A SYNTAX-DIRECTED TRANSLATOR FOR TEACHING PURPOSES

***

A Thesis
Presented to
The Faculty of Graduate Studies and Research
The University of Manitoba

***

In Partial Fulfilment
of the Requirements for the Degree
Master of Science
in the Department of Computer Science

***

by
Kenneth Carpenter

August, 1971

A C K N O W L E D G E M E N T S

The author wishes to thank Dr. S. Clark for his proposal and assistance in the preparation of this thesis.

Thanks are also due to Dr. P. R. King for his kind and thoughtful guidance during the later stages of writing and editing.

Finally, the author would like to acknowledge the educational groundwork in this domain and general inspiration provided by James Wells.

A B S T R A C T

This thesis presents a generalized compiler with the view to aid students of Computer Science in creation and manipulation of formal programming languages.

The compiler developed will accept the broad base of context free simple formal languages and can be used to generate a similar class of object language.

The compiler hopefully results in a useful educational tool providing a practical facility with which to describe linguistic concepts, an approach to generalized compiling, and the compiler function.

TABLE OF CONTENTS

LIST OF FIGURES

C H A P T E R I

INTRODUCTION

1.1 INTRODUCTION

Evidence in the published literature indicates a proliferation of types of computers and of programming languages. And, in general, a compiler must be written for translation between each source language-object code combination.

There is a trend towards standardization. ALGOL, FORTRAN in the scientific field, COBOL in the commercial field, JOVIAL in the military, and recently PL/1 in all these areas are becoming common (if not standard) languages. However, as the trend continues, the need for special purpose languages (symbol manipulation, information storage and retrieval, simulation, concept processing, machine control, etc.), and the desire for extensions of present day languages in order to adapt to the problems of the future, will perpetuate a source language-object code proliferation.

The consequent need for a large number of compilers has caused computer scientists to research how cheaply, how quickly, and how well any compiler may be created. It should be noted that compiler writing in the past has generally proven to be an expensive and time-consuming occupation. From these efforts has developed a formal theory on the description of programming languages and their implementation.

Appendix F contains a short glossary of words for the reader who is not familiar with the thesis subject.

1.2 FORMAL LANGUAGES AND FORMAL GRAMMARS

The syntactic rules for many programming languages may be specified by formal grammars, generally variants of the phrase-structure grammars developed by the linguist Noam Chomsky [4] during the 1950's as a tool in the study of natural languages. The most representative and fruitful example of this is the use of Backus Normal Form (BNF) to define most of the syntactic rules of ALGOL 60 [3].

A complete set of grammatical (syntactic) rules for a language written in a form equivalent to BNF is a "formal grammar". The language in which a formal grammar is written is a meta-language (a language to describe languages). A language definable by a formal grammar is a "formal language".

1.3 SYNTAX-DIRECTED COMPILERS

The syntactic analysis essential to translation of programming languages can be done entirely mechanically for formal languages. The compiler philosophy on which this is based is generally call "syntax-directed".

All compilers are syntax-directed in the sense that they must obviously recognize the various syntactic structures of a language and the output produced is a function of the syntax. Syntax-directed refers to a compiler which contains a direct encoding of the syntax of the language, this encoding being used by the compiler as data. Hence the alias "parameterized compiler", and since the data is often in tabular form, these compilers are then described as "table-driven". With the other compilers (the ad hoc class), based on non-formal direct methods, the syntactic definition of the source language is essentially buried in the coding of the compiler.

Compilers built upon syntax-directed concepts are very general. Additions to and deletions from the language can be made, and completely new languages introduced with relative ease, the only restriction being that the language must be formal to allow a direct encoding of its syntax to be made corresponding to its formal definition. In fact, this encoding may usually be generated mechanically. A variety of this type of compiler now exists [6, 7, 8, 9, 14, 16].

Moreover, by considering languages definable by a more restrictive class of definition, syntax-directed compilers can realize significant economies in syntax recognition (parsing). Such techniques as precedence and LR(k) yield efficient parsers [13, 14, 15, 17, 18, 19].

1.4 PURPOSE OF THIS THESIS

This thesis presents an implementation and refinement of an existing syntax-directed translator created by Reeves at the University of Leeds, England [1]. It is an attempt to provide students of Computer Science with a facility for practical experimentation in the definition and manipulation of formal languages.

1.4.1 IMPLEMENTATION GOALS

(1) Relative Machine Independence:

The syntax-directed compiler under discussion has been written in the universally accepted algebraic language ALGOL 60. Since the input/output routines in ALGOL have only been recently defined [5] and are, in fact, locally defined at many computer centres, a complete ALGOL input/output package was added to this program and it may now be implemented using any ALGOL translator having the primitives defined by the IFIP Working Group 2.1 [5].

(2) Ease of Use:

The student should not encode by hand the internal representation of the syntax of a language and its associated semantics. He can, in fact, define the formal grammar in an extended version of BNF and the compiler will convert the formal defini-

tion into the proper encoding. This formal definition encompasses semantics as well as syntax.

The definitions should be concise and easily learned.

(3) Flexibility:

The system should be workable for a large range of formal languages. Facilities should exist for the control of input/output devices. For instance, initialization data should be available from disk as well as cards.

1.4.2 EXTENSIONS

An investigation is made into the possibility of useful extensions to the system, such as increasing the scope of the input and output languages.

1.5 REEVES COMPILING MACHINE

The overall organization of Reeves' syntax-directed compiler is based on a description by Metcalfe [2]. It consists of a KDF9 ALGOL program that simulates a special-purpose stored program digital computer. A program of instructions for this machine represents the syntax and semantics of some language (i.e. its program is the internal encoding referred to above). The machine, under control of its program, reads in a text written in the source

language, determines if it is grammatically correct, and then outputs a text written in the target language. This output may be either object code or some rearrangement and modification of this input. The machine may thus act as a translator or as an editor.

The compiling machine need not be directly programmed in its assembly language. Reeves uses a boot strap technique which reads the syntactic and semantic definition of a language in a simple formalized form (a meta-language) and compiles this into the required machine program. The machine will then use this program to handle the defined language.

1.6 MODIFICATIONS TO REEVES COMPILING MACHINE

1.6.1 IMPLEMENTATION MODIFICATIONS

Reeves compiling machine was designed for a paper tape environment (KDF9) using an upper and lower case alphabet, special characters and underlining facilities. Also it was written in KDF9 ALGOL which has a locally defined set of input/output procedures [10].

The following changes were required:

(1) Reeves meta-language for formal language description was based on underlined lower case symbols and special characters which are not available on card oriented systems such as the IBM

360/65 used for this implementation. It and the corresponding software were adapted to the relatively reduced card character set.

(2) A different internal character set coding was designed and routines modified accordingly. See Appendix C.

(3) A different set of input/output procedures were used and data transfers suitably altered. These I/O routines will work with any ALGOL compiler using the procedures described in the ACM [5].

