

UNCLASSIFIED

AD NUMBER
AD462935
LIMITATION CHANGES
TO: Approved for public release; distribution is unlimited.
FROM: Distribution authorized to U.S. Gov't. agencies and their contractors; Administrative/Operational Use; MAY 1964. Other requests shall be referred to Defense Advanced Research Projects agency, 675 North Randolph Street, Arlington, VA 22203-2114.
AUTHORITY
ARPA ltr, 17 Jul 1969

THIS PAGE IS UNCLASSIFIED

UNCLASSIFIED

AD 4 6 2 9 3 5

DEFENSE DOCUMENTATION CENTER

FOR

SCIENTIFIC AND TECHNICAL INFORMATION

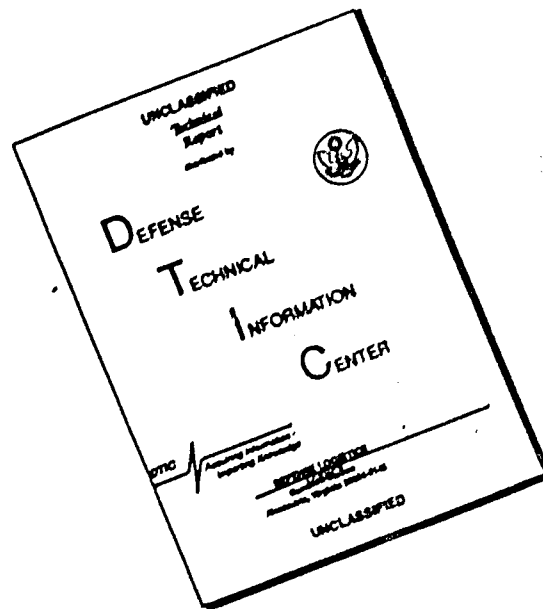
CAMDEN TATION ALEXANDRIA, VIRGINIA



UNCLASSIFIED

NOTICE: When government or other drawings, specifications or other data are used for any purpose other than in connection with a definitely related government procurement operation, the U. S. Government thereby incurs no responsibility, nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use or sell any patented invention that may in any way be related thereto.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

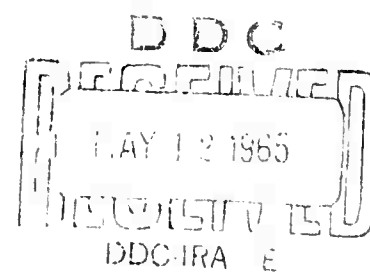
AD No. 462935

A FORMAL SEMANTICS FOR COMPUTER ORIENTED LANGUAGES

Jerome A. Feldman

May, 1964

4 6 2 9 3 5



THE WORK REPORTED HERE WAS SUPPORTED BY THE
ADVANCED RESEARCH PROJECT AGENCY OF THE
OFFICE OF THE SECRETARY OF DEFENSE;
CONTRACT SD-146

CARNEGIE INSTITUTE OF TECHNOLOGY

③ C. I. ...
Tech., P. ... , P. ...

⑥ A FORMAL SEMANTICS FOR COMPUTER-ORIENTED LANGUAGES,

⑦ 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

⑧ 1964

⑨

14

4

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

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

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

2.

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.

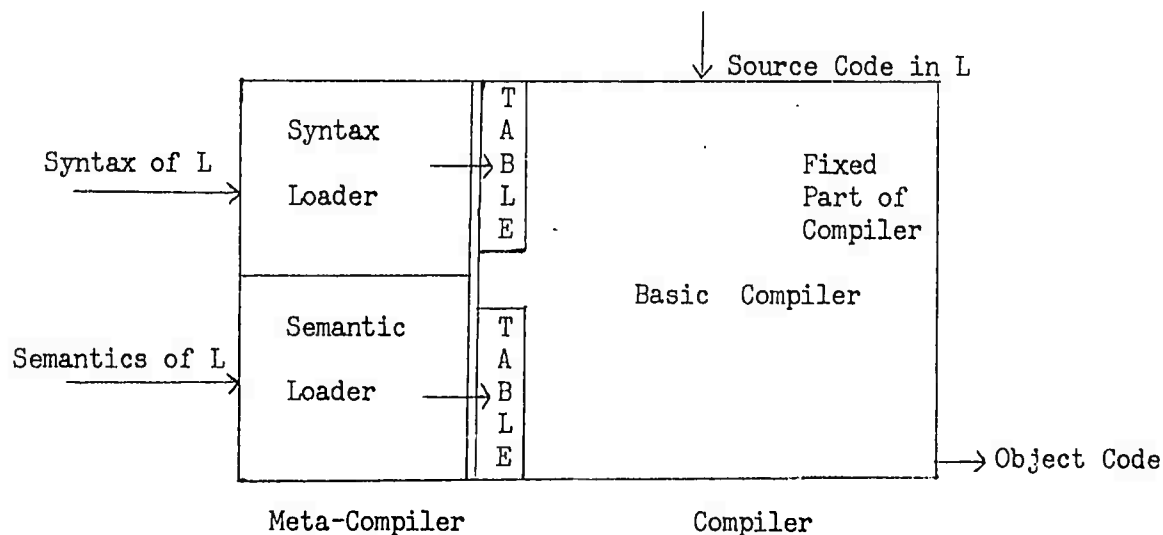


Figure 1.

6.

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

8.

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.

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

10.

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

12.

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 $\mathcal{N}$ characters ($1 \leq \mathcal{N} \leq 5$) 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 'Ø' 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

14.

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...R1 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...R1 must be blank. Thus L5...L1 refer to the top of the stack before and R3...R1 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 σ

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

T σ

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 '*P1'. The asterisk in 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 P1. 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

16.

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

<u>Letter</u>	<u>May be Read</u>
I	Identifier
P	Primary
T	Term
E	Expression

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

18.

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 P1 to pick up the other operand for that multiplication.

The productions under E1 (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 P1 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. $\langle I \rangle ::= \langle \text{letter} \rangle \mid \langle I \rangle \langle \text{letter} \rangle \mid \langle I \rangle \langle \text{digit} \rangle$
2. $\langle P \rangle ::= \langle I \rangle \mid (\langle E \rangle)$
3. $\langle T \rangle ::= \langle P \rangle \mid \langle T \rangle * \langle P \rangle$
4. $\langle E \rangle ::= \langle T \rangle \mid - \langle T \rangle \mid \langle E \rangle - \langle T \rangle$

5. Production Syntax for the Same Fragment

6.	LABEL		LEFT	RIGHT	ACTION	NEXT
7.	P1		-			*P1
8.			(			*P1
9.			I →	P	EXEC 1	*T1
10.			∅		ERROR 1	Q1
11.	T1	T *	P ∅ →	T ∅	EXEC 2	T2
12.			P ∅ →	T ∅		T2
13.	T2		T *			*P1
14.	E1	E -	T ∅ →	E ∅	EXEC 3	E2
15.		-	T ∅ →	E ∅	EXEC 4	E2
16.			T ∅ →	E ∅		E2
17.	E2		E -			*P1
18.		(	E) →	P	EXEC 5	*T1
19.			∅		ERROR 2	Q2

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 is 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 it is in the situation (a, S_i, b) . There is an element q_o 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 q_o in every square and the input tape containing a string $b_1 \dots b_n$ 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 $(q_o, S_0, \#)$ we say it

24.

accepts (recognizes) the string $b_1 \dots b_n$.

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

$$(1) : \quad (a, S_i, b) \rightarrow (S_j, X, K)$$

where $a \in A_I$, $b \in A_0$; 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

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

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 ' $\emptyset$ ' 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:

$$W ::= 0 \mid 1 \mid 0W0 \mid 1W1$$

28.

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

SUBR L

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.

A. Introduction

This section of the paper is devoted to a detailed discussion of the Formal Semantic Language, FSL. This language embodies an attempt to state in fixed terms the meaning of a statement in any of a large class of programming languages.

The formalization of the semantics of some language, L, will involve representing the meanings of statements in L in terms of an appropriate meta-language. If L is a computer language, the semantics of L must involve a description of its translator. Throughout this paper we will use the neutral word 'translator' to mean either a compiler or an interpreter. The distinction between these latter concepts is not clear cut, since many compilers use run-time routines and many interpreters incorporate a preprocessing phase. We will use terms like 'highly interpretive translator' to indicate that a translator is more like one or the other of these extremes.

The meanings of statements in the meta-language are assumed to be known (primitive). An example of a semantic meta-language would be the order code of a computer. An order code or assembly language can certainly describe any translation possible on its machine and, in fact, these have been the only kinds of semantic meta-languages in general use to date.

Unfortunately, these lack several properties which one would like to see in a semantic meta-language.

A semantic meta-language should allow the description of source languages to be as easy as possible. It should be readable so that other people can understand the meaning of the source language being defined. It should enable a description of a language which is precise and complete enough to allow for automatic compilation. Finally, the meta-language description should be independent of any particular machine.

These criteria are easily recognizable as those relevant to the specification of a problem-oriented computer language. The problem involved is the representation of the meanings of statements in a computer language. The Formal Semantic Language presented here is an attempt to provide a complete programming system possessing these properties.

The semantic meta-language, FSL, was designed to have just those constructs which are useful in describing translators. These were chosen with no thought of implementation and no particular computer in mind. After the system was complete, users began to attempt the writing of semantics for some common programming languages and its implementation was begun. The interaction of these three forces, preconceived formalism, hardware implementation, and human users, forged the system into

its final form. An FSL system is running on the Control Data G-21 at Carnegie Tech, and translators for several languages including FORTRAN, ALGOL, and LISP have been written.

The basic unit in an FSL program is the labeled statement or sentence. These labels are integers and form the main link between the productions described in Section II and the semantic routines discussed here. As we recall, a production may include an action of the form

EXEC n

where n is an integer. The semantic routine labeled 'n' will be executed each time that production is matched.

Whenever possible, the discussion of a construct in FSL will be accompanied by an example. The examples will be drawn from the translator for a subset of ALGOL 60 which we call the "small language". The BNF syntax of the small language is Appendix A of this paper. The productions for the small language constitute Appendix B while the formal semantics is Appendix C. Appendices B and C comprise a complete description of the small language and have been used in the construction of a compiler for that language.

In many ways, FSL is similar to other programming languages. Various concepts have been borrowed from different algebraic and list processing languages. As is the case in programming research, it is impossible

to accurately credit one author as the originator of an idea. There is also a 'folklore' of unpublished techniques upon which we have drawn freely. But there are also many constructs in FSL which are peculiar to the translation problem and which have not appeared before. The introduction of each new construct was prompted by some problem in language design.

One of the most difficult concepts in translator writing is the distinction between actions done at translate time and those done at run time. Anyone who has mastered this difference has taken the basic step towards gaining an understanding of computer languages. This difference between run time and compile time actions is a crucial one in FSL and is always represented explicitly. The means of doing this is the pair of code brackets 'CODE(' and ')'. Any statement enclosed in code brackets specifies an action to be taken at run time, while any statement not so enclosed describes action taken at translate time. As we will see, each FSL construct has two meanings, one when it appears inside code brackets and the other when it does not. In some cases, one of the meanings will not be defined.

Some of the other unique features of FSL are the tagging facilities. There are special statements for putting on or removing tags and a predicate used to test for the presence of a given tag. These are usually used in connection with various data types and are discussed in Section III-C.

There are also FSL constructs which effect the translation of forward references in a source language program. These constructs formalize some efficient techniques used by translator writers to handle forward references. They include floating addresses, discussed in Section III-D and chains, discussed in Section III-E. Also used in this connection is an 'assign' statement described in Section III-F.

In addition, FSL includes indirect addressing at both translate time and at run time. It also allows a more general replacement statement than is usually found in programming languages. These features are useful in computing addresses for use in compiling code and are discussed in Sections III-D and III-F.

Some of the more interesting features arise in connection with storage for the translator itself. Each FSL program describes a translator and thus must include a description of the storage the translator will use in building up code. The various storage types available in FSL are described below.

B. Storage Declarations

One of the major difficulties involved in the translation of computer languages arises because the meaning of a statement will, in general, depend on that of preceding and even of succeeding statements. For example, most programming languages allow the declaration of data

types and the subsequent use of defined variables in various contexts. In this case the meaning of a statement depends on the data type declared for the variables involved in that statement.

There is at least one well known example of the use of succeeding information in programming languages. It is very common to have a jump or transfer to a label which is not yet defined. The meaning of such a statement is not available until the label is defined somewhere in the succeeding text.

For reasons such as these, one might expect some fairly sophisticated storage operations in a language which purports to formalize the translation process. In the Formal Semantic Language (FSL) there are five distinct types of storage available to the language designer. These are introduced as declarations in a FSL description of a language and become part of the translator for that language. All of these declarations must be at the beginning of the FSL program and no declaration may occur within code brackets. The declaration of translator storage is covered in this section while the operations available for declared storage will be treated in III-C.

The first and most complex data type in FSL is the TABLE. Conventional translators often have tables for variables, for labels, for subroutine names and for a variety of other constructs. These usually exist only at compile time but in highly interpretive translators, tables

may be required at run time as well. In FSL a run time table is declared by a RTABL declaration while compile time tables are declared by a TABLE declaration. The syntax is:

```

< table dec > ::= TABLE < table specifier > |
                    RTABL < table specifier > |
                    < table dec > , < table specifier >
< table specifier > ::= < identifier > [ < integer > , < integer > ]

```

The <identifier> is the name of the table being declared. The first integer specifies the maximum length of the table and the second integer specifies the number of entries (width) of the table. In the next section we will discuss the operators for entering information into and retrieving it from these tables.

The small language described in Appendices A - C is a pure compiler and thus has no run-time tables. The one table, SYMB, declared in Appendix C would be the symbol table for the translator. It would hold the identifier (name) of a variable and its location, data type, and level for each variable declared in a particular source language program. SYMB, as declared, can hold four entries for each of a maximum of 800 symbols.

The description of all the other storage types makes use of the construct <identifier list> :

$$\langle \text{identifier list} \rangle :: = \langle \text{identifier} \rangle \mid \langle \text{identifier list} \rangle, \langle \text{identifier} \rangle$$

Thus each declarator operates on a list of variables, eliminating the use of redundant declarators.

In recent years the push-down stack has become an indispensable tool to translator writers. It is possible in FSL to declare stacks which will exist either at compile-time or at run-time. The syntax is:

$$\begin{aligned} \langle \text{stack dec} \rangle :: = & \text{STACK } \langle \text{identifier list} \rangle \mid \\ & \text{RSTAK } \langle \text{identifier list} \rangle \end{aligned}$$

There are no run-time stacks in the small language. The two stacks, STR and SYM, declared for the compiler will be used in connection with the block structure. At the beginning of each block, STR will have stored in it the value of the storage pointer and SYM the value of the symbol table pointer. At the end of a block these two pointers can be replaced, allowing the translator to reuse both storage and table space. These operations occur in semantic routines 2↓ and 29↓.

Individual storage locations for use in the translator are declared to be of type CELL:

$$\langle \text{Cell dec} \rangle :: = \text{CELL } \langle \text{identifier list} \rangle$$

A cell may be used as a switch, a temporary or any other object requiring a single machine location at translate time.

It is often useful in translator writing to have certain marks of constant value. One may declare alphabetic constants in FSL as variables of type TITLE:

`<title dec> :: = TITLE <identifier list>`

The title words used in the small language are the three data types: REAL, BOOL, LABE. When an identifier is declared, the appropriate title word is entered into the symbol table. Then when an identifier occurs in a statement, a test is made to see if the identifier is of the proper type, i.e. if its TITLE word is one acceptable in this context. If it is not, an error message will be printed.

The fifth type of declaration in FSL is DATA:

`<data dec> :: = DATA <identifier list>`

The variables declared to be of type DATA specify bit configurations rather than entire computer words. A translator works primarily with addresses and in most computers an address is not as long as a word. The operators associated with DATA identifiers enable the translator writer to tag an address with certain bits and later test if the bits are present. The use of DATA variables will be discussed in detail in Section III-C.

The data variables used in the small language are LOGIC, INTEGER, SINGLE, DOUBLE. There are several instances of their use in the

42.

semantics of the small language (Appendix C).

The various declarations described above must occur together at the beginning of an FSL program. The syntax of the declaration part of a program is:

```
<declaration part> :: = <declaration> |  
                        <declaration part> ; <declaration>  
<declaration> :: = <table dec> | <stack dec> | <cell dec>  
                  <title dec> | <data dec>
```

C. Storage Operations

We have already introduced several means of declaring storage for a translator. Each type of storage has associated with it a set of operators which enables the language designer to use the storage effectively.

Storage variables declared as type CELL or of type TITLE are single storage locations and require no special operators. Any such variable will be of type <operand> and as such will be discussed in Section III-D.

There are two basic operations on STACKS, PUSH and POP. Each has two parameters, the first being the name of a stack and the second specifying a location. An execution of PUSH causes the contents of specified locations to be appended to the top of the stack. The command POP

will cause the top element of the stack to be stored in the specified location and to be removed from the top of the stack. The syntax of stack commands is:

$$\begin{aligned} \langle \text{stack command} \rangle ::= & \text{PUSH [} \langle \text{identifier} \rangle , \langle \text{arithmetic} \rangle \text{] } | \\ & \text{POP [} \langle \text{identifier} \rangle , \langle \text{arithmetic} \rangle \text{]} \end{aligned}$$

In addition, the name of any stack can be used as an $\langle \text{operand} \rangle$. In this case the top element of the stack is treated as a variable which can be accessed or stored into. Thus there are commands for the destructive and non-destructive reading and writing of stacks.

In the small language semantics, stacks are used only in connection with block structure. Semantic routine 1↓ has two instances of the destructive store into a stack.

$$1\downarrow \quad \text{LEV} \leftarrow 0; \quad \text{STR} \leftarrow \text{STORLOC}; \quad \text{SYM} \leftarrow \text{LOC}[\text{SYMB}] \quad \downarrow$$

Routine 2↓ demonstrates the non-destructive (pushdown) store into the same stacks. Two examples of destructive read out occurs in 29↓.

There are no non-destructive reads in the small language.

$$\begin{aligned} 2\downarrow \quad \text{LEV} \neq 0 \rightarrow & \text{PUSH}[\text{STR}, \text{STORLOC}]; \text{PUSH}[\text{SYM}, \text{LOC}[\text{SYMB}]]\$; \\ & \text{TALLY}[\text{LEV}] \downarrow \end{aligned}$$

$$29\downarrow \quad \text{MINUS}[\text{LEV}]; \quad \text{POP}[\text{STR}, \text{STORLOC}]; \text{POP}[\text{SYM}, \text{LOC}[\text{SYMB}]] \downarrow$$

If a stack command occurs within code brackets the identifier must have been declared to be of type RSTAK. In this case the stack operation will be performed at run time. Since the small language requires no RSTAK variables, all stack commands in Appendix C are outside code brackets.

Both STACK and RSTAK are implemented using a pointer. That is, when a stack identifier is used in a PUSH command, the pointer for that stack increases by one. Similarly, a POP causes the pointer to be decreased by one, after the store is done. The pointer associated with any stack is also available to the FSL user. It is referred to as

LOC [<stack id>]

where the <stack id> is the name of the stack as declared with a STACK or RSTAK statement.

Since the run time stacks are not available at compile time, the use of RSTAK variables outside code brackets is forbidden. The general rules for compile time operands within code brackets will be discussed in Section III-D.

The variables declared to be of type DATA are used only with specific data operators. There is a special command, SET, for putting the tag associated with a data identifier into a machine word. The syntax of the set command is

SET [<arithmetic> , <data id>]

where the value of <arithmetic> is the location which is to be set. The effect of a SET is to unite the current contents of the location specified by the arithmetic expression with the bits corresponding to the <data id>. An example may be found in 4↓ of Appendix C which is executed when a real variable has been declared. The contents of TO are united with the bits for DOUBLE and the result put back in TO, which is then put in the symbol table, SYMB.

4↓ TO ← STORLOC; SET[TO,DOUBLE]; ENTER [SYMB;LEFT2,TO,REAL,LEV];
STORLOC ← STORLOC+2 ↓

The FSL system itself can currently recognize and compile code for the four data types: LOGIC, INTEGER, SINGLE, and DOUBLE. Any variable that is set to have one of these types will be treated automatically. In addition, the language designer may choose to use other data names such as COMPLEX, ARRAY, or TREE. Any variable declared of type DATA will have a unique set of bits associated with it. One can, for example, design a language with complex arithmetic as a basic set of operations. Each complex variable could be set COMPLEX and recognized as such in a later arithmetic statement. The actual semantics of each complex operation would, however, have to be written in terms of the operations on SINGLE or DOUBLE variables which are part of the system.

To make effective use of DATA variables, one must have a way to test for the presence of bit patterns. This is provided in FSL by means of a binary predicate, TEST. This predicate is one instance of a <boolean primary>. The syntax is:

TEST [<arithmetic> , <data id>]

The value of the predicate is TRUE if the bit pattern corresponding to the <data id> is present in the location named by the <arithmetic>. The value is FALSE in any other case. The location named by <arithmetic> is not changed by the TEST.

In the small language described in the appendices, real variables are used in double precision form. Therefore in 4↓, when an identifier is declared to be real, its location is set to DOUBLE. It should be noted that none of the statements dealing with arithmetic operations (10↓ - 14↓) contains a TEST. This is because the small language contains only data types which the system treats directly. The addition of two variables set with the bits for DOUBLE will automatically be compiled by the system as a double precision operation.

A potential difficulty arises in connection with variables which have been tagged. If one were to attempt to do arithmetic at compile time on tagged addresses, the tags would affect the arithmetic operations in unpredictable ways. For this reason, there is a special operator,

CLEAR, in FSL. The syntax is:

```
CLEAR [ <arithmetic> ]
```

The location specified by the <arithmetic> names the location to be cleared. The effect of a clear is to remove all the tag bits from a machine word. The DATA identifiers name only portions of a machine word and thus may not be used except as described above.

The most complicated data structure in FSL is the table. In Section III-B we described the manner in which tables are declared in FSL. The one table in the small language translator was seen to have four entries for each variable used in a small language program. These entries are filled in by the operation ENTER. The syntax of the enter command is:

```
<enter> :: = ENTER [ <table id> ; <expression list> ]
```

```
<expression list> :: = <arithmetic> | <expression list>, <arithmetic>
```

The number of expressions in the list should equal the width declared for the table. The ENTER statement will form the expressions in the list, in order, as a row of the table named by the <table id>. If the table is already filled, no stores will be done and the cell SIGNAL will be set FALSE.

48.

In the semantics of the small language (Appendix C) there are three examples of the use of ENTER. These occur in connection with the declaration of the three kinds of variables in routines 4↓, 5↓, and 30↓. As each variable is declared, the small language translator alters its symbol table, SYMB. In each case an enter command is used to form a new row of the symbol table.

For example, in 4↓ a new row of the symbol table SYMB is formed as follows. The next available storage cell is picked up and SET to have the type title. The row of SYMB will consist of the name of the variable (in LEFT2), the tagged address, the title REAL and the current value of the cell LEV.

```
4↓  TO ← STORLOC; SET[TO,DOUBLE]; ENTER[SYMB;LEFT2,TO,REAL,LEV];  
    STORLOC ← STORLOC+2 ↓
```

We now know how to declare tables and how to store into them. In order for tables to be useful, there should also be a convenient way of retrieving information. Since a table element will most often be used as a operand, the retrieval routine is of the syntactic type <operand>. This enables the FSL programmer to use table entries directly as operands. The syntax is:

```
<table operand> ::= <table id> [<arithmetic> <commas> $ <commas>]  
<commas>       ::= , | <commas> , | <empty>
```

The selection of a table operand takes place in three distinct steps. First the value of the <arithmetic> is computed and saved. Then the table named by the <table id> is searched for an occurrence of this value as the first entry of a row. When a match is found the position of the \$ is used to pick out the desired entry in the matched row.

An example should help clarify the use of table operands. Consider semantic routine 16↓, which is called when an arithmetic assignment statement has been scanned.

```
16↓ SYMB[LEFT4,,,$,]=REAL→COMT 2←SYMB[LEFT4,$,,];
    CODE(COMT 2←LEFT2) ; TEMPLOC←T3 : FAULT 3$ ↓
```

The fourth element in the stack at this point (LEFT4) is an identifier which names the location to be stored into. In 16↓ the identifier is looked up in the symbol table and the third entry in the matched row is selected as the operand. As we recall, this entry should be the data type of the variable. If the data type is not REAL, a semantic fault is detected.

If the identifier was declared correctly the assignment statement should be compiled. For this purpose the address which the variable will have at run time is needed. The next statement in 16↓ selects this address (the second entry in the row named by the identifier) and stores it into a cell named COMT 2. We will see later how this address is used in compiling the code for the store operation.

As we mentioned, the selection of a table operand takes place in three distinct steps. First the expression to be looked up is computed. Then the table is searched for a match and finally the correct entry in the matched row is selected. It is often the case that more than one entry from a specific row is needed in a single routine. In such cases, it should be possible to bypass the first two parts of the selection process for all but the first of these operands.

There is a special convention in FSL to enable the user to take advantage of this possibility. If the <arithmetic> in the specification of the table operation is '0' the last row selected is automatically used, bypassing the first two steps of the selection process. This is useful when successive references to the same row of one table are used within a semantic routine. Although this feature could have been used in many of the semantic routines in Appendix C (e.g. 8↓, 9↓, 16↓) it was omitted for the sake of clarity. The one example of its use is in 26↓. Each table declared in FSL is implemented using a pointer to the next available row. This pointer is also used by the retrieval routine as the starting points of its search. In addition, the pointer for each table is available to the user as an <operand> of the form

LOC [<table id>]

In the small language, the table pointer is used in connection with the block structure. At the beginning of each block, 2↓ will be called and it will cause the pointer for SYMB to be saved. Routine 29↓, which

is called at the end of each block, resets the pointer to its value on entering that block. This effects the elimination of all variables which were declared to be local to the block.

It is important to remember that all of the storage constructs described above are defined for the translator only. The language designer may choose to incorporate any of a number of different kinds of storage in his source language. The storage allocation for programs written in a particular source language is part of the semantic description of that language. We will discuss in Section III-G some special features of FSL which facilitate the solution of storage problems in a source language.

D. Operands and Primaries

The basic unit in FSL is the operand. We have already encountered several instances of operands in the discussion of storage in III-B and III-C. In this section we will introduce the other operands and discuss their use. The material below presupposes an understanding of the significance of code brackets as presented in III-A. The meaning and use of each operand will depend upon whether or not it is inside code brackets.

We will first present the syntax of <operand> and then discuss each instance in the definition.

$$\begin{aligned} \langle \text{operand} \rangle ::= & \langle \text{storage operand} \rangle \mid \langle \text{production operand} \rangle \mid \text{COMT} _ \langle \text{identifier} \rangle \mid \\ & \text{RUNT} _ \langle \text{identifier} \rangle \mid \langle \langle \text{arithmetic} \rangle \rangle \end{aligned}$$

The storage operands have been discussed in some detail in III-C. Their syntax is:

$$\langle \text{storage operand} \rangle :: = \langle \text{table operand} \rangle \mid \langle \text{cell id} \rangle \mid \langle \text{stack id} \rangle \mid \langle \text{title id} \rangle$$

The identifiers of type TITLE each name an alphabetic string extant at compile time and thus have no meaning within code brackets. Both table and stack operands involve address computations each time they are used. For this reason variables associated with RSTAK and RTABL can occur only within code brackets, specifying actions to be performed at run time. Similarly, STACK and TABLE operands must be computed at compile time and thus never appear within code brackets. This is in keeping with the convention, adhered to in FSL, of distinguishing explicitly between actions performed at run time and those done at compile time. Identifiers of type CELL represent single machine locations and are treated in a straightforward manner.

The production operands form the main link between the syntax (production) phase of translator (cf. Section II) and the semantic phase being discussed here. The definition is:

$$\langle \text{production operand} \rangle :: = \begin{array}{c} \text{LEFT1} \mid \text{LEFT2} \mid \text{LEFT3} \mid \text{LEFT4} \mid \text{LEFT5} \\ \text{RIGHT1} \mid \text{RIGHT2} \mid \text{RIGHT3} \end{array}$$

The operands LEFT1 - LEFT5 refer to the main stack before the production has been executed. The element at the top of the stack is

always referred to as LEFT1. The operands RIGHT1 - RIGHT3 name the top three positions in the main stack after the execution of the same production. As we recall, each position in the main stack is actually two machine words, one which names the syntactic construct and the other which describes its meaning. The purpose of the stack operands is to enable the language designer to use or alter the semantics of constructs as they are recognized by his productions.

A simple example may be found in Appendix B on the line labeled B3. At this point in the syntax, an 'IF CLAUSE' of the small language has been recognized. The semantics of the 'IF CLAUSE' is the same as that of the boolean expression in LEFT2; namely a location to be tested for TRUE or FALSE. The action field in the production B3 contains a call of the routine 21↓. We see that in Appendix C, 21↓ consists of the statement

$$21\downarrow \quad \text{RIGHT1} \leftarrow \text{LEFT2} \downarrow$$

This routine specifies that the semantics of the construct which was second from the top of the stack when the production was matched is to be assigned as the semantics of the element which is on top of the stack after production has been executed. We will see later that stack operands are also used in more sophisticated ways.

The next type of operand to be discussed is the compile time temporary or COMT. They are available as temporaries to the FSL user and

and are also used for communication with compile time routines. The character '␣' in the definition denotes a space and the <identifier> is assumed to be a subscript. This subscript can be either a constant or a variable of type CELL.

A similar construct is the run time temporary or RUNT. The RUNT's are system cells extant at run time and will be used for communication with run time routines. The subscript on a RUNT must also be a constant or a CELL. Since RUNT's exist only at run time they may not occur outside code brackets except in the form LOC [RUNT i] (see III-E) which specifies the location at run time of a RUNT as an operand at compile time.

Because of the simplicity of the small language in the appendices, there is no use of RUNT variables. There are several uses of COMT variables, for example in 16↓, the cell COMT 2 is used to hold the address found in the table for the identifier. The use of the temporary is necessary because the table-look-up operation cannot occur within code brackets.

The final instance of operand is an arithmetic expression within the brackets '<' and '>'. These are called contents brackets and are used in FSL for indirect addressing. The value of the arithmetic expression is assumed to be an address and the operand is taken to be the contents of this address. Since the arithmetic expression may itself be an operand, the contents brackets may be nested. An example of the use of contents brackets may be found in 25↓ of Appendix C.

Two FSL constructs closely related to operands are the <primary> and the <boolean primary>. The distinction between the three concerns the contexts in which each is legal.

The primaries take on numerical values and are legal in connection with arithmetic operations. Boolean primaries take on logical values and are used accordingly. Operands are constructs which can be used in either of these ways. We will first describe the various primaries and boolean primaries and then discuss the properties which they have in common with operands.

The primaries in FSL are described by the following formal syntax:

$$\begin{aligned} \langle \text{primary} \rangle &::= \langle \text{operand} \rangle \mid \langle \text{constant} \rangle \mid (\langle \text{arithmetic} \rangle) \mid \text{LOC} [\langle \text{table id} \rangle] \mid \\ &\quad \text{LOC} [\langle \text{stack id} \rangle] \mid \langle \text{system cell} \rangle \mid \langle \text{flad} \rangle \\ \langle \text{system cell} \rangle &::= \text{CODELOC} \mid \text{STORLOC} \mid \text{TEMPLOC} \end{aligned}$$

The first element of the definition indicates that any operand may be used as an arithmetic primary. A constant used in an FSL program will become part of the translator being described. Constants are expressed in the same way as ALGOL 60 constants [/].

The syntactic definition of a parenthesized arithmetic expression as a primary also follows ALGOL 60. This is needed to incorporate the usual grouping rules connected with arithmetic operations and will be discussed in III-E.

As we mentioned in Section III-C, both tables and stacks are implemented using a pointer to the next available location in their reserved storage areas. These pointers are used as operands of the form LOC [<table id>] and LOC [<stack id>] respectively. The value of either of these operands is an address and must be used accordingly. Following the usual convention, the pointers to STACK or TABLE operands cannot occur within code brackets. Similarly, since the pointers to RSTAK's and RTABL's are non-existent at compile time they cannot be used outside code brackets. The pointer to the symbol table, LOC [SYMB], is used as an operand in routines 2↓ and 29↓.

The system cells mentioned as the next instance in the definition of primary are also pointers. The cell CODELOC contains the address into which the next word of the compiled code will be placed. CODELOC is used by the system and is incremented as code is compiled. In addition, the language designer could use CODELOC in connection with his translation of labels, conditionals or subroutine definitions.

STORLOC is the pointer to the next available storage location for a particular source language program. It is used by the Storage Allocation Routine in the system and may also be accessed by the FSL user. In the semantics of the small language (Appendix C) the value of STORLOC at the beginning of each block is saved by 2↓. Then at the end of each block, STORLOC is reset to its previous value by 29↓. Since the Storage Allocation Routine is not used in the small language, STORLOC is also used in

assigning storage to real variables (4↓) and boolean variables (5↓) as they are declared in a small language program.

```
4↓ TO ← STORLOC; SET[TO, DOUBLE]; ENTER[SYMB; LEFT2, TO, REAL, LEV];
    STORLOC ← STORLOC + 2
```

Notice that in 4↓ STORLOC is increased by two to allow space for a double precision number.

The cell TEMPLOC is also used by both the system and the language designer. It points to the next in the list of temporaries used by the system in translating complicated expressions. In the small language semantics, conditional boolean expressions have been implemented using TEMPLOC explicitly in 22↓. In most source languages the FSL user would like to reinitialize TEMPLOC whenever values stored in temporaries would no longer be needed. In the small language this is done at every assignment statement (16↓, 20↓, 24↓). The cell T3 was assigned the initial value of TEMPLOC in 0↓.

The last alternative in the definition of <primary> concerns floating addresses or FLAD's. The syntax is

$$\langle \text{flad} \rangle :: = \text{FLAD1} \mid \text{FLAD2} \mid \text{FLAD3} \mid \text{FLAD4}$$

The flads are used to help implement undefined forward references, such as those occurring in conditional statements. When a flad is used

58.

as the operand of a transfer command appearing in an FSL statement, the system notes that an address will have to be added to that statement. When the proper place in the source program is reached an ASSIGN statement (cf. III-F) should be used. The system will then put the address into the transfer command and free the flad for further use. Each of the flads is actually a pushdown stack so that uses of flads may be nested.

