 12 (also 16) only changes the name of the symbol in the stack. This device is used to enable the productions to realize the rule of arithmetic that requires multiplication to bind more tightly than subtraction. The production in line 13 embodies this implementation. If a term is followed by an '*', there is a multiplication which must be performed
before that term is used in any subtraction. For this reason line 13 brings a new symbol into the stack and causes a transfer to Pl to pick up the other operand for that multiplication.

The productions under El (lines 14-16) recognize subtractions and form the final expressions. Those under E2 (line 17-20) process an expression once it is recognized. If the expression is followed by a minus, it is part of a larger expression so line 17 returns control to Pl to process the remainder. The production on line 18 embodies the rule that all operations within parentheses must be executed before the parenthesized expression acts as an operand. If the fragment were imbedded in a full language (e.g., Appendix B) other uses of arithmetic expressions would be checked for under E2. In the fragment, expressions are used in just the two ways described above so that any other character in the stack leads to an error condition.

The label Q1 is assumed to lead to an error recovery routine. For an example of a sophisticated error recovery scheme using production language see reference [13].

The production syntax for a complete programming language is included as Appendix B of this paper. The Backus Normal Form syntax of the same language is Appendix A. A word is in order regarding the equivalence of a BNF and a PL grammar. Nowhere in this paper is there an attempt to present a formal proof of the equivalence of two such grammars. The writing of
productions is in essence a constructive programming task and discrepancies should be treated as 'bugs' in a program. To the best of our knowledge the examples are all debugged.

The task of writing productions is not a difficult one. The productions for the small language (Appendix B) were assigned, with good results, as a homework problem in an undergraduate programming course. In fact, one of the students wrote a program in IPL-V which converts a BNF specification of a language into productions for a large class of languages. This work will be reported in a forthcoming paper.
BNF Syntax of a Language Fragment

1. $<I> ::= <letter> | <I> <letter> | <I> <digit>
2. $<P> ::= <I> ( <E> )
3. $<T> ::= <P> | <T> * <P>
4. $<E> ::= <T> | <T> | <E> - <T>

Production Syntax for the Same Fragment

<table>
<thead>
<tr>
<th>LABEL</th>
<th>LEFT</th>
<th>RIGHT</th>
<th>ACTION</th>
<th>NEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.</td>
<td>P1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>(</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>T1</td>
<td>T</td>
<td>T</td>
<td>EXEC 2</td>
</tr>
<tr>
<td>12.</td>
<td>P</td>
<td>T</td>
<td>T</td>
<td>T2</td>
</tr>
<tr>
<td>13.</td>
<td>T2</td>
<td>T</td>
<td>T</td>
<td>*P1</td>
</tr>
<tr>
<td>14.</td>
<td>E1</td>
<td>-</td>
<td>E</td>
<td>EXEC 3</td>
</tr>
<tr>
<td>15.</td>
<td>-</td>
<td>T</td>
<td>E</td>
<td>EXEC 4</td>
</tr>
<tr>
<td>16.</td>
<td>T</td>
<td>T</td>
<td>E</td>
<td>E2</td>
</tr>
<tr>
<td>17.</td>
<td>E2</td>
<td>E</td>
<td>-</td>
<td>*P1</td>
</tr>
<tr>
<td>18.</td>
<td>(</td>
<td>E</td>
<td>P</td>
<td>EXEC 5</td>
</tr>
<tr>
<td>19.</td>
<td></td>
<td></td>
<td></td>
<td>ERROR 2</td>
</tr>
</tbody>
</table>

Figure 2.
B. Formal Properties

The Production Language used to describe the syntax of various source languages is a formal grammar. There has been much research done on the mathematical properties of such formal grammars and in this section we will discuss the relationship between Production Language and some well-known grammars.

Most of the formal grammars whose properties have been studied are generator-oriented. A generator-oriented grammar for a language, L, is a set of rules for generating the sentences of L. The generation rules are applied recursively to strings of characters, starting with a fixed initial symbol. Various schemes have been devised to classify these grammars according to the complexity of the languages they specify. We will follow the terminology of Chomsky [6] in describing a hierarchy of languages types.

In his definitive paper, Chomsky describes four types of grammar and shows that each is capable of generating more complex languages than its successor in the hierarchy. The most general grammar he considers in Type 0, which allows as a grammar essentially any finite set of rules for transforming character strings. The other types are obtained by placing various restrictions on the permissible rules.