1.6.2 EXTENSIONS

(1) Format Control (spacing, tab positioning and carriage control) was added to the meta-language to facilitate output readability.

(2) Modifications to the semantic specification facility in the meta-language were made to ease usage.

(3) A method for allowing the output of object code for Princeton-type machines was designed but not implemented. However, the approach is given.

(4) An indication of the efficiency of a particular language definition was added as an option.

CHAPTER II

DESCRIPTION OF THE SYNTAX-DIRECTED COMPILER

2.1 INTRODUCTION

The purpose of this chapter is to give a detailed description of the revised compiler. This will be done by looking separately at the two major areas of the compiler function, syntax-analysis and semantics.

2.2 SYNTAX ANALYSIS

If the structure (syntax) of a language may be described by a set of syntactic rules, these rules may be fed into the compiler in a standard form (meta-syntactic language) and the compiler will produce an internal program or encoding of the rules. Using this internal program, the compiler reads in source text of the language described and attempts to parse the text.

2.2.1 META-SYNTACTIC LANGUAGE

In order for the student to unambiguously state a set of syntactic rules, we require a rigorous meta-syntactic language.

For this purpose an extended version of Backus Normal Form has been chosen (see Appendix E).

Composing the meta-syntactic language is a set of meta-symbols {<, >, ::=, |, ", (,), ⌐, *, 'U', ;, 'BEGIN', 'END'}. The structure of a language may then be described by a sequence of syntax rules separated by semicolons and enclosed between 'BEGIN' and 'END'. Each rule has the form <syntactic variable>::=<expression>. 'Syntactic variables' (often referred to as non-basic or non-terminal types in the literature) are denoted by enclosing the name of the variable in meta-brackets '<>'. 'Syntactic constants' (basic or terminal types) are denoted by enclosing them in meta-quotes '"'. An 'expression' is composed of a series of 'alternatives' separated by '|', which is read as 'or'. An 'alternative', which defines a basic phrase or structure in the language, is composed of zero or many constructions which are called 'elements'. The juxtaposition of the elements defines the actual ordering of their physical appearance in the language. An element is composed of a 'syntactic variable' or a 'constant'.

Thus far the meta-syntactic language is equivalent to B.N.F. The three differences being that, first, the whole set of definitions must be enclosed in 'BEGIN' 'END' brackets; second, definitions are separated from each other by semicolons;

and third, syntactic constants are placed in quotes. Now, the extensions are considered.

(1) Left-Recursive Definitions

In the ALGOL report, many structures are defined partially in the terms of themselves e.g.

<unsigned integer>::=<digit>|<unsigned integer><digit>.

An unsigned integer is defined to be a digit or whatever was previously defined as an unsigned integer followed by a digit.

Left-recursive definitions (the name of the variable being defined appears as the first element in one or more of the alternatives) of this type are not allowed in this meta-syntactic language. Instead, there is a meta-symbol '*' which may precede an element and is read as 'a sequence of zero or more' e.g.

<unsigned integer>::=<digit>*<digit>

is read 'an unsigned integer is defined to be a digit followed by a sequence of zero or more digits'.

(2) Structural Element

Whenever, within a definition, several alternatives start with the same element(s), replace all the alternatives by a single alternative, whose first component is the common element(s) and whose last component is the non-common elements

separated by '|' and enclosed in meta-brackets '()', e.g.

<a>::=<b><c><d><e>|<b><c>|<b><c><f>|<g>

would be transformed into

<a>::=<b><c>(<d><e>|<f>|)|<g>.

The expression enclosed in '()' is called a structural element. (Note that since the compiler will consider alternatives in the order in which they appear, the null alternative is placed last in the structural element.)

As an example of handling both a left-recursive definition and a structural element,

<a>::=<b><c>|<a><d><e>|<a><f>|<g><h>

would be transformed into

<a>::=(<b><c>|<g><h>)*(<d><e>|<f>).

(3) Any Symbol Other Than

The meta-symbol '¬' preceding an element denotes 'any symbol other than', e.g.

¬"3" is read 'any symbol other than the constant 3'.

¬("ab"|"ac") is read as 'any symbol other than an 'a' followed by a 'b' or an 'a' followed by a 'c'. Thus the acceptability of 'a' depends on its context.

(4) Any Symbol

The meta-symbol ' 'U' ' denotes 'any symbol'.

We can now describe the syntax of our meta-syntactic language, using the meta-syntactic language.

```
'BEGIN'       <grammar>;
<grammar>::="'BEGIN'"<subject>";"<definition>*(";"<definition>)"'END'";
<definition>::=<variable>"::="<expression>;
<variable>::="<"*←">""">";
<expression>::=<alternative>*("|"<alternative>);
<alternative>::=*<element >;
<element>::=<variable>|<constant>|"*"<element>|"("<expression>")"
            | "'U'"|"←"<element>;
<constant>::="""'U'*←""" """"
'END'
```

Note that the title or subject of the grammar is listed following the first "'BEGIN'". This is done for ease of code generation as will be seen later. This "subject" must be the syntactic variable name of the root definition; that is the definition whose variable name does not appear in the expressions of any definitions.

This syntactic description does not include such semantic considerations as the restriction on left recursion as discussed on page 11.

2.2.2 THE PARSING MACHINE

The ALGOL implementation simulates a special purpose stored program digital computer, the compiler machine, a program for which defines the syntax and the semantics of some source language. That part of the compiler machine dealing only with syntax analysis will be referred to as the parsing machine. It determines the syntactic structures that comprise a source text (i.e. parsing) and determines whether this text is grammatically correct. This section will describe the parsing machine and the set of instructions associated with it.