Consider the example of the conditional statement in the small language. Routine 3↓ is called whenever an 'IF CLAUSE' which is part of a conditional statement is scanned.

3↓ PUSH[FLAD1,0]; CODE(¬LEFT1 → JUMP[FLAD1]\$) ↓

If the value at run time of the associated boolean expression is FALSE, the next statement is not executed. In terms of a translator -- a transfer around the statement must be compiled. Since the length of the statement is unknown at this point, 3↓ contains a JUMP[FLAD1] in code brackets. When the end of the statement is found (7↓) the current value of CODELOC is assigned to FLAD1 by the statement ASSIGN [FLAD1].

If the small language conditional statement had included an ELSE the flow would have been through 6↓ and 27↓ and would require two flads. Notice that each flad is used in a PUSH command before it is used as an operand. This allows for nested conditional statements in a small language source program. The ASSIGN command automatically does a POP of the flad after assigning it.

There are more boolean primaries in FSL than in conventional programming languages because of the nature of the translation task. While generating each individual piece of object code is not difficult, the job of choosing the right piece of code for each stage of the translation is quite involved. The variety of boolean primaries in FSL should alleviate the burden of determining the state of the translator. The syntax is:

$$\begin{aligned} \langle \text{boolean primary} \rangle ::= & \langle \text{arithmetic} \rangle \langle \text{relation} \rangle \langle \text{arithmetic} \rangle \mid \langle \text{operand} \rangle \mid \\ & (\langle \text{boolean} \rangle) \mid \text{TRUE} \mid \text{FALSE} \mid \text{SIGNAL} \mid \text{OK} \mid \text{TEST} \\ & [\langle \text{arithmetic} \rangle, \langle \text{data id} \rangle] \mid \text{CONST} [\langle \text{arithmetic} \rangle] \end{aligned}$$

The relations defined in FSL are '=', '≠', '>', and '<'. These all have their customary meanings and are defined both inside and outside of code brackets. The second instance of $\langle \text{boolean primary} \rangle$ is the type $\langle \text{operand} \rangle$ described at the beginning of this section. Thus any construct of type $\langle \text{operand} \rangle$ may be used in either a boolean or an arithmetic context. The definition of a parenthesized $\langle \text{boolean} \rangle$ as a $\langle \text{boolean primary} \rangle$ parallels the situation with arithmetic expressions. Both will be discussed in the next section.

The logical constants, TRUE and FALSE are also boolean primaries. They may occur either inside or outside of code brackets. The cell, SIGNAL, is set TRUE or FALSE by various routines which are part of the system. For example, if the routine to select an operand from a table

cannot match its entry, SIGNAL is set FALSE. The status cell, OK, is set by the language designer and used by the system. If OK is FALSE, no code will be generated for the source program being scanned. The FSL user can set OK in the event of a non-recoverable source language error. Even if OK is FALSE, syntax checking will continue until a halt command is given. There is no way to reset OK once it has been given the value FALSE.

The next instance of <boolean primary> is the TEST statement, which involves data types. As we mentioned in Section III-B, FSL allows the declaration of tags which can be placed into a computer word. The value of a TEST is TRUE if the location specified by its first parameter contains the bits corresponding to the <data id> which is its second parameter. Further information on the use of data types in FSL will be provided in Section III-E.

The final instance of boolean primary is the test for constants, CONST[<arithmetic>]. The arithmetic expression is assumed to specify a location to be tested. CONST will have the value TRUE if the location being tested contains a numerical or logical constant. This test is necessary because the FSL system is set up to process constants directly when used in normal mode. In the small language, we test for constants when an identifier is used in an arithmetic expression ($8\downarrow$, $9\downarrow$). If the identifier has the semantics of a constant, no action is necessary. Otherwise the identifier is a variable and must be looked up in the symbol table, SYMB.

9 ↓ CONST[LEFT1] → RIGHT1 ← LEFT1 :

SYMB[LEFT1,,,\$,]= REAL → RIGHT1 ← SYMB[LEFT1,\$,,]:FAULT 1\$\$ ↓

Of the boolean primaries, OK, SIGNAL, and CONST are meaningful only at compile time. These may appear outside code brackets only, while the other boolean primaries can occur either inside or outside of code brackets. Although the arithmetic primaries and operands exist only at compile time, there is a special convention that allows most of these to appear within code brackets.

This is possible because the usual value at compile time of an arithmetic operand is an address. Thus while, for example, CODELOC exists only at compile time, it contains an address which is itself meaningful at run time. For these reasons, we have adopted the convention that the occurrence of a compile time cell within code brackets specifies that we use the compile time contents of that cell as the operand of the code to be compiled.

A simple example will help clarify the situation. In 12 ↓ the expression

LEFT4 + LEFT2

appears within code brackets. Assume that, as will usually be the case, both LEFT4 and LEFT2 contain addresses at compile time. If the the expression above were not enclosed in code brackets, the two addresses

would be added at compile time to produce another address. When the addition occurs within code brackets an entirely different action takes place. In this case the system will generate code to do an addition at run time. The addends at run time would be the run time contents of the addresses contained in LEFT4 and LEFT2 at compile time.

The process described above may be thought of as adding an extra level of indirectness to compile time operands occurring within code brackets. Since the convention holds for most operands and arithmetic primaries, we will point out the exceptions.

This convention obviously cannot apply to RUNT's which do not exist at compile time and only appear inside code brackets. It also does not apply to table and stack operands extant at compile time because these implicitly involve compile time actions. In addition, FLAD's are a special type of operand to which the convention does not apply.

As we will see, the above convention is part of the overall rule that all operators appearing within code brackets specify run time actions. Another example of this concerns contents brackets. Any occurrence of contents brackets within code brackets specifies indirect addressing to be done at run time. More explanation and examples of this situation will be presented in the next section, which deals with the construction of expressions in FSL.

E. Expressions

There are two kinds of expressions, arithmetic and boolean, in FSL. The description of expressions in this section assumes an understanding of Section III-D, which deals with operands and primaries. Each FSL expression is formed by combining the appropriate primaries with a simple set of operators.

The arithmetic expressions in FSL appear in the syntax more frequently than any other syntactic type. This is largely because addresses at compile time are of this type. The syntax of arithmetic expressions is:

$$\begin{aligned}
 \langle \text{factor} \rangle &::= \langle \text{primary} \rangle \mid \langle \text{factor} \rangle \uparrow \langle \text{primary} \rangle \\
 \langle \text{term} \rangle &::= \langle \text{factor} \rangle \mid \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle / \langle \text{factor} \rangle \\
 \langle \text{arithmetic} \rangle &::= \langle \text{term} \rangle \mid \langle \pm \rangle \langle \text{term} \rangle \mid \\
 &\quad \langle \text{arithmetic} \rangle \langle \pm \rangle \langle \text{term} \rangle \mid \text{INT} [\langle \text{arithmetic} \rangle] \mid \\
 &\quad \text{ABS} [\langle \text{arithmetic} \rangle] \mid \text{LOC} [\langle \text{arithmetic} \rangle] \mid \\
 &\quad \text{CHAIN} [\langle \text{arithmetic} \rangle]
 \end{aligned}$$

The practice of breaking up the definition of arithmetic expressions into parts follows the ALGOL 60 report. This notation is a convenient way to represent the normal grouping rules of arithmetic. The operator ' $\uparrow$ ' represents exponentiation which actually implemented by a subroutine. The definition of the other arithmetic operations is straightforward.

The one potential source of difficulty is the proper use of code brackets. Any occurrence of an arithmetic operator within code brackets specifies an action to be performed at run time. This implies that all computations on addresses must occur outside code brackets.

Since the arithmetic operations in the small language include only those operators treated directly by the system, the examples (10↓ - 14↓, 28↓) in Appendix C should be clear. Each of these routines carries out code generation for one of the basic arithmetic operations. The use of VALUE operands in these examples will be explained in the next section.

In addition to the operations described above, there are four other ways of describing an arithmetic expression. The value of INT [<arithmetic>] is the largest integer which is not greater than the value specified by the <arithmetic>. This function is identical to ENTIER described in the ALGOL 60 report [31]. The value of ABS [<arithmetic>] is the absolute value of the operand specified by the <arithmetic>. Either of these can appear inside or outside of code brackets describing actions to be taken at run time or at compile time, respectively.

The LOC operations is used with two slightly different meanings in FSL. If the operand of LOC is a stack or table identifier, LOC will have the appropriate pointer as its value (cf. Section III-C, III-D). Any other use of LOC describes a reduction in the indirectness of the addressing of its operand. In this case LOC acts as the inverse of the contents brackets '<' and '>'.

The use of CHAIN as a unary operator is more involved. The CHAIN of an arithmetic expression is used when some operand, usually a label, is used before its value is defined. An example of this may be found in 25↓ of the Appendix C. If the label in a 'GO TO' statement of the small language is undefined, the tests in 25↓ lead to the statement

CODE (JUMP [CHAIN [COMT 2]])

This command will cause the system to compile a jump command with the address portion to be specified later. The cell COMT 2 contains the location in the SYMB table where the address associated with the label is to be stored. If there are several uses of the same undefined label the system will build a linked list or chain to keep track of them. Then when the label is defined (26↓) an ASSIGN statement would be used. This will cause the system to fill in all previous references to the undefined label, including the one in SYMB. We will have more to say about ASSIGN in Section III-F. An appearance of CHAIN outside code brackets, is meaningless since the system automatically handles all undefined references in an FSL program.

The syntax of boolean expressions also consists of several steps. This is set up so that the following equivalence holds for boolean expressions

$$\neg A \wedge B \vee C \wedge D \equiv ((\neg A) \wedge B) \vee (C \wedge D)$$

In other words ' $\neg$ ' binds most tightly, then ' $\wedge$ ' and finally ' $\vee$ '.

The syntax is:

$$\begin{aligned} \langle \text{boolean factor} \rangle &::= \langle \text{boolean primary} \rangle \mid \neg \langle \text{boolean primary} \rangle \\ \langle \text{boolean term} \rangle &::= \langle \text{boolean factor} \rangle \mid \langle \text{boolean term} \rangle \wedge \\ &\quad \langle \text{boolean factor} \rangle \\ \langle \text{boolean} \rangle &::= \langle \text{boolean term} \rangle \mid \langle \text{boolean} \rangle \vee \langle \text{boolean term} \rangle \end{aligned}$$

All boolean expressions may take on only the values TRUE or FALSE. Since the value of a compile time boolean expression is never an address, these may never occur within code brackets. The boolean operations themselves may occur within code brackets, specifying logical operations to be performed at run time. Boolean expressions are most often used in connection with conditional statements which will be discussed in the next section.

F. Statements

The statement types in FSL include most of the usual constructs found in programming languages. The unconditional statements are formed according to the following rules

$$\begin{aligned} \langle \text{unconditional} \rangle &::= \langle \text{storage} \rangle \mid \langle \text{assignment} \rangle \mid \\ &\quad \langle \text{transfer} \rangle \mid \langle \text{auxiliary} \rangle \mid \\ &\quad \text{CODE } (\langle \text{unconditional} \rangle) \end{aligned}$$

The storage statements are

$$\langle \text{storage} \rangle :: = \langle \text{stack command} \rangle \mid \langle \text{enter} \rangle$$

These were discussed in detail in Section III-B and III-C.

The assignment statement in FSL is more general than that usually found in programming languages. The syntax is:

$$\begin{aligned} \langle \text{assignment} \rangle &:: = \langle \text{left side} \rangle \leftarrow \langle \text{arithmetic} \rangle \\ &\quad \langle \text{left side} \rangle \leftarrow \langle \text{boolean} \rangle \\ \langle \text{left side} \rangle &:: = \langle \text{primary} \rangle \mid \text{VALUE1} \mid \text{VALUE2} \mid \text{VALUE3} \end{aligned}$$

Notice that the left side of an assignment statement can be any arithmetic primary, rather than just an identifier. This proves useful in translator writing, where addresses are often computed before being stored into. One must be careful in using this feature inside code brackets. Any address computation inside code brackets will be done at run time.

The use of a VALUE word as the left side of an assignment statement requires more explanation. Consider the semantic routine 10↓ in Appendix C, which is called whenever a multiplication in the small language is recognized.

$$10\downarrow \quad \text{CODE} (\text{VALUE2} \leftarrow \text{LEFT4} * \text{LEFT2}) \downarrow$$

We can see from the productions that the operands of the multiplication are second and fourth in the stack. Thus `LEFT4 * LEFT2` in code brackets will cause the code for the multiplication to be compiled. The problem is that we will want to use the result of this multiplication in some future operation. This is handled in the syntax by putting the symbol 'T' into the second position of the syntax stack. The purpose of `VALUE2` is to assign semantics to the 'T' which represents the results of the multiplication. It is important to note that the semantics is filled in at compile time. This represents the only deviation from the convention on operators inside code brackets and the special `VALUE` operands mark this fact. A store into `VALUE1` inside code brackets has the same effect as a store into `RIGHT1` outside code brackets.

The next type of statement to be considered is the jump or transfer statement. The syntax is:

$$\begin{aligned} \langle \text{transfer} \rangle ::= & \text{JUMP} [\langle \text{arithmetic} \rangle] \mid \text{JUMP} [\langle \text{label} \rangle] \mid \\ & \text{MARKJUMP} [\langle \text{arithmetic} \rangle] \mid \text{MARKJUMP} [\langle \text{subroutine} \rangle] \mid \\ & \text{EXECUTE} [\langle \text{arithmetic} \rangle]. \end{aligned}$$

The `JUMP` statement is the well known unconditional transfer of control in a program. If the operand is an arithmetic expression it is presumed to specify the address to be transferred to. Since an FSL user has no knowledge of absolute address in his translator, computed transfers can occur only within code brackets. Transfers outside of code brackets are

done with labels (in the translator itself). Any FSL statement can be labeled and a transfer can be made to any labeled statement. No label may occur within code brackets, but bracketed statements can be labeled.

The MARKJUMP command specifies a return-jump or subroutine-jump to its operand. If a MARKJUMP [β] occurs in location α two distinct actions take place: First the address $\alpha + 1$ is put in location β and then control passes to location $\beta + 1$. As with JUMP, a MARKJUMP to a computed address can occur only within code brackets. The operation MARKJUMP [<subroutine>] can occur either inside or outside of code brackets. These call various subroutines which are part of the FSL system. Calls of run time subroutines appear inside code brackets and calls of compile time routines occur only outside code brackets. The system subroutines will be described in Section IX.

The EXECUTE command requires an <arithmetic> as its operand and thus occurs only within code brackets. An EXECUTE will cause the command in the location specified to be executed and control to be returned to the statement following the EXECUTE. This is the same as a MARKJUMP to a subroutine of one instruction. The EXECUTE command is useful in sharing the code for two slightly different routines.

The auxiliary statements in FSL include a number of different types of commands which do not fit into the other categories. The syntax is:

```

<auxiliary> ::= SET [<arithmetic> , <data id>] |
              CLEAR [<arithmetic>] | FAULT <identifier> |
              STOP | TALLY [<arithmetic>] | MINUS [<arithmetic>]
              | ASSIGN [<arithmetic>].

```

The SET and CLEAR statements are used in tagging and removing tags from addresses. Both are discussed in III-C where it is shown that neither can appear inside code brackets. The FAULT statement enables the language designer to provide error messages in the event of a semantic error. For example, in 8↓ Appendix C, it is a FAULT if an identifier used arithmetically has not been declared REAL.

```

8↓  CONST[LEFT2] → RIGHT2 ← LEFT2 :
      SYMB[LEFT2,,$,]= REAL → RIGHT2 ← SYMB[LEFT2,$,,]:FAULT 1$$↓

```

After printing the message 'FAULT 1' the system will return to the next statement. In some cases one might then want to set OK to FALSE, eliminating all further code generation.

In any event, once a FAULT has been executed the system will not run the source program which had the semantic error.

The statement STOP may occur either inside or outside code brackets. When a STOP is executed the entire program will be stopped. A STOP outside code brackets will stop the translator, while a STOP inside will stop the running program. There is an action HALT in the production language

(entirely distinct from FSL) which causes translation to cease and execution of the compiled program to commence.

The commands TALLY and MINUS are merely shorthands for arithmetic statements. TALLY means add one to the operand specified and MINUS means subtract one from the operand. Either may be used inside or outside of code brackets and can lead to more efficient code. There are several instances of TALLY and MINUS in Appendix C.

The ASSIGN statement is used in connection with CHAIN and with FLADS. In either case the parameter to ASSIGN is the address of the head of a linked list. On this list are all the past references to the entity being assigned. ASSIGN has the effect of filling in the current value of CODELOC as the address portion of all these commands. If the parameter named a FLAD, a POP is then executed. For more information on chaining and forward references see Sections III-D and III-E. In Appendix C, routines 6↓, 7↓, and 27↓ handle the assigning of flads while chained labels are assigned in 26↓. An ASSIGN statement cannot appear within code brackets.

All of the statements described above were of the type <unconditional>. Except as noted, any of these may appear inside or outside code brackets. We also define a sequence of statements:

$$\begin{aligned} \langle \text{statement sequence} \rangle :: = & \langle \text{statement} \rangle \mid \\ & \langle \text{statement sequence} \rangle ; \langle \text{statement} \rangle \mid \\ & \text{CODE } (\langle \text{statement sequence} \rangle) \end{aligned}$$

We may also combine statements with boolean expressions to form conditional statements. The syntax is:

```

<conditional> ::= <if clause> <statement sequence> $ |
               <if clause> <statement sequence> :
               <statement sequence> $ |
               CODE (<conditional>)

<if clause> ::= <boolean> →

```

The two types of conditionals correspond to the ALGOL 60 conditional statements with and without 'ELSE'. By requiring the special punctuator '\$' to terminate all conditional statements, FSL is able to permit great flexibility in the construction of the two statement sequences. Either the 'true' part or the 'false' part of a conditional statement may be any sequence of FSL statements including other conditionals. This seems to be more convenient than other schemes considered for conditionals.

If a conditional statement appears inside code brackets, the tests and transfers will all be compiled. If not, the sequence of statements in either part of a conditional may still contain any combination of bracketed and unbracketed statements.

There are several occurrences of conditionals outside code brackets in Appendix C. For example, routine 8↓ contains a set of nested conditional statements. For examples of both types of conditional statements

within code brackets see routines 3 ↓ and 22 ↓. In these latter routines all the tests and transfers required will be put in the running program.

Any FSL statement may be prefixed by a label. The syntax is:

$$\langle \text{statement} \rangle :: = \langle \text{unconditional} \rangle \mid \langle \text{conditional} \rangle \mid$$

$$' \langle \text{label} \rangle ' \langle \text{statement} \rangle .$$

As was mentioned at the beginning of this section, labels may not occur within code brackets. The quotation marks set off the label in the FSL program and are not used when the label in the operand of a JUMP command. These labels are internal to an FSL sentence and should not be confused with integer names linking the semantics with the productions of Section II.

G. Subroutines

Subroutines appear in many forms in FSL. In this section, we will discuss the use of subroutines which are part of the system itself. To implement subroutines in a particular source language, one would use the MARKJUMP command described in Section III-F.

All of the source languages described in FSL will have access to a set of library subroutines. The library will be expandable, but will contain at least the subroutines described in the syntax.

`<subroutine> :: = SIN | COS | LOG | EXP | SQRT | ARCTAN | SIGN`

To process a call on one of these routines the language designer will use the systems run-time cells, the RUNT's. Each library routine expects to receive its parameters in RUNT 0 ... RUNT n where n+1 is the number of parameters involved. The value of any system subroutine will be left in RUNT 0.

After putting the parameters into the RUNT's the proper subroutine must be called. Library routines are called by a statement of the form

`MARKJUMP [<subroutine>].`

The library subroutines are available only at run time and thus any call on a library routine must occur within code brackets.

In addition to the library routines, there are some subroutines which are part of the system at compile time. The purpose of these routines is to relieve the FSL user of handling the details of some complex operations executed at compile time.

We have already encountered some examples of compile time routines in the discussion of tables in Section III-B. The operations of entering data into tables and of retrieving it from them were described as implicit subroutine calls. The one explicit call of a compile time subroutine is used in connection with the allocation of a storage in a source language program.

`<subroutine> :: = SIN | COS | LOG | EXP | SQRT | ARCTAN | SIGN`

To process a call on one of these routines the language designer will use the systems run-time cells, the RUNT's. Each library routine expects to receive its parameters in RUNT 0 ... RUNT n where n+1 is the number of parameters involved. The value of any system subroutine will be left in RUNT 0.

After putting the parameters into the RUNT's the proper subroutine must be called. Library routines are called by a statement of the form

`MARKJUMP [<subroutine>].`

The library subroutines are available only at run time and thus any call on a library routine must occur within code brackets.

In addition to the library routines, there are some subroutines which are part of the system at compile time. The purpose of these routines is to relieve the FSL user of handling the details of some complex operations executed at compile time.

We have already encountered some examples of compile time routines in the discussion of tables in Section III-B. The operations of entering data into tables and of retrieving it from them were described as implicit subroutine calls. The one explicit call of a compile time subroutine is used in connection with the allocation of a storage in a source language program.

The task of allocating storage and building access functions for complicated data structures is among the most difficult to be found in translator writing. There is in FSL a Storage Allocation Routine, SAR, which helps the designer solve this problem. In its present form, the SAR is capable of handling rectangular arrays of any dimension whose elements are all of the same precision.

The SAR expects a description of the data structure to appear in the compile time temporaries, the COMT's. The contents of COMT 0 will be a '1' if storage for an array is desired and should be '0' if a simple variable is to be allocated space. The contents of COMT 1 should be the number of words per element (precision) of the array and COMT 2 the number, n , of dimensions of the array. The next $2n$ COMT's will contain the lower and upper limits for each dimension of the array. Each of those must be a positive or negative integer which is fixed at compile time.

After all the parameters have been filled in, a statement of the form

MARKJUMP [SAR]

should appear in a semantic routine. The SAR will then allocate storage for the array and build an access function for later use. The SAR will leave an integer in COMT 0 when it is finished. This integer is the internal name of the array and will presumably be put into a symbol table.

The task of allocating storage and building access functions for complicated data structures is among the most difficult to be found in translator writing. There is in FSL a Storage Allocation Routine, SAR, which helps the designer solve this problem. In its present form, the SAR is capable of handling rectangular arrays of any dimension whose elements are all of the same precision.

The SAR expects a description of the data structure to appear in the compile time temporaries, the COMT's. The contents of COMT 0 will be a '1' if storage for an array is desired and should be '0' if a simple variable is to be allocated space. The contents of COMT 1 should be the number of words per element (precision) of the array and COMT 2 the number, n , of dimensions of the array. The next $2n$ COMT's will contain the lower and upper limits for each dimension of the array. Each of those must be a positive or negative integer which is fixed at compile time.

After all the parameters have been filled in, a statement of the form

MARKJUMP [SAR]

should appear in a semantic routine. The SAR will then allocate storage for the array and build an access function for later use. The SAR will leave an integer in COMT 0 when it is finished. This integer is the internal name of the array and will presumably be put into a symbol table.

When a reference to an element in the array occurs in the source code another compile time routine, PLACE is executed. This routine compiles code to generate the proper element of the array as an operand at run time. Like the table operand, this operand involves an implicit call of a compile time routine. However, certain information must be in the COMT's before the use of a storage retrieval statement.

The contents of COMT 0 should be the internal name of the array supplied by the SAR. It should be set to the data type of the array element (cf. Section III-B). If the array was n dimensional, the next n COMT's will describe the subscripts of the array element. Each description of a subscript is of one of two types. If the subscript appears in the source code as a constant, the constant itself will describe the subscript. If the subscript is to be computed, information on where the value of the subscript will be at run time will be provided in the COMT's. If the language designer uses the main stack in the usual way, this distinction is not important to him. His statements would probably be of the form

COMT 3 ← LEFT2, etc.

After the subscripts have been described a retrieval routine must be called. The purpose of this routine is to compile the code which will access the array element. To the FSL user, what is important is the address of this element, which will be used in compiling more code. A

statement of the form

CODE (VALUE1 $\leftarrow$ LOC [PLACE])

will produce code to find the location of the element and will also put the appropriate information in the stack at compile time. The information is placed in the desired stack position by using the appropriate VALUE operand as described in Section III-D.

Notice that the statement above will leave a description of the location of the array element in the stack. This is useful for multiple precision arrays where a number of operations may be performed on the several successive locations which comprise the array element. To retrieve the array element in normal addressing mode, one would use a statement of the form

CODE (VALUE1 $\leftarrow$ PLACE)

which will leave the description of the array element in the stack with the normal addressing mode. Since PLACE involves actions both at compile time and at run time, it is a special case and may only occur as described above.

The use of the SAR and of PLACE as described above is so complicated that one might question its usefulness to the language designer. There is a system cell, STORLOC, in FSL which points to the next available

storage location and is used by the SAR. The language designer can do his own storage allocation using STORLOC and, in fact, this is done in the small language where there are no subscripted variables. It seems likely that the problem is intrinsically complicated and that no simple algorithms for array accessing exist.

There is another reason for using the system routines in storage allocation. Subscripting is often done quite differently on different computers. The relative efficiency of using index registers or other special features is a consideration best left to the system as it functions on a particular machine. The description above specifies no particular implementation. It is also expected that the SAR will eventually include provision for other data structures such as linked lists and files. The main reason these were not put in is that there is as yet no precise and generally accepted operational definition of either construct. The FSL system is set up so that more system subroutines can be easily added if the need arises.

H. Program Structure

Any program in FSL is the complete semantic description of some programming language. This description is also the specification of a translator for that language. Structurally, an FSL program consists of two separate parts, a declaration part and the main body.

The declaration part of an FSL program describes the storage to be used by the translator. The several types of storage were discussed in Sections III-B and III-C. The syntax is:

$$\begin{aligned} \langle \text{declaration part} \rangle &::= \langle \text{declaration} \rangle \mid \\ &\quad \langle \text{declaration part} \rangle ; \langle \text{declaration} \rangle \\ \langle \text{declaration} \rangle &::= \langle \text{table dec} \rangle \mid \langle \text{stack dec} \rangle \mid \langle \text{cell dec} \rangle \mid \\ &\quad \langle \text{title dec} \rangle \mid \langle \text{data dec} \rangle \end{aligned}$$

All of the declarations must occur together at the beginning of an FSL program. The remainder of the program consists of the semantic description of individual constructs in the source language.

These semantic descriptions of source language constructs are linked to the syntax (productions) by means of statement numbers. Each numbered semantic routine corresponds to an action 'EXEC' appearing in the syntax of the source language. Those semantic descriptions are combined with the declarations to form the semantic description of a particular source language. The syntax is:

$$\begin{aligned} \langle \text{semantic program} \rangle &::= \langle \text{program head} \rangle \text{ END} \\ \langle \text{program head} \rangle &::= \text{BEGIN } \langle \text{declaration part} \rangle ; \langle \text{sentence} \rangle \mid \\ &\quad \langle \text{program head} \rangle ; \langle \text{sentence} \rangle \\ \langle \text{sentence} \rangle &::= \langle \text{integer} \rangle \downarrow \langle \text{statement sequence} \rangle \downarrow \end{aligned}$$

The construct <statement sequence> has been discussed in III-F. The <integer> is the statement number mentioned in the preceding paragraph. The complete formal syntax of FSL may be found in Appendix D.

In its present form, FSL includes no constructs for formally describing input-output operations. We have chosen, rather, to include a common input-output language in each source language in the system. This input-output language is very similar to the one described by Perlis [32].

This approach has been taken for several reasons. It seems to be in the nature of input-output languages that they depend more on hardware than on the source language into which they are imbedded. Requiring each language designer to specify a complete format language for his translator would contradict the aim of the project. In the current system, the language designer need worry about input-output only at a very superficial level.

To imbed the entire input-output language in a source language, the FSL user must make two additions to his formal description. In the syntax (productions) of the source language, he must check for the occurrence of certain special characters. The appearance of one of these characters specifies that the following source code is a format description. Upon the scanning such a symbol the productions of the source language are expected to execute a transfer to a predefined label. In addition, the user must place a predefined label on the production to be executed at

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, 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 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.

IV. IMPLEMENTATION

The formal semantic language described in the previous chapter was originally developed with no implementation in mind. Once a complete and precise formalism was described, it was felt that it could easily be put on the machine. While it had been planned to stop short of implementation, we were moved to implement precisely because so many other efforts had stopped at just this point. Predictably, the nature of the formalism underwent many changes during implementation.

When, in August of 1963, we became committed to implementation, the natural first study was of translators already running on the Carnegie Tech computer. The hope was that we could adapt an existing compiler to the task of building a compiler-compiler. There were two reasons for choosing this path rather than starting afresh. Obviously, this method would be quicker and easier. The second reason was that part of the theory behind the system states that many translation tasks are independent of the language being translated and thus an existing translator should suffice.

At this time the Carnegie Tech ALGOL compiler [13] was just beginning to operate as a full-fledged running system. The syntax phase of the ALGOL translator was based on a version of Production Language and operated in an almost language-independent fashion. There were two difficulties involved in using the ALGOL syntax phase. The productions used for ALGOL were based on the conversion of the source code into a Polish

postfix intermediate language. Further, this result was achieved by using a hierarchy table to implement the binding rules for the various ALGOL operators. The problem was that neither of these techniques was easily adaptable to a general purpose translator.

With great ingenuity, A. Evans devised a scheme whereby such ALGOL separators as FOR, ELSE, and WHILE would fit into the postfix form and the hierarchy system. We were loath to require such ingenuity of every designer using our system. Besides, we felt that an intermediate language such as postfix would add complexity to a system that already threatened to become unmanageable.

The solution to these two problems are interrelated. Instead of hierarchies we have implemented grouping rules by changing the names of constructs in the syntax stack. This technique is in essence inverting the BNF grammar and is described in Section II-A. This technique is easy to use and has the additional advantage that it is fail-safe. As each construct is recognized by the productions an appropriate action is taken. If a construct would lead to the generation of code, this code is built at the time the construct is recognized.

Fortunately, the implementation of these changes required relatively minor alterations in the ALGOL production loader. These were carried out and we were then in possession of a general purpose syntax phase. As we have mentioned in Section I, the syntax of a source language is

handled in two steps. The productions of the source language are fed into the Syntax Loader which produces from them a set of tables. These tables are then imbedded in the Basic Compiler and determine the syntax phase of a translator for the source language.

This syntax technique had one marked advantage over previous table driven compilers. Since the productions were recognizer-oriented and non-recursive the syntax phase of the translator could be executed quite rapidly. The slow operation of recursive recognizers had even led to statements that no general purpose recognizer would ever be efficient. Although this difficulty was now overcome the real problem was getting a general purpose system to produce code.

Part of the code generation problem is no more difficult in a compiler-compiler than in conventional translators. Individual pieces of code are usually generated by small special purpose routines. For example, there might be a generator for addition and subtraction statements, one for transfers and one for conditionals. The nature of these generator routines depends very little on the source language being translated. However, if the code produced is to be at all efficient, the generators must have a great deal of information about in-transit states of the translator during the code production process.

Since we would be trying to generate code directly from the main stack, it was convenient to keep much of the information in the stack.

This was accomplished by having each element of the stack consist of two machine words. The first of these two words holds the syntactic type of the element in the stack and is used by the syntax phase in matching productions. The second word contains the semantics of the construct in the stack and is used by the code generators. In an FSL language description, a reference to LEFT3 specifies the second word of the element in the third position of the main stack.

The basic unit of information in any compiler is a machine address. The semantic word in the main stack normally consists of an address and some identification bits. The following table shows the use of the G-20's 32-bit word in describing an operand. In all cases the bit is set if the predicate is true.

Bit	Use
0 - 14	Address
15	Integer Constant ?
16	Floating Constant ?
17	In Accumulator ?
18	Floating Address (FLAD)?
19	In a Temporary ?
20	Negation (Arithmetic or Logical)?
21 - 24	Data Type: LOGIC, INTEGER, ...
25	COMT ?
26	RUNT ?
27 - 29	Addressing Depth (0 - 7)
30 - 31	Interrupt Flags

Figure 3.

It is not within the scope of this paper to explain how all of this information is used by generators to produce G-20 machine code. What is of interest here is that the language designer using FSL need not deal directly with the complex problem of code generation. For example, consider the statement describing the addition of two numbers in the small language (l2↓ of Appendix C)

l2↓ CODE(VALUE2 ← LEFT4+LEFT2) ↓

When a production with the action 'EXEC l2' is matched, an addition is to be compiled. Using the information in the stack positions LEFT4 and LEFT2 the system is able to compile locally optimal code from the designers simple semantic statement. For example, one of the operands may already be specified as being in the accumulator at run time. If one operand is of the wrong sign a 'subtract' or 'reverse subtract' command must be compiled. These and many other tests are handled automatically by the system. In short, any local optimization techniques available on a particular computer should be feasible in an FSL system.

The significance of VALUE operands should now be much clearer. In processing an addition statement using l2↓, one not only must generate code but must also assign semantics to the position in the main stack which is to hold the result. After the addition has been processed the result is to be placed in the second stack position, hence VALUE2. In this case RIGHT2 will have the syntactic type 'E' and semantics specifying

that its value will be in the accumulator at run time. Should the system need the accumulator for another computation, it would automatically compile a 'STORE' command and change the semantics of 'E' to indicate that its value will be in a temporary.

The writing of generators and auxiliary routines to use this semantic information was not very difficult. The complex problems arose in trying to build the semantic descriptions. The goal was to establish a system whereby the proper semantic information could be assembled with as little effort as possible on the part of the FSL user. To discuss this topic we must present the third basic program in the system, the Semantic Loader.

As we mentioned in Chapter I, the system consists of a Production Loader, a Semantic Loader and a Basic Compiler. We present the Semantic Loader last because, in a sense, we have already discussed its essential properties. The semantic meta-language, FSL, is a well-specified programming language and thus its syntax is expressible in Production Language. This Production Language syntax could then serve as the basis for an FSL translator. The Semantic Loader is built in precisely this way -- the productions for FSL constitute Appendix E of this paper.