In TYPE 1 grammars, the restriction is that each application of a generation rule does not decrease the length of the sentence being
generated. The TYPE 2 grammars may be characterized by the restriction that each rule specifies the replacement of a single character, independent of its context. The weakest grammars, TYPE 3, allow only those replacements which effect a transformation on only one end of the sentence being generated. Chomsky gives a precise definition of those four types, establishes the hierarchy, and then relates these formalisms to some well-known mathematical constructs.

The class of TYPE 0 grammars is shown to be equivalent to the class of all Turing machines, i.e., for any computations performable on a Turing machine there is a TYPE 0 grammar which will perform equivalent string transformations. Turing machines, and thus TYPE 0 grammars, are capable of executing any computable function. (cf. Davis [10])

TYPE 1 grammars are called context-sensitive because the replacement rules may depend on the context of the character being replaced. These have been used less than the other types, but do have the interesting property that they are decidable, i.e., for any TYPE 1 language there exists an algorithm for determining if a given string in its alphabet is a sentence of that language.

The TYPE 2 or context-free languages have received the widest attention in the literature. These are equivalent to the formalisms: Simple Phrase Structure Grammars [1], Definable Sets [20], and to Backus Normal Form [32]. It is this last equivalence which has made TYPE 2 grammars
interesting to workers in programming languages. Despite its theoretical inadequacies, Backus Normal Form has established itself as a canonical form for the description of syntax of artificial languages. In this section, we will discuss the formal properties of the Production Language as compared with those of Backus Normal Form.

In this discussion we will make use of results by Chomsky [7] and Evey [14]. In independent efforts, they attempted to construct automata which would be equivalent to Backus Normal Form. We will follow Chomsky in presenting the definition of a Push Down Store Automaton. This automaton is recognizer-oriented and will, in fact, be shown equivalent to the Production Language. A pushdown store automaton, M, consists of a finite state control unit and two unbounded tapes, an input and a storage tape. The control unit can read symbols of the input alphabet $A_I$ from the input tape and the symbols of the output alphabet $A_O \supset A_I$ from the storage tape. It can also write the symbols of $A_O$ on the storage tape. If M is in the state $S_i$ scanning the symbols a on the input tape and b on the storage tape, we say that is in the situation $(a, S_i, b)$. There is an element $\_\#$ which cannot appear on the input tape nor be written on the storage tape. At the beginning of a computation, M is in the state $S_0$, with the storage tape containing $\_\#$ in every square and the input tape containing a string $b_1 \ldots b_n$ in the first n squares. After $b_n$, the input tape has a special symbol $\#$ in every square. A computation ends when the M returns to the state $S_0$. If at this time M is in the situation $(\_\#, S_0, \#)$ we say it
accepts (recognizes) the string $b_1 \ldots b_n$.

A computation of $M$ is controlled by a set of rules of the form

$$ (a, S_i, b) \rightarrow (S_j, X, k) $$

where $a \in A_1$, $b \in A_0$; $X$ is a string in $A_0$ and $S_i, S_j$ are states of the control unit. If $X = \lambda$, then $k = 0$ or $-1$, if $X \neq \lambda$, then $k = \text{length of } X$. We can assume, without loss of generality that $b = \lambda$ only if $S_i = S_0$ and $K \leq 0$. Then the instruction (1) applies when $M$ is in the situation $(a, S_i, b)$ and has the following effect: The control unit switches to state $S_j$, one symbol is removed from the end of the input tape and the storage tape is moved $k$ squares left, with $X$ being filled into these squares. After the instruction, $M$ is in the state $(c, S_j, d)$ where $c$ is the symbol to the right of $a$ on the input tape and $d$ is the rightmost symbol of $X$. If $k$ is $-1$ the storage tape is shifted one to the right, effectively erasing the symbol which was rightmost on this stack. In the special case that $a = \lambda$, the contents of the input tape are ignored and that tape does not move.

We will now show that each pushdown store automaton is equivalent to a program in the Production Language of Section II. For this purpose we first rename some of Chomsky's constructs. The storage tape will be called the stack. The input tape is just a device for holding the input text and will not be mentioned explicitly.
For each state $S_i$ in $M$ we set up a label $S_i$ in PL and then associate all situations in state $S_i$ with productions occurring under the label $S_i$. Then we make a transformation on the format of the instructions of $M$. The situation on the left of an $M$-instruction looks at the top of the stack and the next input symbol. Since all input symbols are contained in the output alphabet, it is equivalent to look at the top two characters in the stack. Then we could write an $M$-rule as