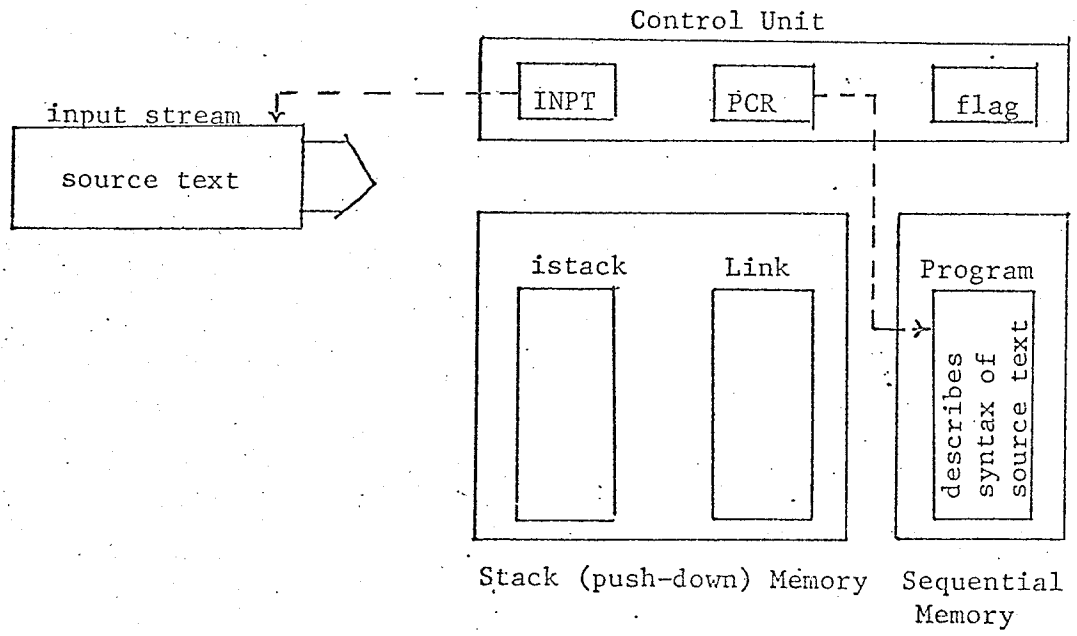


FIG. 1 THE PARSING MACHINE

The parsing machine is a sequential computer similar to a Turing machine. It consists of the following elements:

(1) An input stream of symbols forming the source text.

(2) A set of operations which perform the functions necessary to parsing, and a set of instructions which invoke these operations.

(3) A Boolean register "flag" whose values true or false indicate the success or failure of different stages in the parsing process.

(4) A register "inpt" (input pointer) which points to the symbol currently being scanned in the input stream.

(5) A register "pcr" (parser control register) which points to the next instruction in the program to be executed.

(6) An instruction memory to contain the "program" of instructions to be executed in parsing the input source text.

(7) Two stack memories to contain "istack" and "link". These are push-down lists that save values of "inpt" and "pcr" respectively at various stages in the parsing process.

Initially "flag" is set to true, "inpt" points to the first symbol in the source text, and "pcr" points to the first instruction in the program. "istack" and "link" commence as empty stacks.

The machine has the common fetch-increment-execute control cycle. The instruction specified by "pcr" is fetched, "pcr" is incremented to point to the successive instruction, and then the instruction is executed. This cycle continues until terminated by a STOP instruction. The instructions are normally executed sequentially, but this order can be broken by branch instructions.

The program which the machine obeys is a sequence of instructions defining the parsing algorithm for the source language. The instructions are of single-address type, having the form (label, operation code, operand). The label names the location of an instruction. An operation code identifies the operation to be performed. And an operand is the label, symbol of the defined language or null character which the operation requires.

The parsing instruction set follows:

CALL Saves current value of "inpt" in "istack".
Saves, if the operand is null, then null else "pcr", the location of the next instruction in the program sequence, in "link".
Sets "flag" true.
If the operand is not null, transfers control to address specified by the operand.

MATCH Matches the source symbol at the current point of scan with the symbol in the operand. If they are the same, "flag" is set true and "inpt" is incremented. Otherwise, "flag" is set false.

NEXT Sets "flag" true and increments "inpt" to the next source symbol.

TRUE If "flag" is true the program control is transferred to the address specified by the operand. Otherwise, "flag" is set true and "inpt" is restored from the value in the top of "istack".

FALSE If "flag" is false then program control is transferred to the address specified by the operand.

FLAG If "flag" is true then program control is transferred to the address specified by the operand. Otherwise, "flag" is set true.

NOT The setting of "flag" is reversed.
"inpt" is restored to the value in the top of "istack".
If "flag" is now true, "inpt" is incremented to the next source symbol.
The top value in each stack is deleted.

RETURN If "flag" is false then "inpt" is restored to the value in the top of "istack".
If the operand is not null, then control is transferred to its address.
The top value from each stack is deleted.

STOP Stops the machine.

If "flag" is false, displays the message "faulty syntax".

The purpose of these nine instructions will be explained in the next section. Briefly, however, CALL and RETURN are subroutine call and link instructions which use the stack to administer the return addresses. TRUE, FALSE and FLAG are conditional branching instructions, a branch occurring by replacing "pcr" by the label in the operand field of the jump instruction. MATCH, NEXT and NOT are input control instructions. STOP is an output instruction.

2.2.3 PROGRAMMING THE PARSING MACHINE

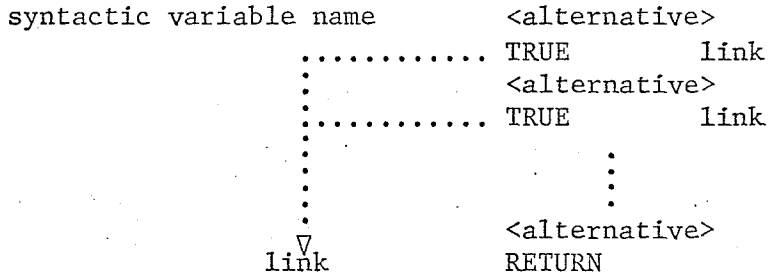
This section explains how, given the grammar (written in the meta-syntactic language) defining the structure of a language, the program for the parsing machine may be written. The general method will be first shown, followed by two examples.

2.2.3.1 THE GENERAL APPROACH

In order to write the program that will parse a given source text, we first examine the language's grammatical defin-

ition. Each syntactic definition in the meta-language is represented by a subroutine in the computer program. The head of the subroutine is labelled by the syntactic variable name and the instructions comprising the subroutine correspond to the expression part of the definition. These instructions are portioned into sequences corresponding to each alternative. The function of each such sequence is to set the flag corresponding to the acceptability of that alternative. Alternative structure sequences are separated by a TRUE instruction which causes a conditional jump to the end of the subroutine. Thus as soon as an acceptable alternative is found, the others are ignored. Leaving a subroutine causes a return to the calling instruction that invoked the subroutine.

That is, since a <grammar> is a sequence of <definition>s, and <definition>::=<variable>"::="<expression> where <expression> ::=<alternative>*<alternative>, the parsing machine program corresponding to each <definition> has the form:



PROGRAM CORRESPONDING TO A DEFINITION

For each alternative, <alternative>::=*<elements>, the program examines each element in turn and sets the flag true or false depending on that element's acceptability. Between consecutive elements is a FALSE instruction causing a conditional jump to the end of the alternative. Thus the elements are examined one by one, and any unacceptable one causes the flag to be set false and the rest are skipped.

```
        <element>
        FALSE       next alt.
        <element>
        FALSE       next alt.
             :
             :
             :
        <element>
next alt.
```

PROGRAM CORRESPONDING TO AN ALTERNATIVE

Most elements require some sort of matching test and are called <active> but others do not. The latter are called <passive> and are always acceptable. They include "*<element>" and "'U'", and are not followed by a TRUE instruction.

The instruction sequences for each type of element are listed below.