These productions were processed by the Production Loader which produced a set of syntax tables to be used by the Basic Compiler. These productions had calls on semantic routines just as those in source

languages. The semantics of FSL, however, were written in G-20 assembly language. In addition, some minor changes were made in the Basic Compiler to facilitate its use as a special purpose translator. We will discuss the question of writing the FSL semantics in FSL in the next chapter.

Many interesting problems arose in coding the semantics of FSL. We were attempting to compile efficient programs which would in turn compile efficient programs. One of the greatest difficulties was keeping track of the level at which each action was to occur. To aid in this distinction we use the terms meta-compile time, compile time, and run time. It is important to remember that communication goes only one way in this chain. We will not permit situations where the compiler must refer to the meta-compiler for information in processing a piece of source code.

As one might suspect, the main stack in the Semantic Loader is arranged just as that of the compiler described at the beginning of this chapter. This means, among other things, that the same optimization techniques are available at both levels. This reflects the basic tenant of the design philosophy that the Semantic Loader and the Basic Compiler be as similar as possible.

Keeping these two programs nearly identical was a key step in the implementation of the system. To see why this was important, we must



consider the function of the Semantic Loader. The semantic description of a source language will specify some actions to be performed at compile time and some to be executed at run time. The semantic loader will generate code (for the compiler) to execute compile time actions and will produce calls on compile time generators to build the code for actions to be performed at run time. In other words, any FSL operation occurring outside code brackets will lead to meta-compile time generation and compile time execution of code. An operation that appears within code brackets will be processed by a compile time generator and be executed in the running program.

It should now be easy to see the advantage of keeping the Semantic Loader and Basic Compiler similar. In our system, the two programs share the same basic generators, auxiliary routines, and tables. To simplify matters further, the Semantic Loader builds code for the compiler in the same locations as that code will occupy at compile time. Considerable pains were taken to have addresses coincide in the two programs so that symbolic addresses could be used in the Semantic Loader and the correct absolute addresses would appear in the semantic tables.

From the discussion above it should not be difficult to understand the operation of the processors for the usual arithmetic and boolean operations. Many of the other FSL constructs have no counterpart in the compiler and are thus handled non-uniformly in the Semantic Loader. We will discuss some of these to show how the system operates. In view of

the table in Figure 3. it should be easy to discern the implementation of the various operations on DATA variables: SET, TEST, CLEAR. Each of these leads to the appropriate bit commands on the data type field (bits 21-24) of the specified operand.

Similarly, the indirect addressing operations effect bit operations on the addressing level portion (bits 27-29) of a word. The predicate CONST leads to a test for the presence of the bit marking a constant. All of these operations are built into code for the compiler. Some of the more complicated operations are implemented by calls on compile time subroutines.

When a compile time routine was used to implement an FSL construct, considerable effort was used to take advantage of the special features of the G-20. Such operations as entering data into tables and retrieving it from them involved fairly complex subroutines at compile time. The handling of chaining and FLAD's was also done with closed routines. The calls for these compile time routines are processed exactly as calls on generators. The only serious difficulty in this process concerns the description of operands for the compile time generators and routines.

Like all high level programming languages, FSL depends on a great deal of implicit information. Each operand occurring in an FSL statement will be translated into a 32-bit semantic word, (see Figure 3.) at meta-compile time. If the operand is to be used by a compile time

generator or routine, the appropriate semantic word must be built at compile time. The routine to make meta-compile time descriptions into compile time ones is as complicated as any in the system. If the operand is a constant, its description at compile time is the same as at meta-compile time. For RUNT or COMT variables, the semantic word specifies a subscript which might be fixed or variable. References to stacks, variables and FLADs are handled indirectly. The communication of addressing depth is also a problem. Thus if the contents brackets occur within code brackets, the indirect addressing is to be done at run time and this information must be given to the compiler. Further, there is an implied extra level of indirectness for compile time operands. Since all of these considerations are treated by the one operand preparation routine, the rest of the Semantic Loader is relatively straightforward.

The entire Semantic Loader was built in modular form. This modular organization was natural for a production-based translator and proved invaluable in the development of the system. As users discovered operations which would improve the system, these were added without changing the rest of the system.

One of the most valuable features of our implementation was its similarity to the ALGOL translator. The ALGOL system carries most of the usage load at Carnegie Tech and underwent considerable independent development during the course of this project. As various debugging aids were added to ALGOL, we were able to incorporate these in the FSL system

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.

A. Review and Critique

In this chapter we will review the principal developments in the paper and point out both weaknesses and possible extensions of the system. The basic idea is that, through appropriate formalization, much of the difficulty involved in translating computer languages can be eliminated.

The formalization of semantics for computer-oriented languages requires that syntax and semantics be carefully separated. In Chapter I we defined notions of syntax and semantics and the concepts of syntax meta-language and semantic meta-language. After comparing our definitions with alternative ones, we described how a compiler-compiler, based on our definitions of syntax and semantics, could be organized.

Chapter II contained a description of the syntax meta-language, Production Language, used in our system. Section II-A was basically a users guide to Production Language as a programming language. The remainder of that chapter discussed some of the more abstract properties of syntax meta-languages with the emphasis on the properties of the Production Language. It was shown there that, under various restrictions, Production Language could be considered either as quite weak or as extremely powerful in comparison with other formal systems.

The longest and most important part of this thesis is Chapter III, which discusses the Formal Semantic Language (FSL). Although many syntax meta-languages have been developed over the years, FSL is the first attempt to provide an adequate semantic meta-language for computer-oriented languages. Many of the features of FSL are formalizations of techniques which have proved useful in translator writing. Among the more interesting constructs are the operations on tables and stacks and the facilities for handling indirect references. Probably the most important feature is the clear distinction between actions taken at translate time and those performed at run time. In Chapter III FSL was described strictly as a language for describing the semantic phase of translators.

The implementation of one general purpose translator was described briefly in Chapter IV. By making use of programs already running on the G-20, we were able to implement the system in a relatively short time. The most difficult problems were those involved in the interface between the three separate programs which comprise the compiler-compiler. Those problems were solved and many peripheral advantages gained by using a high degree of parallelism in the component programs.

One of the topics discussed in the earlier chapters was a set of criteria for semantic meta-languages. We mentioned that a semantic meta-language should be easy to use and should be readable so that others can understand the languages described in it. It should be sufficiently

flexible to allow descriptions which can specify an automatic translator for any of a large and varied class of source languages. Further, a good semantic meta-language should be independent of any particular computer. An additional and very important consideration is the quality of translators produced by such a system.

The Formal Semantic Language, FSL, embodies an attempt to satisfy all these conditions. There is now considerable evidence that FSL is easy to write in and not too difficult to read. The justification of these statements is based on the author's success in teaching FSL to undergraduates.

The system was presented in an advanced undergraduate programming course at Carnegie Tech. None of the students in either term had any previous experience in translator design, but all had taken the equivalent of two semesters of elementary programming. The material presented in the course was the non-theoretical parts of Chapters II and III of this thesis. In the sixth week of each semester the students chose projects which would constitute the main part of their semester grade. Among the topics suggested were the construction of translators for some common programming languages.

The projects in the first term class were not completed on time, largely due to errors in the FSL system itself. These projects were of tremendous value in pointing out coding errors and conceptional

weaknesses in the system. Many of the FSL constructs described here were first suggested by problems encountered in these projects. However, student interest in the project remained high and almost all enrolled for the second term so as to complete their projects.

During the second term the students had a much more stable system to work with. The course outline was accelerated to allow an earlier start on the term projects. Although the FSL system still was not completely debugged, the better students were able to complete translators during this term. A detailed presentation of these projects will be released later this year as a research report.

One of the interesting features of the second term class was the use of students as instructors. One hour a week was denoted to progress reports by students working on term projects. The discussions included detailed presentations of the formal syntax and semantics of the language under consideration. The class was usually able to follow the presentation in sufficient detail to suggest possible flaws in the translator being designed.

Of interest here is that a group of new programmers (undergraduates) were able to learn, use and expound the FSL system in a one term course. Besides learning the FSL system, the students were forced to consider and solve many of the unique, logically complex, and generally tedious problems involved in translator writing. The problems, such as local

optimization of code, which are handled automatically by the system were discussed in supplementary lectures.

Among the translators attempted by students were ones for such diverse languages as ALGOL, LISP, FORTRAN, COMIT, and SIMSCRIPT. Not all efforts were equally successful, but it is difficult to determine whether the discrepancy is due to the difference between students or between languages. We can, however, make some remarks about the class of languages for which the FSL system is adequate.

The FSL system is for the description of languages which can be compiled, i.e., those for which the course of action can be fixed at translate time. If a language involves constructs whose meanings are not available until run time, one must, of course, use run time routines as part of a translator for this language. There are provisions for constructing run time translator routines in FSL, but they are not as convenient as they might be. For this reason, among others, LISP and COMIT were somewhat tricky to design.

The other difficulty in LISP and COMIT was the absence of fixed FSL primitives to act on linked lists. These were not put into the system because, as yet, there is no universally accepted way of representing and operating upon such lists. Similarly, the SIMSCRIPT effort was made more difficult by the absence of any good primitives for operating on data files. All of these difficulties can be overcome,

but an ideal system would make them unnecessary.

A much more serious difficulty arises in connection with using FSL to describe assembly languages. An assembly language consists of the symbolic representation of a machine order code along with certain high level operators called macro-operations and pseudo-operations. The crucial feature is that an assembly language has a basic 1:1 correspondence with the computer on which it is to run. There is no simple way to capture this essential feature in FSL.

The missing concept here is a formalism for describing the characteristics of a computer. This is one of the most important problems in the theory of computing at the present time and will be discussed further in Section V-B. This same difficulty occurs in trying to describe FSL in FSL. In order to build good translators, one must make a particular implementation of FSL highly dependent on the characteristics of the object machine. One could write the semantics of FSL in FSL, but the result would not be at all indicative of the structure of the language. This is not so surprising when one recalls that FSL was designed to replace assembly languages in describing translators.

The remaining question concerns the quality of translators produced by an FSL system. This is very difficult to evaluate because there are many qualities of a good translator and these are not all mutually compatible. Language designers have always had to evaluate the relative

emphasis to be placed on time and storage space at compile time and at run time. In addition, error detection and recovery features are a major consideration.

We have attempted, in the FSL system, to allow the individual language designer to make these choices for himself. All of the translators designed so far have been influenced by the nature of the program load at Carnegie Tech. Since there are a large number of student runs and input-output is on-line, at least half of the computer's time is spent translating programs. For this reason, a translator which produces marvelous code at the expense of compile time is not locally desirable.

The translators for algebraic language produced by the FSL system seem comparable with hand coded translators on the G-20. The compile speeds are equally good, (most of the time is spent in character scanning in any case) and the code produced is locally good though the code does not have interesting global features. In Section V-B we discuss the question of producing highly efficient code using an FSL system.

A major weakness of the current system is the space it occupies at compile time. This arises because all features of the system are present in every translator. This difficulty can be eliminated by using a relocater routine which loads only those features needed for a particular language. For example, the small language translator could

be made about 20 per cent smaller by these techniques.

The error detection and recovery features of the system are potentially quite good. An example of an elaborate error scheme for ALGOL 60 using the Production Language may be found in reference [13]. Several debugging aids are also included as a basic part of the FSL system.

The subject of machine independence is one on which we have little information. In the discussion of implementation in Chapter IV we described a technique for using existing translators as part of an FSL system. There is no apparent reason why such a system could not be implemented on any large general purpose machine. However, since there is only one extant implementation of FSL, no more precise claim can be made at this time. These questions on the adequacy of FSL will be considered further in the next section, which deals with areas for future research.

B. Future Research

As mentioned in the preceding section, there are many improvements which could be added to the FSL system. There are several programming constructs, notably linked lists, which have not yet been formalized in the system. A relocater would enable the Basic Compiler to use only required space at translate time. With a relocater comes the ability to add new routines to the repertoire of the Basic Compiler directly in an FSL program.

Another set of additions would be useful for improving the quality of code produced by the system. Although global optimization rules such as the recognition of common subexpressions can be programmed in FSL, we would suggest an alternative course of action. At least on the G-20, much global optimization can be done on the resulting object code. With very little information about the source language (e.g., location of brackets and branch points) one can produce remarkably good code. A routine to perform these operations will be added to the system with a switch making its use optimal. This has the advantage that it will be available to all users of the FSL system, but need not be used in programs not yet debugged.

We also plan to add a complete assembly language to the FSL system. While this is distasteful from an abstract viewpoint, it will enable one to utilize machine dependent features when their use is warranted. This should not interfere with communication between humans because the published version of a translator would be written wholly in FSL with notes about the parts which were recoded in assembly language. This technique is used currently in communicating algorithms written in problem-oriented source languages such as ALGOL or LISP.

Another set of improvements concern the input text character scanner which is part of all languages written in FSL. At the present time the scanner contains several fixed conventions such as which characters can be juxtaposed to form an identifier. One would like a set of declarations

in the productions of source language to describe such variable conditions as fixed field scanning as well as those mentioned above.

The major weakness in the system is its inability to describe assembly language and other machine dependent systems. Since the FSL system is equivalent to a Turing machine and hence, "all things are possible", we must clarify what is meant by 'inability'.

If one were given the task of writing an assembler for the G-20 in FSL there are two basic approaches one could take. Since an assembly language is just another computer language, we could write it machine-independent semantics just as we would those for ALGOL. This would have the pleasant property that the assembly language would function on any machine having an FSL implementation. The problem is that on the G-20 the code produced by this assembler would be intolerable. Where a conventional assembler is 1:1 with machine code in its basic operations, the FSL assembler would, in essence, be simulating the computer on itself.

There is another way of attacking this problem in FSL. Since we know the internal representation of G-20 commands we could build a table of them using FSL. FSL also contains operations for uniting in, and extracting off, bit patterns so we could build G-20 commands much as a conventional assembler would. Although it might take longer, there is every reason to suspect that the FSL assembler would create good code. With this approach the difficulty is that the FSL semantics does not

describe the nature of the language being described and would be absolutely useless on a machine different than the G-20.

The solution to this dilemma requires a formal language capable of expressing the properties of a computer, as FSL does for a computer language. Since there is no clear dividing line between the properties of computers and of computer languages, this additional language might well be an extension of FSL. The difficulties involved in formalizing the properties of computing machines and machine-like languages are immense. Although no solutions exist at this time, there are several efforts in this direction and results can be hoped for in the near future.

An extension to FSL which would allow the process description of computers would do far more than solve the problem of building assembly languages. If one had such a language the model of the compiler-compiler could be extended to include a formal description of the target (object) language. This description could then be processed by a program which would produce generators to be used by the old FSL system. The object language would always be machine-like but the source language could vary in type. If the source language were a problem-oriented language like those discussed in this paper, the compiler-compiler would be the same except that it could produce a variety of machine and assembly codes. Assembly languages could be designed so that they would work efficiently on machines that matched their characteristics. Further, if the source

language were also machine-like, the compiler-compiler would produce a simulator of the source machine on a target machine.

It appears that the picture above assumes that once the meta-language is prescribed there will be little difficulty involved in the construction of processors. Our experience with programming in general and FSL in particular indicates that this is largely the case. Perhaps the point is that a sine qua non for an adequate computer language is that it contain an implicit description of its processor. In any event, a formalization of the properties of computers would be a major advance in the theory of programming languages.

The introduction of a semantic meta-language such as FSL raises another set of questions entirely distinct from those considered above. These all deal with the properties of FSL as a formal system. The most interesting question regards the relationship of FSL to various notions of formalized semantics used in mathematical logic. A preliminary investigation of this question left us with no concrete results.

One might also want to use the idea of formal semantics to define a notion of formal translation. Intuitively, one would describe translation as a process which preserves meaning. The many treatments of translation as a purely syntactical process, while very valuable, do not capture this intuitive concept of translation. If there were a fixed semantic meta-language, one could discuss translation as a process

conserving semantics in that meta-language. This would leave the problem of determining the equivalence of two statements in the meta-language, but at least it would create a well-defined problem area where before there was none.

Among the questions one might want to ask are the following.

—Is this program in ALGOL equivalent to that LISP routine? Is language A strictly weaker than (or equivalent to) language B? Can I find a shorter program in another language which will have the same meaning as this one? While all of these questions will all still be undecidable in the formal sense, representation in a fixed meta-language should make particular cases more tractable, i.e., provide representations for special proofs.

We have presented in this paper a number of ideas, some borrowed, some new, about the formalization of computer-oriented languages. The most significant was a semantic meta-language, FSL, which seems to provide new opportunities in several areas of programming research. Besides its use on a compiler-compiler, FSL has proved useful in teaching language design and shows promise of helping to provide insight into the nature of computer-oriented languages.

Whatever the merits of our particular system, some such formalized system will have a marked effect on language design. Picture, if you will, a time when there are standard formalisms for expressing the syntax

and semantics of computer languages.

Then suppose a language designer devises a language for expressing a certain class of problems. First of all, in specifying formally its syntax and semantics, he is forced to make his ideas precise. Since the formalisms are translator oriented it will be hard to specify those constructs which are unnatural to implement. Once the language is specified the meta-compiler will automatically produce a translator on which the designer can try his problems. Should the language prove acceptable he could then publish his formal description much as algorithms are now published in the Communications of the ACM. But, this published version is also the complete specification of a translator from his language to any machine which has an implementation of the formal system. All of this does more than present an intriguing picture, it describes a situation which should be realized in the near future.

APPENDICES

Notes on the Appendices

The material contained in these appendices forms an integral part of the thesis. The basis for most of them is one or more machine listings. In all cases these are actual runs of the programs being described.

Appendices A-C comprise a complete description of the subset of ALGOL 60 which we call the small language. The BNF syntax of the small language which is Appendix A is what might be given in a published paper on the small language. The Appendices B and C are a solution to the problem of building a compiler for the small language.

Appendix B contains a run of the production loader on the syntax of the small language. The results of this run are some tightly packed tables which will control the scanning of small language source programs. The notation used in these tables is described in that section.

Appendix C contains a run of the semantic loader on the FSL semantics of the small language. The tables produced there are actually short programs which will be executed by the Basic Compiler. We will give a capsule description of the G-20 machine code at the end of these notes. This description will also be of use in reading Appendix F, which contains the translations of sample programs in the small language.

Appendices D and E describe the semantic meta-language, FSL. The Backus Normal Form syntax constitutes Appendix D while the Production Language Syntax is Appendix E. The tables produced in Appendix E follow the same format as those for the productions of the small language (Appendix B).

In order to fully understand the appendices, one must have some knowledge of G-20 machine code. We will present a brief description of the machine, emphasizing its differences from the better known computers of its class.

The Control Data (née Bendix) G-20 at Carnegie Tech is a large single address machine. Among its more interesting features are a floating point accumulator and a special operand assembly (OA) register. These are the only hardware registers with which we will be concerned. Although the G-20 has 64 index registers, these are in its 6 μ -second core memory and thus have appreciable access times. We will describe the G-20 command structure as it will appear in Appendices C and F. The format of a command is:

```
f xxx m dddd , ii
```

where 'f' denotes the interrupt flag (0-3) used by the machine hardware. The symbols 'xxx' denote one of the 3-letter mnemonic opcodes which we will describe below. The 'm' denotes the addressing mode (0-3) and will also be described below. The field marked 'dddd' stands for the five

digit octal address used in a G-20 command and the 'ii' denotes one of the octal 100 index registers (0-77). The internal representation of a command is slightly different than our picture.

Before describing the commands, we must say something about the address structure of the G-20. This is based on the use of the OA register which automatically combines with the address field of every command according to the mode specified in the command. If we represent the address field of a command by A, the index field by I and the contents of OA register as (OA) we attain the following rules for building X, the effective address.

<u>Mode</u>	<u>Effective Address</u>
0	$(OA) + A + (I) = X$
1	$(OA) + (A) + (I) = X$
2	$((OA) + A + (I)) = X$
3	$((OA) + (A) + (I)) = X$

The normal mode for stores and transfer commands is mode 0, while all other commands are normally in mode 2. There are special commands for operating on the OA and, except for these instructions, the OA is set to zero after every command.

In the table of opcodes (Figure 4), (ACC) stands for the accumulator and all the symbols defined above retain their meanings. Notice that for all conditional statements the next command is executed if the

Partial List of G-20 Opcodes

Address Preparation		Index Register Codes	
OCA	$X \rightarrow (OA)$	LXP	$X \rightarrow I$
OCA	$-X \rightarrow (OA)$	LXM	$-X \rightarrow I$
OAD	$(ACC) + X \rightarrow (OA)$	ADX	$(I) + X \rightarrow I$
OSU	$(ACC) - X \rightarrow (OA)$	SUX	$(I) - X \rightarrow I$
Add and Subtract		AXT	$(I) + X \rightarrow I \quad (=0?)$
CLA	$X \rightarrow (ACC)$	SXT	$(I) - X \rightarrow I \quad (=0?)$
CLS	$-X \rightarrow (ACC)$	Transfer of Control	
ADD	$(ACC) + X \rightarrow (ACC)$	TRA	$X \rightarrow NC$
SUB	$(ACC) - X \rightarrow (ACC)$	TRM	$(NC) \rightarrow X; X + 1 \rightarrow NC$
ADN	$-(ACC) - X \rightarrow (ACC)$		
SUN	$-(ACC) + X \rightarrow (ACC)$		
ADA	$(ACC) + X \rightarrow (ACC)$		
SUA	$(ACC) - X \rightarrow (ACC)$		
Arithmetic Tests			
FOM	$X < 0$		
FOP	$X > 0$		
FLO	$(ACC) < X$		
FGO	$(ACC) > X$		
Multiply and Divide			
MPY	$(ACC) * X \rightarrow (ACC)$		
DIV	$(ACC) / X \rightarrow (ACC)$		
RDV	$X / (ACC) \rightarrow (ACC)$		
Logic Operations			
CAL	$X \rightarrow (ACC)$		
CCL	$X \rightarrow (ACC)$		
ADL	$(ACC) + X \rightarrow (ACC)$		
SUL	$(ACC) - X \rightarrow (ACC)$		
EXL	$(ACC) \wedge X \rightarrow (ACC)$		
UNL	$(ACC) \vee X \rightarrow (ACC)$		
Logic Tests			
IOZ	$X = 0$		
IOZ	$X = 0$		
IUO	$(ACC) - X \neq 0$		
Store			
STL	$(ACC) \rightarrow X$		
STD	$(ACC) \rightarrow X$		
STI	$(ACC) \rightarrow X$		
STZ	$0 \rightarrow X$		

Figure 4.

condition holds. With this capsule view of the G-20 and the descriptions accompanying the examples, it should be possible to follow the code generated in Appendices C and F. A much more complete set of examples, including a number of source languages, will appear as a separate paper.

Appendix A

Syntax of a Small Language

$\langle \text{arithmetic expression} \rangle ::= \langle \text{term} \rangle \mid \pm \langle \text{term} \rangle \mid \langle \text{arithmetic expression} \rangle \pm \langle \text{term} \rangle$
 $\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{term} \rangle * / \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle ::= \langle \text{primary} \rangle \mid \langle \text{factor} \rangle \uparrow \langle \text{primary} \rangle$
 $\langle \text{primary} \rangle ::= \langle \text{identifier} \rangle \mid (\langle \text{arithmetic expression} \rangle)$
 $\langle \text{simple boolean} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{arithmetic expression} \rangle \langle \text{relation} \rangle \langle \text{arithmetic expression} \rangle$
 $\langle \text{boolean} \rangle ::= \langle \text{simple boolean} \rangle \mid \langle \text{if clause} \rangle \langle \text{simple boolean} \rangle \text{ ELSE } \langle \text{boolean} \rangle$
 $\langle \text{if clause} \rangle ::= \text{IF } \langle \text{boolean} \rangle \text{ THEN}$
 $\langle \text{if statement} \rangle ::= \langle \text{if clause} \rangle \langle \text{unconditional} \rangle$
 $\langle \text{assignment} \rangle ::= \langle \text{identifier} \rangle \leftarrow \langle \text{arithmetic expression} \rangle \mid \langle \text{identifier} \rangle \leftarrow \langle \text{boolean} \rangle$
 $\langle \text{go to statement} \rangle ::= \text{GO TO } \langle \text{identifier} \rangle$
 $\langle \text{conditional} \rangle ::= \langle \text{if statement} \rangle \mid \langle \text{if statement} \rangle \text{ ELSE } \langle \text{statement} \rangle$
 $\langle \text{unconditional} \rangle ::= \langle \text{assignment} \rangle \mid \langle \text{go to statement} \rangle \mid \langle \text{block} \rangle$
 $\langle \text{declaration} \rangle ::= \langle \text{type} \rangle \langle \text{type list} \rangle \mid \langle \text{declaration} \rangle ; \langle \text{type} \rangle \langle \text{type list} \rangle$
 $\langle \text{type} \rangle ::= \text{REAL} \mid \text{BOOLEAN} \mid \text{LABEL}$
 $\langle \text{type list} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{type list} \rangle , \langle \text{identifier} \rangle$
 $\langle \text{head} \rangle ::= \text{BEGIN} \mid \text{BEGIN } \langle \text{declaration} \rangle \mid \langle \text{head} \rangle ; \langle \text{statement} \rangle$
 $\langle \text{block} \rangle ::= \langle \text{head} \rangle \text{ END}$
 $\langle \text{statement} \rangle ::= \langle \text{conditional} \rangle \mid \langle \text{unconditional} \rangle \mid \langle \text{empty} \rangle \mid \langle \text{identifier} \rangle : \langle \text{statement} \rangle$

Notes on Appendix B

The appendix is a machine run of the Production Language syntax of the small language. Besides the productions themselves there are three tables given as input to the Production Loader. The first is a table of reserved identifiers, both those used internally and those which appear in the source code. The second table defines the class names (meta-characters) as discussed in Chapter II. The table of actions contains all those mentioned in Chapter II as well as others used internally by the system.

As we have mentioned, the tables built by the Production Loader are tightly packed and thus difficult to read. The small table at 37715 to 40011 contains initializing information for the Basic Compiler. The two tables (other stuff and LABELS) before the production table contain debugging information which will not concern us here.

To read the production table we must understand the internal numbering system for symbols. Reserved symbols are given octal integers starting at 200 as internal names. After the last number assigned to a reserved symbol (261 for the small language), the source code identifiers are numbered sequentially.

The production table consists of two types of words, head cells and operands. Consider the first entry (62245) in the production table.

0 105 02 10000

The '105' marks this as the head cell of a production and the '02' gives the relative address of the next head cell. The '10000' is the relative address of the interpretation table entry corresponding to this production. We will discuss the interpretation table below.

The next cell contains '252' in the low order positions. This is the octal integer which corresponds to 'BEGIN'. A word of all zeros in the production table corresponds to a 'Ø' in a production.

The interpretation table is easier to follow. The integer in the high order position represents one of the actions in the Action Table. These are numbered sequentially starting with '1' for EXEC. The parameter to the action is contained in the low order bits of the same word. These tables are interpreted by the Basic Compiler in recognizing source language text.

Appendix B

Productions for the Small Language

*	SYMBOLS	
*	12 INTERNAL SYMBOLS	
		I
		P
		F
		T
		E
		SBE
		BE
		ICL
		UN
		S
		HEAD
		I →
+		+
-		-
*		*
/		/
=		=
√		√
≠		≠
^		^
<		<
>		>
→		→
←		←
↔		↔
↑		↑
↓		↓
.		.
,		,
;		;
:		:
+		+
+		+
\$		\$
(		(
[		[
]		]
)		)
REAL		REAL
BOOLEAN		BOOL
LABEL		LABL
BEGIN		BEGN
END		END
GOTO		GO
IF		IF
THEN		THEN
ELSE		ELSE
TRUE		TRUE
FALSE		FALS
*	METACHARACTERS	

M <OP> + - * / †
 M <IP> REAL BOOL LABL
 M <HL> = # < > -< ->
 M <PM> + -
 M <IO> * /
 * ACTIONS

EXEC
 VSTK
 STAK
 VALU
 SUBR
 SCAN
 NEXT
 GET
 BOOK
 RETURN
 ERROR
 HALT
 CON
 NUM
 FOUT
 STRIN

* PRODUCTIONS
 37715 00000000074 00053117163 00003754224 00001020101 00003720323 00000000523 00001336065 00000016307
 37725 00017232775 00125776240 20000034000 01177202242 00003402170 00000000073 00000000072 00000000071
 37735 00000000070 00000000065 00000000035 00000000034 00000000076 00000000067 00000000037 00000000053
 37745 00000000074 00000000075 00000000036 00000000036 00000000036 00000000066 00000000064 00000000063
 37755 00000000062 00000000061 00000000060 00000000057 00000000056 00000000055 00000000054 30000000000
 37765 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000
 37775 30000000000 30000000000 30000000000 00000000261 30000000000 02000000046 00000000032 00000000014
 40005 30000000000 02000000062 00000000213 30000000000 30000000000 30000000000
 ...

S0			BEGN				EXEC 1	*D1
			<SG>				ERROR 0	Q1
D1			<TP>				SCAN	*D2
			BEGN	END	→	S		*S1
			BEGN	;	→	BEGN		*D1
			BEGN	I →	<SG>	→	HEAD ;	S1
			BEGN	<SG>	→	HEAD ;	<SG>	S1
D2			REAL	I	<SG>		EXEC 2	D3
			BOOL	I	<SG>		EXEC 4	D3
			LABL	I	<SG>		EXEC 5	D3
D3			<TP>	I	,	→	<TP>	*D2
	I →		<TP>	I	;	→	I →	*D1
			<TP>	I	;	→	I →	*D1
			<SG>				ERROR 1	Q1
S1			BEGN					*D1
			IF					*B1
			GO					*G1
			;	→	S	;		S9
			END					S9
			I					*S2
S2			I	:	→		EXEC 26	*S1
			I	+				*EX1
			<SG>				ERROR 2	Q1
I1			ICL					*B1
			IF	ICL				*B1
	SBE	ELSE	ICL					*B1
			ICL				EXEC 3	*S1
			<SG>				ERROR 4	Q1
UN1			ICL	UN	<SG>			C1
			UN	<SG>	→	S	<SG>	S9
			<SG>				ERROR 5	Q1
C1			ICL	UN	ELSE		EXEC 6	*S1
			ICL	UN	<SG>	→	EXEC 7	S9
			<SG>			S	<SG>	Q1
P1			I	→		P	ERROR 6	*F1
			(				EXEC 9	*P1
			+	→				*P1
			-					*P1
			<SG>				ERROR 7	Q1
F1	F	↑	P	<SG>	→	F	<SG>	F2
			P	<SG>	→	F	<SG>	F2
F2			F	↑				*P1
T1	T	*	F	<SG>	→	T	<SG>	T2
	T	/	F	<SG>	→	T	<SG>	T2
			F	<SG>	→	T	<SG>	T2
T2			T	<TD>				*P1
E1	E	+	T	<SG>	→	E	<SG>	E2
	E	-	T	<SG>	→	E	<SG>	E2
			T	<SG>	→	E	<SG>	E2
			T	<SG>	→	E	<SG>	E2
E2			E	<PM>				*P1
			(	E	)	→	P	EXEC 15
			E	<RL>				*P1
	I	+	E	<SG>	→	UN	<SG>	UN1
	E	=	E	<SG>	→	SBE	<SG>	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
B1				<SG>						ERROR 8	Q1
				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
B2				I							*B4
			SBE	ELSE							*B1
			SBE	<SG>		→	BE	<SG>			B3
B3			IF	THEN		→				EXEC 21	I1
	ICL	SBE	ELSE	<SG>		→	BE	<SG>		EXEC 22	B3
		I	+	<SG>		→	UN	<SG>		EXEC 23	UN1
EX1				I							*EX2
				IF							*B1
				+		→					*P1
				-							*P1
				(							*P1
EX2			I	<OP>		→	P	<OP>		EXEC 8	F1
			I	<RL>		→	E	<RL>		EXEC 8	*P1
	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	;	S	;		HEAD	;			*S1
		HEAD	;	S	END					EXEC 29	ND1
ND1			I→	UN		→		UN		EXEC 31	Q1
				<SG>							*UN1
Q1				<SG>						HALT	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

I17	2
I18	293
I19	0
I20	28
I21	16333
J 0	287
J 1	0
J 2	1
J 3	61
J 4	67
J 5	7
J 6	51
J 7	12

LABEL TABLE

	LABEL NAME	VALUE
1	S0	0
2	D1	4
3	Q1	285
4	D2	19
5	S1	46
6	D3	31
7	B1	190
8	G1	253
9	S9	258
10	S2	58
11	EX1	230
12	I1	66
13	UN1	80
14	C1	89
15	P1	99
16	F1	109
17	F2	117
18	T1	120
19	T2	133
20	E1	136
21	E2	153
22	B2	209
23	B4	198
24	B3	215
25	EX2	240
26	ND1	280
27		1

PRODUCTION TABLE

62245	01050210000	00000000252	01050210003	00000000000	01050210005	01050300006	01050310010	00000000253
62255	00000000252	01050310014	00000000235	00000000252	01050410017	00000000000	00000000213	00000000252
62265	01050310025	00000000000	00000000252	01050410033	00000000000	00000000200	00000000246	01050410035
62275	00000000000	00000000200	00000000250	01050410037	00000000000	00000000200	00000000251	01050410041
62305	00000000254	00000000200	01050300006	01050510045	00000000235	00000000200	01050300006	00000000213
62315	01050410050	00000000235	00000000200	01050400006	01050210054	00000000000	01050210056	00000000252
62325	01050210060	00000000255	01050210062	00000000254	01050210064	00000000235	01050210070	00000000253
62335	01050210071	00000000200	01050310073	00000000236	00000000200	01050310077	00000000237	00000000200