$$ (2) \quad (ba, S_i) \rightarrow (S_j, X, K) $$

Now notice that the parameter $k$ is redundant except when $X$ is null and in this case it is either 0 or -1. In the latter case, the corresponding production would either leave the stack alone or execute a production which removes the top element of the stack. Then eliminating $k$ and remembering that labels in productions correspond to states in $M$, we get the rule

$$ (3) \quad S_i : b a \rightarrow X \quad *S_j $$

which is almost in production language form. The remaining difference is that $X$ may have more than three characters in $M$ and that productions may look at as many as five characters on the left. In either case, the longer symbol string can be replaced by a succession of shorter replacements by adding extra symbols to $A_0$ in the usual manner.
To complete the correlation between productions and pushdown store automata we need two other facts. In productions one can have a rule using ‘\(\epsilon\)’ which is always recognized - the automaton M allows the symbol \(a_0\) to appear in a rule specifying that any character will match that rule. Further, in this case the input string does not move, corresponding to a production rule without ‘\(*\)’ in the label field.

The preceding paragraphs outline a procedure for converting a pushdown store automaton into a production language program. Since all of the steps are reversible, the two concepts are equivalent in the sense that they will recognize the same class of languages. This is of interest here because of some known results on the formal properties of pushdown store automata. We will present Chomsky's main theorem, which is apparently a complete characterization of Production Language, and then show why this is not the case.

Theorem (Chomsky) A language \(L\) is accepted by a pushdown store automaton if and only if it is a TYPE 2 language.

In the proof of this theorem Chomsky first reduces the replacement rules of any TYPE 2 grammar to a simple canonical form. Then, through an intermediate step, he is able to invert each replacement rule and produce a set of recognition rules which specify the equivalent automaton. This would, indeed, solve the problem if it were not for the following difficulty.
In the inversion process it may happen that many automaton instructions with the same left side are formed. There is no specification of which rule is to be executed first in such a case. What the theorem states is that, in each case, there is a choice which will lead to the recognition of a particular TYPE 2 sentence. Chomsky describes this as the non-deterministic operation of an automaton and does not concern himself with the implementation of such a device.

For us, however, implementation is a key issue. To implement such a scheme would require another storage tape to hold temporary information while each alternative was tried. The maximum amount of storage is a function of the length of the input string and the complexity of the automaton. Whether such an automaton could be weaker than a Turing machine is an interesting question which we do not pursue here. These non-deterministic automata are related to the multiple-path syntax analyzers used by Kuno and Oettinger [27] on natural languages. For artificial languages where syntax is more precise, we consider non-deterministic automata too inefficient to be of interest.

A question arises as to whether there are any TYPE 2 languages which actually require a non-deterministic recognizer. One well-known example of such a language is the odd palindromes over the alphabet '0' and '1'. The syntax of this language in Backus Normal Form is:

\[ W ::= 0 | 1 | 0WO | 1WL \]
where $W$ names the set of well-formed sentences. A pushdown store automaton for this language would require non-deterministic control.

The results of the preceding paragraphs may be summarized as follows. There is a sense in which the Production Language is equivalent to Backus Normal Form. This formal equivalence is unsatisfactory because we are not willing to use productions in the inefficient manner specified by the equivalence theorem. Further, there are simple languages which are expressible in Backus Normal Form and which cannot be efficiently recognized by productions.

There is, however, a construct in Production Language which we have not yet mentioned. This device wasn't used in any of the example in Section II-A and apparently would not add to the formal power of the language. The extra construct is simply the ability to call any labeled set of productions as a subroutine. This is done by placing a statement of the form

\begin{verbatim}
SUBR L
\end{verbatim}

in the 'action' field of a production. The parameter 'L' is the label of the set of productions being called. The action RETU is executed at the end of a subroutine. These subroutine calls are allowed to be recursive; a subroutine can contain a call of itself.
It is the last feature which indirectly introduces the extra power into Production Language. Since a subroutine may be called many times before it finishes once, a stack of return addresses (marks) must be kept. As is well known, one stack can keep the marks for all the subroutines. For simplicity we will assume that all implementations of Production Language are of this form. The crucial point is that there may be any number of nested subroutine calls so this stack of marks must be unbounded.

Now, as Evey [14] has shown, an automaton with two unbounded stacks is equivalent to Turing machine. This is intuitively clear if we picture the two stacks as the two ends of a tape which is unbounded in either direction. Then the Turing machine operation of moving the tape is paralleled by switching a character from one stack to the other. There is one difficulty with this picture in the case of the Production Language. Here one of the stacks contains symbols of $A_0$ and the other stack only marks for subroutines. This difficulty is circumscribed by what can only be called a programming device.