	<ELEMENT>	CORRESPONDING PROGRAM		NOTE
1.	<variable>	CALL	variable name	the operand is the name of the head of the relevant subroutine

	<ELEMENT>	CORRESPONDING PROGRAM		NOTE
2.	<constant>	MATCH	constant	the operand is the constant under test
3.	*<element>	L:<element> FLAG L		
4.	(<expression>)	CALL <expression> RETURN		the operand is null
5.	'U'	NEXT		
6.	⌐<element>	CALL <element> NOT		

Finally, the program is terminated by placing a STOP instruction at the end of the principle subroutine.

2.2.3.2 EXAMPLES OF PARSING MACHINE PROGRAMS

EXAMPLE 1

Consider the following grammar that defines the structure of a sentence:

```
'BEGIN'    <sentence> ;
<sentence>::=<subject><predicate>;
<subject>::=<noun phrase>;
<noun phrase>::=<article><noun>;
<predicate>::=<verb><object>;
<object>::=<noun phrase>;
<article>::="THE"|"A";
<noun>::="BOY"|"TREE";
<verb>::="SEES"
'END'
```

The program that will parse a sentence defined by this grammar would be:

```
STMT #      LABEL              OPERATION      OPERAND

1           main routine         CALL         sentence
2                                STOP

3           sentence             CALL         subject
4                                FALSE        L1
5                                CALL         predicate
6           L1                   RETURN

7           subject              CALL         noun phrase
8                                RETURN

9           noun phrase          CALL         article
10                               FALSE        L2
11                               CALL         noun
12          L2                   RETURN

13          predicate            CALL         verb
14                               FALSE        L3
15                               CALL         object
16          L3                   RETURN

17          object               CALL         noun phrase
18                               RETURN

19          article              MATCH        T
20                               FALSE        L4
21                               MATCH        H
22                               FALSE        L4
23                               MATCH        E
24          L4                   TRUE         L5
25                               MATCH        A
26          L5                   RETURN

27          noun                 MATCH        B
28                               FALSE        L6
29                               MATCH        O
30                               FALSE        L6
31                               MATCH        Y
32          L6                   TRUE         L7
```

```
STMT #          LABEL              OPERATION      OPERAND

33                                 MATCH          T
34                                 FALSE          L7
35                                 MATCH          R
36                                 FALSE          L7
37                                 MATCH          E
38                                 FALSE          L7
39                                 MATCH          E
40              L7                 RETURN

41              verb               MATCH          S
42                                 FALSE          L8
43                                 MATCH          E
44                                 FALSE          L8
45                                 MATCH          E
46                                 FALSE          L8
47                                 MATCH          S
48              L8                 RETURN
```

If required to analyze the structure of source text "The boy sees a tree", the parser machine would begin at the first instruction in the main routine and perform a hierarchical series of subroutine calls and returns. All possibilities would be examined until either the sentence could not be matched with the ultimate constants, or it was completely acceptable. The latter results in a well-defined sentence, the former in "faulty syntax".

A left-to-right scan of the source text would be made. By use of the stacks "istack" and "link", subroutines called could remember the current point of scans in the parse at all times and correct linkage could be effected. For a detailed explanation of the execution of the parser machine in parsing the above source text, see Appendix G.

EXAMPLE 2

Let us consider the syntax of the meta-syntactic language itself. The corresponding parsing program to analyze its syntax would be:

```
STMT #      LABEL              OPERATION      OPERAND

1                              CALL           grammar
2                              STOP

3           grammar            MATCH          'B'
4                              FALSE          A1
5                              MATCH          'E'
6                              FALSE          A1
7                              MATCH          'G'
8                              FALSE          A1
9                              MATCH          'I'
10                             FALSE          A1
11                             MATCH          'N'
12                             FALSE          A1
13                             CALL           definition
14                             FALSE          A1
15          A2                 CALL
16                             MATCH          ;
17                             FALSE          A3
18                             CALL           definition
19          A3                 RETURN
20                             FLAG           A2
21                             MATCH          'E'
22                             FALSE          A1
23                             MATCH          'N'
24                             FALSE          A1
25                             MATCH          'D'
26          A1                 RETURN

27          definition         CALL           variable
28                             FALSE          B1
29                             MATCH          :
30                             FALSE          B1
31                             MATCH          :
32                             FALSE          B1
33                             MATCH          =
34                             FALSE          B1
35                             CALL           expression
36          B1                 RETURN
```

```
STMT #        LABEL                OPERATION     OPERAND

37            variable             MATCH         <
38                                 FALSE         C1
39            C2                   CALL
40                                 MATCH         >
41                                 NOT
42                                 FLAG          C2
43                                 MATCH         >
44            C1                   RETURN

45            expression           CALL          alternative
46                                 FALSE         D1
47            D3                   CALL
48                                 MATCH         |
49                                 FALSE         D2
50            D2                   CALL          alternative
51            D2                   RETURN
52            D1                   FLAG          D3
53            D1                   RETURN

54            alternative          CALL          element
55                                 FLAG          alternative
56                                 RETURN

57            e lement             CALL          variable
58                                 TRUE          F1
59                                 CALL          constant
60                                 TRUE          F1
61                                 MATCH         *
62                                 FALSE         F2
63                                 CALL          element
64            F2                   TRUE          F1
65                                 MATCH         (
66                                 FALSE         F3
67                                 CALL          expression
68                                 FALSE         F3
69                                 MATCH         )
70            F3                   TRUE          F1
71                                 MATCH         'U'
72                                 TRUE          F1
73                                 MATCH         —
74                                 FALSE         F1
75                                 CALL          elememt
76                                 RETURN
```

STMT #	LABEL	OPERATION	OPERAND
76	constant	MATCH	"
77		FALSE	G1
78		NEXT	
79	G2	CALL	
80		MATCH	"
81		NOT	
82		FLAG	G2
83		MATCH	"
84	G1	RETURN	

With respect to this section, the following facts should be noted:

1) Symbols Within Quotes

Placing a symbol(s) within quotes is equivalent to underlining it (them) in Reeve's compiler. For example,

'BEGIN' ≡ 'B' 'E' 'G' 'I' 'N' ≡ BEGIN

2) Integer Form of Symbolic Parsing Programs

The above two examples of symbolic parsing programs are written in a mnemonic notation for labels, operation codes, and operands. This was done for ease of understanding. For direct input to the compiler machine, however, a "symbolic" (assembly language) form of parsing program is written completely in integer notation. The compiler will assemble this assembly program, resolving all symbolic labels into absolute addresses, to generate the "machine" form of the parsing program which will then drive the parsing machine.

This will be fully explained, in terms of the total compiling machine later in this chapter, and what follows is a brief summary. Labels are negative integers. Operation codes are the positive integers 1 - 19 (see Appendix A). Operands may be either positive or negative integers or 0, depending upon whether they represent characters in the language (see Appendix C), labels, or the null character.

3) Blanks in the Source Text

The above two examples of parsing programs appear not to consider blank characters in the source text. This is correct because the parsing machine ignores all blanks and skips along until finding a non-blank symbol.