62345	01050210101	00000000000	01050310103	00000000207	00000000237	01050310105	00000000207	00000000255
62355	01050410107	00000000207	00000000257	00000000205	01050210111	00000000207	01050210114	00000000000
62365	01050410116	00000000000	00000000210	00000000207	01050310117	00000000000	00000000210	01050210123
62375	00000000000	01050410125	00000000257	00000000210	00000000207	01050410130	00000000000	00000000210
62405	00000000207	01050210135	00000000000	01050210137	00000000200	01050210144	00000000242	01050210146
62415	00000000214	01050210151	00000000215	01050210153	00000000000	01050510155	00000000000	00000000201
62425	00000000231	00000000202	01050310163	00000000000	00000000201	01050310165	00000000231	00000000202
62435	01050510167	00000000000	00000000202	00000000216	00000000203	01050510173	00000000000	00000000202
62445	00000000217	00000000203	01050310177	00000000000	00000000202	01050310203	01050200021	00000000203
62455	01050510205	00000000000	00000000203	00000000214	00000000204	01050510211	00000000000	00000000203
62465	00000000215	00000000204	01050410215	00000000000	00000000203	00000000215	01050310222	00000000000
62475	00000000203	01050310226	01050200017	00000000204	01050410230	00000000245	00000000204	00000000242
62505	01050310235	01050600011	00000000204	01050510237	00000000000	00000000204	00000000237	00000000200
62515	01050510244	00000000000	00000000204	00000000220	00000000204	01050510251	00000000000	00000000204
62525	00000000224	00000000204	01050510256	00000000000	00000000204	00000000225	00000000204	01050510263
62535	00000000000	00000000204	01050600011	00000000204	01050210270	00000000000	01050210272	00000000255
62545	01050210274	00000000214	01050210277	00000000215	01050210301	00000000242	01050310303	01050500001
62555	00000000200	01050310310	01050600011	00000000200	01050310316	00000000000	00000000200	01050210323
62565	00000000200	01050310325	00000000257	00000000205	01050310327	00000000000	00000000205	01050410333
62575	00000000256	00000000206	00000000255	01050410337	00000000000	00000000206	00000000257	00000000205
62605	00000000207	01050510344	00000000000	00000000206	00000000237	00000000200	01050210351	00000000200
62615	01050210353	00000000255	01050210355	00000000214	01050210360	00000000215	01050210362	00000000242
62625	01050310364	01050500001	00000000200	01050310371	01050600011	00000000200	01050510377	00000000000
62635	00000000200	00000000237	00000000200	01050210404	00000000000	01050310406	00000000200	00000000254
62645	01050210413	00000000000	01050610415	00000000000	00000000211	00000000257	00000000210	00000000207
62655	01050610422	00000000000	00000000210	00000000257	00000000210	00000000207	01050510427	00000000235
62665	00000000211	00000000235	00000000212	01050510432	00000000253	00000000211	00000000235	00000000212
62675	01050310436	00000000210	00000000213	01050210441	00000000000	01050210443	00000000000	00000000000

AUXILIARY PRODUCTION TABLE

65202	00000000000	00000000210	00000000213	00000000253	00000000210	00000000235	00000000000	00000000000
65212	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000
65222	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000

INTERPRETATION TABLE

65227	00010000001	00060000001	00070000004	00130000000	000700000435	00060000000	00060000001	00070000023
65237	00020000002	00030000211	00060000001	00070000056	00020000001	00060000001	00070000004	00020000003
65247	00030000212	00030000235	00030000000	00010000002	00070000056	00020000002	00030000212	00030000235
65257	00030000000	00010000002	00070000056	00010000004	00070000037	00010000005	00070000037	00010000036
65267	00070000037	00020000002	00060000000	00060000001	00070000023	00020000003	00060000001	00070000004
65277	00020000003	00030000213	00060000001	00070000004	00130000001	000700000435	00060000001	00070000004
65307	00060000001	00070000276	00060000001	00070000375	00020000001	00030000211	00030000000	00070000402
65317	00070000402	00060000001	00070000072	00020000002	00010000032	00060000001	00070000056	00060000001
65327	00070000346	00130000002	00070000435	00060000001	00070000276	00060000001	00070000276	00060000001
65337	00070000276	00010000003	00060000001	00070000056	00130000004	00070000435	00070000131	00020000002
65347	00030000211	00030000000	00070000402	00130000005	00070000435	00010000006	00060000001	00070000056
65357	00020000003	00030000211	00030000000	00010000007	00070000402	00130000006	00070000435	00020000001
65367	00030000201	00010000011	00060000001	00070000155	00060000001	00070000143	00020000001	00060000001
65377	00070000143	00060000001	00070000143	00130000007	00070000435	00020000003	00030000000	00010000034
65407	00070000165	00020000002	00030000202	00030000000	00070000165	00060000001	00070000143	00020000003
65417	00030000000	00010000012	00070000205	00020000003	00030000000	00010000013	00070000205	00020000002
65427	00030000203	00030000000	00070000205	00060000001	00070000143	00020000003	00030000000	00010000014
65437	00070000231	00020000003	00030000000	00010000015	00070000231	00020000003	00030000204	00030000000
65447	00010000016	00070000231	00020000002	00030000204	00030000000	00070000231	00060000001	00070000143
65457	00020000003	00030000201	00010000017	00060000001	00070000155	00060000001	00070000143	00020000004

```

65467 00030000210 00030000000 00010000020 00070000120 00020000004 00030000205 00030000000 00010000021
65477 00070000321 00020000004 00030000205 00030000000 00010000022 00070000321 00020000004 00030000205
65507 00030000000 00010000023 00070000321 00020000004 00030000205 00030000000 00010000021 00070000321
65517 00130000010 00070000435 00060000001 00070000276 00020000001 00060000001 00070000143 00060000001
65527 00070000143 00060000001 00070000143 00020000002 00030000201 00030000000 00010000010 00070000155
65537 00020000002 00030000204 00030000000 00010000010 00060000001 00070000143 00020000002 00030000205
65547 00030000000 00010000024 00070000321 00060000001 00070000306 00060000001 00070000276 00020000002
65557 00030000206 00030000000 00070000327 00020000003 00030000207 00010000025 00070000102 00020000005
65567 00030000001 00030000000 00010000026 00070000327 00020000004 00030000210 00030000000 00010000027
65577 00070000120 00060000001 00070000360 00060000001 00070000276 00020000002 00030000201 00030000000
65607 00060000001 00070000143 00060000001 00070000143 00020000002 00030000201 00030000000 00010000010
65617 00070000155 00020000002 00030000204 00030000000 00010000010 00060000001 00070000143 00020000004
65627 00030000210 00030000000 00010000030 00070000120 00130000010 00070000435 00020000002 00030000210
65637 00010000031 00060000001 00070000120 00130000011 00070000435 00020000005 00030000001 00030000000
65647 00010000033 00070000402 00020000005 00030000211 00030000000 00010000033 00070000402 00020000002
65657 00060000001 00070000056 00020000004 00030000210 00010000035 00070000430 00020000001 00010000037
65667 00070000435 00060000001 00070000120 00140000000 00070000435

```

• TAPE 210

RECORD TYPE	NUMBER OF RECORDS	STARTING RECORD
SYMBOL TABLE	1	21 *+01
HIERARCHY TABLE	1	21 *+01
PRODUCTION TABLE	1	22 *+01
METACHARACTER LISTS	1	22 *+01

TIME USED: 00:02:43

PAGES USED: 8

19:14:09

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 $12 \downarrow$ (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

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

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.

OPERATOR-040 14 MAY 64 19:14:10 FSL PAGES: 100 TIME: 5

00:00:03

CO
CO

APPENDIX C FORMAL SEMANTICS OF THE SMALL LANGUAGE

SN DUMP

BEGIN TABLE SYMB(200,4) ; CELL LEV,T0,T1,T2,T3,T4,T5; STACK STR,SYM;
TITLE REAL, BOOL, LABE; DATA LOGIC, INTEGER, SINGLE, DOUBLE;

0+ T3 ← TEMPLOC ↓

1+ LEV ← 0; STR ← STORLOC; SYM ← LOC(SYMB) ↓

2+ LEV ≠ 0 → PUSH(STR,STORLOC); PUSH (SYM, LOC(SYMB))\$;
TALLY(LEV) ↓

3+ PUSH(FLAD1,0); CODE(-LEFT1 → JUMP(FLAD1))\$ ↓

4+ T0 ← STORLOC; SET(T0,DOUBLE); ENTER(SYMB;LEFT2,T0,REAL,LEV);
STORLOC ← STORLOC+2 ↓

5+ ENTER(SYMB;LEFT2,STORLOC,BOOL ,LEV);TALLY(STORLOC) ↓

6+ PUSH(FLAD2,0); CODE(JUMP(FLAD2)); ASSIGN(FLAD1)↓

7+ ASSIGN(FLAD1)↓

8+ CONST(LEFT2) → RIGHT2 ← LEFT2 :
SYMB(LEFT2,,,\$,)= REAL → RIGHT2 ← SYMB(LEFT2,\$,,):FAULT 1\$\$ ↓

9+ CONST(LEFT1) → RIGHT1 ← LEFT1 :
SYMB(LEFT1,,,\$,)= REAL → RIGHT1 ← SYMB(LEFT1,\$,,):FAULT 1\$\$ ↓

10+ CODE(VALUE2+LEFT4*LEFT2)↓

11+ CODE(VALUE2+LEFT4/LEFT2)↓

12+ CODE(VALUE2+LEFT4+LEFT2)↓

13+ CODE(VALUE2+LEFT4-LEFT2)↓

14+ CODE(VALUE2+--LEFT2)↓

```

15+ RIGHT1←LEFT2+
16+ SYMB(LEFT4,,,$,)=REAL→ COMT 2← SYMB(LEFT4,,,$,);
    CODE(COMT 2← LEFT2) ; TEMPLOC ← T3 ; FAULT 3$+
17+ CODE(VALUE2←LEFT4=LEFT2)+
18+ CODE(VALUE2←LEFT4<LEFT2)+
19+ CODE(VALUE2←LEFT4>LEFT2)+
20+CONST(LEFT2)→ RIGHT2← LEFT2 ;
    SYMB(LEFT2,,,$,)=BOOL→RIGHT2←SYMB(LEFT2,,,$,);
    SET(RIGHT2,LOGIC) ; TEMPLOC ← T3 ; FAULT 3$$+
21+ RIGHT1 ← LEFT2 +
22+ CODE( LEFT5→ TEMPLOC← LEFT4; TEMPLOC← LEFT2$) ;
    RIGHT2← TEMPLOC; SET(RIGHT2,LOGIC) ;
    TALLY(TEMPLOC) +
23+ SYMB(LEFT4,,,$,)=BOOL→ COMT 2← SYMB(LEFT4,,,$,);
    CODE( COMT 2← LEFT2); FAULT 4$ +
24+ COMT 4← SYMB(LEFT4,,,$,);
    CONST(LEFT2) → COMT 2← LEFT2;
    SYMB(LEFT2,,,$,)=SYMB(LEFT4,,,$,) → COMT 2← SYMB(LEFT2,,,$,);
    FAULT 5 $ $; TEMPLOC ← T3;
    CODE( COMT 4← COMT 2) +
25+ SYMB(LEFT1,,,$,) ≠ LABE → FAULT 6:
    COMT 2← LOC(SYMB(LEFT1,,,$,));
    COMT 3← <COMT 2>; SYMB(LEFT1,,,$,) ≠ 0 → CODE( JUMP(COMT 3)) ;
    CODE( JUMP(CHAIN( COMT 2) )) $$ +
26+ SYMB(LEFT2,,,$,) ≠ LABE → FAULT 6:
    SYMB(0,,,$) ← 1 ;
    ASSIGN(LOC ( SYMB( LEFT2,,,$,))) $+
27+ ASSIGN(FLAD2) +
28+ CODE( VALUE2 ← LEFT4 + LEFT2) +
29+ MINUS(LEV); POP(STR,STORLOC); POP(SYM,LOC(SYMB)) +
30+ ENTER(SYMB;LEFT2,0,LABE, 0') +
31+ CODE( STOP) +

```

```

65600 0 CLA 0 00004,00 0 STI 0 45775,00 0 CLA 0 45774,00 0 STI 0 45777,00 0 CLA 0 00310,00
65605 0 STI 0 45776,00 0 LXP 0 44161,40 0 LXP 0 44015,41 0 CLA 0 00370,00 0 STL 0 44014,00
65612 0 CLA 0 00371,00 0 STL 0 44013,00 0 CLA 0 00372,00 0 STL 0 44012,00 0 CLA 2 00076,00
65617 0 STL 0 44327,00 0 TRA 0 17020,00 0 CLA 0 00000,00 0 STL 0 44333,00 0 CLA 2 00071,00
65624 0 STL 2 00040,00 0 CAL 2 45777,00 0 STL 2 00041,00 0 TRA 0 62110,00 0 CAL 2 44333,00
65631 0 FUO 0 00000,00 0 TRA 0 65634,00 0 TRA 0 65643,00 0 LXM 0 00001,74 0 ADX 0 00001,40
65636 0 CLA 2 00071,00 0 STL 2 00040,00 0 ADX 0 00001,41 0 CAL 2 45777,00 0 STL 2 00041,00
65643 0 CAL 2 44333,00 0 ADD 0 00001,00 0 STL 0 44333,00 0 TRA 0 62110,00 0 ADX 0 00001,53
65650 0 CLA 0 00000,00 0 STL 2 00053,00 0 CAL 2 63334,00 0 STL 0 63347,00 0 TRM 0 63516,00
65655 0 CAL 2 63351,00 0 STL 0 63346,00 0 TRM 0 63664,00 0 LXM 0 00001,74 0 CAL 0 00053,00
65662 0 UNL 2 63251,00 0 STL 0 63346,00 0 TRM 0 64057,00 0 OCA 2 63365,77 0 CAL 2 00000,00
65667 0 UNL 2 00070,00 0 STL 2 63365,77 0 SUX 0 00001,77 0 LXM 0 00001,74 0 TRA 0 62110,00
65674 0 CLA 2 00071,00 0 STL 0 44332,00 0 CAL 2 44332,00 0 UNL 2 63275,00 0 UNL 2 63313,00
65701 0 STL 0 44332,00 0 CAL 2 63335,00 0 STL 0 47022,00 0 CAL 2 44332,00 0 STL 0 47023,00
65706 0 CAL 2 44014,00 0 STL 0 47024,00 0 CAL 2 44333,00 0 STL 0 47025,00 0 CLA 0 45777,00
65713 0 TRM 0 64626,00 0 CLA 2 00071,00 0 ADD 0 00002,00 0 STI 0 00071,00 0 TRA 0 62110,00
65720 0 CAL 2 63335,00 0 STL 0 47022,00 0 CAL 2 00071,00 0 STL 0 47023,00 0 CAL 2 44013,00
65725 0 STL 0 47024,00 0 CAL 2 44333,00 0 STL 0 47025,00 0 CLA 0 45777,00 0 TRM 0 64626,00
65732 0 ADX 0 00001,71 0 TRA 0 62110,00 0 ADX 0 00001,54 0 CLA 0 00000,00 0 STL 2 00054,00
65737 0 CAL 0 00054,00 0 UNL 2 63251,00 0 STL 0 63346,00 0 TRM 0 64057,00 0 CAL 0 00053,00
65744 0 UNL 2 63251,00 0 TRM 0 64134,00 0 TRA 0 62110,00 0 CAL 0 00053,00 0 UNL 2 63251,00
65751 0 TRM 0 64134,00 0 TRA 0 62110,00 0 CAL 2 63335,00 0 IEZ 2 63225,00 0 OCS 0 00001,00
65756 0 CAL 2 14377,00 0 IUO 2 14377,00 0 TRA 0 65765,00 0 LXM 0 00001,74 0 CAL 2 63335,00
65763 0 STL 0 63342,00 0 TRA 0 66013,00 0 LXM 0 00001,74 0 LXP 0 00002,52 0 CAL 2 63335,00
65770 0 STI 0 47034,00 0 CLA 0 45777,00 0 TRM 0 64570,00 0 CAL 3 47034,00 0 FUO 2 44014,00
65775 0 TRA 0 66007,00 0 LXM 0 00001,74 0 LXP 0 00001,52 0 CAL 2 63335,00 0 STI 0 47034,00
66002 0 CLA 0 45777,00 0 TRM 0 64570,00 0 CAL 3 47034,00 0 STL 0 63342,00 0 TRA 0 66012,00
66007 0 LXM 0 00001,74 0 CLA 0 00001,00 0 TRM 0 64161,00 0 LXM 0 00001,74 0 LXM 0 00001,74
66014 0 TRA 0 62110,00 0 CAL 2 63334,00 0 IEZ 2 63225,00 0 OCS 0 00001,00 0 CAL 2 14377,00
66021 0 IUO 2 14377,00 0 TRA 0 66027,00 0 LXM 0 00001,74 0 CAL 2 63334,00 0 STL 0 63341,00
66026 0 TRA 0 66055,00 0 LXM 0 00001,74 0 LXP 0 00002,52 0 CAL 2 63334,00 0 STI 0 47034,00
66033 0 CLA 0 45777,00 0 TRM 0 64570,00 0 CAL 3 47034,00 0 FUO 2 44014,00 0 TRA 0 66051,00
66040 0 LXM 0 00001,74 0 LXP 0 00001,52 0 CAL 2 63334,00 0 STI 0 47034,00 0 CLA 0 45777,00
66045 0 TRM 0 64570,00 0 CAL 3 47034,00 0 STL 0 63341,00 0 TRA 0 66054,00 0 LXM 0 00001,74
66052 0 CLA 0 00001,00 0 TRM 0 64161,00 0 LXM 0 00001,74 0 LXM 0 00001,74 0 TRA 0 62110,00
66057 0 ADX 0 00001,74 0 CAL 2 63337,00 0 STL 0 63346,00 0 CAL 2 63335,00 0 STL 0 63347,00
66064 0 TRM 0 63455,00 0 LXP 0 00001,52 0 TRM 0 62472,00 0 TRA 0 62110,00 0 ADX 0 00001,74
66071 0 CAL 2 63337,00 0 STL 0 63346,00 0 CAL 2 63335,00 0 STL 0 63347,00 0 TRM 0 63461,00
66076 0 LXP 0 00001,52 0 TRM 0 62472,00 0 TRA 0 62110,00 0 ADX 0 00001,74 0 CAL 2 63337,00
66103 0 STL 0 63346,00 0 CAL 2 63335,00 0 STL 0 63347,00 0 TRM 0 63405,00 0 LXP 0 00001,52
66110 0 TRM 0 62472,00 0 TRA 0 62110,00 0 ADX 0 00001,74 0 CAL 2 63337,00 0 STL 0 63346,00
66115 0 CAL 2 63335,00 0 STL 0 63347,00 0 TRM 0 63411,00 0 LXP 0 00001,52 0 TRM 0 62472,00
66122 0 TRA 0 62110,00 0 CAL 2 63335,00 0 SIL 0 63347,00 0 TRM 0 63516,00 0 LXP 0 00001,52
66127 0 TRM 0 62472,00 0 TRA 0 62110,00 0 CAL 2 63335,00 0 STL 0 63341,00 0 TRA 0 62110,00
66134 0 LXP 0 00002,52 0 CAL 2 63337,00 0 SIL 0 47034,00 0 CLA 0 45777,00 0 TRM 0 64570,00
66141 0 CAL 3 47034,00 0 FUO 2 44014,00 0 TRA 0 66166,00 0 LXM 0 00001,74 0 LXP 0 00001,52
66146 0 CAL 2 63337,00 0 STI 0 47034,00 0 CLA 0 45777,00 0 TRM 0 64570,00 0 CAL 3 47034,00
66153 0 STL 0 46465,00 0 ADX 0 00001,74 0 CAL 2 46465,00 0 STL 0 63346,00 0 CAL 2 63335,00
66160 0 STL 0 63347,00 0 TRM 0 63762,00 0 LXM 0 00001,74 0 CAL 2 44327,00 0 STI 0 00076,00
66165 0 TRA 0 66171,00 0 LXM 0 00001,74 0 CLA 0 00003,00 0 TRM 0 64161,00 0 LXM 0 00001,74
66172 0 TRA 0 62110,00 0 ADX 0 00001,74 0 CAL 2 63337,00 0 STL 0 63346,00 0 CAL 2 63335,00
66177 0 STL 0 63347,00 0 TRM 0 63527,00 0 LXP 0 00001,52 0 TRM 0 62472,00 0 TRA 0 62110,00
66204 0 ADX 0 00001,74 0 CAL 2 63337,00 0 STL 0 63346,00 0 CAL 2 63335,00 0 STL 0 63347,00
66211 0 TRM 0 63537,00 0 LXP 0 00001,52 0 TRM 0 62472,00 0 TRA 0 62110,00 0 ADX 0 00001,74

```

66216	0 CAL 2 63337,00	0 STL 0 63346,00	0 CAL 2 63335,00	0 STL 0 63347,00	0 TRM 0 63543,00
66223	0 LXP 0 00001,52	0 TRM 0 62472,00	0 TRA 0 62110,00	0 CAL 2 63335,00	0 IEZ 2 63225,00
66230	0 OCS 0 00001,00	0 CAL 2 14377,00	0 IVO 2 14377,00	0 TRA 0 66240,00	0 LXM 0 00001,74
66235	0 CAL 2 63335,00	0 STL 0 63342,00	0 TRA 0 66274,00	0 LXM 0 00001,74	0 LXP 0 00002,52
66242	0 CAL 2 63335,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00
66247	0 FUD 2 44013,00	0 TRA 0 66270,00	0 LXM 0 00001,74	0 LXP 0 00001,52	0 CAL 2 63335,00
66254	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00	0 STL 0 63342,00
66261	0 CAL 2 63342,00	0 UNL 2 63272,00	0 UNL 2 63313,00	0 STL 0 63342,00	0 CAL 2 44327,00
66266	0 STI 0 00076,00	0 TRA 0 66273,00	0 LXM 0 00001,74	0 CLA 0 00003,00	0 TRM 0 64161,00
66273	0 LXM 0 00001,74	0 LXM 0 00001,74	0 TRA 0 62110,00	0 CAL 2 63335,00	0 STL 0 63341,00
66300	0 TRA 0 62110,00	0 CAL 2 63340,00	0 STL 0 63346,00	0 TRM 0 63664,00	0 LXM 0 00001,74
66305	0 ADX 0 00001,74	0 CAL 2 00076,00	0 STL 0 63346,00	0 CAL 2 63337,00	0 STL 0 63347,00
66312	0 TRM 0 63762,00	0 LXM 0 00001,74	0 TRM 0 63731,00	0 LXM 0 00001,74	0 ADX 0 00001,74
66317	0 CAL 2 00076,00	0 STL 0 63346,00	0 CAL 2 63335,00	0 STL 0 63347,00	0 TRM 0 63762,00
66324	0 LXM 0 00001,74	0 OCA 2 63365,77	0 CAL 2 00000,00	0 UNL 2 00070,00	0 STL 2 63365,77
66331	0 SUX 0 00001,77	0 LXM 0 00001,74	0 CLA 2 00076,00	0 STL 0 63342,00	0 CAL 2 63342,00
66336	0 UNL 2 63272,00	0 UNL 2 63313,00	0 STL 0 63342,00	0 ADX 0 00001,76	0 TRA 0 62110,00
66343	0 LXP 0 00002,52	0 CAL 2 63337,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00
66350	0 CAL 3 47034,00	0 FUD 2 44013,00	0 TRA 0 66373,00	0 LXM 0 00001,74	0 LXP 0 00001,52
66355	0 CAL 2 63337,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00
66362	0 STL 0 46465,00	0 ADX 0 00001,74	0 CAL 2 46465,00	0 STL 0 63346,00	0 CAL 2 63335,00
66367	0 STL 0 63347,00	0 TRM 0 63762,00	0 LXM 0 00001,74	0 TRA 0 66376,00	0 LXM 0 00001,74
66374	0 CLA 0 00004,00	0 TRM 0 64161,00	0 LXM 0 00001,74	0 TRA 0 62110,00	0 LXP 0 00001,52
66401	0 CAL 2 63337,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00
66406	0 STL 0 46467,00	0 CAL 2 63335,00	0 IEZ 2 63225,00	0 OCS 0 00001,00	0 CAL 2 14377,00
66413	0 IVO 2 14377,00	0 TRA 0 66421,00	0 LXM 0 00001,74	0 CAL 2 63335,00	0 STL 0 46465,00
66420	0 TRA 0 66454,00	0 LXM 0 00001,74	0 LXP 0 00002,52	0 CAL 2 63335,00	0 BTI 0 47034,00
66425	0 CLA 0 45777,00	0 TRM 0 64570,00	0 LXP 0 00002,52	0 CAL 2 63337,00	0 STI 0 47034,00
66432	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00	0 FUD 3 47034,00	0 TRA 0 66450,00
66437	0 LXM 0 00001,74	0 LXP 0 00001,52	0 CAL 2 63335,00	0 STI 0 47034,00	0 CLA 0 45777,00
66444	0 TRM 0 64570,00	0 CAL 3 47034,00	0 STL 0 46465,00	0 TRA 0 66453,00	0 LXM 0 00001,74
66451	0 CLA 0 00005,00	0 TRM 0 64161,00	0 LXM 0 00001,74	0 LXM 0 00001,74	0 CAL 2 44327,00
66456	0 STI 0 00076,00	0 ADX 0 00001,74	0 CAL 2 46467,00	0 STL 0 63346,00	0 CAL 2 46465,00
66463	0 STL 0 63347,00	0 TRM 0 63762,00	0 LXM 0 00001,74	0 TRA 0 62110,00	0 LXP 0 00002,52
66470	0 CAL 2 63334,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00
66475	0 FUD 2 44012,00	0 TRA 0 66500,00	0 TRA 0 66504,00	0 LXM 0 00001,74	0 CLA 0 00006,00
66502	0 TRM 0 64161,00	0 TRA 0 66543,00	0 LXM 0 00001,74	0 LXP 0 00001,52	0 CAL 2 63334,00
66507	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 2 47034,00	0 STL 0 46465,00
66514	0 CAL 3 46465,00	0 STL 0 46466,00	0 LXP 0 00003,52	0 CAL 2 63334,00	0 STI 0 47034,00
66521	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00	0 FUD 0 00000,00	0 TRA 0 66527,00
66526	0 TRA 0 66534,00	0 LXM 0 00001,74	0 CAL 2 46466,00	0 STL 0 63346,00	0 TRM 0 64057,00
66533	0 TRA 0 66542,00	0 LXM 0 00001,74	0 CAL 2 46465,00	0 TRM 0 64126,00	0 CAL 2 63351,00
66540	0 STL 0 63346,00	0 TRM 0 64057,00	0 LXM 0 00001,74	0 LXM 0 00001,74	0 TRA 0 62110,00
66545	0 LXP 0 00002,52	0 CAL 2 63335,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00
66552	0 CAL 3 47034,00	0 FUD 2 44012,00	0 TRA 0 66556,00	0 TRA 0 66562,00	0 LXM 0 00001,74
66557	0 CLA 0 00006,00	0 TRM 0 64161,00	0 TRA 0 66601,00	0 LXM 0 00001,74	0 LXP 0 00003,52
66564	0 CLA 0 00000,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CLA 0 00001,00
66571	0 STL 2 47034,00	0 LXP 0 00001,52	0 CAL 2 63335,00	0 STI 0 47034,00	0 CLA 0 45777,00
66576	0 TRM 0 64570,00	0 CAL 2 47034,00	0 TRM 0 64134,00	0 LXM 0 00001,74	0 TRA 0 62110,00
66603	0 CAL 0 00054,00	0 UNL 2 63251,00	0 TRM 0 64134,00	0 TRA 0 62110,00	0 ADX 0 00001,74
66610	0 CAL 2 63337,00	0 STL 0 63346,00	0 CAL 2 63335,00	0 STL 0 63347,00	0 TRM 0 64024,00
66615	0 LXP 0 00001,52	0 TRM 0 62472,00	0 TRA 0 62110,00	0 CAL 2 44333,00	0 SUB 0 00001,00
66622	0 STL 0 44333,00	0 CAL 3 00040,00	0 STI 0 00071,00	0 SUX 0 00001,40	0 CAL 3 00041,00
66627	0 STL 0 45777,00	0 SUX 0 00001,41	0 TRA 0 62110,00	0 CAL 2 63335,00	0 STL 0 47022,00
66634	0 CLA 0 00000,00	0 STL 0 47023,00	0 CAL 2 44012,00	0 STL 0 47024,00	0 CLA 0 00000,00

66641 0 STL 0 47025,00 0 CLA 0 45777,00 0 TRM 0 64626,00 0 TRA 0 62110,00 0 CLA 0 05300,00
 66646 0 UNL 2 63166,00 0 TRM 0 64320,00 0 TRA 0 62110,00

37000	00000065616	00000065621	00000065630	00000065647	00000065674	00000065720	00000065734	00000065747
37010	00000065753	00000066015	00000066057	00000066070	00000066101	00000066112	00000066123	00000066131
37020	00000066134	00000066173	00000066204	00000066215	00000066226	00000066276	00000066301	00000066343
37030	00000066400	00000066467	00000066545	00000066603	00000066607	00000066620	00000066632	00000066645
37040	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37050	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37060	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37070	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37100	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37110	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37120	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37130	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37140	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000

TIME USED: 00:00:29 PAGES USED: 5 19:14:50

Appendix D

BNF Syntax of the Formal Semantic Language

```

<program head> ::= BEGIN <declaration part> ; <sentence> |
                  <program head> ; <sentence>

<semantic program> ::= <program head> END

<declaration part> ::= <declaration> | <declaration part> ;
                      <declaration>

<declaration> ::= <table dec> | <stack dec> | <cell dec> | <title dec> |
                  <data dec>

<table dec> ::= TABLE <table specifier> | RTABL <table specifier> |
               <table dec> , <table specifier>

<stack dec> ::= STACK <identifier list> | RSTAK <identifier list>

<cell dec> ::= CELL <identifier list>

<title dec> ::= TITLE <identifier list>

<data dec> ::= DATA <identifier list>

<table specifier> ::= <table id> [<integer> , <integer>]

<sentence> ::= <integer> ↓ <statement sequence> ↓

<statement sequence> ::= <statement> | <statement sequence> ;
                       <statement> | CODE (<statement sequence>)

<statement> ::= <unconditional> | <conditional> | '<label>'
               <statement>

<unconditional> ::= <assignment> | <storage> | <transfer> |
                   <auxiliary> | CODE (<unconditional>)

<conditional> ::= <if clause> <statement sequence> $ |
                  <if clause> <statement sequence> :
                  <statement sequence> $ | CODE (<conditional>)

<assignment> ::= <left side> ← <arithmetic> |
                 <left side> ← <boolean>

```

$\langle \text{left side} \rangle ::= \langle \text{primary} \rangle \mid \text{VALUE1} \mid \text{VALUE2} \mid \text{VALUE3}$
 $\langle \text{storage} \rangle ::= \langle \text{stack command} \rangle \mid \langle \text{enter} \rangle$
 $\langle \text{stack command} \rangle ::= \text{PUSH} [\langle \text{stack id} \rangle, \langle \text{arithmetic} \rangle] \mid \text{POP} [\langle \text{stack id} \rangle, \langle \text{arithmetic} \rangle]$
 $\langle \text{enter} \rangle ::= \text{ENTER} [\langle \text{table id} \rangle; \langle \text{expression list} \rangle]$
 $\langle \text{expression list} \rangle ::= \langle \text{arithmetic} \rangle \mid \langle \text{expression list} \rangle, \langle \text{arithmetic} \rangle$
 $\langle \text{transfer} \rangle ::= \text{JUMP} [\langle \text{arithmetic} \rangle] \mid \text{MARKJUMP} [\langle \text{arithmetic} \rangle] \mid \text{JUMP} [\langle \text{label} \rangle] \mid \text{MARKJUMP} [\langle \text{subroutine} \rangle] \mid \text{EXECUTE} [\langle \text{arithmetic} \rangle]$
 $\langle \text{auxiliary} \rangle ::= \text{SET} [\langle \text{arithmetic} \rangle, \langle \text{data id} \rangle] \mid \text{FAULT} [\langle \text{identifier} \rangle] \mid \text{TALLY} [\langle \text{arithmetic} \rangle] \mid \text{MINUS} [\langle \text{arithmetic} \rangle] \mid \text{ASSIGN} [\langle \text{arithmetic} \rangle] \mid \text{STOP}$
 $\langle \text{operand} \rangle ::= \langle \langle \text{arithmetic} \rangle \rangle \mid \text{COML} [\langle \text{identifier} \rangle] \mid \text{RUNL} [\langle \text{identifier} \rangle] \mid \langle \text{production operand} \rangle \mid \langle \text{storage operand} \rangle$
 $\langle \text{production operand} \rangle ::= \text{LEFT1} \mid \text{LEFT2} \mid \text{LEFT3} \mid \text{LEFT4} \mid \text{LEFT5} \mid \text{RIGHT1} \mid \text{RIGHT2} \mid \text{RIGHT3}$
 $\langle \text{storage operand} \rangle ::= \langle \text{table operand} \rangle \mid \langle \text{cell id} \rangle \mid \langle \text{stack id} \rangle \mid \langle \text{title id} \rangle$
 $\langle \text{table operand} \rangle ::= \langle \text{table id} \rangle [\langle \text{arithmetic} \rangle \langle \text{commas} \rangle \$ \langle \text{commas} \rangle]$
 $\langle \text{commas} \rangle ::= , \mid \langle \text{commas} \rangle, \mid \langle \text{empty} \rangle$
 $\langle \text{primary} \rangle ::= \langle \text{operand} \rangle \mid \langle \text{constant} \rangle \mid (\langle \text{arithmetic} \rangle) \mid \text{LOC} [\langle \text{table id} \rangle] \mid \text{LOC} [\langle \text{stack id} \rangle] \mid \langle \text{system cell} \rangle \mid \langle \text{flad} \rangle$
 $\langle \text{system cell} \rangle ::= \text{CODELOC} \mid \text{STORLOC} \mid \text{TEMPLOC}$
 $\langle \text{factor} \rangle ::= \langle \text{primary} \rangle \mid \langle \text{factor} \rangle \uparrow \langle \text{primary} \rangle$
 $\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{term} \rangle \langle * / \rangle \langle \text{factor} \rangle$
 $\langle \text{arithmetic} \rangle ::= \langle \text{term} \rangle \mid \langle \pm \rangle \langle \text{term} \rangle \mid \langle \text{arithmetic} \rangle \langle \pm \rangle \langle \text{term} \rangle \mid \text{INT} [\langle \text{arithmetic} \rangle] \mid \text{ABS} [\langle \text{arithmetic} \rangle] \mid \text{LOC} [\langle \text{arithmetic} \rangle] \mid \text{CHAIN} [\langle \text{arithmetic} \rangle]$