We add to a Production Language program two sets of productions for each symbol of $A_0$. The first set will have the effect of removing the top character from the stack and leaving an address in the mark stack. Thus when a character is to be moved to the second stack a unique address is put there in its stead. Now, when a symbol is to be put back in the main stack, the action RETU is executed. This causes a transfer to the 'next'
field of the production which called the subroutine. The label in the 'next' field will lead to a production which puts the appropriate symbol back in the main stack. Thus we have established a 1:1 correspondence between symbols in the main stack and addresses in the mark stack. This correspondence and the two transfer functions allow us to treat the two stacks jointly as the tape of a Turing Machine.

In this section we have attempted to define the formal properties of the Production Language. By varying slightly the side conditions on the use of productions we arrived at three very different characterizations of their power as an automaton. There are two important lessons to be learned from these examples. The first, and most important, concerns the relation of actual computer techniques and the formal structures used to model them. Since the formal model is precise and relatively tractable, there is a tendency to discuss the model as if it were the system itself. The examples of this section show that there may be several formal models, with quite different properties, of one programming construct.

The other point to be made is a practical one. We were able, through clever programming, to make Production Language equivalent to Turing machines. The problem is that one would rarely use productions in the manner required in the demonstration for translating a programming language. We have shown that the power is in the system, but have not shown how to use this power effectively. It would be interesting to see if there were some practical way to utilize the extended production language
to develop efficient recognizers for more complex programming languages.

In the remainder of this paper, productions will only be used in the weakest of the three ways described here. Although it is theoretically possible to do complete syntax checking with productions, we have found it more efficient to defer some of the checking to the semantic phase. An interesting side effect of this decision is that it is impossible to specify an ambiguous language using productions in the weak sense. We will consider some of the practical problems connected with production grammars in Chapters IV and V.
Page 32

blank

(Reverse side of page 31)
the end of a format statement.

The second addition to the source language description occurs in the semantics. A format statement specifies how a construct is to be printed or read but does not determine which machine location is to hold the data. There is, in FSL, a special run-time stack, \texttt{NAME}, in which the input-output processor expects to find this information. In his semantic description, the FSL user must have a statement which fill the stack \texttt{NAME} with the locations to be processed by the format routines.

With these two additions to a source language description, the designer buys the full power of a sophisticated input-output language. The format language is now available only in the source languages, but could easily be added to FSL itself if this proved desirable. The 'easily' results from the fact that the system has been implemented in such a way that the Basic Compiler and the FSL translator are very similar. A discussion of how this similarity was attained and the advantages it provides are contained in the next section.
at both the meta-compile and compile-time levels. The format language used in the FSL system is also closely modeled on the ALGOL format.

Because of these simplifying features, the entire implementation took less than six man months. The two basic components of the system are a syntax directed compiler and a Production Loader, each of which is used in two places. An ALGOL 60 version of the Production Loader will be available from A. Evans [12] later this year. Basing our system on existing programs involved certain compromises. The effect of these compromises will be discussed as part of the review and critique in the next chapter.


Appendix B

Productions for the Small Language

SYMBOLS

12 INTERNAL SYMBOLS

- I P F T E SBE BE ICL UN S HEAD

+ - * / = < > ^ ~ + _ , ; : + $ ( )