4) Number of Labels

There may be up to three labels per program statement.

5) Main Routine

The ordering of definition subroutines is not significant. However, the main routine must appear first.

```
CALL        SUBJECT
STOP
```

6) Mechanical Generation of a Parser Program

The direct relationship binding a meta-syntactic definition to the required parser program is what enables the compiler to read in a grammar definition and mechanically generate the required program that will parse the defined source language. This is the bootstrap technique that was referred to in Section 1.5

2.3 SEMANTICS

Thus far the thesis has dealt with source language parsing. Now it is necessary to consider the semantics associated with the source text. Therefore, this section will describe the method for generating the elements of the target language. The compiler machine consists of two distinct successive processors, the first of which being the parsing machine and the second being an editor machine (see Figure 2 on page 29). The parsing machine analyzes the structure of the incoming source text, as previously detailed, and concurrently generates a semantically oriented output stream (the interface) which is then rearranged and modified by the editor machine, resulting in the appropriate target code.

This section will be devoted to a detailed description of the two levels of semantic outflow, parser output and editor

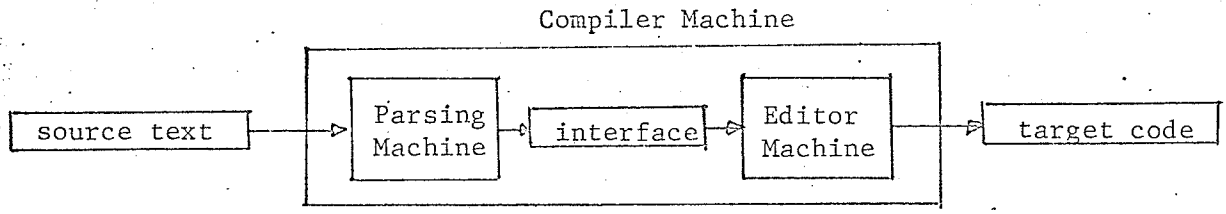


FIG. 2 THE COMPILER MACHINE IN A MACROSCOPIC VIEW

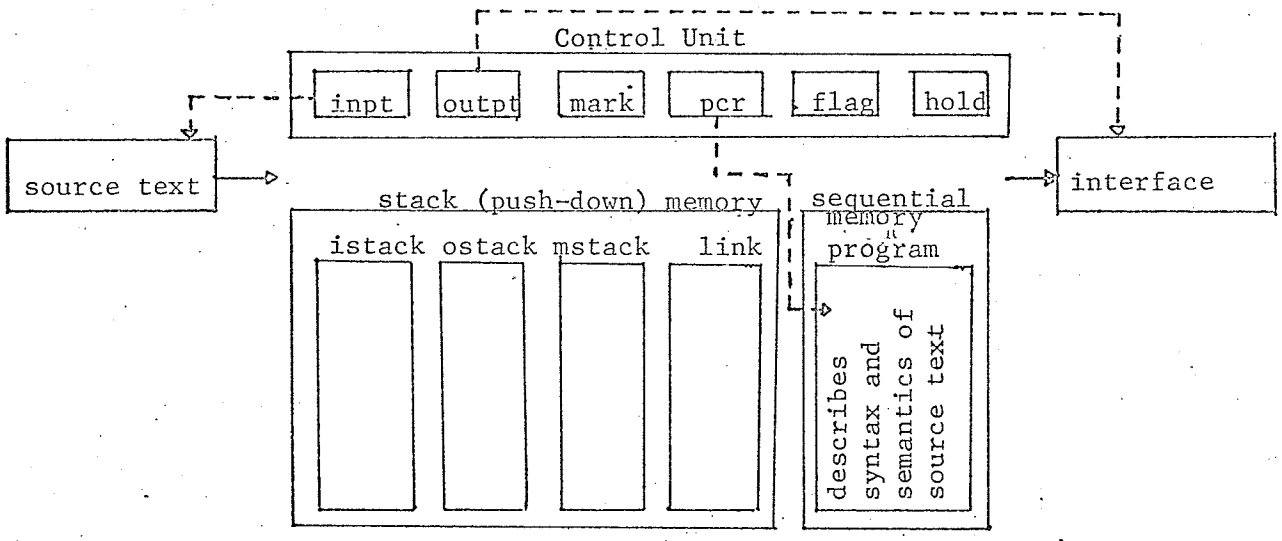


FIG. 3 THE PARSING MACHINE

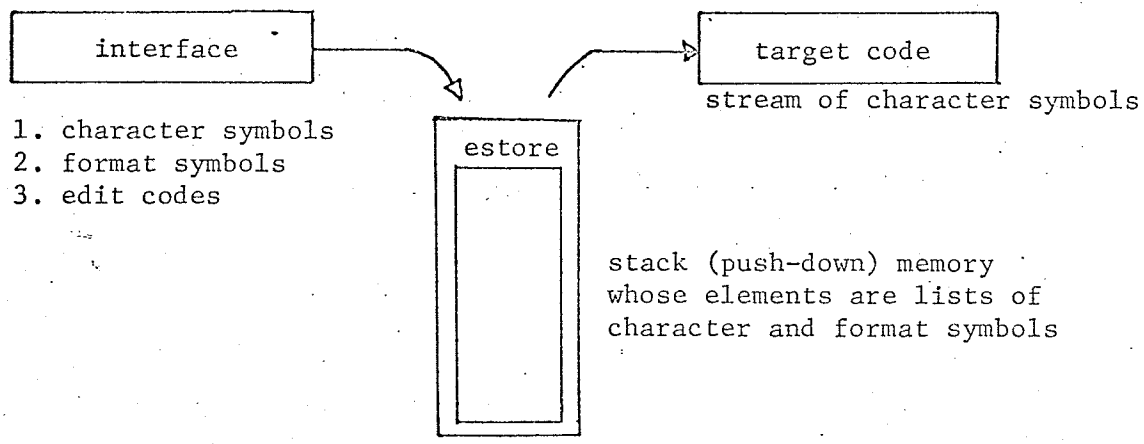


FIG. 4 THE EDITOR MACHINE

output, and of the meta-semantic language necessary for their specification. The meta-syntactic language described in Section 2.2.1 will be extended into a complete meta-language capable of describing both syntax and semantics, enabling the student to fully describe a computer language and to mechanically generate the appropriate program to perform the entire compiling function.

2.3.1 PARSING MACHINE OUTPUT

The parsing machine provides semantic output generation in a fashion consistent with and embedded in the parsing logic. The machine as previously described must be expanded for this facility. The elements to be added are:

1) an output stream (the "interface") consisting of a linear sequence of output elements. See Figure 3.
2) a register "outpt" (output pointer) which points to the location in "interface" where the next output symbol will be placed.
3) a stack memory "ostack" to save values of "outpt" at various stages in the parsing process.
4) a set of operations which perform the functions necessary for output, and set of instructions which invoke these operations.