$\langle \text{boolean primary} \rangle ::= \langle \text{arithmetic} \rangle \langle \text{relation} \rangle \langle \text{arithmetic} \rangle \mid$
 $\langle \text{operand} \rangle \mid (\langle \text{boolean} \rangle) \mid \text{TRUE} \mid \text{FALSE} \mid \text{SIGNAL} \mid$
 $\text{TEST} [\langle \text{arithmetic} \rangle, \langle \text{data id} \rangle] \mid$
 $\text{CONST} [\langle \text{arithmetic} \rangle] \mid \text{OK}$

$\langle \text{boolean factor} \rangle ::= \langle \text{boolean primary} \rangle \mid \neg \langle \text{boolean primary} \rangle$

$\langle \text{boolean term} \rangle ::= \langle \text{boolean factor} \rangle \mid \langle \text{boolean term} \rangle \wedge \langle \text{boolean factor} \rangle$

$\langle \text{boolean} \rangle ::= \langle \text{boolean term} \rangle \mid \langle \text{boolean} \rangle \vee \langle \text{boolean term} \rangle$

$\langle \text{if clause} \rangle ::= \langle \text{boolean} \rangle \rightarrow$

$\langle \text{stack id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{data id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{table id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{cell id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{title id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{label} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle$

$\langle \text{identifier list} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{identifier list} \rangle, \langle \text{identifier} \rangle$

$\langle \text{subroutine} \rangle ::= \text{SIN} \mid \text{COS} \mid \text{EXP} \mid \text{LOG} \mid \text{ARCTAN} \mid \text{SIGN} \mid$

$\langle \text{flad} \rangle ::= \text{FLAD1} \mid \text{FLAD2} \mid \text{FLAD3} \mid \text{FLAD4}$

$\langle \text{relation} \rangle ::= = \mid > \mid < \mid \neq$

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
		SO
		I→
+		+
-		-
*		*
/		/
=		=
≠		≠
<		<
>		>
┌		┌
└		└
^		^
↑		↑
↓		↓
,		,
;		;
:		:
←		←
→		→
\$		\$
(		(
[		[
]		]
)		)
,		,
BEGIN		BEGN
END		END
STOP		STOP
STACK		STAK
CELL		CELL
TITLE		NAME
TABLE		TABL
DATA		DATA

JUMP	TR
MARKJUMP	TM
EXECUTE	X
ENTER	E
PUSH	ST
POP	RS
CHAIN	CH
ASSIGN	UC
CODE	CD.
FAULT	FALT
TALLY	TAL
MINUS	MIN
SET	SET
TEST	TEST
INT	IN
LOC	L
CODELOC	A
STORLOC	W
TEMPLOC	TLOC
COMT	CT
RUNT	RT
VALUE1	V1
VALUE2	V2
VALUE3	V3
LEFT1	Y1
LEFT2	Y2
LEFT3	Y3
LEFT4	Y4
LEFT5	Y5
RIGHT1	Z1
RIGHT2	Z2
RIGHT3	Z3
FLAD1	FL1
FLAD2	FL2
FLAD3	FL3
FLAD4	FL4
SIGNAL	SIG
RSTAK	RSTK
RTABL	RTAB
CONST	CON
CLEAR	CLE
OK	OK
PLACE	PLAC
DEPTH	DEP
SAR	SAR
SIN	SIN
COS	COS
EXP	EXP
LN	LN
SQRT	SQRT
ARCTAN	ARC
SIGN	SIGN
TRUE	TRUE
FALSE	FALS

* METACHARACTERS

M <TD> * /
 M <PM> + -
 M <RL> = < > *
 M <EN> = < > *)) , . :
 M <BN> ^ v ~
 M <TP> TABL STAK CELL NAME DATA RSTK RTAB
 M <V> V1 V2 V3
 M <Y> Y1 Y2 Y3 Y4 Y5
 M <Z> Z1 Z2 Z3
 M <FL> FL1 FL2 FL3 FL4
 M <CO> TR TM TAL MIN X CH UC ST RS SET CON CLE
 M <EN> = < > *)) , . :
 M <AR> + - * / = < > *
 M <SR> SAR SIN COS EXP LN SQRT ARC SIGN
 * ACTIONS

EXEC
 NSTK
 STAK
 VALU
 SUBR
 SCAN
 NEXT
 GET
 BUOK
 RETURN
 EHROR
 HALT
 CON
 NUM
 FOUT
 STRIN

* PRODUCTION TABLE
 37621 00000000170 00053117163 00003754224 00003560231 00611714205 00003606376 00000000712 00000017105
 37631 00000011111 00000063366 00000062722 00035514621 00164555471 00000001066 00027626043 00030330167
 37641 00204435745 00204355575 11743611142 00055161016 00164555471 00055161015 00055161014 00055161013 11333155110
 37651 11333155107 11333155106 00126625667 00126625666 00126625665 00126625664 00126625663 13305117277
 37661 13305117276 13305117275 00003455761 00000521703 20000006000 12320674355 20000006000 12063765243
 37671 20000006000 01605475232 00000041132 00000031073 00003711324 00000063150 00136603374 00217175262
 37701 00053146640 00000521147 00624052521 00027000157 00000053713 00003150344 00046435212 20000012000
 37711 02727146472 20001742000 06571415646 00002026340 00000617662 00217142444 00220652116 00000467034
 37721 00213552372 00003616226 00000016307 00017232775 00000000077 00000000073 00000000072 00000000071
 37731 00000000070 00000000065 00000000035 00000000034 00000000076 00000000067 00000000037 00000000053
 37741 00000000074 00000000075 00000000063 00000000061 00000000036 00000000066 00000000064 00000000062
 37751 00000000060 00000000057 00000000056 00000000055 00000000054 30000000000 30000000000 30000000000
 37761 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000
 37771 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000
 40001 30000000000 02000000053 00000000031 00000000103 30000000000 02000000156 00000000221 30000000000
 40011 30000000000
 ...

D0			BEGN				EXEC 1	*D1
			<SG>				ERROR 0	Q2
D1			<TP>				SCAN	*D2
			↓					*S1
			END	→			HALT	D1
			I	→			EXEC 2	*D1
			<SG>				ERROR 1	Q2
D2		<TP> I	,	→		<TP>	EXEC 3	D1
		<TP> I	;	→			EXEC 3	*D1
			I	→			SCAN	*TA
			<SG>				ERROR 2	Q2
TA	TABL I	I	,				SCAN	*TA1
TA1 I	I	,	I	→			EXEC 4	*TA2
TA2			TABL	→		TABL		D1
			TABL	→				*D1
			TABL	→		<SG>		D1
			<SG>				ERROR 3	Q2
S1			<CO>				SCAN	*AP1
			E				SCAN	*SP1
			<V>				SCAN	*EX1
			CD.				EXEC 5	*CD
			FALT					*F1
			STOP	→		UN	EXEC 57	*UN
			'				SCAN	*LAB
			;	→				*S1
		I	↓	→			ERROR 19	*S1
EX1			OK	→		BP	EXEC 6	*BS1
			W	→		OP	EXEC 15	*OP1
			TLOC	→		OP	EXEC 16	*OP1
			SIG	→		BP	EXEC 17	*BS1
			TEST				SCAN	*AP1
			<SG>					AP1
LAB		I	'	→			EXEC 7	*S1
OP1		<AR>	OP	→	<AR> AP	<SG>		AF1
			OP	→	BP	<BN>		BS1
			OP	→	AP	<SG>		AF1
			<SG>				ERROR 5	Q1
ID	L	I	I	→		OP	EXEC 58	*OP1
		I	I	→		AE	EXEC 8	AE3
		I	I	→		AE	EXEC 8	AE2
	TL	I	I	→	TL	AE	EXEC 9	T0
		I	I	→		TL	EXEC 10	*AP1
		I	I	→		OP	EXEC 11	OP1
			<SG>				ERROR 6	Q1
			<SG>				EXEC 12	*OP1
B1		CT	I	→		OP	EXEC 13	*OP1
		RT	I	→		OP	ERROR 7	Q1
			<SG>					*S1
CD		CD.	(	→		CD.	ERROR 4	Q1
			<SG>					*EX1
AP1			(					*ID
			I					*AP1
			<PM>				SCAN	*AP1
			IN				SCAN	*AP1
			L					

				CH			SCAN	
				CT				*AP1
				RT				*B1
				A	→	AP		*B1
				W	→	AP	EXEC 14	*AF1
				TLOC	→	AP	EXEC 15	*AF1
				<CO>			EXEC 16	*AF1
				<SR>	→	AE	SCAN	*AP1
				<Y>	→	OP	EXEC 18	*AE3
				<Z>	→	AP	EXEC 19	*OP1
				<			EXEC 20	*AF1
				PLAC	→	AE		*AP1
				<FL>	→	AP	EXEC 49	*AE2
				DEP	→	AP	EXEC 22	*AF1
				<SG>			EXEC 66	*AF1
AF1	AF	+	AP	<SG>	→	AF	ERROR 18	Q1
				AP			EXEC 21	AF2
				AP	→	AF		*EX1
AF2			AF	<SG>				AF2
AT1	AT	<TD>	AF	<SG>	→	AT		*AP1
				AF	→	AT	EXEC 23	AT2
AT2			AT	<TD>				AT2
AE1	AE	<PM>	AT	<SG>	→	AE		*AP1
				AT	→	AE	EXEC 24	AE2
				AT	→	AE		AE2
				AT	→	AE	EXEC 25	AE2
				AT	→	AE		AE2
				<SG>			ERROR 8	Q1
AE2			AE	<PM>				*AP1
				AE				AE3
				<	→	OP	EXEC 26	*OP1
				AE				*AP1
				AE	→	AP	EXEC 27	*AF1
				AE	→	AE		TO
TL	TL		AE	,	→	E	EXEC 27	*AP1
				AE	→	BP	EXEC 28	*EX1
E	TL		AE	,	→	UN	EXEC 29	BS1
AE	<RL>		AE	<SG>	→	UN	EXEC 30	UN
AP			AE	<SG>	→	BP	EXEC 31	BS1
<V>			AE	<SG>	→			BS1
				AE	→	AE	EXEC 32	*AE2
AE3	IN	(	AE	)	→	UN	EXEC 33	*UN
				AE	→	UN	EXEC 34	*UN
				AE	→	AP	EXEC 35	*AF1
				AE	→	UN	EXEC 36	*UN
				AE	→	UN	EXEC 37	*UN
				AE	→	UN	EXEC 38	*UN
				AE	→	AE	EXEC 39	*AE2
				AE	→	UN	EXEC 40	*UN
				AE	→	BP	EXEC 64	*BS1
				AE	→	UN	EXEC 65	*UN
				AE	→	UN	EXEC 41	*UN
				AE	→	UN	EXEC 42	*UN
				AE	→	UN	EXEC 54	*UN
				AE	→	BP	EXEC 56	*BS1
				AE	→	UN	EXEC 43	*UN

BS1			BP	<SG>	BS	<SG>	ERROR 9	Q1
			BP	<SG>	BS	<SG>	EXEC 44	BF1
BF1	BF	^	BS	<SG>	BF	<SG>	EXEC 45	BF1
			BS	<SG>	BF	<SG>		BF2
BF2			BF	^				BF2
BE1	BE	^	BF	<SG>	BE	<SG>	EXEC 46	*EX1
			BF	<SG>	BE	<SG>		BE2
				<SG>			ERROR 10	BE2
BE2			BE	^				Q1
			BE	^				*EX1
			BE	)	IC		EXEC 47	*S1
	<V>	^	BE	<SG>	BP		EXEC 48	*BS1
	AP	^	BE	<SG>	UN	<SG>	EXEC 31	UN
			BE	<SG>	UN	<SG>	EXEC 50	UN
UN			UN	<SG>			ERROR 11	Q1
				<SG>	S	<SG>		S9
S9			S	<SG>			ERROR 12	Q1
IC	SQ	]	S	\$	SQ	<SG>		SQ1
	SQ	]	S	)	S		EXEC 53	*S9
		CD.	S	)	S		EXEC 52	*S9
			S	<SG>	SQ	<SG>		SQ1
				<SG>			ERROR 13	Q1
SQ1	IC		SQ	\$	S		EXEC 53	*S9
			SQ	)	S		EXEC 52	*S9
			SQ	]			EXEC 59	*D1
			SQ	]			EXEC 55	*S1
			SQ	\$	S		EXEC 51	*S9
			SQ	]				*S1
				<SG>			ERROR 15	Q1
SP1		E	I	]	E	TL	EXEC 63	*SP2
SP2			TL	]		TL		*EX1
				<SG>			ERROR 16	Q1
T0		TL	AE	,	TL	AE	EXEC 60	*T0
		TL	AE	\$		OP	EXEC 61	*T1
T1				,				*T1
				]				*OP1
				<SG>			ERROR 17	Q1
Q1				]				*D1
				END			HALT	Q1
				<SG>				*Q1
Q2				]				D1
				END			HALT	Q1
				<SG>				*Q2
F1			FALT	<SG>	UN		EXEC 62	*UN
				<SG>			ERROR 20	Q1
*	END							

OTHER STUFF

I 1	4
I 2	0
I 3	0
I 4	0
I 5	0
I 6	20

I 7	62
I 8	67
I 9	0
I10	1
I11	67
I12	6
I13	15
I14	3
I15	0
I16	36
I17	2
I18	565
I19	* 0
I20	28
I21	16273
J 0	492
J 1	0
J 2	1
J 3	65
J 4	67
J 5	3
J 6	51
J 7	18

LABEL TABLE

	LABEL NAME	VALUE
1	D0	0
2	D1	4
3	Q2	481
4	D2	14
5	S1	48
6	TA	26
7	TA1	31
8	TA2	37
9	AP1	136
10	SP1	452
11	EX1	67
12	CD	131
13	F1	487
14	UN	400
15	LAB	81
16	BS1	350
17	OP1	85
18	AF1	176
19	Q1	475
20	ID	97
21	AE3	268
22	AE2	219
23	T0	461
24	B1	123
25	AF2	187
26	AT1	190
27	AT2	198

28	AE1	201
29	BF1	357
30	BF2	365
31	BE1	368
32	BE2	378
33	S9	405
34	SQ1	425
35	SP2	456
36	T1	469
37		1

PRODUCTION TABLE

62245	01050210000	00000000253	01050210003	00000000000	01050210003	01050700026	01050210010	00000000236
62255	01050210012	00000000254	01050210015	00000000200	01050210021	00000000000	01050410023	00000000240
62263	00000000200	01050700026	01050410026	00000000241	00000000200	01050700026	01050210032	00000000247
62275	01050210036	00000000000	01050510040	00000000240	00000000200	00000000200	00000000261	01050610043
62305	00000000250	00000000200	00000000240	00000000200	00000000200	01050310047	00000000240	00000000261
62315	01050310051	00000000241	00000000261	01050310054	00000000000	00000000261	01050210057	00000000000
62325	01050210061	01051400054	01050210064	00000000270	01050210067	01050300035	01050210072	00000000275
62335	01050210075	00000000276	01050210077	00000000255	01050210104	00000000252	01050210107	00000000241
62345	01050310112	00000000236	00000000200	01050210116	00000000232	01050210120	00000000341	01050210125
62355	00000000310	01050210132	00000000312	01050210137	00000000334	01050210144	00000000302	01050210147
62365	00000000000	01050410150	00000000252	00000000200	00000000252	01050410154	00000000000	00000000201
62375	01051000102	01050310160	01050300023	00000000201	01050310164	00000000000	00000000201	01050210170
62405	00000000000	01050510172	00000000250	00000000200	00000000247	00000000304	01050410177	00000000250
62415	00000000200	00000000247	01050410204	01051200011	00000000200	00000000243	01050510211	00000000240
62425	00000000200	00000000247	00000000206	01050310216	00000000247	00000000200	01050310224	00000000000
62435	00000000200	01050210231	00000000000	01050310233	00000000200	00000000313	01050310240	00000000200
62445	00000000314	01050210245	00000000000	01050310247	00000000246	00000000275	01050210252	00000000000
62455	01050210254	00000000246	01050210256	00000000200	01050210260	01050200003	01050210262	00000000303
62465	01050210265	00000000304	01050210270	00000000273	01050210273	00000000313	01050210275	00000000314
62475	01050210277	00000000306	01050210304	00000000310	01050210311	00000000312	01050210316	01051400054
62503	01050210321	01051000112	01050210326	01050500040	01050210333	01050300045	01050210340	00000000230
62515	01050210342	00000000342	01050210347	01050400050	01050210354	00000000343	01050210361	00000000000
62525	01050510363	00000000000	00000000202	00000000235	00000000203	01050310367	00000000243	00000000202
62535	01050310371	00000000000	00000000202	01050310375	00000000235	00000000203	01050510377	00000000000
62545	00000000203	01050200001	00000000204	01050310403	00000000000	00000000203	01050310407	01050200001
62555	00000000204	01050510411	00000000000	00000000204	01050200003	00000000205	01050410415	00000000000
62565	00000000204	00000000222	01050410421	00000000000	00000000204	00000000223	01050310426	00000000000
62575	00000000204	01050210432	00000000000	01050310434	01050200003	00000000205	01050310436	00000000250
62605	00000000205	01050410437	00000000231	00000000205	00000000230	01050310444	01050400005	00000000205
62615	01050410446	00000000251	00000000205	00000000246	01050510453	00000000240	00000000205	00000000247
62625	00000000206	01050410457	00000000240	00000000205	00000000247	01050510464	00000000240	00000000205
62635	00000000206	00000000270	01050510470	00000000000	00000000205	01050400005	00000000205	01050510475
62645	00000000000	00000000205	00000000243	00000000202	01050510502	00000000000	00000000205	00000000243
62655	01050300035	01050310507	00000000000	00000000205	01050510513	00000000250	00000000205	00000000247
62665	00000000303	01050510520	00000000250	00000000205	00000000247	00000000263	01050510525	00000000250
62675	00000000205	00000000247	00000000265	01050510532	00000000250	00000000205	00000000247	00000000304
62705	01050510537	00000000250	00000000205	00000000247	00000000267	01050510544	00000000250	00000000205
62715	00000000247	00000000277	01050510551	00000000250	00000000205	00000000247	00000000300	01050510556
62725	00000000250	00000000205	00000000247	00000000273	01050510563	00000000250	00000000205	00000000247
62735	00000000274	01050510570	00000000250	00000000205	00000000247	00000000337	01050510575	00000000250
62745	00000000205	00000000247	00000000340	01050510602	00000000250	00000000205	00000000205	00000000271
62755	01050510607	00000000250	00000000205	00000000205	00000000272	01050510614	00000000250	00000000205
62765	00000000205	00000000301	01050510621	00000000250	00000000205	00000000205	00000000302	01050510626

627/5	00000000250	00000000205	00000000206	00000000270	01050210633	00000000000	01050410635	00000000000
63005	00000000210	00000000232	01050310642	00000000000	00000000210	01050510646	00000000000	00000000211
63015	00000000234	00000000212	01050310652	00000000000	00000000211	01050310656	00000000234	00000000212
63025	01050510660	00000000000	00000000212	00000000233	00000000213	01050310664	00000000000	00000000212
63035	01050210670	00000000000	01050310672	00000000233	00000000213	01050310674	00000000244	00000000213
63045	01050410701	00000000251	00000000213	00000000246	01050510706	00000000000	00000000213	00000000243
63055	01050300035	01050510713	00000000000	00000000213	00000000243	00000000202	01050210720	00000000000
63065	01050310722	00000000000	00000000215	01050210726	00000000000	01050510730	00000000000	00000000217
63075	00000000241	00000000220	01050610733	00000000245	00000000217	00000000242	00000000220	00000000214
63105	01050410740	00000000251	00000000217	00000000275	01050310745	00000000000	00000000217	01050210751
63115	00000000000	01050610753	00000000245	00000000220	00000000242	00000000220	00000000214	01050410760
63125	00000000251	00000000220	00000000275	01050410765	00000000236	00000000220	00000000236	01050410771
63135	00000000242	00000000220	00000000214	01050410774	00000000245	00000000220	00000000214	01050311001
63145	00000000241	00000000220	01050211003	00000000000	01050411005	00000000200	00000000247	00000000270
63155	01050311012	00000000241	00000000206	01050211015	00000000000	01050411017	00000000240	00000000205
63165	00000000206	01050411023	00000000245	00000000205	00000000206	01050211030	00000000240	01050211033
63175	00000000250	01050211036	00000000000	01050211040	00000000236	01050211043	00000000254	01050211045
63205	00000000000	01050211050	00000000236	01050211051	00000000254	01050211053	00000000000	01050311056
63215	00000000000	00000000276	01050211063	00000000000	30000000000			

AUXILIARY PRODUCTION TABLE

65202	00100000000	00000000000	00000000215	00000000201	00000000205	00000000217	00000000240	30000000000
65212	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
65222	30000000000	30000000000	30000000000	30000000000				

INTERPRETATION TABLE

65227	00010000001	00060000001	00070000004	00130000000	00070000741	00060000000	00060000001	00070000016
65237	00060000001	00070000000	00020000001	00140000000	00070000004	00020000001	00010000002	00060000001
65247	00070000004	00130000001	00070000741	00020000002	00010000003	00070000004	00020000003	00010000003
65257	00060000001	00070000004	00020000001	00060000000	00060000001	00070000032	00130000002	00070000741
65267	00060000000	00060000001	00070000037	00020000005	00010000004	00060000001	00070000045	00020000001
65277	00070000004	00020000002	00060000001	00070000004	00020000002	00030000000	00070000004	00130000003
65307	00070000741	00060000000	00060000001	00070000210	00060000000	00060000001	00070000704	00060000000
65317	00060000001	000700000103	00010000005	00060000001	00070000203	00060000001	00070000747	00020000001
65327	00030000215	00010000071	00060000001	00070000620	00060000000	00060000001	00070000121	00020000001
65337	00060000001	00070000060	00020000002	00130000023	00060000001	00070000060	00060000001	00070000103
65347	00020000001	00030000210	00010000006	00060000001	00070000536	00020000001	00030000201	00010000017
65357	00060000001	00070000125	00020000001	00030000201	00010000020	00060000001	00070000125	00020000001
65367	00030000210	00010000021	00060000001	00070000536	00060000000	00060000001	00070000210	00070000210
65377	00020000003	00010000007	00060000001	00070000060	00020000002	00030000202	00030000000	00070000260
65407	00020000002	00030000210	00030000000	00070000536	00020000002	00030000202	00030000000	00070000260
65417	00130000005	00070000733	00020000004	00030000201	00010000072	00060000001	00070000125	00020000002
65427	00030000205	00030000000	00010000010	00070000414	00020000002	00030000205	00030000000	00010000010
65437	00070000333	00020000003	00030000205	00030000000	00010000011	00070000715	00020000002	00030000206
65447	00030000000	00010000012	00060000001	00070000210	00020000002	00030000201	00030000000	00010000013
65457	00070000125	00130000006	00070000733	00020000002	00030000201	00010000014	00060000001	00070000125
65467	00020000002	00030000201	00010000015	00060000001	00070000125	00130000007	00070000733	00020000001
65477	00060000001	00070000060	00130000004	00070000733	00060000001	00070000103	00060000001	00070000141
65507	00060000001	00070000210	00060000000	00060000001	00070000210	00060000000	00060000001	00070000210
65517	00060000000	00060000001	00070000210	00060000001	00070000173	00060000001	00070000173	00020000001
65527	00030000202	00010000016	00060000001	00070000260	00020000001	00030000202	00010000017	00060000001
65537	00070000260	00020000001	00030000202	00010000020	00060000001	00070000260	00060000000	00060000001
65547	00070000210	00020000001	00030000205	00010000022	00060000001	00070000414	00020000001	00030000201
65557	00010000023	00060000001	00070000125	00020000001	00030000202	00010000024	00060000001	00070000260
65567	00060000001	00070000210	00020000000	00030000205	00010000061	00060000001	00070000333	00020000001

```

65577 00030000202 00010000026 00060000001 00070000260 00020000001 00030000202 00010000102 00060000001
65607 00070000260 00130000022 00070000733 00020000003 00030000000 00010000025 00070000273 00060000001
65617 00070000103 00020000002 00030000203 00030000000 00070000273 00060000001 00070000210 00020000003
65627 00030000000 00010000027 00070000306 00020000002 00030000204 00030000000 00070000306 00060000001
65637 00070000210 00020000003 00030000000 00010000030 00070000333 00020000003 00030000205 00030000000
65647 00070000333 00020000003 00030000205 00030000000 00010000031 00070000333 00020000002 00030000205
65657 00030000000 00070000333 00130000010 00070000733 00060000001 00070000210 00070000414 00020000003
65667 00030000201 00010000032 00060000001 00070000125 00060000001 00070000210 00020000003 00030000202
65677 00010000033 00060000001 00070000260 00020000003 00030000001 00030000000 00070000715 00020000003
65707 00030000001 00010000033 00060000001 00070000210 00020000002 00010000034 00060000001 00070000103
65717 00020000004 00030000210 00030000000 00010000035 00070000536 00020000004 00030000215 00030000000
65727 00010000036 00070000620 00020000004 00030000215 00030000000 00010000037 00070000620 00020000002
65737 00030000210 00030000000 00070000536 00020000004 00030000001 00010000040 00060000001 00070000333
65747 00020000004 00030000215 00010000041 00060000001 00070000620 00020000004 00030000215 00010000042
65757 00060000001 00070000620 00020000004 00030000202 00010000043 00060000001 00070000260 00020000004
65767 00030000215 00010000044 00060000001 00070000620 00020000004 00030000215 00010000045 00060000001
65777 00070000620 00020000004 00030000215 00010000046 00060000001 00070000620 00020000001 00030000001
66007 00010000047 00060000001 00070000333 00020000004 00030000215 00010000050 00060000001 00070000620
66017 00020000004 00030000210 00010000100 00060000001 00070000536 00020000004 00030000215 00010000101
66027 00060000001 00070000620 00020000004 00030000215 00010000051 00060000001 00070000620 00020000004
66037 00030000215 00010000052 00060000001 00070000620 00020000004 00030000215 00010000066 00060000001
66047 00070000620 00020000004 00030000210 00010000070 00060000001 00070000536 00020000004 00030000215
66057 00010000053 00060000001 00070000620 00130000011 00070000733 00020000003 00030000211 00030000000
66067 00010000054 00070000545 00020000002 00030000211 00030000000 00070000545 00020000003 00030000000
66077 00010000055 00070000555 00020000002 00030000212 00030000000 00070000555 00060000001 00070000103
66107 00020000003 00030000000 00010000056 00070000572 00020000002 00030000213 00030000000 00070000572
66117 00130000012 00070000733 00060000001 00070000103 00020000002 00030000214 00010000057 00060000001
66127 00070000060 00020000003 00030000210 00010000060 00060000001 00070000536 00020000004 00030000215
66137 00030000000 00010000037 00070000620 00020000004 00030000215 00030000000 00010000062 00070000620
66147 00130000013 00070000733 00020000002 00030000217 00030000000 00070000625 00130000014 00070000733
66157 00020000003 00030000000 00070000651 00020000005 00030000001 00010000065 00060000001 00070000625
66167 00020000003 00030000001 00010000064 00060000001 00070000625 00020000002 00030000220 00030000000
66177 00070000651 00130000015 00070000733 00020000005 00030000217 00010000065 00060000001 00070000625
66207 00020000003 00030000217 00010000064 00060000001 00070000625 00020000003 00010000073 00060000001
66217 00070000004 00010000067 00060000001 00070000060 00020000003 00030000217 00010000063 00060000001
66227 00070000625 00060000001 00070000060 00130000017 00070000733 00020000002 00030000206 00010000077
66237 00060000001 00070000710 00020000001 00060000001 00070000103 00130000020 00070000733 00020000001
66247 00010000074 00060000001 00070000715 00020000003 00030000201 00010000075 00060000001 00070000725
66257 00020000001 00060000001 00070000725 00020000001 00060000001 00070000125 00130000021 00070000733
66267 00020000001 00060000001 00070000004 00140000000 00070000733 00020000001 00060000001 00070000733
66277 00070000004 00140000000 00070000733 00020000001 00060000001 00070000741 00020000002 00030000215
66307 00010000076 00060000001 00070000620 00130000024 00070000733

```

* TAPE 8 4

RECORD TYPE	NUMBER OF RECORDS	STARTING RECORD
SYMBOL TABLE	1	8
HIERARCHY TABLE	2	9
PRODUCTION TABLE	1	13
METACHARACTER LISTS	2	14

TIME USED: 00:05:02 PAGES USED: 11 19:08:18

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.

 OPERATOR-040 14 MAY 64 19:10:11 MAGIC PAGES: 50 TIME: 1

CC35.013

J. FELDMAN
COMPUTER CENTER00:00:04^①

CO EXAMPLE 1

SN DUMP 1

BEGIN REAL X,Y,ZEBRA; BOOLEAN P,Q; LABEL L1,L2;

L1: Y← X*Y-ZEBRA*(Y/2);

IF IF P THEN Q ELSE X=3 THEN BEGIN X← X-1; GOTO L1 END

ELSE GOTO L2;

X← X+1; GOTO L1;

L2: Q← TRUE

END

20000	0	CLA	2	40000,00	0	MPY	2	40002,00	0	SID	0	72400,77	0	CLA	2	40002,00	0	DIV	0	00002,00
20005	0	STD	0	72652,00	0	CLA	2	40004,00	0	STD	0	72650,00	0	TRM	0	10253,00	0	SUN	2	72400,77
20012	0	STD	0	40002,00	0	CLA	2	40000,00	0	FUO	0	00003,00	0	OCA	0	00001,00	0	CCL	2	14376,00
20017	0	IOZ	2	40006,00	0	TRA	0	20024,00	0	CAL	2	40007,00	0	STL	0	72400,77	0	TRA	0	20025,00
20024	0	STL	0	72400,77	0	ICZ	2	72400,77	0	TRA	0	20030,00	0	TRA	0	20035,00	0	CLA	2	40000,00
20031	0	SUB	0	00001,00	0	STD	0	40000,00	0	TRA	0	20000,00	0	TRA	0	20036,00	0	TRA	0	20042,00
20036	0	CLA	2	40000,00	0	ADD	0	00001,00	0	STD	0	40000,00	0	TRA	0	20000,00	0	CAL	2	14377,00
20043	0	STL	0	40007,00	0	TRM	0	05300,00												

TIME USED: 00:00:07^② PAGES USED: 1 19:10:17

1. The compiler took four seconds to read in the tables of the small language.
2. Three seconds were used to compile and dump the program.
3. ↑ is executed by a library subroutine.
4. Location 5300 is the monitor HALT routine.
5. Temporaries start at 72400; the index register allows recursive use of temporaries.