REAL BOOLEAN LABEL BEGIN END GOTO IF THEN ELSE TRUE FALSE

METACHARACTERS
S0  BEGIN | EXEC 1 | *D1
    <SG> | ERROR 0 | Q1
D1  <TP> | SCAN | *D2
    BEGIN | *S1
    END | *D1
    BEGIN | <TP> | SCAN | *D2
    BEGIN | <TP> | EXEC 2 | S1
    END | <TP> | EXEC 2 | S1
D2  REAL | EXEC 4 | D3
    <SG> | EXEC 5 | D3
    BOOL | EXEC 30 | D3
    <SG> | 
D3  <TP> | EXEC 3 | S1
    I = <TP> | EXEC 4 | D3
    I = <TP> | EXEC 5 | D3
    <TP> | EXEC 9 | *F1
    I = <TP> | EXEC 10 | T2
    I / <TP> | EXEC 11 | T2
    F = <TP> | EXEC 12 | E2
T1  T = F <SG> |  E <SG> | EXEC 13 | E2
    T / F <SG> |  E <SG> | EXEC 14 | E2
    F <SG> |  E <SG> |  E <SG> | E2
T2  T <TD> |  E <SG> |  E <SG> | E2
    T <TD> |  E <SG> |  E <SG> | E2
E1  E + T <SG> |  E <SG> | EXEC 15 | *F1
    E - T <SG> |  E <SG> | EXEC 16 | UN1
    E - T <SG> |  E <SG> | EXEC 17 | B2
E < E <SG> | → SBE <SG> | EXEC 18 B2
E > E <SG> | → SBE <SG> | EXEC 19 B2
E <RL> E <SG> | → SBE <SG> | EXEC 17 B2
<SG> | → ERROR 8 Q1
B1
IF | → *B1
+ | → *
P1
- | → *
P1
( | → *
P1
B4
I <OP> | → P <OP> | EXEC 8 F1
I <RL> | → E <RL> | EXEC 8 *P1
I <SG> | → SBE <SG> | EXEC 20 B2
I | → *
P1
B3
SBE ELSE | → BE <SG> | EXEC 22 B3
B3
IF BE THEN | → ICL | EXEC 21 I1
ICL SBE ELSE BE <SG> | → BE <SG> | EXEC 22 B3
I I + BE <SG> | → UN <SG> | EXEC 23 UN1
EX1
I | → *
IF | → *EX2
+ | → *
P1
( | → *
P1
EX2
I <OP> | → P <OP> | EXEC 8 F1
I <RL> | → E <RL> | EXEC 8 *P1
I I + I <SG> | → UN <SG> | EXEC 24 UN1
<SG> | → ERROR 8 Q1
G1
GO I | → UN | EXEC 25 *UN1
<SG> | → ERROR 9 Q1
S9 ICL UN ELSE S <SG> | → S <SG> | EXEC 27 S9
ICL UN ELSE UN <SG> | → S <SG> | EXEC 27 S9
HEAD J | S | → HEAD ; |
HEAD ; S END | → UN | EXEC 29 ND1
ND1
I | → UN | → I | → EXEC 31 Q1
<SG> | → *
Q1
END

OTHER STUFF
I 1 4
I 2 0
I 3 0
I 4 1
I 5 0
I 6 14
I 7 61
I 8 67
I 9 0
I10 56
I11 67
I12 12
I13 12
I14 5
I15 0
I16 26
Notes on Appendix C

This is a complete run of the semantics for the small language. All of the locations are in octal form and the meaning of the mnemonic opcodes is given at the beginning of the appendices. The semantic tables start at 65600 and extend to 66650. The table of addresses starting at 37000 is called the switching table. Each entry in this table is the first location of the code for a semantic routine. For example, the code for 12 (octal 14) starts at 66101 as described in cell 37014. The switching table is used by the compiler in executing semantic routines.

As an example we will consider the code generated for

\[12 \downarrow \text{CODE(VALUE2 } \leftarrow \text{LEFT4 + LEFT2)} \downarrow\]

The first command increments a pointer because we are entering code brackets. The next four commands put LEFT4 (63337) and LEFT2 (63335) into the parameter region. Then the generator for '+' is called by 'TRM 0 63405'. The next two commands set up and call the processor for 'VALUE2' which will adjust the compile time stack and put an accumulator symbol in RIGHT2. Finally, the semantic routine returns control to the basic compiler at 62110.

Tables and other storage for the translator start at 45777 and go down in memory. Addresses between 62000 and 65000 refer to routines in the basic compiler. Locations near 14400 are used for constants and any address below 10000 is in the monitor, except for the index registers 0-77.
## APPENDIX C