The parser output stream, "interface", is a string of symbols each of which may be a character symbol, an edit code, or format control symbol. An output instruction for the parsing machine loads the position indicated by the current value of "outpt" and advances "outpt" to the successor position. Certain instructions cause "outpt" to be restored to an earlier value (saved in "ostack") and have the effect of erasing all intermediate items from the interface. Initially the interface is empty and "outpt" indicates the leading position.

The basic output instructions of the parsing machine are:

(1) PRINT character

Outputs the symbol in the operand part of the instruction.

(2) COPY

Outputs the symbol just read in from the source text, i.e.: the one immediately preceding that specified by "inpt" as the next to be read.

(3) NULL

Outputs a special null symbol. This occupies a space in the interface and causes "outpt" to be advanced, unlike a null input, but will not be written out by the editor later.

(4) EDIT edit code

Outputs the edit code in the operand part of the instruction.

(5) SPACE positive integer

Outputs a special format control symbol for spacing.

(6) TAB positive integer

Outputs a special format control symbol for tab positioning and carriage return.

We differentiate between character, edit and format codes on the parsing machine output stream. Character codes represent the symbols to be eventually listed on an output device. Edit codes are editor instructions invoking modifications and rearrangements of the character codes before they are listed. And format codes control layout at final listing time. PRINT, COPY, and NULL produce character codes, EDIT generates edit codes, and SPACE and TAB yield the format codes.

The pushdown stack "ostack" is augmented at each level in the parsing process by a location which records the value of "outpt". This is handled by the expansion of the instructions:

CALL subroutine name or null (Λ)

records the value of "outpt" in "ostack".

RETURN If "flag" is false then restore to "outpt" the value from the top of "ostack" and delete it from "ostack".

NOT Restore to "outpt" the value from the top of the "ostack" and delete it from the "ostack".

TRUE If "flag" is false restore to "outpt" the value from the top of "ostack".

These last operations serve to erase any output that may have taken place in the course of an unsuccessful recognition attempt in the parsing procedure.

2.3.2 EDITING AND THE EDITOR MACHINE

The method of combining and rearranging the character (and format) symbols on the parser output stream, "interface", must now be considered. It is accomplished by passing "interface" through a second computer called the editor machine which is formed by: (see Figure 4)

(1) Input stream, "interface", consisting of a sequential list of character symbols, format symbols, and edit codes.

(2) Output stream, the target code, consisting of a sequence of rearranged character and format codes.

(3) A stack memory containing a pushdown list structure, "estore", whose elements are lists of one or more character and format codes.

(4) An operating cycle consisting of:

a) "Pull" the next element from the list "interface"

b) If the element is a character or format symbol, "push" it into the stack "estore".

If the element is an edit code, execute the specified operation upon the elements of "estore".

(5) A set of operations that are invoked when the corresponding edit code is detected in the input stream.

Thus the input stream, "interface", acts as combined data and program for the editor machine. The character and format codes are the data and the edit codes the instructions.

The edit codes are:

1) X Exchange the top two lists of "estore".

2) C Concatenate the top two lists of "estore" by attaching the head of the top list to the tail of the next.

3) W Output (write) the elements in the editor stack in sequence list by list from the bottom up and from head to tail within each list, ignoring the null symbol whenever it appears, and obeying all format control symbols. Then return control to the parsing machine at the instruction following the EDIT W which initially invoked the editor.

The editor machine is initiated by the execution of an EDIT W instruction by the parsing machine. This causes the parser to output the edit code W as the last element of "interface" and to pass control to the editor machine. The editor scans "interface" one element at a time from its head to the final W, pushing down all character and format symbols into the editor stack and executing edit codes. When editing is finished control is returned to the parsing machine.

For edification purposes, let

1) "interface" consist of tog X Ca C X W where capital letters represent edit codes and lower case specify the character symbols toga.

2) "estore", commencing as empty, be represented by a series of elemental lists separated by commas, the elements being ordered such that the bottom level is on the left.

The activation of the editor machine upon the parser-editor interface would change toga into goat as follows:

interface	estore	target code
t o g X C a C X W		
o g X C a C X W	t	
g X C a C X W	t,o	
X C a C X W	t,o,g	
C a C X W	t,g,o	
a C X W	t,go	
C X W	t,go,a	
X W	t,goa	
W	goa,t	
		goat

2.3.3 META-SEMANTIC LANGUAGE

In order to describe the structure of the source text by a set of syntactic rules, the meta-syntactic language was designed. Now, in order to describe the semantics associated with the structural elements of the source text, we require a meta-semantic language.

The first level of the semantic process is the output from the parsing machine. The meta symbols {(/,/), 'C', 'X','S','B', 'W', 'O', 'K'} which compose the meta-semantic language therefore correspond to the output instructions in the parsing machine program. The correspondence is as follows:

	meta-semantic language	⟷	parsing machine program
(1)	(/ abc....z/) where a,b,....,z are character symbols		PRINT a PRINT b EDIT C PRINT c EDIT C ⋮ PRINT z EDIT C
(2)	'Sα' where α is a positive integer		SPACE α
(3)	'Bα' where α is a positive integer		TAB α
(4)	'K'		COPY
(5)	'O'		NULL
(6)	'W'		EDIT W

(7) 'C' EDIT C

(8) 'X' EDIT X

In case (1) the print definition (/ abc...z/) where abc....z represents a string of character symbols, produces a PRINT a instruction followed by a series of PRINT and EDIT C instructions. This ensures the character string will, in the editor phase, have its elements concatenated and will thus be treated by later editing operations as an entire unit (instead of having only its last character operated on).

The semantic rules must correspond and be embedded with the syntactic structures of a source language. Therefore, the meta-semantic and meta-syntactic languages are integrated to form a "meta-language", that is, a language that will completely describe (the syntax and semantics of) another language.

To clarify the semantic concepts described in Section 2.3, there follow two examples, the first illustrating the power of parsing machine output, and the second illustrating the editing facilities.