 OPERATOR-040 14 MAY 64 19:18:37 MAGIC PAGES: 20 TIME: 3

CC35.013 J. FELDMAN
 COMPUTER CENTER

00:00:04

CO EXAMPLE 2

SN ① CODE 1
 SN ② EXEC 1
 SN DUMP

BEGIN REAL X,Y,ZEBRA; BOOLEAN P,Q; LABEL L1,L2;

00061	0000000001								
63334	0000000252	0000000000	0000000000	0000000000	0000000000	0000000252	0000000213	0000000213	
00061	0000000004								
63334	0000000234	0000000262	0000000246	0000000000	0000000000	0000000234	0000000262	0000000246	
00061	0000000004								
63334	0000000234	0000000263	0000000246	0000000000	0000000000	0000000234	0000000263	0000000246	
00061	0000000004								
63334	0000000235	0000000264	0000000246	0000000000	0000000000	0000000235	0000000264	0000000246	
00061	0000000005								
63334	0000000234	0000000265	0000000250	0000000000	0000000000	0000000234	0000000265	0000000250	
00061	0000000005								
63334	0000000235	0000000266	0000000250	0000000000	0000000000	0000000235	0000000266	0000000250	
00061	0000000036								
63334	0000000234	0000000267	0000000251	0000000000	0000000000	0000000234	0000000267	0000000251	
00061	0000000036								
63334	0000000235	0000000270	0000000251	0000000000	0000000000	0000000235	0000000270	0000000251	
	L1: IF (X*Y)/(X+Y) = ZEBRA THEN								
00061	0000000002								
63334	0000000267	0000000246	0000000252	0000000000	0000000000	0000000267	0000000235	0000000252	
00061	0000000032								
63334	0000000236	0000000267	0000000252	0000000000	0000000000	0000000235	0000000252	0000000213	
00061	0000000011								
63334	0000000262	0000000267	0000000252	0000000000	0000000000	0000000262	0000000242	0000000255	
00061	0000000011								
63334	0000000263	0103004000	0000000252	0000000000	0000000000	0000000263	0000000216	0103004000	
00061	0000000012								
63334	0000000245	0103004002	0000000216	0103004000	0000000000	0000000245	0103004000	0000000242	
20000	0405004000								
20001	0477004002								
00061	0000000017								
63334	0000000245	0003040000	0000000242	0103004000	0000000000	0000000242	0000000255	0000000235	
00061	0000000011								
63334	0000000262	0003040000	0000000242	0103004000	0000000000	0000000262	0000000242	0000000217	
00061	0000000011								
63334	0000000263	0103004000	0000000242	0103004000	0000000000	0000000263	0000000214	0103004000	
00061	0000000014								
63334	0000000245	0103004002	0000000214	0103004000	0000000000	0000000245	0103004000	0000000242	
20002	0153772400								
20003	0405004000								
20004	0445004002								
00061	0000000017								
63334	0000000245	0003040000	0000000242	0103004000	0000000000	0000000242	01032072400	0003040000	

```

00061 00000000013
63334 000000000220 00030400000 01032072400 00030400000 00000000000 00000000220 00030400000 00000000255
20005 00570040002
00061 00000000011
63334 000000000264 00030400000 01032072400 00030400000 00000000000 00000000264 00000000220 00030400000
00061 00000000021
63334 000000000256 01030040004 00000000220 00030400000 00000000000 00000000256 00030400000 00000000255
20005 05610040004
20007 00000000001
20010 04150014376
20010 04350014376
00061 00000000025
63334 000000000256 00000000000 00000000255 00030400000 00000000000 00000000255 00000000235 00000000252
00061 00000000003
63334 00000000000 00000400000 00000000255 00030400000 00000000000 00000400000 00000000235 00000000252
20007 00170000000
20010 00170000000
BEGIN X+ -X+ZEBRA; GOTO L2 END ELSE P= FALSE;
00061 00000000002
63334 000000000262 00000000252 00000000255 00030400000 00000000000 00000000262 00000000235 00000000252
00061 00000000011
63334 000000000262 00000000252 00000000255 00030400000 00000000000 00000000262 00000000215 00000000237
00061 00000000016
63334 000000000214 01030040000 00000000215 00030400000 00000000000 00000000214 00000000215 00000000237
00061 00000000011
63334 000000000264 01030040000 00000000215 00030400000 00000000000 00000000264 00000000214 01034040000
00061 00000000014
63334 000000000235 01030040004 00000000214 01034040000 00000000000 00000000235 01034040000 00000000237
20011 04250040000
20012 04450040004
00061 00000000020
63334 000000000235 00030400000 00000000237 00000000262 00000000000 00000000235 00000000262 00000000235
20013 01530040000
00061 00000000031
63334 000000000270 00000000254 00000000237 00000000262 00000000000 00000000254 00000000235 00000000252
20014 00170000000
00061 00000000035
63334 000000000253 00000000254 00000000235 00000000252 00000000000 00000000252 00000400000 00000000235
00061 00000000006
63334 000000000257 00000000252 00000400000 00000000252 00000000000 00000000257 00000000252 00000400000
20015 00170000000
00061 00000000030
63334 000000000235 01000214376 00000000237 00000000265 00000000000 00000000235 00000000265 00000000257
20016 04150014376
20017 01730040006
00061 00000000033
63334 000000000235 00000000265 00000000257 00000000252 00000400000 00000000235 00000000265 00000000235
L2: IF P THEN X+ Y+ZEBRA; Q+ X<Y
00061 00000000032
63334 000000000236 00000000270 00000000257 00000000252 00000400000 00000000235 00000000252 00000000213
00061 00000000024
63334 000000000256 00000000265 00000000257 00000000252 00000400000 00000000256 00000000265 00000000255
00061 00000000025
63334 000000000256 01000040006 00000000255 00000000252 00000400000 00000000255 00000000235 00000000252
00061 00000000003

```

151.

```

63334 01000040006 01000040006 00000000255 00000000252 00000400000 01000040006 00000000235 00000000252
20020 04310040006
20021 00170000000
20022 00170000000
00061 00000000010
63334 00000000231 00000000263 00000000255 00000000252 00000400000 00000000231 00000000263 00000000237
00061 00000000011
63334 00000000264 01030040002 00000000255 00000000252 00000400000 00000000264 00000000231 01030040002
00061 00000000034
63334 00000000235 01030040004 00000000231 01030040002 00000400000 00000000235 01030040002 00000000237
20023 04050040004
20024 01530072652
20025 04050040002
20026 01530072650
20027 01770010253
00061 00000000020
63334 00000000235 00030400000 00000000237 00000000262 00000400000 00000000235 00000000262 01000040006
20030 01530040000
00061 00000000007
63334 00000000235 00000000262 01000040006 00000000262 00000400000 00000000235 01000040006 00000000235
00061 00000000010
63334 00000000224 00000000262 01000040006 00000000262 00000400000 00000000224 00000000262 00000000237
00061 00000000011
63334 00000000263 00000000262 01000040006 00000000262 00000400000 00000000263 00000000224 01030040000
END
00061 00000000022
63334 00000000253 01030040002 00000000224 01030040000 00000400000 00000000253 01030040000 00000000237
20031 04050040000
20032 05210040002
20033 00000000001
20034 04150014376
00061 00000000027
63334 00000000253 00000400000 00000000237 00000000266 00000400000 00000000253 00000000266 00000000235
20035 01730040007
00061 00000000035
63334 00000000253 00000000266 00000000235 00000000252 00000400000 00000000252 00000000213 00000000213
00061 00000000037
63334 00000000252 00000000213 00000000235 00000000252 00000400000 00000000213 00000000213 00000000213
20036 01770005300
20000 0 CLA 2 40000,00 0 MPY 2 40002,00 0 STD 0 72400,77 0 CLA 2 40000,00 0 ADD 2 40002,00
20005 0 RDV 0 40002,00 0 FUC 2 40004,00 0 TRA 0 20011,00 0 TRA 0 20016,00 0 CLS 2 40000,00
20012 0 ADD 2 40004,00 0 STD 0 40000,00 0 TRA 0 20020,00 0 TRA 0 20020,00 0 CAL 2 14376,00
20017 0 STL 0 40006,00 0 ICZ 2 40006,00 0 TRA 0 20023,00 0 TRA 0 20031,00 0 CLA 2 40004,00
20024 0 STD 0 72652,00 0 CLA 2 40002,00 0 STD 0 72650,00 0 TRM 0 10253,00 0 STD 0 40000,00
20031 0 CLA 2 40000,00 0 FLO 2 40002,00 0 OCA 0 00001,00 0 CAL 2 14376,00 0 STL 0 40007,00
20036 0 TRM 0 05300,00

```

TIME USED: 00:00:14 PAGES USED: 3 19:18:50

1. This causes the code to be printed as it is compiled.
2. Causes the routine number and LEFT1-LEFT5 and RIGHT1-RIGHT5 to be printed.
3. EXEC 12 ↓ is called to add -X and ZERRA, it knows to subtract X because of (4)
4. The negation is bit set in LEFT4.

 OPERATOR-040 14 MAY 64 19:11:05 MAGIC PAGES: 50 TIME: 1

CC35.013 J. FELDMAN
 COMPUTER CENTER

00:00:04

CO SN DUMP EXAMPLE 3

BEGIN
 BEGIN REAL X,Y, ZEDD; BOOLEAN P,Q,ZEDE;
 ZEDE ← IF X/Y+ZEDD-X = X THEN P ELSE Q;
 X ← X*X + X*X END

BEGIN REAL X, Y, Z; BOOLEAN P,Q;

X ← Y + X*X*Z; ⁽⁵⁾
 X ← 1+1; Y ← 1.5; Z ← X/3;

① FAULT
 FAULT

P ← TRUE; Q ← FALSE ;

P ← Q END

END

20000	0 CLA 2 40000,00	0 DIV 2 40002,00	0 ADD 2 40004,00	0 SUB 2 40000,00	0 FUD 2 40000,00
20005	0 TRA 0 20011,00	0 CAL 2 40006,00	0 STL 0 72400,77	0 TRA 0 20013,00	0 CAL 2 40007,00
20012	0 STL 0 72400,77	0 CAL 2 72400,77	0 STL 0 40010,00	0 CLA 2 40000,00	0 MPY 2 40000,00
20017	0 STD 0 72401,77	0 CLA 2 40000,00	0 MPY 2 40000,00	0 ADD 2 72401,77	0 STD 0 40000,00
20024	0 CLA 2 40011,00	0 MPY 2 40011,00	0 MPY 2 40015,00	0 ADD 2 40013,00	0 STD 0 40011,00
20031	② 0 CAL 0 00271,00	0 STD 0 72652,00	② 0 CAL 0 00271,00	0 STD 0 72650,00	0 TRM 0 10253,00
20036	0 STD 0 40011,00	0 CLA 2 14401,00	0 STD 0 40013,00	0 CLA 2 40011,00	0 DIV 0 00003,00
20043	0 STD 0 40015,00	0 CAL 2 14377,00	④ 0 STL 0 40017,00	0 CAL 2 14376,00	④ 0 STL 0 40020,00
20050	0 CAL 0 40020,00	0 STL 0 40017,00	0 TRM 0 05300,00		

TIME USED: 00:00:08 PAGES USED: 1 19:11:12

1. The semantic error message FAULT 1 is printed twice because 'I' was not declared.
2. Since OK was not set FALSE, translation continues using the internal name of 'I' as its semantics.
3. The floating point constant '1.5' is put in a cell in the constant region.
4. TRUE and FALSE are also absolute constants.

Bibliography

1. Bar Hillel, Y., M. Perles and E. Shamir, "On Formal Properties of Simple Phrase Structure Grammars," Technical Report No. 4 ONR No. 62558-2214.
2. Brooker, R. and D. Morris, "An Assembly Program for a Phrase Structure Language". Computer Journal Vol. 3 p. 168 (1960).
3. Brooker, MacCallum, Morris, and Rohl, "The Compiler Compiler" in Annual Review of Automatic Programming. New York, Pergamon, 1963.
4. Cheatham, T. and K. Sattley, "Syntax Directed Compiling" Proceedings of AFIP Conference, Washington D.C., 1964.
5. Cheatham, T. and G. Leonard, "Introduction to the CL-II Programming System" CA-63-7-SD, Computer Associates Inc., June 1963.
6. Chomsky, N., "Three Models for the Description of Language", IRE Transactions on Information Theory, IT-2, 113-124, (1956).
7. Chomsky, N., "On Certain Formal Properties of Grammars", Information and Control 2, 137-167 (1959).
8. Chomsky, N., "Formal Properties of Grammars", in Handbook of Mathematical Psychology Vol. 2, New York 1963, Wiley.
9. Church, A., "Introduction to Mathematic Logic", Princeton, New Jersey, 1956, Princeton University Press.
10. Davis, M., "Computability and Unsolvability," McGraw-Hill, 1958.
11. Eickel, J., M. Paul, F. Bauer and K. Samelson, "A Syntax Controlled Generator of Formal Language Processors". Communications of the Association for Computing Machinery, August, 1963.
12. Evans, A., Paper to appear describing an ALGOL production loader.
13. Evans, A., "An ALGOL 60 Compiler" National Conference of the ACM, Denver, Colorado, 1963.
14. Evey, R., "The Theory and Applications of Pushdown Store Machines", Doctoral Dissertation, Harvard, May 1963.

29. McNaughton, R., "The Theory of Automata, A Survey," in Advances in Computers 2, (F. Alt, Editor), Academic Press, 1961.
30. Markov, A. A., "Theory of Algorithms", available from Office of Technical Service OTS 60-51085.
31. Naur, P., "Report on the Algorithm Language ALGOL 60", Communications of the ACM 3, 299-314, (1960).
32. Oettinger, A. G., "Automatic Syntax Analysis and the Pushdown Store", Proceedings Symposia in Applied Mathematics 12, 1961.
33. Perlis, A. J., "A Format Language" Communications of the ACM, February 1964.
34. Reynolds, J. C., "CONGENT, A Compiler and Generalized Translator", Applied Mathematics Report, Argonne National Laboratory, December 1962.
35. Ross, D., "On Context and Ambiguity in Parsing" Communications of the ACM, February, 1964.
36. Samelson, K. and F. Bauer, "Sequential Formula Translation", Communications of the ACM, Vol. 3, No. 2 (1960).
37. Scheinberg, S., "Note on the Boolean Properties of Context-Free Languages", Information and Control, Vol. 3 (1960)
38. Shamir, E., "On Sequential Languages", Technical Report No. 7, ONR No. 62558-2214.
39. Simon, H., "Experiments with a Heuristic Compiler", Journal of the ACM Vol. 10, No. 4, October 1963.
40. Steel, T., "Beginnings of a Theory of Information Handling", Communications of the ACM, February 1964.
41. Warshall, S., "Summary of a Method for the Automatic Construction of Syntax Directed Compilers". Computer Associates Report CA-62-3, December 1962.
42. Warshall, S. and R. Shapiro, "A General Purpose Table Driver Compiler", Proceedings of AFIP Conference, Washington D.C. 1964.
43. Yngve, V., "An Introduction to COMIT Programming", M.I.T. Press 1960.
44. Floyd, R., "On Ambiguity in Phrase Structure Languages", Communications of the ACM, Vol. 10, No. 5 October 1962.

Errata Sheet

This is an errata sheet for the paper entitled, "A Formal Semantics for Computer Oriented Languages", by Jerome A. Feldman, Computation Center, Carnegie Institute of Technology.

Page	Line(s)	Correction
61	12	Should read: ...as a computer system.
3	3	Should read: ...theorems in the propositional calculus
24	3, 7	K should be k
25	3, 8	
63	3	Should read: ...exponentiation which is usually...
67	11	Should read: ...computed before being...
84	8	Should read: ... Besides, we felt...
95	13	Should read: ...share the same...
98	12	Should read: ...as an abstract...
98	12	Should read: ...was devoted to progress
103	9	Should read: ...making 'in the optional
112	End operation	Should read: DCS ... (13)

* Number of lines, counting from the bottom of the page

The author's address is now:

Lincoln Laboratory
Lexington, Massachusetts

The FSI system at Carnegie Tech is being maintained by:

Mr. R. A. Krieger
Computation Center
Carnegie Institute of Technology
Pittsburgh, Pennsylvania 15213



Notes on the Appendices

The material contained in these appendices forms an integral part of the thesis. The basis for most of them is one or more machine listings. In all cases these are actual runs of the programs being described.

Appendices A-C comprise a complete description of the subset of ALGOL 60 which we call the small language. The BNF syntax of the small language which is Appendix A is what might be given in a published paper on the small language. The Appendices B and C are a solution to the problem of building a compiler for the small language.

Appendix B contains a run of the production loader on the syntax of the small language. The results of this run are some tightly packed tables which will control the scanning of small language source programs. The notation used in these tables is described in that section.

Appendix C contains a run of the semantic loader on the FSL semantics of the small language. The tables produced there are actually short programs which will be executed by the Basic Compiler. We will give a capsule description of the G-20 machine code at the end of these notes. This description will also be of use in reading Appendix F, which contains the translations of sample programs in the small language.

Appendices D and E describe the semantic meta-language, FSL. The Backus Normal Form syntax constitutes Appendix D while the Production Language Syntax is Appendix E. The tables produced in Appendix E follow the same format as those for the productions of the small language (Appendix B).

In order to fully understand the appendices, one must have some knowledge of G-20 machine code. We will present a brief description of the machine, emphasizing its differences from the better known computers of its class.

The Control Data (née Bendix) G-20 at Carnegie Tech is a large single address machine. Among its more interesting features are a floating point accumulator and a special operand assembly (OA) register. These are the only hardware registers with which we will be concerned. Although the G-20 has 64 index registers, these are in its 6 μ -second core memory and thus have appreciable access times. We will describe the G-20 command structure as it will appear in Appendices C and F. The format of a command is:

f xxx m ddddd , ii

where 'f' denotes the interrupt flag (0-3) used by the machine hardware. The symbols 'xxx' denote one of the 3-letter mnemonic opcodes which we will describe below. The 'm' denotes the addressing mode (0-3) and will also be described below. The field marked 'dddd' stands for the five

digit octal address used in a G-20 command and the 'ii' denotes one of the octal 100 index registers (0-77). The internal representation of a command is slightly different than our picture.

Before describing the commands, we must say something about the address structure of the G-20. This is based on the use of the OA register which automatically combines with the address field of every command according to the mode specified in the command. If we represent the address field of a command by A, the index field by I and the contents of OA register as (OA) we attain the following rules for building X, the effective address.

<u>Mode</u>	<u>Effective Address</u>
0	$(OA) + A + (I) = X$
1	$(OA) + (A) + (I) = X$
2	$((OA) + A + (I)) = X$
3	$((OA) + (A) + (I)) = X$

The normal mode for stores and transfer commands is mode 0, while all other commands are normally in mode 2. There are special commands for operating on the OA and, except for these instructions, the OA is set to zero after every command.

In the table of opcodes (Figure 4), (ACC) stands for the accumulator and all the symbols defined above retain their meanings. Notice that for all conditional statements the next command is executed if the

Partial List of G-20 Opcodes

Address Preparation		Index Register Codes	
OCA	$X \rightarrow (OA)$	LXP	$X \rightarrow I$
OCA	$-X \rightarrow (OA)$	LXM	$-X \rightarrow I$
OAD	$(ACC) + X \rightarrow (OA)$	ADX	$(I) + X \rightarrow I$
OSU	$(ACC) - X \rightarrow (OA)$	SUX	$(I) - X \rightarrow I$
Add and Subtract		AXT	$(I) + X \rightarrow I \quad (=0?)$
CLA	$X \rightarrow (ACC)$	SXT	$(I) - X \rightarrow I \quad (=0?)$
CLS	$-X \rightarrow (ACC)$	Transfer of Control	
ADD	$(ACC) + X \rightarrow (ACC)$	TRA	$X \rightarrow NC$
SUB	$(ACC) - X \rightarrow (ACC)$	TRM	$(NC) \rightarrow X; X + 1 \rightarrow NC$
ADN	$-(ACC) - X \rightarrow (ACC)$		
SUN	$-(ACC) + X \rightarrow (ACC)$		
ADA	$(ACC) + X \rightarrow (ACC)$		
SUA	$(ACC) - X \rightarrow (ACC)$		
Arithmetic Tests			
FOM	$X < 0$		
FOP	$X > 0$		
FLO	$(ACC) < X$		
FGO	$(ACC) > X$		
Multiply and Divide			
MPY	$(ACC) * X \rightarrow (ACC)$		
DIV	$(ACC) / X \rightarrow (ACC)$		
RDV	$X / (ACC) \rightarrow (ACC)$		
Logic Operations			
CAL	$X \rightarrow (ACC)$		
CCL	$X \rightarrow (ACC)$		
ADL	$(ACC) + X \rightarrow (ACC)$		
SUL	$(ACC) - X \rightarrow (ACC)$		
EXL	$(ACC) \wedge X \rightarrow (ACC)$		
UNL	$(ACC) \vee X \rightarrow (ACC)$		
Logic Tests			
IOZ	$X = 0$		
IOZ	$X = 0$		
IUO	$(ACC) - X \neq 0$		
Store			
STL	$(ACC) \rightarrow X$		
STD	$(ACC) \rightarrow X$		
STI	$(ACC) \rightarrow X$		
STZ	$0 \rightarrow X$		

Figure 4.

condition holds. With this capsule view of the G-20 and the descriptions accompanying the examples, it should be possible to follow the code generated in Appendices C and F. A much more complete set of examples, including a number of source languages, will appear as a separate paper.

Appendix A

Syntax of a Small Language

$\langle \text{arithmetic expression} \rangle ::= \langle \text{term} \rangle \mid \pm \langle \text{term} \rangle \mid \langle \text{arithmetic expression} \rangle \pm \langle \text{term} \rangle$
 $\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{term} \rangle * / \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle ::= \langle \text{primary} \rangle \mid \langle \text{factor} \rangle \uparrow \langle \text{primary} \rangle$
 $\langle \text{primary} \rangle ::= \langle \text{identifier} \rangle \mid (\langle \text{arithmetic expression} \rangle)$
 $\langle \text{simple boolean} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{arithmetic expression} \rangle \langle \text{relation} \rangle \langle \text{arithmetic expression} \rangle$
 $\langle \text{boolean} \rangle ::= \langle \text{simple boolean} \rangle \mid \langle \text{if clause} \rangle \langle \text{simple boolean} \rangle \text{ ELSE } \langle \text{boolean} \rangle$
 $\langle \text{if clause} \rangle ::= \text{IF } \langle \text{boolean} \rangle \text{ THEN}$
 $\langle \text{if statement} \rangle ::= \langle \text{if clause} \rangle \langle \text{unconditional} \rangle$
 $\langle \text{assignment} \rangle ::= \langle \text{identifier} \rangle \leftarrow \langle \text{arithmetic expression} \rangle \mid \langle \text{identifier} \rangle \leftarrow \langle \text{boolean} \rangle$
 $\langle \text{go to statement} \rangle ::= \text{GO TO } \langle \text{identifier} \rangle$
 $\langle \text{conditional} \rangle ::= \langle \text{if statement} \rangle \mid \langle \text{if statement} \rangle \text{ ELSE } \langle \text{statement} \rangle$
 $\langle \text{unconditional} \rangle ::= \langle \text{assignment} \rangle \mid \langle \text{go to statement} \rangle \mid \langle \text{block} \rangle$
 $\langle \text{declaration} \rangle ::= \langle \text{type} \rangle \langle \text{type list} \rangle \mid \langle \text{declaration} \rangle ; \langle \text{type} \rangle \langle \text{type list} \rangle$
 $\langle \text{type} \rangle ::= \text{REAL} \mid \text{BOOLEAN} \mid \text{LABEL}$
 $\langle \text{type list} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{type list} \rangle , \langle \text{identifier} \rangle$
 $\langle \text{head} \rangle ::= \text{BEGIN} \mid \text{BEGIN } \langle \text{declaration} \rangle \mid \langle \text{head} \rangle ; \langle \text{statement} \rangle$
 $\langle \text{block} \rangle ::= \langle \text{head} \rangle \text{ END}$
 $\langle \text{statement} \rangle ::= \langle \text{conditional} \rangle \mid \langle \text{unconditional} \rangle \mid \langle \text{empty} \rangle \mid \langle \text{identifier} \rangle : \langle \text{statement} \rangle$

The appendix is a machine run of the Production Language syntax of the small language. Besides the productions themselves there are three tables given as input to the Production Loader. The first is a table of reserved identifiers, both those used internally and those which appear in the source code. The second table defines the class names (meta-characters) as discussed in Chapter II. The table of actions contains all those mentioned in Chapter II as well as others used internally by the system.

As we have mentioned, the tables built by the Production Loader are tightly packed and thus difficult to read. The small table at 37715 to 40011 contains initializing information for the Basic Compiler. The two tables (other stuff and LABELS) before the production table contain debugging information which will not concern us here.

To read the production table we must understand the internal numbering system for symbols. Reserved symbols are given octal integers starting at 200 as internal names. After the last number assigned to a reserved symbol (261 for the small language), the source code identifiers are numbered sequentially.

The production table consists of two types of words, head cells and operands. Consider the first entry (62245) in the production table.

0 105 02 10000

The '105' marks this as the head cell of a production and the '02' gives the relative address of the next head cell. The '10000' is the relative address of the interpretation table entry corresponding to this production. We will discuss the interpretation table below.

The next cell contains '252' in the low order positions. This is the octal integer which corresponds to 'BEGIN'. A word of all zeros in the production table corresponds to a 'Ø' in a production.

The interpretation table is easier to follow. The integer in the high order position represents one of the actions in the Action Table. These are numbered sequentially starting with '1' for EXEC. The parameter to the action is contained in the low order bits of the same word. These tables are interpreted by the Basic Compiler in recognizing source language text.

Appendix B

Productions for the Small Language

* *	SYMBOLS	
	12 INTERNAL SYMBOLS	
		I
		P
		F
		T
		E
		SBE
		BE
		ICL
		UN
		S
		HEAD
		I→
+		+
-		-
*		*
/		/
=		=
<		<
>		>
<		<
>		>
<		<
>		>
↑		↑
↓		↓
.		.
,		,
;		;
:		:
←		←
→		→
\$		\$
(		(
[		[
]		]
)		)
REAL		REAL
BOOLEAN		BOOL
LABEL		LABL
BEGIN		BEGN
END		END
GOTO		GO
IF		IF
THEN		THEN
ELSE		ELSE
TRUE		TRUE
FALSE		FALS
* *	METACHARACTERS	

M <OP> + - * / +
 M <IP> REAL BOOL LABL
 M <RL> = # < > ~< ~>
 M <PM> + -
 M <ID> * /
 * ACTIONS

EXEC
 VSTK
 STAK
 VALU
 SUBR
 SCAN
 NEXT
 GET
 BOOK
 RETURN
 ERROR
 HALT
 CON
 NUM
 FOUT
 STRIN

	PRODUCTIONS								
37715	00000000074	00053117163	00003754224	00001020101	00003720323	000000000523	00001336065	00000016307	
37725	00017232775	00125776240	20000034000	01177202242	00003402170	000000000073	000000000072	000000000071	
37735	000000000070	000000000065	000000000035	000000000034	000000000076	000000000067	000000000037	000000000053	
37745	000000000074	000000000075	000000000036	000000000036	000000000036	000000000066	000000000064	000000000063	
37755	000000000062	000000000061	000000000060	000000000057	000000000056	000000000055	000000000054	300000000000	
37765	300000000000	300000000000	300000000000	300000000000	300000000000	300000000000	300000000000	300000000000	
37775	300000000000	300000000000	300000000000	000000000261	300000000000	020000000046	000000000032	000000000014	
40005	300000000000	020000000062	000000000213	300000000000	300000000000				

	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							*B4
B2			SBE	ELSE							*B1
			SBE	<SG>		→	BE	<SG>			B3
B3			IF	THEN		→		ICL		EXEC 21	I1
	ICL	SBE	ELSE	<SG>		→	BE	<SG>		EXEC 22	B3
		I	+	<SG>		→	UN	<SG>		EXEC 23	UN1
EX1				I							*EX2
				IF							*B1
				+		→					*P1
				-							*P1
				(							*P1
EX2			I	<OP>		→	P	<OP>		EXEC 8	F1
			I	<RL>		→	E	<RL>		EXEC 8	*P1
		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	;	S	;		HEAD	;			*S1
		HEAD	;	S	END			UN		EXEC 29	ND1
ND1			I→	UN		→		I→		EXEC 31	Q1
				<SG>							*UN1
Q1				<SG>						HALT	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

117	2
118	293
119	0
120	28
121	16333
J 0	287
J 1	0
J 2	1
J 3	61
J 4	67
J 5	7
J 6	51
J 7	12

LABEL TABLE

	LABEL NAME	VALUE
1	S0	0
2	D1	4
3	Q1	285
4	D2	19
5	S1	46
6	D3	31
7	B1	190
8	G1	253
9	S9	258
10	S2	58
11	EX1	230
12	I1	66
13	UN1	80
14	C1	89
15	P1	99
16	F1	109
17	F2	117
18	T1	120
19	T2	133
20	E1	136
21	E2	153
22	B2	209
23	B4	198
24	B3	215
25	EX2	240
26	ND1	280
27		1

PRODUCTION TABLE

62243	01050210000	00000000252	01050210003	00000000000	01050210005	01050300006	01050310010	00000000253
62255	00000000252	01050310014	00000000235	00000000252	01050410017	00000000000	00000000213	00000000252
62265	01050310025	00000000000	00000000252	01050410033	00000000000	00000000200	00000000246	01050410035
62275	00000000000	00000000200	00000000250	01050410037	00000000000	00000000200	00000000251	01050410041
62305	00000000234	00000000200	01050300006	01050510045	00000000235	00000000200	01050300006	00000000213
62315	01050410050	00000000235	00000000200	01050400006	01050210054	00000000000	01050210056	00000000252
62325	01050210060	00000000255	01050210062	00000000254	01050210064	00000000235	01050210070	00000000253
62335	01050210071	00000000200	01050310073	00000000236	00000000200	01050310077	00000000237	00000000200

62345	01050210101	00000000000	01050310103	00000000207	00000000237	01050310105	00000000207	00000000255
62355	01050410107	00000000207	00000000257	00000000205	01050210111	00000000207	01050210114	00000000000
62365	01050410116	00000000000	00000000210	00000000207	01050310117	00000000000	00000000210	01050210123
62375	00000000000	01050410125	00000000257	00000000210	00000000207	01050410130	00000000000	00000000210
62405	00000000207	01050210135	00000000000	01050210137	00000000200	01050210144	00000000242	01050210146
62415	00000000214	01050210151	00000000215	01050210153	00000000000	01050510155	00000000000	00000000201
62425	00000000231	00000000202	01050310161	00000000000	00000000201	01050310165	00000000231	00000000202
62435	01050510167	00000000000	00000000202	00000000216	00000000203	01050510173	00000000000	00000000202
62445	00000000217	00000000203	01050310177	00000000000	00000000202	01050310203	01050200021	00000000203
62455	01050510205	00000000000	00000000203	00000000214	00000000204	01050510211	00000000000	00000000203
62465	00000000215	00000000204	01050410215	00000000000	00000000203	00000000215	01050310222	00000000000
62475	00000000203	01050310226	01050200017	00000000204	01050410230	00000000245	00000000204	00000000242
62505	01050310235	01050600011	00000000204	01050510237	00000000000	00000000204	00000000237	00000000200
62515	01050510244	00000000000	00000000204	00000000220	00000000204	01050510251	00000000000	00000000204
62525	00000000224	00000000204	01050510256	00000000000	00000000204	00000000225	00000000204	01050510263
62535	00000000000	00000000204	01050600011	00000000204	01050210270	00000000000	01050210272	00000000255
62545	01050210274	00000000214	01050210277	00000000215	01050210301	00000000242	01050310303	01050500001
62555	00000000200	01050310310	01050600011	00000000200	01050310316	00000000000	00000000200	01050210323
62565	00000000200	01050310325	00000000257	00000000205	01050310327	00000000000	00000000205	01050410333
62575	00000000256	00000000206	00000000255	01050410337	00000000000	00000000206	00000000257	00000000205
62605	00000000207	01050510344	00000000000	00000000206	00000000237	00000000200	01050210351	00000000200
62615	01050210353	00000000255	01050210355	00000000214	01050210360	00000000215	01050210362	00000000242
62625	01050310364	01050500001	00000000200	01050310371	01050600011	00000000200	01050510377	00000000000
62635	00000000200	00000000237	00000000200	01050210404	00000000000	01050310406	00000000200	00000000254
62645	01050210413	00000000000	01050610415	00000000000	00000000211	00000000257	00000000210	00000000207
62655	01050610422	00000000000	00000000210	00000000257	00000000210	00000000207	01050510427	00000000235
62665	00000000211	00000000235	00000000212	01050510432	00000000253	00000000211	00000000235	00000000212
62675	01050310436	00000000210	00000000213	01050210441	00000000000	01050210443	00000000000	00000000000

AUXILIARY PRODUCTION TABLE

65202	00000000000	00000000210	00000000213	00000000253	00000000210	00000000235	00000000000	00000000000
65212	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000
65222	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000

INTERPRETATION TABLE

65227	00010000001	00060000001	00070000004	00130000000	000700000435	00060000000	00060000001	00070000023
65237	00020000002	000300000211	00060000001	00070000056	00020000001	00060000001	00070000004	00020000003
65247	000300000212	000300000235	00030000000	00010000002	00070000056	00020000002	000300000212	000300000235
65257	00030000000	00010000002	00070000056	00010000004	00070000037	00010000005	00070000037	00010000036
65267	00070000037	00020000002	00060000000	00060000001	00070000023	00020000003	00060000001	00070000004
65277	00020000003	000300000213	00060000001	00070000004	00130000001	000700000435	00060000001	00070000004
65307	00060000001	000700000276	00060000001	000700000375	00020000001	000300000211	00030000000	000700000402
65317	000700000402	00060000001	00070000072	00020000002	00010000032	00060000001	00070000056	00060000001
65327	000700000346	00130000002	000700000435	00060000001	000700000276	00060000001	000700000276	00060000001
65337	000700000276	00010000003	00060000001	00070000056	00130000004	000700000435	000700000131	00020000002
65347	000300000211	00030000000	000700000402	00130000005	000700000435	00010000006	00060000001	00070000056
65357	00020000003	000300000211	00030000000	00010000007	000700000402	00130000006	000700000435	00020000001
65367	000300000201	00010000011	00060000001	000700000155	00060000001	000700000143	00020000001	00060000001
65377	000700000143	00060000001	000700000143	00130000007	000700000435	00020000003	00030000000	00010000034
65407	000700000165	00020000002	000300000202	00030000000	000700000165	00060000001	000700000143	00020000003
65417	00030000000	00010000012	000700000205	00020000003	00030000000	00010000013	000700000205	00020000002
65427	000300000203	00030000000	000700000205	00060000001	000700000143	00020000003	00030000000	00010000014
65437	000700000231	00020000003	00030000000	00010000015	000700000231	00020000003	000300000204	00030000000
65447	00010000016	000700000231	00020000002	000300000204	00030000000	000700000231	00060000001	000700000143
65457	00020000003	000300000201	00010000017	00060000001	000700000155	00060000001	000700000143	00020000004

```

65467 00030000210 00030000000 00010000020 00070000120 00020000004 00030000205 00030000000 00010000021
65477 00070000321 00020000004 00030000205 00030000000 00010000022 00070000321 00020000004 00030000205
65507 00030000000 00010000023 00070000321 00020000004 00030000205 00030000000 00010000021 00070000321
65517 00130000010 00070000435 00060000001 00070000276 00020000001 00060000001 00070000143 00060000001
65527 00070000143 00060000001 00070000143 00020000002 00030000201 00030000000 00010000010 00070000155
65537 00020000002 00030000204 00030000000 00010000010 00060000001 00070000143 00020000002 00030000205
65547 00030000000 00010000024 00070000321 00060000001 00070000306 00060000001 00070000276 00020000002
65557 00030000206 00030000000 00070000327 00020000003 00030000207 00010000025 00070000102 00020000005
65567 00030000001 00030000000 00010000026 00070000327 00020000004 00030000210 00030000000 00010000027
65577 00070000120 00060000001 00070000360 00060000001 00070000276 00020000001 00060000001 00070000143
65607 00060000001 00070000143 00060000001 00070000143 00020000002 00030000201 00030000000 00010000010
65617 00070000155 00020000002 00030000204 00030000000 00010000010 00060000001 00070000143 00020000004
65627 00030000210 00030000000 00010000030 00070000120 00130000010 00070000435 00020000002 00030000210
65637 00010000031 00060000001 00070000120 00130000011 00070000435 00020000005 00030000001 00030000000
65647 00010000033 00070000402 00020000005 00030000211 00030000000 00010000033 00070000402 00020000002
65657 00060000001 00070000056 00020000004 00030000210 00010000035 00070000430 00020000001 00010000037
65667 00070000435 00060000001 00070000120 00140000000 00070000435

```

TAPE 210

RECORD TYPE	NUMBER OF RECORDS	STARTING RECORD
SYMBOL TABLE	1	21 +01
HIERARCHY TABLE	1	21 +01
PRODUCTION TABLE	1	22 +01
METACHARACTER LISTS	1	22 +01

TIME USED: 00102143 PAGES USED: 8 19114109

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↓ CODE(VALUE2 ← LEFT4 + LEFT2) ↓

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.