### FORMAL SEMANTICS OF THE SMALL LANGUAGE

<table>
<thead>
<tr>
<th>SN</th>
<th>DUMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BEGIN TABLE Symb[200,4]</td>
</tr>
<tr>
<td>2</td>
<td>T3 = TEMPLOC</td>
</tr>
<tr>
<td>3</td>
<td>LEV = 0; Str = STORLOC; Sym = Loc(Symb)</td>
</tr>
<tr>
<td>4</td>
<td>LEV = 0; Push[Str,STORLOC]; Push [Sym, Loc(Symb)]$; Tally[LEV]</td>
</tr>
<tr>
<td>5</td>
<td>Push[FLAD1,0]; Code( -LEFT1 = JUMP[FLAD1]$)</td>
</tr>
<tr>
<td>6</td>
<td>T0 = STORLOC; Set(T0,DOUBLE); Enter(Symb;LEFT2,T0,REAL,LEV); STORLOC = STORLOC+2</td>
</tr>
<tr>
<td>7</td>
<td>Enter(Symb;LEFT2,STORLOC,Bool ,LEV);Tally(STORLOC)</td>
</tr>
<tr>
<td>8</td>
<td>Push[FLAD2,0]; Code( JUMP[FLAD2]); Assign[FLAD1]</td>
</tr>
<tr>
<td>9</td>
<td>/ Assign[FLAD1]</td>
</tr>
<tr>
<td>10</td>
<td>Const[LEFT2] = RIGHT2 = LEFT2 : Sym[LEFT2,$,1] = REAL = RIGHT2 = Sym[LEFT2,$,$] : Fault 1$</td>
</tr>
<tr>
<td>11</td>
<td>Const[LEFT1] = RIGHT1 = LEFT1 : Sym[LEFT1,$,1] = REAL = RIGHT1 = Sym[LEFT1,$,$] : Fault 1$</td>
</tr>
<tr>
<td>12</td>
<td>Code(VALUE2+LEFT4*LEFT2)</td>
</tr>
<tr>
<td>13</td>
<td>Code(VALUE2+LEFT4/LEFT2)</td>
</tr>
<tr>
<td>14</td>
<td>Code(VALUE2+LEFT4+LEFT2)</td>
</tr>
<tr>
<td>15</td>
<td>Code(VALUE2+LEFT4-LEFT2)</td>
</tr>
<tr>
<td>16</td>
<td>Code(VALUE2-LEFT2)</td>
</tr>
</tbody>
</table>
154 RIGHT1<LEFT2

155 Symb[LEFT4,$,]=REAL+ ComT 2+ SYMB[LEFT4,$,]
   Code(ComT 2+ LEFT2); TempLoc + T3; Fault 3$ 

174 Code(VALUE2+LEFT4=LEFT2)

184 Code(VALUE2+LEFT4<LEFT2)

194 Code(VALUE2+LEFT4>LEFT2)

204 Const[LEFT2] + RIGHT2 + LEFT2:
   Symb[LEFT2,$,]=BOOL+RIGHT2+SYMB[LEFT2,$,];
   Set(RIGHT2, Logic); TempLoc + T3; Fault 3$$

214 RIGHT1+LEFT2

224 Code( LEFT5+ TempLoc+ LEFT4; TempLoc+ LEFT2$);
   RIGHT2+ TempLoc; Set(RIGHT2, Logic);
   Tally(TempLoc)

234 Symb[LEFT4,$,]=BOOL+ ComT 2+ Symb[LEFT4,$,];
   Code( ComT 2+ LEFT2); Fault 4$

244 ComT 4+ Symb[LEFT4,$,];
   Const[LEFT2] + ComT 2+ LEFT2;
   Symb[LEFT2,$,]=Symb[LEFT4,$,] + ComT 2+ Symb[LEFT2,$,];
   Fault 5 $ $; TempLoc + T3;
   Code( ComT 4+ ComT 2)

254 Symb[LEFT1,$,] ≠ LabE + Fault 6:
   ComT 2+ Loc[Symb[LEFT1,$,]];  
   ComT 3+ <ComT 2> ; Symb[LEFT1,$,]$ ≠ 0 + Code( Jump[ComT 3] ) ;  
   Code( Jump[Chain[ ComT 2 ] ] ) $$

255 Symb[LEFT2,$,] ≠ LabE + Fault 6:
   Symb[0,$,]$ + 1 ;  
   Assign[Loc [ Symb[ LEFT2,$, ] ] ] $

274 Assign[FLA1V2]

284 Code( VALUE2 + LEFT4 + LEFT2)

294 MINUS[LEV]; Pop[STR,STORLoc]; Pop[SYM,Loc[Symb]]

304 ENTER[Symb]=LEFT2,0,LabE,0 ...
Appendix E

Productions for the Formal Semantic Language

* SYMBOLS
* 18 INTERNAL SYMBOLS
  I
  OP
  AP
  AF
  AT
  AE
  TL
  LO
  BP
  BS
  BF
  BE
  IC
  UN
  UQ
  S
  SQ
  +
  -
  *
  /
  =
  #
  <
  >
  ~
  ^
  *
  ;
  :
  