EXAMPLE 1:

Suppose we wish to translate the sentence "The boy sees a tree" into its equivalent German counterpart "Der knabe seht einen baum." The meta-language definition would be:

```
'BEGIN'     <sentence>;
<sentence>::=<subject><predicate>(/./);
<subject>::=<noun phrase>;
<predicate>::=<verb><object>;
<verb>::= "sees" 'S1' (/ seht /) 'C';
<object>::=<noun phrase>;
<noun phrase>::=<article><noun>;
<article>::= "The" (/ der /)|"a" 'S1' (/ einen /) 'C' ;
<noun>::= "boy" 'S1' (/ knabe /) 'C'|"tree" 'S1' (/ baum /) 'C'
'END'
```

The parsing machine program would be similar to that described in Example 1 of Section 2.2.3.2 on page 21 with the following modifications to add semantic rules to routines:

STMT #	LABEL	OPERATION	OPERAND
3	sentence	CALL	subject
		FALSE	L1
		CALL	predicate
		FALSE	L1
		PRINT	.
	L1	RETURN	
19	article	MATCH	T
		FALSE	L4
		MATCH	H
		FALSE	L4
		MATCH	E
		FALSE	L4
		PRINT	D
		PRINT	E
		EDIT	C
		PRINT	R
		EDIT	C
	L4	TRUE	L5
		MATCH	A
		FALSE	L5
		SPACE	1
		PRINT	E
		PRINT	I

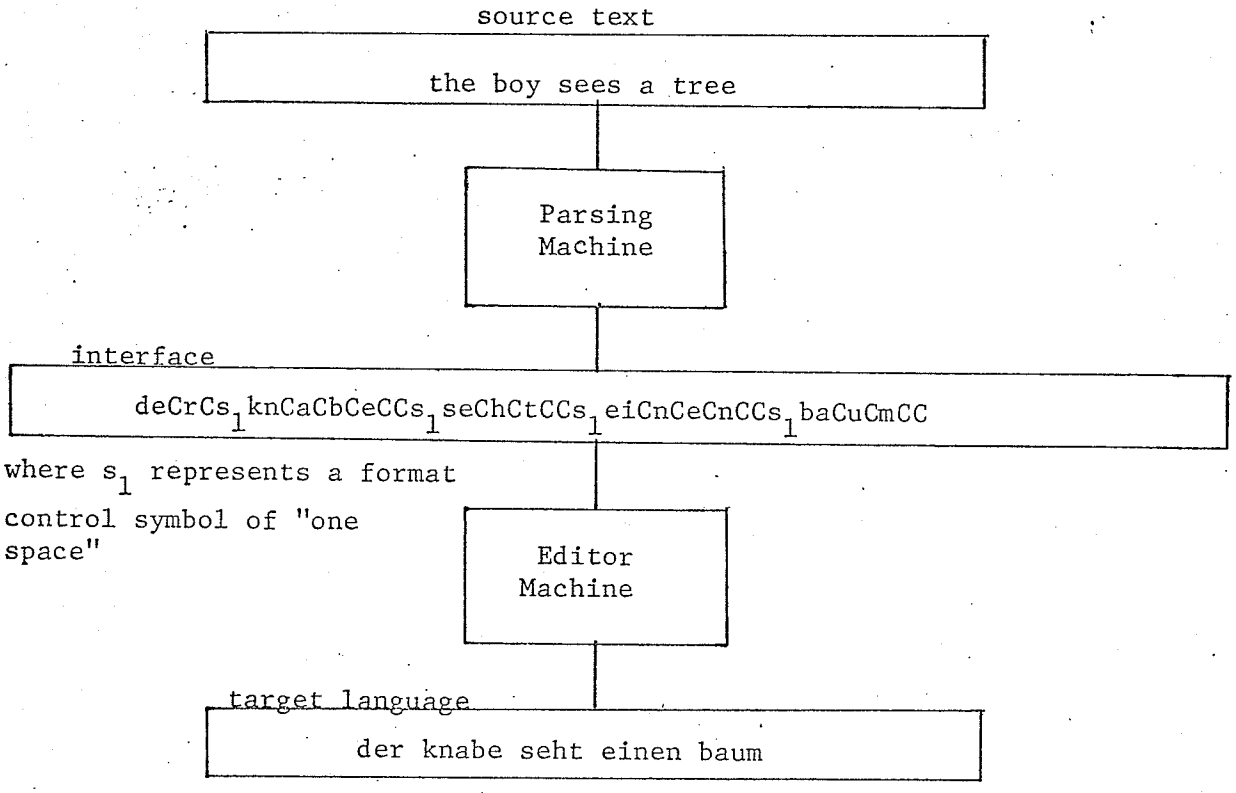
```
STMT #      LABEL       OPERATION       OPERAND

                        EDIT            C
                        PRINT           N
                        EDIT            C
                        PRINT           E
                        EDIT            C
                        PRINT           N
                        EDIT            C
                        EDIT            C
            L5          RETURN

27          noun        MATCH           B
                        FALSE           L6
                        MATCH           O
                        FALSE           L6
                        MATCH           Y
                        FALSE           L6
                        SPACE           1
                        PRINT           K
                        PRINT           N
                        EDIT            C
                        PRINT           A
                        EDIT            C
                        PRINT           B
                        EDIT            C
                        PRINT           E
                        EDIT            C
                        EDIT            C
            L6          TRUE            L7
                        MATCH           T
                        FALSE           L7
                        MATCH           R
                        FALSE           L7
                        MATCH           E
                        FALSE           L7
                        MATCH           E
                        FALSE           L7
                        SPACE           1
                        PRINT           B
                        PRINT           A
                        EDIT            C
                        PRINT           U
                        EDIT            C
                        PRINT           M
                        EDIT            C
                        EDIT            C
            L7          RETURN
```

STMT #	LABEL	OPERATION	OPERAND
41	verb	MATCH	S
		FALSE	L8
		MATCH	E
		FALSE	L8
		MATCH	E
		FALSE	L8
		MATCH	S
		FALSE	L8
		PRINT	S
		PRINT	E
		EDIT	C
		PRINT	H
		EDIT	C
		PRINT	T
		EDIT	C
	L8	RETURN	

With regard to Figure 2 on page 29, this source text and parsing machine program would cause the following output:



It should be noted that this example uses little editing facilities. The numerous concatenation commands "C" to the editor are not really necessary here, but are included for generality. The main purpose was to stress the output capabilities of the parser machine.