#####

OPERATOR-040 14 MAY 64 19:14:10 FSL PAGES: 100 TIME: 5

00:00:03

CO
CO

APPENDIX C FORMAL SEMANTICS OF THE SMALL LANGUAGE

SN DUMP

BEGIN TABLE SYMB(200,4) ; CELL LEV,T0,T1,T2,T3,T4,T5; STACK STR,SYM;
TITLE REAL, BOOL, LABE; DATA LOGIC, INTEGER, SINGLE, DOUBLE;

0+ T3 ← TEMPLOC ↓

1+ LEV ← 0; STR ← STORLOC; SYM ← LOC(SYMB) ↓

2+ LEV ≠ 0 → PUSH(STR,STORLOC); PUSH (SYM, LOC(SYMB))\$;
TALLY(LEV) ↓

3+ PUSH(FLAD1,0); CODE(-LEFT1 → JUMP(FLAD1))\$ ↓

4+ T0 ← STORLOC; SET(T0,DOUBLE); ENTER(SYMB;LEFT2,T0,REAL,LEV);
STORLOC ← STORLOC+2 ↓

5+ ENTER(SYMB;LEFT2,STORLOC,BOOL ,LEV);TALLY(STORLOC) ↓

6+ PUSH(FLAD2,0); CODE(JUMP(FLAD2)); ASSIGN(FLAD1)↓

7+ ASSIGN(FLAD1)↓

8+ CONST(LEFT2)→ RIGHT2← LEFT2 :
SYMB(LEFT2,,,\$,)= REAL→ RIGHT2← SYMB(LEFT2,\$,,):FAULT 1\$\$ ↓

9+ CONST(LEFT1) → RIGHT1 ← LEFT1 :
SYMB(LEFT1,,,\$,)= REAL→ RIGHT1← SYMB(LEFT1,\$,,):FAULT 1\$\$ ↓

10+ CODE(VALUE2←LEFT4*LEFT2)↓

11+ CODE(VALUE2←LEFT4/LEFT2)↓

12+ CODE(VALUE2←LEFT4+LEFT2)↓

13+ CODE(VALUE2←LEFT4-LEFT2)↓

14+ CODE(VALUE2←-LEFT2)↓

```

15+ RIGHT1←LEFT2+
16+ SYMB[LEFT4,,,$,]=REAL← COMT 2← SYMB[LEFT4,$,,,];
   CODE(COMT 2← LEFT2) ; TEMPLOC ← T3 ; FAULT 3$ ↓
17+ CODE(VALUE2←LEFT4=LEFT2)↓
18+ CODE(VALUE2←LEFT4<LEFT2)↓
19+ CODE(VALUE2←LEFT4>LEFT2)↓
20+CONST(LEFT2)→ RIGHT2← LEFT2 ;
   SYMB[LEFT2,,,$,]=BOOL←RIGHT2←SYMB[LEFT2,$,,,];
   SET(RIGHT2,LOGIC) ; TEMPLOC ← T3 ; FAULT 3$$$↓
21+ RIGHT1 ← LEFT2 ↓
22+ CODE( LEFT5← TEMPLOC← LEFT4; TEMPLOC← LEFT2$) ;
   RIGHT2← TEMPLOC; SET(RIGHT2,LOGIC) ;
   TALLY(TEMPLOC) ↓
23+ SYMB[LEFT4,,,$,]=BOOL← COMT 2← SYMB[LEFT4,$,,,];
   CODE( COMT 2← LEFT2); FAULT 4$ ↓
24+ COMT 4← SYMB[LEFT4,$,,,];
   CONST(LEFT2) → COMT 2← LEFT2;
   SYMB[LEFT2,,,$,]=SYMB[LEFT4,,,$,] → COMT 2← SYMB[LEFT2,$,,,];
   FAULT 5 $ $; TEMPLOC ← T3;
   CODE( COMT 4← COMT 2) ↓
25+ SYMB[LEFT1,,,$,] ≠ LABE → FAULT 6:
   COMT 2← LOC(SYMB[LEFT1,$,,,]);
   COMT 3← <COMT 2>; SYMB[LEFT1,,,$,] ≠ 0 → CODE( JUMP(COMT 3)) ;
   CODE( JUMPICHAIN( COMT 2) )) $$ ↓
26+ SYMB[LEFT2,,,$,] ≠ LABE → FAULT 6:
   SYMB[0,,,$,] ← 1 ;
   ASSIGN(LOC ( SYMB[ LEFT2,$,,,])) $↓
27+ ASSIGN(FLAD2) ↓
28+ CODE( VALUE2 ← LEFT4 + LEFT2) ↓
29+ MINUS(LEV); POP(STR,STORLOC); POP(SYM,LOC(SYMB)) ↓
30+ ENTER(SYMB;LEFT2,0,LABE, 0 ) ↓
31+ CODE( STOP) ↓

```

```

ENDJ
65600 0 CLA 0 00004,00 0 STI 0 45775,00 0 CLA 0 45774,00 0 STI 0 45777,00 0 CLA 0 00310,00
65605 0 STI 0 45776,00 0 LXP 0 44161,40 0 LXP 0 44015,41 0 CLA 0 00370,00 0 STL 0 44014,00
65612 0 CLA 0 00371,00 0 STL 0 44013,00 0 CLA 0 00372,00 0 STL 0 44012,00 0 CLA 2 00076,00
65617 0 STL 0 44327,00 0 TRA 0 17020,00 0 CLA 0 00000,00 0 STL 0 44333,00 0 CLA 2 00071,00
65624 0 STL 2 00040,00 0 CAL 2 45777,00 0 STL 2 00041,00 0 TRA 0 62110,00 0 CAL 2 44333,00
65631 0 FUO 0 00000,00 0 TRA 0 65634,00 0 TRA 0 65643,00 0 LXM 0 00001,74 0 ADX 0 00001,40
65636 0 CLA 2 00071,00 0 STL 2 00040,00 0 ADX 0 00001,41 0 CAL 2 45777,00 0 STL 2 00041,00
65643 0 CAL 2 44333,00 0 ADD 0 00001,00 0 STL 0 44333,00 0 TRA 0 62110,00 0 ADX 0 00001,53
65650 0 CLA 0 00000,00 0 STL 2 00053,00 0 CAL 2 63334,00 0 STL 0 63347,00 0 TRM 0 63516,00
65655 0 CAL 2 63351,00 0 STL 0 63346,00 0 TRM 0 63664,00 0 LXM 0 00001,74 0 CAL 0 00053,00
65662 0 UNL 2 63251,00 0 STL 0 63346,00 0 TRM 0 64057,00 0 OCA 2 63365,77 0 CAL 2 00000,00
65667 0 UNL 2 00070,00 0 STL 2 63365,77 0 SUX 0 00001,77 0 LXM 0 00001,74 0 TRA 0 62110,00
65674 0 CLA 2 00071,00 0 STL 0 44332,00 0 CAL 2 44332,00 0 UNL 2 63275,00 0 UNL 2 63313,00
65701 0 STL 0 44332,00 0 CAL 2 63335,00 0 STL 0 47022,00 0 CAL 2 44332,00 0 STL 0 47023,00
65706 0 CAL 2 44014,00 0 STL 0 47024,00 0 CAL 2 44333,00 0 STL 0 47025,00 0 CLA 0 45777,00
65713 0 TRM 0 64626,00 0 CLA 2 00071,00 0 ADD 0 00002,00 0 STI 0 00071,00 0 TRA 0 62110,00
65720 0 CAL 2 63335,00 0 STL 0 47022,00 0 CAL 2 00071,00 0 STL 0 47023,00 0 CAL 2 44013,00
65725 0 STL 0 47024,00 0 CAL 2 44333,00 0 STL 0 47025,00 0 CLA 0 45777,00 0 TRM 0 64626,00
65732 0 ADX 0 00001,71 0 TRA 0 62110,00 0 ADX 0 00001,54 0 CLA 0 00000,00 0 STL 2 00034,00
65737 0 CAL 0 00054,00 0 UNL 2 63251,00 0 STL 0 63346,00 0 TRM 0 64057,00 0 CAL 0 00053,00
65744 0 UNL 2 63251,00 0 TRM 0 64134,00 0 TRA 0 62110,00 0 CAL 0 00053,00 0 UNL 2 63251,00
65751 0 TRM 0 64134,00 0 TRA 0 62110,00 0 CAL 2 63335,00 0 IEZ 2 63225,00 0 OCS 0 00001,00
65756 0 CAL 2 14377,00 0 IUO 2 14377,00 0 TRA 0 65765,00 0 LXM 0 00001,74 0 CAL 2 63335,00
65763 0 STL 0 63342,00 0 TRA 0 66013,00 0 LXM 0 00001,74 0 LXP 0 00002,52 0 CAL 2 63335,00
65770 0 STI 0 47034,00 0 CLA 0 45777,00 0 TRM 0 64570,00 0 CAL 3 47034,00 0 FUO 2 44014,00
65775 0 TRA 0 66007,00 0 LXM 0 00001,74 0 TRM 0 64570,00 0 STL 0 63342,00 0 STI 0 47034,00
66002 0 CLA 0 45777,00 0 TRM 0 64570,00 0 CLA 0 00001,00 0 LXM 0 00001,74 0 TRA 0 66012,00
66007 0 LXM 0 00001,74 0 CLA 0 00001,00 0 CAL 2 63334,00 0 IEZ 2 63225,00 0 LXM 0 00001,74
66014 0 TRA 0 62110,00 0 TRA 0 66027,00 0 LXM 0 00001,74 0 LXP 0 00002,52 0 CAL 2 63334,00
66021 0 IUO 2 14377,00 0 LXM 0 00001,74 0 TRM 0 64570,00 0 CAL 3 47034,00 0 FUO 2 44014,00
66026 0 TRA 0 66055,00 0 LXP 0 00001,52 0 CAL 2 63334,00 0 STL 0 63341,00 0 STI 0 47034,00
66033 0 CLA 0 45777,00 0 TRM 0 64570,00 0 CAL 3 47034,00 0 STL 0 63341,00 0 TRA 0 66051,00
66040 0 LXM 0 00001,74 0 CLA 0 00001,00 0 TRM 0 64161,00 0 LXM 0 00001,74 0 CLA 0 45777,00
66045 0 TRM 0 64570,00 0 CAL 2 63337,00 0 CAL 2 63337,00 0 LXM 0 00001,74 0 LXM 0 00001,74
66052 0 CLA 0 00001,00 0 TRM 0 64161,00 0 TRM 0 62472,00 0 CAL 2 63335,00 0 TRA 0 62110,00
66057 0 ADX 0 00001,74 0 CAL 2 63337,00 0 LXP 0 00001,52 0 STL 0 63346,00 0 STL 0 63347,00
66064 0 TRM 0 63455,00 0 STL 0 63346,00 0 TRM 0 62472,00 0 CAL 2 63335,00 0 ADX 0 00001,74
66071 0 CAL 2 63337,00 0 CAL 2 63335,00 0 TRM 0 62472,00 0 TRA 0 62110,00 0 TRM 0 63461,00
66076 0 LXP 0 00001,52 0 STL 0 63346,00 0 CAL 2 63335,00 0 ADX 0 00001,74 0 CAL 2 63337,00
66103 0 STL 0 63346,00 0 CAL 2 63335,00 0 TRM 0 62472,00 0 TRM 0 63405,00 0 LXP 0 00001,52
66110 0 TRM 0 62472,00 0 TRA 0 62110,00 0 STL 0 63347,00 0 LXP 0 00001,52 0 STL 0 63346,00
66115 0 CAL 2 63335,00 0 STL 0 63347,00 0 TRM 0 63411,00 0 TRM 0 63516,00 0 TRM 0 62472,00
66122 0 TRA 0 62110,00 0 CAL 2 63335,00 0 STL 0 63347,00 0 LXP 0 00001,52 0 LXP 0 00001,52
66127 0 TRM 0 62472,00 0 TRA 0 62110,00 0 CAL 2 63335,00 0 STL 0 63341,00 0 TRA 0 62110,00
66134 0 LXP 0 00002,52 0 CAL 2 63337,00 0 TRA 0 66166,00 0 CLA 0 45777,00 0 TRM 0 64570,00
66141 0 CAL 3 47034,00 0 FUO 2 44014,00 0 LXM 0 00001,74 0 TRM 0 64570,00 0 CAL 3 47034,00
66146 0 CAL 2 63337,00 0 STI 0 47034,00 0 CLA 0 45777,00 0 TRM 0 64570,00 0 CAL 2 63335,00
66153 0 STL 0 46465,00 0 ADX 0 00001,74 0 CAL 2 46465,00 0 STL 0 63346,00 0 CAL 2 63335,00
66160 0 STL 0 63347,00 0 TRM 0 63762,00 0 LXM 0 00001,74 0 CAL 2 44327,00 0 STI 0 00076,00
66165 0 TRA 0 66171,00 0 LXM 0 00001,74 0 CLA 0 00003,00 0 TRM 0 64161,00 0 LXM 0 00001,74
66172 0 TRA 0 62110,00 0 ADX 0 00001,74 0 CAL 2 63337,00 0 STL 0 63346,00 0 CAL 2 63335,00
66177 0 STL 0 63347,00 0 TRM 0 63527,00 0 LXP 0 00001,52 0 TRM 0 62472,00 0 TRA 0 62110,00
66204 0 ADX 0 00001,74 0 CAL 2 63337,00 0 STL 0 63346,00 0 CAL 2 63335,00 0 STL 0 63347,00
66211 0 TRM 0 63537,00 0 LXP 0 00001,52 0 TRM 0 62472,00 0 TRA 0 62110,00 0 ADX 0 00001,74

```

66216	0 CAL 2 63337,00	0 STL 0 63346,00	0 CAL 2 63335,00	0 STL 0 63347,00	0 TRM 0 63543,00
66223	0 LXP 0 00001,52	0 TRM 0 62472,00	0 TRA 0 62110,00	0 CAL 2 63335,00	0 IEZ 2 63225,00
66230	0 OCS 0 00001,00	0 CAL 2 14377,00	0 IUD 2 14377,00	0 TRA 0 66240,00	0 LXM 0 00001,74
66235	0 CAL 2 63335,00	0 STL 0 63342,00	0 TRA 0 66274,00	0 LXM 0 00001,74	0 LXP 0 00002,52
66242	0 CAL 2 63335,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00
66247	0 FUD 2 44013,00	0 TRA 0 66270,00	0 LXM 0 00001,74	0 LXP 0 00001,52	0 CAL 2 63335,00
66254	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00	0 STL 0 63342,00
66261	0 CAL 2 63342,00	0 UNL 2 63272,00	0 UNL 2 63313,00	0 STL 0 63342,00	0 CAL 2 44327,00
66266	0 STI 0 00076,00	0 TRA 0 66273,00	0 LXM 0 00001,74	0 CLA 0 00003,00	0 TRM 0 64161,00
66273	0 LXM 0 00001,74	0 LXM 0 00001,74	0 TRA 0 62110,00	0 CAL 2 63335,00	0 STL 0 63341,00
66300	0 TRA 0 62110,00	0 CAL 2 63340,00	0 STL 0 63346,00	0 TRM 0 63664,00	0 LXM 0 00001,74
66305	0 ADX 0 00001,74	0 CAL 2 00076,00	0 STL 0 63346,00	0 CAL 2 63337,00	0 STL 0 63347,00
66312	0 TRM 0 63762,00	0 LXM 0 00001,74	0 TRM 0 63731,00	0 LXM 0 00001,74	0 ADX 0 00001,74
66317	0 CAL 2 00076,00	0 STL 0 63346,00	0 CAL 2 63335,00	0 STL 0 63347,00	0 TRM 0 63762,00
66324	0 LXM 0 00001,74	0 OCA 2 63365,77	0 CAL 2 00000,00	0 UNL 2 00070,00	0 STL 2 63365,77
66331	0 SUX 0 00001,77	0 LXM 0 00001,74	0 CLA 2 00076,00	0 STL 0 63342,00	0 CAL 2 63342,00
66336	0 UNL 2 63272,00	0 UNL 2 63313,00	0 STL 0 63342,00	0 ADX 0 00001,76	0 TRA 0 62110,00
66343	0 LXP 0 00002,52	0 CAL 2 63337,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00
66350	0 CAL 3 47034,00	0 FUD 2 44013,00	0 TRA 0 66373,00	0 LXM 0 00001,74	0 LXP 0 00001,52
66355	0 CAL 2 63337,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00
66362	0 STL 0 46465,00	0 ADX 0 00001,74	0 CAL 2 46465,00	0 STL 0 63346,00	0 CAL 2 63335,00
66367	0 STL 0 63347,00	0 TRM 0 63762,00	0 LXM 0 00001,74	0 TRA 0 66376,00	0 LXM 0 00001,74
66374	0 CLA 0 00004,00	0 TRM 0 64161,00	0 LXM 0 00001,74	0 TRA 0 62110,00	0 LXP 0 00001,52
66401	0 CAL 2 63337,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00
66406	0 STL 0 46467,00	0 CAL 2 63335,00	0 IEZ 2 63225,00	0 OCS 0 00001,00	0 CAL 2 14377,00
66413	0 IUD 2 14377,00	0 TRA 0 66421,00	0 LXM 0 00001,74	0 CAL 2 63335,00	0 STL 0 46465,00
66420	0 TRA 0 66454,00	0 LXM 0 00001,74	0 LXP 0 00002,52	0 CAL 2 63335,00	0 STI 0 47034,00
66425	0 CLA 0 45777,00	0 TRM 0 64570,00	0 LXP 0 00002,52	0 CAL 2 63337,00	0 STI 0 47034,00
66432	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00	0 FUD 3 47034,00	0 TRA 0 66450,00
66437	0 LXM 0 00001,74	0 LXP 0 00001,52	0 CAL 2 63335,00	0 STI 0 47034,00	0 CLA 0 45777,00
66444	0 TRM 0 64570,00	0 CAL 3 47034,00	0 STL 0 46465,00	0 TRA 0 66453,00	0 LXM 0 00001,74
66451	0 CLA 0 00005,00	0 TRM 0 64161,00	0 LXM 0 00001,74	0 LXM 0 00001,74	0 CAL 2 44327,00
66456	0 STI 0 00076,00	0 ADX 0 00001,74	0 CAL 2 46467,00	0 STL 0 63346,00	0 CAL 2 46465,00
66463	0 STL 0 63347,00	0 TRM 0 63762,00	0 LXM 0 00001,74	0 TRA 0 62110,00	0 LXP 0 00002,52
66470	0 CAL 2 63334,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00
66475	0 FUD 2 44012,00	0 TRA 0 66500,00	0 TRA 0 66504,00	0 LXM 0 00001,74	0 CLA 0 00006,00
66502	0 TRM 0 64161,00	0 TRA 0 66543,00	0 LXM 0 00001,74	0 LXP 0 00001,52	0 CAL 2 63334,00
66507	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 2 47034,00	0 STL 0 46465,00
66514	0 CAL 3 46465,00	0 STL 0 46466,00	0 LXP 0 00003,52	0 CAL 2 63334,00	0 STI 0 47034,00
66521	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CAL 3 47034,00	0 FUD 0 00000,00	0 TRA 0 66527,00
66526	0 TRA 0 66534,00	0 LXM 0 00001,74	0 CAL 2 46466,00	0 STL 0 63346,00	0 TRM 0 64057,00
66533	0 TRA 0 66542,00	0 LXM 0 00001,74	0 CAL 2 46465,00	0 TRM 0 64126,00	0 CAL 2 63351,00
66540	0 STL 0 63346,00	0 TRM 0 64057,00	0 LXM 0 00001,74	0 LXM 0 00001,74	0 TRA 0 62110,00
66545	0 LXP 0 00002,52	0 CAL 2 63335,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00
66552	0 CAL 3 47034,00	0 FUD 2 44012,00	0 TRA 0 66556,00	0 TRA 0 66562,00	0 LXM 0 00001,74
66557	0 CLA 0 00006,00	0 TRM 0 64161,00	0 TRA 0 66601,00	0 LXM 0 00001,74	0 LXP 0 00003,52
66564	0 CLA 0 00000,00	0 STI 0 47034,00	0 CLA 0 45777,00	0 TRM 0 64570,00	0 CLA 0 00001,00
66571	0 STL 2 47034,00	0 LXP 0 00001,52	0 CAL 2 63335,00	0 STI 0 47034,00	0 CLA 0 45777,00
66576	0 TRM 0 64570,00	0 CAL 2 47034,00	0 TRM 0 64134,00	0 LXM 0 00001,74	0 TRA 0 62110,00
66603	0 CAL 0 00054,00	0 UNL 2 63251,00	0 TRM 0 64134,00	0 TRA 0 62110,00	0 ADX 0 00001,74
66610	0 CAL 2 63337,00	0 STL 0 63346,00	0 CAL 2 63335,00	0 STL 0 63347,00	0 TRM 0 64024,00
66615	0 LXP 0 00001,52	0 TRM 0 62472,00	0 TRA 0 62110,00	0 CAL 2 44333,00	0 SUB 0 00001,00
66622	0 STL 0 44333,00	0 CAL 3 00040,00	0 STI 0 00071,00	0 SUX 0 00001,40	0 CAL 3 00041,00
66627	0 STL 0 45777,00	0 SUX 0 00001,41	0 TRA 0 62110,00	0 CAL 2 63335,00	0 STL 0 47022,00
66634	0 CLA 0 00000,00	0 STL 0 47023,00	0 CAL 2 44012,00	0 STL 0 47024,00	0 CLA 0 00000,00

66641 0 STL 0 47025,00 0 CLA 0 45777,00 0 TRM 0 64626,00 0 TRA 0 62110,00 0 CLA 0 05300,00
 66646 0 UNL 2 63166,00 0 TRM 0 64320,00 0 TRA 0 62110,00

37000	00000065616	00000065621	00000065630	00000065647	00000065674	00000065720	00000065734	00000065747
37010	00000065753	00000066015	00000066057	00000066070	00000066101	00000066112	00000066123	00000066131
37020	00000066134	00000066173	00000066204	00000066215	00000066226	00000066276	00000066301	00000066343
37030	00000066400	00000066467	00000066545	00000066603	00000066607	00000066620	00000066632	00000066645
37040	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37050	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37060	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37070	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37100	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37110	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37120	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37130	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
37140	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000

TIME USED: 00:00:29 PAGES USED: 5 19:14:50

Appendix D

BNF Syntax of the Formal Semantic Language

```

<program head> ::= BEGIN <declaration part> ; <sentence> |
                  <program head> ; <sentence>

<semantic program> ::= <program head> END

<declaration part> ::= <declaration> | <declaration part> ;
                     <declaration>

<declaration> ::= <table dec> | <stack dec> | <cell dec> | <title dec> |
                  <data dec>

<table dec> ::= TABLE <table specifier> | RTABL <table specifier> |
               <table dec> , <table specifier>

<stack dec> ::= STACK <identifier list> | RSTAK <identifier list>

<cell dec> ::= CELL <identifier list>

<title dec> ::= TITLE <identifier list>

<data dec> ::= DATA <identifier list>

<table specifier> ::= <table id> [<integer> , <integer>]

<sentence> ::= <integer> ↓ <statement sequence> ↓

<statement sequence> ::= <statement> | <statement sequence> ;
                      <statement> | CODE (<statement sequence>)

<statement> ::= <unconditional> | <conditional> | '<label>'
               <statement>

<unconditional> ::= <assignment> | <storage> | <transfer> |
                   <auxiliary> | CODE (<unconditional>)

<conditional> ::= <if clause> <statement sequence> $ |
                 <if clause> <statement sequence> :
                 <statement sequence> $ | CODE (<conditional>)

<assignment> ::= <left side> ← <arithmetic> |
                 <left side> ← <boolean>

```

$\langle \text{left side} \rangle ::= \langle \text{primary} \rangle \mid \text{VALUE1} \mid \text{VALUE2} \mid \text{VALUE3}$
 $\langle \text{storage} \rangle ::= \langle \text{stack command} \rangle \mid \langle \text{enter} \rangle$
 $\langle \text{stack command} \rangle ::= \text{PUSH} [\langle \text{stack id} \rangle, \langle \text{arithmetic} \rangle] \mid$
 $\text{POP} [\langle \text{stack id} \rangle, \langle \text{arithmetic} \rangle]$
 $\langle \text{enter} \rangle ::= \text{ENTER} [\langle \text{table id} \rangle; \langle \text{expression list} \rangle]$
 $\langle \text{expression list} \rangle ::= \langle \text{arithmetic} \rangle \mid \langle \text{expression list} \rangle, \langle \text{arithmetic} \rangle$
 $\langle \text{transfer} \rangle ::= \text{JUMP} [\langle \text{arithmetic} \rangle] \mid \text{MARKJUMP} [\langle \text{arithmetic} \rangle] \mid$
 $\text{JUMP} [\langle \text{label} \rangle] \mid \text{MARKJUMP} [\langle \text{subroutine} \rangle] \mid$
 $\text{EXECUTE} [\langle \text{arithmetic} \rangle]$
 $\langle \text{auxiliary} \rangle ::= \text{SET} [\langle \text{arithmetic} \rangle, \langle \text{data id} \rangle] \mid$
 $\text{FAULT} [\langle \text{identifier} \rangle] \mid \text{TALLY} [\langle \text{arithmetic} \rangle] \mid$
 $\text{MINUS} [\langle \text{arithmetic} \rangle] \mid \text{ASSIGN} [\langle \text{arithmetic} \rangle] \mid \text{STOP}$
 $\langle \text{operand} \rangle ::= \langle \text{arithmetic} \rangle \mid \text{COMT} [\langle \text{identifier} \rangle] \mid \text{RUNT} [\langle \text{identifier} \rangle] \mid$
 $\langle \text{production operand} \rangle \mid \langle \text{storage operand} \rangle$
 $\langle \text{production operand} \rangle ::= \text{LEFT1} \mid \text{LEFT2} \mid \text{LEFT3} \mid \text{LEFT4} \mid \text{LEFT5} \mid \text{RIGHT1} \mid$
 $\text{RIGHT2} \mid \text{RIGHT3}$
 $\langle \text{storage operand} \rangle ::= \langle \text{table operand} \rangle \mid \langle \text{cell id} \rangle \mid \langle \text{stack id} \rangle \mid$
 $\langle \text{title id} \rangle$
 $\langle \text{table operand} \rangle ::= \langle \text{table id} \rangle [\langle \text{arithmetic} \rangle \langle \text{commas} \rangle \$ \langle \text{commas} \rangle]$
 $\langle \text{commas} \rangle ::= , \mid \langle \text{commas} \rangle, \mid \langle \text{empty} \rangle$
 $\langle \text{primary} \rangle ::= \langle \text{operand} \rangle \mid \langle \text{constant} \rangle \mid (\langle \text{arithmetic} \rangle) \mid \text{LOC} [\langle \text{table id} \rangle] \mid$
 $\text{LOC} [\langle \text{stack id} \rangle] \mid \langle \text{system cell} \rangle \mid \langle \text{flad} \rangle$
 $\langle \text{system cell} \rangle ::= \text{CODELOC} \mid \text{STORLOC} \mid \text{TEMPLOC}$
 $\langle \text{factor} \rangle ::= \langle \text{primary} \rangle \mid \langle \text{factor} \rangle \uparrow \langle \text{primary} \rangle$
 $\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{term} \rangle \langle * / \rangle \langle \text{factor} \rangle$
 $\langle \text{arithmetic} \rangle ::= \langle \text{term} \rangle \mid \langle + \rangle \langle \text{term} \rangle \mid \langle \text{arithmetic} \rangle \langle + \rangle \langle \text{term} \rangle \mid$
 $\text{INT} [\langle \text{arithmetic} \rangle] \mid \text{ABS} [\langle \text{arithmetic} \rangle] \mid$
 $\text{LOC} [\langle \text{arithmetic} \rangle] \mid \text{CHAIN} [\langle \text{arithmetic} \rangle]$

$\langle \text{boolean primary} \rangle ::= \langle \text{arithmetic} \rangle \langle \text{relation} \rangle \langle \text{arithmetic} \rangle \mid$
 $\langle \text{operand} \rangle \mid (\langle \text{boolean} \rangle) \mid \text{TRUE} \mid \text{FALSE} \mid \text{SIGNAL} \mid$
 $\text{TEST} [\langle \text{arithmetic} \rangle, \langle \text{data id} \rangle] \mid$
 $\text{CONST} [\langle \text{arithmetic} \rangle] \mid \text{OK}$

$\langle \text{boolean factor} \rangle ::= \langle \text{boolean primary} \rangle \mid \neg \langle \text{boolean primary} \rangle$

$\langle \text{boolean term} \rangle ::= \langle \text{boolean factor} \rangle \mid \langle \text{boolean term} \rangle \wedge \langle \text{boolean factor} \rangle$

$\langle \text{boolean} \rangle ::= \langle \text{boolean term} \rangle \mid \langle \text{boolean} \rangle \vee \langle \text{boolean term} \rangle$

$\langle \text{if clause} \rangle ::= \langle \text{boolean} \rangle \rightarrow$

$\langle \text{stack id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{data id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{table id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{cell id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{title id} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{label} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle$

$\langle \text{identifier list} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{identifier list} \rangle, \langle \text{identifier} \rangle$

$\langle \text{subroutine} \rangle ::= \text{SIN} \mid \text{COS} \mid \text{EXP} \mid \text{LOG} \mid \text{ARCTAN} \mid \text{SIGN} \mid$

$\langle \text{flad} \rangle ::= \text{FLAD1} \mid \text{FLAD2} \mid \text{FLAD3} \mid \text{FLAD4}$

$\langle \text{relation} \rangle ::= = \mid > \mid < \mid \neq$

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
	SO
	I→
+	+
-	-
*	*
/	/
=	=
≠	≠
<	<
>	>
┌	┌
└	└
^	^
↑	↑
↓	↓
·	·
,	,
;	;
:	:
←	←
→	→
\$	\$
(	(
[	[
]	]
)	)
!	!
BEGIN	BEGN
END	END
STOP	STOP
STACK	STAK
CELL	CELL
TITLE	NAME
TABLE	TABL
DATA	DATA

JUMP	TR
MARKJUMP	TM
EXECUTE	X
ENTER	E
PUSH	ST
POP	RS
CHAIN	CH
ASSIGN	UC
CODE	CD.
FAULT	FALT
TALLY	TAL
MINUS	MIN
SET	SET
TEST	TEST
INT	IN
LOC	L
CODELOC	A
STORLOC	W
TEMPLOC	TLLOC
COMT	CT
RUNT	RT
VALUE1	V1
VALUE2	V2
VALUE3	V3
LEFT1	Y1
LEFT2	Y2
LEFT3	Y3
LEFT4	Y4
LEFT5	Y5
RIGHT1	Z1
RIGHT2	Z2
RIGHT3	Z3
FLAD1	FL1
FLAD2	FL2
FLAD3	FL3
FLAD4	FL4
SIGNAL	SIG
RSTAK	RSTK
RTABL	RTAB
CONST	CON
CLEAR	CLE
OK	OK
PLACE	PLAC
DEPTH	DEP
SAR	SAR
SIN	SIN
COS	COS
EXP	EXP
LN	LN
SQRT	SQRT
ARCTAN	ARC
SIGN	SIGN
TRUE	TRUE
FALSE	FALS

METACHARACTERS

```

M <TD> * /
M <PM> + -
M <RL> = < > #
M <EN> = < > # 1 ) , . 1
M <BN> ^ v ~
M <TP> TABL STAK CELL NAME DATA RSTK RTAB
M <V> V1 V2 V3
M <Y> Y1 Y2 Y3 Y4 Y5
M <Z> Z1 Z2 Z3
M <FL> FL1 FL2 FL3 FL4
M <CO> TR TH TAL MIN X CH UC ST RS SET CON CLE
M <EN> = < > # 1 ) , $ 1
M <AR> + - * / * < > #
M <SR> SAR SIN COS EXP LN SQRT ARC SIGN
* ACTIONS

```

```

EXEC
NSTK
STAK
VALU
SUBR
SCAN
NEXT
GET
BUOK
RETURN
EHROR
HALT
CON
VUM
FOUT
SIRIN

```

```

* PRODUCTION TABLE
37621 00000000170 00053117163 00003754224 00003560231 00611714205 00003606376 00000000712 00000017105
37631 00000011111 00000063366 00000062722 00035514621 00164555471 00000001066 00027626043 00030330167
37641 00204435745 00204355575 11743611142 00055161016 00055161015 00055161014 00055161013 11333155110
37651 11333155107 11333155106 00126625667 00126625666 00126625665 00126625664 00126625663 113305117277
37661 13305117276 13305117275 00003455761 00000521703 20000006000 12320674355 20000006000 12063765243
37671 20000006000 01605475232 00000041132 00000031073 00003711324 00000063150 00136603374 00217175262
37701 00053146640 00000521147 00624052521 00027000157 00000053713 00003150344 00046435212 20000012000
37711 02727146472 20001742000 06571415646 00002026340 00000617662 00217142444 00220652116 00000467034
37721 00213552372 00003616226 00000016307 00017232775 00000000077 00000000073 00000000072 00000000071
37731 00000000070 00000000065 00000000035 00000000034 00000000076 00000000067 00000000037 00000000053
37741 00000000074 00000000075 00000000063 00000000061 00000000036 00000000066 00000000064 00000000062
37751 00000000060 00000000057 00000000056 00000000055 00000000054 30000000000 30000000000 30000000000
37761 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000
37771 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000 30000000000
40001 30000000000 02000000053 00000000031 00000000103 30000000000 02000000156 00000000221 30000000000
40011 30000000000
...