BEGIN
END
STOP
STACK
CELL
TITLE
TABLE
DATA
BEGN
END
STOP
STAK
CELL
NAME
TABL
DATA
<table>
<thead>
<tr>
<th>JUMP</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARKJUMP</td>
<td>TM</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>X</td>
</tr>
<tr>
<td>ENTER</td>
<td>E</td>
</tr>
<tr>
<td>PUSH</td>
<td>ST</td>
</tr>
<tr>
<td>POP</td>
<td>RS</td>
</tr>
<tr>
<td>CHAIN</td>
<td>CH</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>UC</td>
</tr>
<tr>
<td>CODE</td>
<td>CD</td>
</tr>
<tr>
<td>FAULT</td>
<td>FALT</td>
</tr>
<tr>
<td>TALLY</td>
<td>TAL</td>
</tr>
<tr>
<td>MINUS</td>
<td>MIN</td>
</tr>
<tr>
<td>SET</td>
<td>SET</td>
</tr>
<tr>
<td>TEST</td>
<td>TEST</td>
</tr>
<tr>
<td>INT</td>
<td>IN</td>
</tr>
<tr>
<td>LOC</td>
<td>L</td>
</tr>
<tr>
<td>CODELOC</td>
<td>A</td>
</tr>
<tr>
<td>STORLOC</td>
<td>W</td>
</tr>
<tr>
<td>TEMPOLOC</td>
<td>TO</td>
</tr>
<tr>
<td>COMT</td>
<td>CT</td>
</tr>
<tr>
<td>HUNT</td>
<td>RT</td>
</tr>
<tr>
<td>VALUE1</td>
<td>V1</td>
</tr>
<tr>
<td>VALUE2</td>
<td>V2</td>
</tr>
<tr>
<td>VALUE3</td>
<td>V3</td>
</tr>
<tr>
<td>LEFT1</td>
<td>Y1</td>
</tr>
<tr>
<td>LEFT2</td>
<td>Y2</td>
</tr>
<tr>
<td>LEFT3</td>
<td>Y3</td>
</tr>
<tr>
<td>LEFT4</td>
<td>Y4</td>
</tr>
<tr>
<td>LEFT5</td>
<td>Y5</td>
</tr>
<tr>
<td>RIGHT1</td>
<td>Z1</td>
</tr>
<tr>
<td>RIGHT2</td>
<td>Z2</td>
</tr>
<tr>
<td>RIGHT3</td>
<td>Z3</td>
</tr>
<tr>
<td>FLAD1</td>
<td>FL1</td>
</tr>
<tr>
<td>FLAD2</td>
<td>FL2</td>
</tr>
<tr>
<td>FLAD3</td>
<td>FL3</td>
</tr>
<tr>
<td>FLAD4</td>
<td>FL4</td>
</tr>
<tr>
<td>SIGNAL</td>
<td>SIG</td>
</tr>
<tr>
<td>HSTAK</td>
<td>RSTK</td>
</tr>
<tr>
<td>HTABL</td>
<td>HTAB</td>
</tr>
<tr>
<td>CONST</td>
<td>CON</td>
</tr>
<tr>
<td>CLEAR</td>
<td>CLE</td>
</tr>
<tr>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>PLACE</td>
<td>PLAC</td>
</tr>
<tr>
<td>DEPTH</td>
<td>DEP</td>
</tr>
<tr>
<td>SAR</td>
<td>SAR</td>
</tr>
<tr>
<td>SIN</td>
<td>SIN</td>
</tr>
<tr>
<td>COS</td>
<td>COS</td>
</tr>
<tr>
<td>EXP</td>
<td>EXP</td>
</tr>
<tr>
<td>LN</td>
<td>LN</td>
</tr>
<tr>
<td>SORT</td>
<td>SORT</td>
</tr>
<tr>
<td>ARCTAN</td>
<td>ARC</td>
</tr>
<tr>
<td>SIGN</td>
<td>SIGN</td>
</tr>
<tr>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALS</td>
</tr>
</tbody>
</table>