```

D0			BEGN				EXEC 1	*D1
			<SG>				ERROR 0	Q2
D1			<TP>				SCAN	*D2
			↓					*S1
			END	→			HALT	D1
			I	→			EXEC 2	*D1
			<SG>				ERROR 1	Q2
D2		<TP> I	,	→		<TP>	EXEC 3	D1
		<TP> I	;	→			EXEC 3	*D1
			I	→			SCAN	*TA
			<SG>				ERROR 2	Q2
TA	TABL I	I	,				SCAN	*TA1
TA1 I	I	I	)	→			EXEC 4	*TA2
TA2		TABL	,	→		TABL		D1
		TABL	;	→				*D1
		TABL	<SG>	→		<SG>		D1
			<SG>				ERROR 3	Q2
S1			<CO>				SCAN	*AP1
			E				SCAN	*SP1
			<V>				SCAN	*EX1
			CD.				EXEC 5	*CD
			FALT					*F1
			STOP	→		UN	EXEC 57	*UN
			'				SCAN	*LAB
			;	→				*S1
		I	↓	→			ERROR 19	*S1
EX1			OK	→		BP	EXEC 6	*BS1
			W	→		OP	EXEC 15	*OP1
			TLOC	→		OP	EXEC 16	*OP1
			SIG	→		BP	EXEC 17	*BS1
			TEST				SCAN	*AP1
			<SG>					AP1
LAB		I	'	→			EXEC 7	*S1
OP1		<AR> OP	<SG>	→	<AR> AP	<SG>		AF1
		OP	<BN>	→	BP	<BN>		BS1
		OP	<SG>	→	AP	<SG>		AF1
			<SG>				ERROR 5	Q1
ID	L	I	)	→		OP	EXEC 58	*OP1
		I	)	→		AE	EXEC 8	AE3
		I	<EN>	→		AE	EXEC 8	AE2
	TL	I	,	→	TL	AE	EXEC 9	T0
		I	I	→		TL	EXEC 10	*AP1
		I	<SG>	→		OP	EXEC 11	OP1
			<SG>				ERROR 6	Q1
81		CT	I	→		OP	EXEC 12	*OP1
		RT	I	→		OP	EXEC 13	*OP1
			<SG>				ERROR 7	Q1
CD		CD.	(	→		CD.		*S1
			<SG>				ERROR 4	Q1
AP1			(					*EX1
			I					*ID
			<PM>					*AP1
			IN				SCAN	*AP1
			L				SCAN	*AP1

			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	Q1
AF1	AF	+	AP	<SG> →	AF	<SG>	EXEC 21	AF2
			AP					*EX1
			AP	<SG> →	AF	<SG>		AF2
AF2			AF					*AP1
AT1	AT	<TD>	AF	<SG> →	AT	<SG>	EXEC 23	AT2
			AF	<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
			AT	<SG> →	AE	<SG>		AE2
			<SG>				ERROR 8	Q1
AE2			AE	<PM>				*AP1
			AE					AE3
		<	AE	> →	OP		EXEC 26	*OP1
			AE	<RL>				*AP1
		(	AE	) →	AP		EXEC 27	*AF1
	TL		AE	, →	AE	,		TO
			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
	<V>	+	AE	<SG> →	UN	<SG>	EXEC 31	UN
			AE	<SG> →	BP	<SG>		BS1
AE3	IN		AE	) →	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

BS1			BP	<SG>		BS	<SG>	ERROR 9	Q1
			BP	<SG>		BS	<SG>	EXEC 44	BF1
BF1	BF	^	BS	<SG>		BF	<SG>	EXEC 45	BF1
			BS	<SG>		BF	<SG>		BF2
BF2			BF	^					*EX1
BE1	BE	~	BF	<SG>		BE	<SG>	EXEC 46	BE2
			BF	<SG>		BE	<SG>		BE2
				<SG>				ERROR 10	Q1
BE2			BE	~					*EX1
			BE	~		IC		EXEC 47	*S1
		(	BE	)		BP		EXEC 48	*BS1
	<V>	~	BE	<SG>		UN	<SG>	EXEC 31	UN
	AP	~	BE	<SG>		UN	<SG>	EXEC 50	UN
				<SG>				ERROR 11	Q1
UN			UN	<SG>		S	<SG>		S9
				<SG>				ERROR 12	Q1
S9	SQ	)	S	<SG>		SQ	<SG>		SQ1
IC	SQ	I	S	\$		S		EXEC 53	*S9
		CD.	S	)		S		EXEC 52	*S9
			S	<SG>		SQ	<SG>		SQ1
				<SG>				ERROR 13	Q1
SQ1	IC	SQ	SQ	\$		S		EXEC 53	*S9
		CD.	SQ	)		S		EXEC 52	*S9
		↓	SQ	↓				EXEC 59	*D1
		IC	SQ	↓				EXEC 55	*S1
		IC	SQ	\$		S		EXEC 51	*S9
			SQ	)					*S1
				<SG>				ERROR 15	Q1
SP1		E	I	I		E	TL	EXEC 63	*SP2
SP2			TL	)			TL		*EX1
				<SG>				ERROR 16	Q1
T0	TL	AE	,	,		TL	AE	EXEC 60	*T0
	-TL	AE	\$	,			OP	EXEC 61	*T1
T1			,	)					*T1
				<SG>					*OP1
Q1				↓				ERROR 17	Q1
				END				HALT	*D1
				<SG>					Q1
Q2				↓					*Q1
				END				HALT	D1
				<SG>					Q1
F1			FALT	<SG>		UN		EXEC 62	*Q2
				<SG>				ERROR 20	*UN
*	END								Q1

OTHER STUFF

I 1	4
I 2	0
I 3	0
I 4	0
I 5	0
I 6	20

I 7	62
I 8	67
I 9	0
I10	1
I11	67
I12	6
I13	15
I14	3
I15	0
I16	36
I17	2
I18	565
I19	* 0
I20	28
I21	16273
J 0	492
J 1	0
J 2	1
J 3	65
J 4	67
J 5	3
J 6	51
J 7	18

LABEL TABLE

	LABEL NAME	VALUE
1	D0	0
2	D1	4
3	Q2	481
4	D2	14
5	S1	48
6	TA	26
7	TA1	31
8	TA2	37
9	AP1	136
10	SP1	452
11	EX1	67
12	CD	131
13	F1	487
14	UN	400
15	LAB	81
16	BS1	350
17	OP1	85
18	AF1	176
19	Q1	475
20	ID	97
21	AE3	268
22	AE2	219
23	T0	461
24	B1	123
25	AF2	187
26	AT1	190
27	AT2	198

28	AE1	201
29	BF1	357
30	BF2	365
31	BE1	368
32	BE2	378
33	S9	405
34	SQ1	425
35	SP2	456
36	T1	469
37		1

144.

PRODUCTION TABLE									
62245	01050210000	00000000253	01050210003	00000000000	01050210005	01050700026	01050210010	00000000236	
62255	01050210012	00000000254	01050210015	00000000200	01050210021	00000000000	01050410023	00000000240	
62263	00000000200	01050700026	01050410026	00000000241	00000000200	01050700026	01050210032	00000000247	
62275	01050210036	00000000000	01050510040	00000000240	00000000200	00000000200	00000000261	01050610043	
62305	00000000250	00000000200	00000000240	00000000200	00000000200	00000000200	00000000261	00000000261	
62315	01050310051	00000000241	00000000261	01050310054	00000000000	00000000261	01050210057	00000000000	
62325	01050210061	01051400054	01050210064	00000000270	01050210067	01050300035	01050210072	00000000275	
62335	01050210075	00000000276	01050210077	00000000255	01050210104	00000000252	01050210107	00000000241	
62345	01050310112	00000000236	00000000200	01050210116	00000000232	01050210120	00000000341	01050210125	
62355	00000000310	01050210132	00000000312	01050210137	00000000334	01050210144	00000000302	01050210147	
62365	00000000000	01050410150	00000000252	00000000200	00000000252	01050410154	00000000000	00000000201	
62375	01051000102	01050310160	01050300023	00000000201	01050310164	00000000000	00000000201	01050210170	
62405	00000000000	01050510172	00000000250	00000000200	00000000247	00000000304	01050410177	00000000250	
62415	00000000200	00000000247	01050410204	00000000206	01050310216	00000000247	00000000200	00000000240	
62425	00000000200	00000000247	00000000000	01050310233	00000000200	00000000313	01050310240	00000000200	
62435	00000000200	01050210231	00000000000	01050310247	00000000200	00000000200	01050210252	00000000000	
62445	00000000314	01050210245	00000000000	00000000273	01050210260	01050200003	01050210262	00000000303	
62455	01050210254	00000000246	01050210256	00000000200	01050210273	00000000313	01050210275	00000000314	
62465	01050210265	00000000304	01050210270	00000000310	00000000310	01050210311	00000000312	01050210316	
62475	01050210277	00000000306	01050210304	00000000040	01050210333	00000000045	01050210340	00000000230	
62505	01050210321	01051000112	01050210326	01050400050	01050210354	00000000343	01050210361	00000000000	
62515	01050210342	00000000342	01050210347	00000000202	00000000203	01050310367	00000000243	00000000202	
62525	01050510363	00000000000	00000000202	01050310375	00000000235	00000000203	01050510377	00000000000	
62535	01050310371	00000000000	00000000202	01050310403	00000000000	00000000203	01050310407	01050200001	
62545	00000000203	01050200001	00000000204	00000000204	01050200003	00000000205	01050410415	00000000000	
62555	00000000204	01050510411	00000000000	01050410421	00000000000	00000000204	00000000205	00000000000	
62565	00000000204	00000000222	01050410432	00000000000	01050310434	01050200003	00000000205	01050310436	
62575	00000000204	00000000222	00000000000	00000000205	00000000205	00000000205	01050400005	00000000205	
62605	00000000205	01050410437	00000000231	00000000205	00000000246	01050510453	00000000240	00000000205	
62615	01050410446	00000000251	00000000205	00000000240	00000000205	00000000247	01050510464	00000000240	
62625	00000000206	01050410457	00000000240	01050510470	00000000205	00000000205	00000000205	01050510475	
62635	00000000206	00000000270	00000000243	00000000202	00000000205	01050510502	00000000000	00000000243	
62645	00000000000	00000000205	00000000000	00000000205	00000000205	01050510513	00000000250	00000000205	
62655	01050300035	01050310507	00000000000	00000000205	00000000247	00000000263	01050510525	00000000247	
62665	00000000303	01050510520	00000000250	00000000265	01050510532	00000000250	00000000205	00000000250	
62675	00000000205	00000000247	00000000250	00000000205	00000000247	01050510544	00000000247	00000000304	
62705	01050510537	00000000250	00000000277	01050510551	00000000250	00000000267	01050510556	00000000205	
62715	00000000247	00000000277	00000000205	00000000247	00000000273	01050510563	00000000250	00000000247	
62725	00000000250	00000000205	00000000250	00000000250	00000000205	00000000247	00000000300	01050510556	
62735	00000000274	01050510570	00000000250	00000000247	00000000205	00000000247	00000000337	01050510575	
62745	00000000205	00000000247	00000000340	00000000205	01050510602	00000000250	00000000205	00000000271	
62755	01050510607	00000000250	00000000205	00000000205	00000000205	01050510614	00000000250	00000000205	
62765	00000000205	00000000301	01050510621	00000000250	00000000205	00000000205	00000000302	01050510626	

627/5	00000000250	00000000205	00000000206	00000000270	01050210633	00000000000	01050410635	00000000000
63005	00000000210	00000000232	01050310642	00000000000	00000000210	01050510646	00000000000	00000000211
63015	00000000234	00000000212	01050310652	00000000000	00000000211	01050310656	00000000234	00000000212
63025	01050510660	00000000000	00000000212	00000000233	00000000213	01050310664	00000000000	00000000212
63035	01050210670	00000000000	01050310672	00000000233	00000000213	01050310674	00000000244	00000000213
63045	01050410701	00000000251	00000000213	00000000246	01050510706	00000000000	00000000213	00000000243
63055	01050300035	01050510713	00000000000	00000000213	00000000243	00000000202	01050210720	00000000000
63065	01050310722	00000000000	00000000215	01050210726	00000000000	01050510730	00000000000	00000000217
63075	00000000241	00000000220	01050610733	00000000245	00000000217	00000000242	00000000220	00000000214
63105	01050410740	00000000251	00000000217	00000000275	01050310745	00000000000	00000000217	01050210751
63115	00000000000	01050610753	00000000245	00000000220	00000000242	00000000220	00000000214	01050410760
63125	00000000251	00000000220	00000000275	01050410765	00000000236	00000000220	00000000236	01050410771
63135	00000000242	00000000220	00000000214	01050410774	00000000245	00000000220	00000000214	01050311001
63145	00000000241	00000000220	01050211003	00000000000	01050411005	00000000200	00000000247	00000000270
63155	01050311012	00000000241	00000000206	01050211015	00000000000	01050411017	00000000240	00000000205
63165	00000000206	01050411023	00000000245	00000000205	00000000206	01050211030	00000000240	01050211033
63175	00000000250	01050211036	00000000000	01050211040	00000000236	01050211043	00000000254	01050211045
63205	00000000000	01050211050	00000000236	01050211051	00000000254	01050211053	00000000000	01050311056
63215	00000000000	00000000276	01050211063	00000000000	30000000000			

AUXILIARY PRODUCTION TABLE

65202	00000000000	00000000000	00000000215	00000000201	00000000205	00000000217	00000000240	30000000000
65212	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000	30000000000
65222	30000000000	30000000000	30000000000	30000000000				

INTERPRETATION TABLE

65227	00010000001	00060000001	00070000004	00130000000	000700000741	00060000000	00060000001	00070000016
65237	00060000001	00070000006	00020000001	00140000000	00070000004	00020000001	00010000002	00060000001
65247	00070000004	00130000001	000700000741	00020000002	00010000003	00070000004	00020000003	00010000003
65257	00060000001	00070000004	00020000001	00060000000	00060000001	000700000032	00130000002	000700000741
65267	00060000000	00060000001	000700000037	00020000005	00010000004	00060000001	000700000045	00020000001
65277	00070000004	00020000002	00060000001	00070000004	00030000002	00030000000	00070000004	00010000003
65307	000700000741	00060000000	00060000001	000700000210	00060000000	00060000001	00070000074	00060000000
65317	00060000001	000700000103	00010000005	00060000001	000700000203	00060000001	000700000747	00020000001
65327	000300000215	000100000071	00060000001	000700000620	00060000000	00060000001	000700000121	00020000001
65337	00060000001	00070000006	00020000002	00130000023	00060000001	00070000006	00060000001	000700000103
65347	00020000001	000300000210	00010000006	00060000001	000700000536	00020000001	000300000201	00010000017
65357	00060000001	000700000125	00020000001	000300000201	000100000020	00060000001	000700000125	00020000001
65367	000300000210	000100000021	00060000001	000700000536	00060000000	00060000001	000700000210	000700000210
65377	00020000003	00010000007	00060000001	00070000006	00020000002	000300000202	00030000000	000700000260
65407	00020000002	000300000210	00030000000	000700000536	00020000002	000300000202	00030000000	000700000260
65417	00130000005	000700000733	00020000004	000300000201	000100000072	00060000001	000700000125	00020000002
65427	000300000205	00030000000	00010000010	000700000414	00020000002	000300000205	00030000000	00010000010
65437	000700000333	00020000003	000300000205	00030000000	00010000011	000700000715	00020000002	000300000206
65447	00030000000	00010000012	00060000001	000700000210	00020000002	000300000201	00030000000	00010000013
65457	000700000125	00130000006	000700000733	00020000002	000300000201	00010000014	00060000001	000700000125
65467	00020000002	000300000201	00010000015	00060000001	000700000125	00130000007	000700000733	00020000001
65477	00060000001	00070000006	00130000004	000700000733	00060000001	000700000103	00060000001	000700000141
65507	00060000001	000700000210	00060000000	00060000001	000700000210	00060000000	00060000001	000700000210
65517	00060000000	00060000001	000700000210	00060000001	000700000173	00060000001	000700000173	00020000001
65527	000300000202	00010000016	00060000001	000700000260	00020000001	000300000202	00010000017	00060000001
65537	000700000260	00020000001	000300000202	000100000020	00060000001	000700000260	00060000000	00060000001
65547	000700000210	00020000001	000300000205	000100000022	00060000001	000700000414	00020000001	000300000201
65557	00010000023	00060000001	000700000125	00020000001	000300000202	00010000024	00060000001	000700000260
65567	00060000001	000700000210	00020000001	000300000205	000100000061	00060000001	000700000333	00020000001

146

65577	00030000202	00010010026	00060000001	00070000260	00020000001	00030000002	00010000102	00060000001
65607	00070000260	00130010022	00070000733	00020000003	00030000000	00010000025	00070000273	00060000001
65617	00070000103	00020010022	00030000203	00030000000	00070000273	00060000001	00070000210	00020000003
65627	00030000000	00010000027	00070000306	00020000002	00030000204	00030000000	00070000306	00060000001
65637	00070000210	00020000023	00030000000	00010000030	00070000333	00020000003	00030000205	00036000000
65647	00070000333	00020000003	00030000205	00030000000	00010000031	00070000333	00020000002	00030000205
65657	00030000000	00070000333	00130000010	00070000733	00060000001	00070000210	00070000414	00020000003
65667	00030000201	00010000032	00060000001	00070000125	00060000001	00070000210	00020000003	00030000202
65677	00010000033	00060000001	00070000260	00020000003	00030000001	00070000210	00070000715	00020000003
65707	00030000001	00010000033	00060000001	00070000210	00020000002	00010000034	00060000001	00070000103
65717	00020000004	00030000210	00030000000	00010000035	00070000536	00020000004	00030000215	00030000000
65727	00010000036	00070000627	00020000004	00030000215	00030000000	00010000037	00070000620	00020000002
65737	00030000210	00030000001	00070000536	00020000004	00030000001	00010000040	00060000001	00070000333
65747	00020000004	00030000215	00010000041	00060000001	00070000620	00020000004	00030000215	00010000042
65757	00060000001	00070000620	00020000004	00030000215	00010000043	00060000001	00070000260	00020000004
65767	00030000215	00010000044	00060000001	00070000620	00020000004	00030000215	00010000045	00060000001
65777	00070000620	00020000004	00030000215	00010000046	00060000001	00070000620	00020000004	00030000001
66007	00010000047	00060000001	00070000620	00020000004	00030000215	00010000047	00060000001	00070000620
66017	00020000004	00030000215	00010000048	00060000001	00070000620	00020000004	00030000215	00010000049
66027	00060000001	00070000620	00020000004	00030000215	00010000050	00060000001	00070000620	00020000004
66037	00030000215	00010000051	00060000001	00070000620	00020000004	00030000215	00010000052	00060000001
66047	00070000620	00020000004	00030000215	00010000053	00060000001	00070000620	00020000004	00030000001
66057	00010000053	00060000001	00070000620	00020000004	00030000215	00010000054	00060000001	00070000620
66067	00010000054	00070000620	00020000004	00030000215	00010000055	00060000001	00070000620	00020000004
66077	00010000055	00070000620	00020000004	00030000215	00010000056	00060000001	00070000620	00020000004
66107	00020000003	00030000000	00070000620	00020000004	00030000215	00010000057	00060000001	00070000620
66117	00130000012	00070000733	00060000001	00070000103	00020000002	00030000214	00010000057	00060000001
66127	00070000060	00020000003	00030000210	00010000060	00060000001	00070000536	00020000004	00030000215
66137	00030000000	00010000037	00070000620	00020000002	00030000217	00030000000	00010000062	00070000620
66147	00130000013	00070000733	00020000002	00030000217	00030000000	00070000625	00130000014	00070000733
66157	00020000003	00030000000	00070000651	00020000005	00030000001	00010000065	00060000001	00070000625
66167	00020000003	00030000001	00010000064	00060000001	00070000625	00020000002	00030000220	00030000000
66177	00070000651	00130000015	00070000733	00020000005	00030000217	00010000065	00060000001	00070000625
66207	00020000003	00030000217	00010000064	00060000001	00070000625	00020000003	00010000073	00060000001
66217	00070000004	00010000067	00060000001	00070000060	00020000003	00030000217	00010000063	00060000001
66227	00070000625	00060000001	00070000060	00130000017	00070000733	00020000002	00030000206	00010000077
66237	00060000001	00070000710	00020000001	00060000001	00070000103	00130000020	00070000733	00020000001
66247	00010000074	00060000001	000700006715	00020000003	00030000201	00010000075	00060000001	00070000675
66257	00020000001	00060000001	00070000725	00020000001	00060000001	00070000125	00130000021	00070000733
66267	00020000001	00060000001	00070000004	00140000000	00070000733	00020000001	00060000001	00070000733
66277	00070000004	00140000000	00070000733	00020000001	00060000001	00070000741	00020000002	00030000215
66307	00010000076	00060000001	00070000620	00130000024	00070000733			

RECORD TYPE	NUMBER OF RECORDS	STARTING RECORD
SYMBOL TABLE	1	8
HIERARCHY TABLE	2	9
PRODUCTION TABLE	1	13
METACHARACTER LISTS	2	14

TIME USED: 00:05:02 PAGES USED: 11 19108:18

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.

OPERATOR-040 14 MAY 64 19:10:11 MAGIC PAGES: 50 TIME: 1

CC35.013 J. FELDMAN
COMPUTER CENTER

00:00:04⁽¹⁾

CO
SN
EXAMPLE 1
DUMP 1
BEGIN REAL X,Y,ZEBRA; BOOLEAN P,Q; LABEL L1,L2;
L1: Y← X*Y-ZEBRA*(Y/2) ;
IF IF P THEN Q ELSE X=3 THEN BEGIN X← X-1; GOTO L1 END
ELSE GOTO L2;
X← X+1; GOTO L1;
L2: Q← TRUE
END
20000 0 CLA 2 40000,00 0 MPY 2 40002,00 0 STD 0 72400,77 0 CLA 2 40002,00 0 DIV 0 00002,00
20005 0 STD 0 72652,00 0 CLA 2 40004,00 0 STD 0 72650,00 0 TRM 0 10253,00⁽³⁾ 0 SUN 2 72400,77
20012 0 STD 0 40002,00 0 CLA 2 40000,00 0 FUD 0 00003,00 0 OCA 0 00001,00 0 CCL 2 14376,00
20017 0 IOZ 2 40006,00 0 TRA 0 20024,00 0 CAL 2 40007,00 0 STL 0 72400,77 0 TRA 0 20025,00
20024 0 STL 0 72400,77⁽⁴⁾ 0 ICZ 2 72400,77 0 TRA 0 20030,00 0 TRA 0 20035,00 0 CLA 2 40000,00
20031 0 SUB 0 00001,00 0 STD 0 40000,00 0 TRA 0 20000,00 0 TRA 0 20036,00 0 TRA 0 20042,00
20036 0 CLA 2 40000,00 0 ADD 0 00001,00⁽⁴⁾ 0 STD 0 40000,00 0 TRA 0 20000,00 0 CAL 2 14377,00
20043 0 STL 0 40007,00 0 TRM 0 05300,00⁽⁴⁾

TIME USED: 00:00:07⁽²⁾ PAGES USED: 1 19:10:17

1. The compiler took four seconds to read in the tables of the small language.
2. Three seconds were used to compile and dump the program.
3. ↑ is executed by a library subroutine.
4. Location 5300 is the monitor HALT routine.
5. Temporaries start at 72400; the index register allows recursive use of temporaries.

OPERATOR-040 14 MAY 64 19:18:37 MAGIC PAGES: 20 TIME: 3

CC35.013 J. FELDMAN
COMPUTER CENTER

00100104

CO
SN ① CODE 1
SN ② EXEC 1
SN DUMP

EXAMPLE 2

BEGIN REAL X,Y,ZEBRA; BOOLEAN P,Q; LABEL L1,L2;

00061	00000000001								
63334	00000000252	00000000000	00000000000	00000000000	00000000000	0000000252	00000000213	00000000213	
00061	00000000004								
63334	00000000234	00000000262	00000000246	00000000000	00000000000	0000000234	00000000262	00000000246	
00061	00000000004								
63334	00000000234	00000000263	00000000246	00000000000	00000000000	0000000234	00000000263	00000000246	
00061	00000000004								
63334	00000000235	00000000264	00000000246	00000000000	00000000000	0000000235	00000000264	00000000246	
00061	00000000005								
63334	00000000234	00000000265	00000000250	00000000000	00000000000	0000000234	00000000265	00000000250	
00061	00000000005								
63334	00000000235	00000000266	00000000250	00000000000	00000000000	0000000235	00000000266	00000000250	
00061	00000000036								
63334	00000000234	00000000267	00000000251	00000000000	00000000000	0000000234	00000000267	00000000251	
00061	00000000036								
63334	00000000235	00000000270	00000000251	00000000000	00000000000	0000000235	00000000270	00000000251	
L1: IF (X*Y)/(X+Y) = ZEBRA THEN									
00061	00000000002								
63334	00000000267	00000000246	00000000252	00000000000	00000000000	0000000267	00000000235	00000000252	
00061	00000000032								
63334	00000000236	00000000267	00000000252	00000000000	00000000000	0000000235	00000000252	00000000213	
00061	00000000011								
63334	00000000262	00000000267	00000000252	00000000000	00000000000	0000000262	00000000242	00000000255	
00061	00000000011								
63334	00000000263	01030040000	00000000252	00000000000	00000000000	0000000263	00000000216	01030040000	
00061	00000000012								
63334	00000000245	01030040002	00000000216	01030040000	00000000000	0000000245	01030040000	00000000242	
20000	04050040000								
20001	04770040002								
00061	00000000017								
63334	00000000245	00030400000	00000000242	01030040000	00000000000	0000000242	00000000255	00000000235	
00061	00000000011								
63334	00000000262	00030400000	00000000242	01030040000	00000000000	0000000262	00000000242	00000000217	
00061	00000000011								
63334	00000000263	01030040000	00000000242	01030040000	00000000000	0000000263	00000000214	01030040000	
00061	00000000014								
63334	00000000245	01030040002	00000000214	01030040000	00000000000	0000000245	01030040000	00000000242	
20002	0153772400								
20003	04050040000								
20004	04450040002								
00061	00000000017								
63334	00000000245	00030400000	00000000242	01030040000	00000000000	0000000242	01032072400	00030400000	

```

00061 00000000013
63334 00000000220 00030400000 01032072400 00030400000 00000000000 00000000220 00030400000 00000000255
20005 00570040002
00061 00000000011
63334 00000000264 00030400000 01032072400 00030400000 00000000000 00000000264 00000000220 00030400000
00061 00000000021
63334 00000000256 01030040004 00000000220 00030400000 00000000000 00000000256 00030400000 00000000255
20005 05610040004
20007 00000000001
20010 04150014376
20010 04350014376
00061 00000000025
63334 00000000256 00000400000 00000000255 00030400000 00000000000 00000000255 00000000235 00000000252
00061 00000000003
63334 00000400000 00000000255 00030400000 00000000000 00000400000 00000000235 00000000252
20007 00170000000
20010 00170000000
      BEGIN X- -X+ZEBRA) GOTO L2 END ELSE P- FALSE)
00061 00000000002
63334 00000000262 00000000252 00000000255 00030400000 00000000000 00000000262 00000000235 00000000252
00061 00000000011
63334 00000000262 00000000252 00000000255 00030400000 00000000000 00000000262 00000000215 00000000237
00061 00000000016
63334 00000000214 01030040000 00000000215 00030400000 00000000000 00000000214 00000000215 00000000237
00061 00000000011
63334 00000000264 01030040000 00000000215 00030400000 00000000000 00000000264 00000000214 01034040000
00061 00000000014
63334 ③ 00000000235 01030040004 00000000214 01034040000 00000000000 00000000235 01034040000 00000000237
20011 04250040000
20012 04450040004
00061 00000000020
63334 00000000235 00030400000 00000000237 00000000262 00000000000 00000000235 00000000262 00000000235
20013 01530040000
00061 00000000031
63334 00000000270 00000000254 00000000237 00000000262 00000000000 00000000254 00000000235 00000000252
20014 00170000000
00061 00000000035
63334 00000000253 00000000254 00000000235 00000000252 00000000000 00000000252 00000400000 00000000235
00061 00000000006
63334 00000000257 00000000252 00000400000 00000000252 00000000000 00000000257 00000000252 00000400000
20015 00170000000
00061 00000000030
63334 00000000235 01000214376 00000000237 00000000265 00000000000 00000000235 00000000265 00000000257
20016 04150014376
20017 01730040006
00061 00000000033
63334 00000000235 00000000265 00000000257 00000000252 00000400000 00000000235 00000000265 00000000235
      L2: IF P THEN X- Y+ZEBRA) Q- X<Y
00061 00000000032
63334 00000000236 00000000270 00000000257 00000000252 00000400000 00000000235 00000000252 00000000213
00061 00000000024
63334 00000000256 00000000265 00000000257 00000000252 00000400000 00000000256 00000000265 00000000255
00061 00000000025
63334 00000000256 01000040006 00000000255 00000000252 00000400000 00000000255 00000000235 00000000252
00061 00000000003

```

151.

```

63334 01000040006 01000040006 00000000255 00000000252 00000400000 01000040006 00000000235 00000000252
20020 04310040006
20021 00170000000
20022 00170000000
00061 00000000010
63334 00000000231 00000000263 00000000255 00000000252 00000400000 00000000231 00000000263 00000000237
00061 00000000011
63334 00000000264 01030040002 00000000255 00000000252 00000400000 00000000264 00000000231 01030040002
00061 00000000034
63334 00000000235 01030040004 00000000231 01030040002 00000400000 00000000235 01030040002 00000000237
20023 04050040004
20024 01530072652
20025 04050040002
20026 01530072650
20027 01770010253
00061 00000000020
63334 00000000235 00030400000 00000000237 00000000262 00000400000 00000000235 00000000262 01000040006
20030 01530040000
00061 00000000007
63334 00000000235 00000000262 01000040006 00000000262 00000400000 00000000235 01000040006 00000000235
00061 00000000010
63334 00000000224 00000000262 01000040006 00000000262 00000400000 00000000224 00000000262 00000000237
00061 00000000011
63334 00000000263 00000000262 01000040006 00000000262 00000400000 00000000263 00000000224 01030040000
END
00061 00000000022
63334 00000000253 01030040002 00000000224 01030040000 00000400000 00000000253 01030040000 00000000237
20031 04050040000
20032 05210040002
20033 00000000001
20034 04150014376
00061 00000000027
63334 00000000253 00000400000 00000000237 00000000266 00000400000 00000000253 00000000266 00000000235
20035 01730040007
00061 00000000035
63334 00000000253 00000000266 00000000235 00000000252 00000400000 00000000252 00000000213 00000000213
00061 00000000037
63334 00000000252 00000000213 00000000235 00000000252 00000400000 00000000213 00000000213 00000000213
20035 01770005300
20000 0 CLA 2 40000,00 0 MPY 2 40002,00 0 STD 0 72400,77 0 CLA 2 40000,00 0 ADD 2 40002,00
20003 0 RDV 0 40002,00 0 FUD 2 40004,00 0 TRA 0 20011,00 0 TRA 0 20016,00 0 CLS 2 40000,00
20012 0 ADD 2 40004,00 0 STD 0 40000,00 0 TRA 0 20020,00 0 TRA 0 20020,00 0 CAL 2 14376,00
20017 0 STL 0 40006,00 0 ICZ 2 40006,00 0 TRA 0 20023,00 0 TRA 0 20031,00 0 CLA 2 40004,00
20024 0 STD 0 72652,00 0 CLA 2 40002,00 0 STD 0 72650,00 0 TRM 0 10253,00 0 STD 0 40000,00
20031 0 CLA 2 40000,00 0 FLO 2 40002,00 0 OCA 0 00001,00 0 CAL 2 14376,00 0 STL 0 40007,00
20035 0 TRM 0 05300,00

```

TIME USED: 00:00:14 PAGES USED: 3 19:18:50

1. This causes the code to be printed as it is compiled.
2. Causes the routine number and LEFT1-LEFT5 and RIGHT1-RIGHT3 to be printed.
3. EXEC 12 ↓ is called to add -X and ZEBRA, it knows to subtract X because of (4)
4. The negation is bit set in LEFT4.

 OPERATOR-040 14 MAY 64 19:11:05 MAGIC PAGES: 50 TIME: 1

CC35 013

00:00:04

CO
 SN DUMP EXAMPLE 3

BEGIN
 BEGIN REAL X,Y, ZEDD; BOOLEAN P,Q,ZEDE;
 ZEDE = IF X/Y+ZEDD-X = X THEN P ELSE Q;
 X = X*X + X*X END

BEGIN REAL X, Y, Z; BOOLEAN P,Q;

X = Y + X*X*Z;
 X = 1+1; Y = 1.5; Z = X/3;

① FAULT
 FAULT

1
 1
 P = TRUE; Q = FALSE;
 P = Q END

20000	0 CLA 2 40000,00	0 DIV 2 40002,00	0 ADD 2 40004,00	0 SUB 2 40006,00	0 CLA 2 40008,00
20005	0 TRA 0 20011,00	0 CAL 2 40006,00	0 STL 0 72400,77	0 TRA 0 20013,00	0 CAL 2 40010,00
20012	0 STL 0 72400,77	0 CAL 2 72400,77	0 STL 0 40010,00	0 CLA 2 40010,00	0 MPY 2 40000,00
20017	0 STD 0 72401,77	0 CLA 2 40000,00	0 MPY 2 40000,00	0 ADD 2 72411,77	0 STD 0 40000,00
20024	0 CLA 2 40011,00	0 MPY 2 40011,00	0 MPY 2 40015,00	0 ADD 2 40013,00	0 STD 0 40011,00
20031	② 0 CAL 0 00271,00	0 STD 0 72652,00	② 0 CAL 0 00271,00	0 STD 0 72650,00	0 TRA 0 40011,00
20036	0 STD 0 40011,00	0 CLA 2 14401,00	0 STD 0 40013,00	0 CLA 2 40011,00	0 DIV 0 00000,00
20043	0 STD 0 40015,00	0 CAL 2 14377,00	③ 0 STL 0 40017,00	0 CAL 2 14376,00	④ 0 ST 0 40017,00
20050	0 CAL 0 40020,00	0 STL 0 40017,00	0 TRM 0 05300,00		

TIME USED: 00:00:08 PAGES USED: 1 19:11:12

1. The semantic error message FAULT 1 is printed twice because 'I' was not declared.
2. Since OK was not set FALSE, translation continues using the internal name of 'I' as 'I'.
3. The floating point constant '1.5' is put in a cell in the constant region.
4. TRUE and FALSE are also absolute constants.