*METACHARACTERS*
CH | | | SCAN | *AP1
CT | | | | *B1
RT | | | | *B1
A | → | AP | EXEC 14 | *AF1
W | → | AP | EXEC 15 | *AF1
TLOC | → | AP | EXEC 16 | *AF1
<CO> | | | SCAN | *AP1
<SR> | → | AE | EXEC 18 | *AE3
<y> | → | OP | EXEC 19 | *OP1
<z> | → | AP | EXEC 20 | *AF1
< | | | | *AP1
PLAC | → | AE | EXEC 49 | *AE2
<FL> | → | AP | EXEC 22 | *AF1
DEP | → | AP | EXEC 66 | *AF1
<SG> | | | | ERROR 18 01
AF1 AF * AP <SG> | → | AF <SG> | EXEC 21 AF2
AP - | | | | *EX1
AP <SG> | → | AF <SG> | AF2
AF + | | | | *AP1
AT1 AT <TD> AF <SG> | → | AT <SG> | EXEC 23 AT2
AT <SG> | → | AT <SG> | AT2
AT2 AT <TD> | | | | *AP1
AE1 AE <PM> AT <SG> | → | AE <SG> | EXEC 24 AE2
+ AT <SG> | → | AE <SG> | AE2
- AT <SG> | → | AE <SG> | EXEC 25 AE2
<SG> | | | ERROR 8 01
AE2 AE <PM> | | | | *AP1
AE | | | | *AE3
< AE > | | | OP | EXEC 26 *OP1
AE <RL> | | | | *AP1
( AE ) | → | AP | EXEC 27 *AF1
TL [ AE | | TL AE | | T0
| AE | | | AE | EXEC 27 | *AP1
E TL AE | | | E TL | EXEC 28 | *EX1
AE <RL> AE <SG> | → | BP <SG> | EXEC 29 BS1
AP + AE <SG> | → | UN <SG> | EXEC 30 UN
AV + AE <SG> | → | UN <SG> | EXEC 31 UN
AE <SG> | → | BP <SG> | BS1
AE3 IN [ AE | | | | EXEC 32 | *AE2
TR [ AE | | | UN | EXEC 33 | *UN
TM [ AE | | | UN | EXEC 34 | *UN
L [ AE | | | AP | EXEC 35 | *AF1
X [ AE | | | UN | EXEC 36 | *UN
TAL [ AE | | | UN | EXEC 37 | *UN
MIN [ AE | | | UN | EXEC 38 | *UN
CH [ AE | | | AE | EXEC 39 | *AE2
UC [ AE | | | UN | EXEC 40 | *UN
CON [ AE | | | BP | EXEC 64 | *BS1
CLE [ AE | | | UN | EXEC 65 | *UN
ST AE AE | | | UN | EXEC 41 | *UN
RS AE AE | | | UN | EXEC 42 | *UN
SET AE AE | | | UN | EXEC 54 | *UN
TEST AE AE | | | BP | EXEC 56 | *BS1
E TL AE | | | UN | EXEC 43 | *UN
<table>
<thead>
<tr>
<th>LABEL</th>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>D1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Q2</td>
<td>481</td>
</tr>
<tr>
<td>4</td>
<td>D2</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>S1</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>TA</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>TA1</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>TA2</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>AP1</td>
<td>136</td>
</tr>
<tr>
<td>10</td>
<td>SP1</td>
<td>452</td>
</tr>
<tr>
<td>11</td>
<td>EX1</td>
<td>67</td>
</tr>
<tr>
<td>12</td>
<td>CD</td>
<td>131</td>
</tr>
<tr>
<td>13</td>
<td>F1</td>
<td>487</td>
</tr>
<tr>
<td>14</td>
<td>UN</td>
<td>400</td>
</tr>
<tr>
<td>15</td>
<td>LAB</td>
<td>81</td>
</tr>
<tr>
<td>16</td>
<td>BS1</td>
<td>350</td>
</tr>
<tr>
<td>17</td>
<td>OP1</td>
<td>85</td>
</tr>
<tr>
<td>18</td>
<td>AF1</td>
<td>176</td>
</tr>
<tr>
<td>19</td>
<td>Q1</td>
<td>475</td>
</tr>
<tr>
<td>20</td>
<td>ID</td>
<td>97</td>
</tr>
<tr>
<td>21</td>
<td>AE3</td>
<td>268</td>
</tr>
<tr>
<td>22</td>
<td>AE2</td>
<td>219</td>
</tr>
<tr>
<td>23</td>
<td>T0</td>
<td>461</td>
</tr>
<tr>
<td>24</td>
<td>B1</td>
<td>123</td>
</tr>
<tr>
<td>25</td>
<td>AF2</td>
<td>187</td>
</tr>
<tr>
<td>26</td>
<td>AT1</td>
<td>190</td>
</tr>
<tr>
<td>27</td>
<td>AT2</td>
<td>198</td>
</tr>
</tbody>
</table>
Appendix F

This appendix consists of three sample programs written in the small language and translated by the compiler built from Appendices B and C. None of the programs are meant to do useful computations, but rather to illustrate the functioning of the compiler.

Example 1 is a correct program involving fairly complicated uses of conditional and arithmetic statements. Example 2 is somewhat simpler, but has been run with several trace options on. Example 3 contains an example of a semantic error, as well as some fairly complex code. The interested reader can get a good idea of the type of code produced by the system from these examples. However, the lack of really involved structures (procedures, etc.) in the small language may yield an unrealistically optimistic picture of the system's performance.
Page 126

blank

(reverse side of 125)