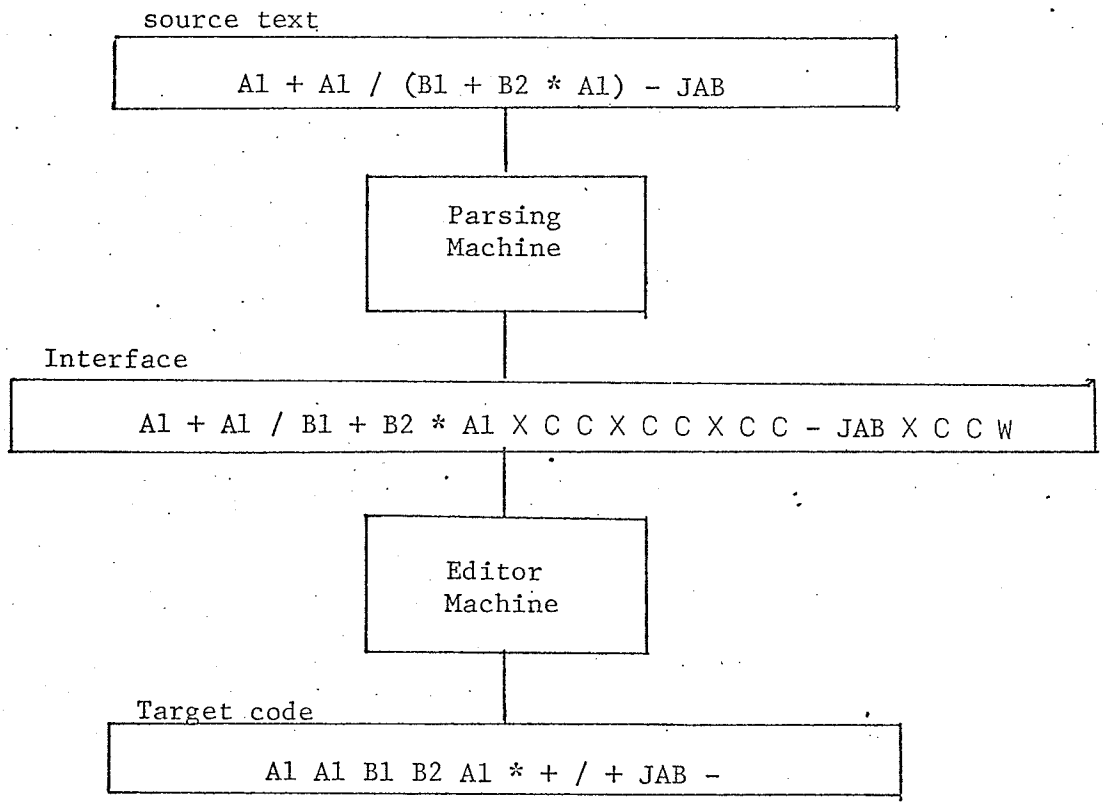
EXAMPLE 2:

To illustrate the compiler machine with an emphasis on editor machine actions, let us consider a problem associated with the compilation of arithmetic expressions - translation into Polish postfix notation.

The meta-language definition is:

```
'BEGIN'   <arithmetic expression>;
<arithmetic expression>::=<term>*(<add op>'K'<term>'XCC')'W';
<add op>::= "+"|"-";
<term>::=<factor>*(<mult.op>'K'<factor>'XCC');
<mult.op>::= "*"|"/";
<factor>::=<variable>   |"("<arithmetic expression>")";
<variable>::=<letter>'K' * (<letter>'KC'|<digit>'KC');
<letter>::= A|B|C|D|E|F|G|H|I|J;
<digit>::= 1|2|3|4|5|6|7|8|9
'END'
```

If the above definition is used to generate a parsing program which in turn reads in the source expression "A1+A1/(B1+B2*A1)-JAB", then the translation process would proceed:



2.3.4 MARKING AND TESTING

There exists a further facility which can be helpful in economizing the description of a source language. It is called "marking and testing" and it consists of flagging the occurrence of a syntactic type and later allowing conditional semantic output in the successively higher level of the meta-language definition, depending upon if the given syntactic type occurred.

2.3.4.1 ILLUSTRATION

This can be shown most easily by illustration: Consider the syntax

```
'BEGIN' <group> ;
<group>::=<item><item>;
<item>::="a"|"b"|"c"
'END'
```

and suppose that the required output when parsing a group is "d" if the first item of the group is "b", otherwise nothing. The normal approach is unattractive, namely,

```
<group>::=<item1><item2>;
<item1>::="a"|"b" (/ d /)|"c";
<item2>::="a"|"b"|"c"
```

Our preferred solution is:

```
<group>::=<item>'T1' ( (/ d /) )<item>;
<item>::="a"|"b" 'M1'|"c"
```

Whenever "b" occurs as an item in a group, bit one in a register called "mark" is turned on (as signified by 'M1'). Then in the next level <group>,bit one is tested (as specified by 'T1') immediately after recognizing the first item. If this bit is on then whatever semantics follow, enclosed within parentheses "()", are executed. If not, these semantics are skipped. In this example, the instruction to print "d" is enclosed within parentheses.

Now consider:

```
<a>::=<b><b><c>'T8'('XCX');
<b>::="p"'K'|"q"'K';
<c>::=<b>|"r"'KM8'
```

This says that an <a> will consist of a sequence of three symbols each of which may be "p" or "q". In addition the last symbol might be an "r". The symbols are copied out as they appear, unless the last symbol is "r" in which case the three symbols are reversed. Bit eight of "mark" is set and tested in this illustration.

If, however, this definition was modified as follows:

```
<c>::=<b>|<reverse>;
<reverse>::="r"'KM8'
```

the testing of bit eight in <a> would not work for <a> is two syntactic levels above <reverse>. There exists a unique "mark" for each syntactic type and it can be altered only by the syntactic type which it explicitly calls.

Finally, a more subtle example, but commonly used, is:

```
<a>::=<b>*(<b>'M1')'T1'( (/ multiple occurences /) ).
```

If <a> consists of a sequence of two or more <b>s, then the message "multiple occurances" is output. Since the second <b> in the definition is enclosed within parentheses (is part of a "structural element") and is programmed within CALL Λ and RETURN instructions, it is in fact a next level and may, therefore, set the marker associated with <a>.

2.3.4.2 IMPLEMENTATION

Marking and testing require the following facilities in the parsing machine: (see Figure 3 on page 29)

(1) a register "mark" that contains a sequence of 31 bits.

(2) a register "hold" to hold temporarily the operand value of a TEST instruction.

(3) a stack memory to contain "mstack". This is a pushdown list that saves values of "mark" at various stages (levels) in the parsing process.

(4) a set of operations which perform the functions necessary to mark and test, and a set of instructions which invoke these operations.

The basic three instructions for this feature are:

(1) MARK α

where α is a positive integer [1,31]. Sets bit α to 1 in the record at the head of "mstack".

(2) SELECT α

where α is a positive integer [1,31]. Copies α into a temporary location called "hold".

(3) TEST label

Tests if the bit position of "mark" specified

by "hold" contains 0 or 1. If 0 control is transferred to label, where label specifies the instruction following the conditional semantic output succeeding the TEST instruction.

In addition, the following parsing machine instructions are extended:

(1) CALL routine name or null

Push "mark" onto the top of "mstack" and set "mark" to zeros.

(2) TRUE label

If "flag" is false reset "mark" to zeros.

(3) RETURN

NOT

Remove the record from the head of "mstack" and copy it into "mark".

Thus each activation of a program routine (representing a syntactic type definition) has a set of 31 marker bits associated with it in "mark", and they are initially zeroed. When this routine calls another, "mark" is saved on the top of "mstack" and may there be altered by MARK instructions within the called routine. When control returns to the calling routine, "mark" is restored from the top of "mstack" and may then be checked by SELECT and TEST instructions to conditionally

allow semantic output. At the end of each alternative "mark" is reset to zeros by the TRUE instruction, thus reinitializing it for the next alternative.

The meta-language definition of marking and testing corresponds directly with the machine program:

	metalanguage	machine program form
(1)	'Mα'	MARK α
	where α is an integer [1,31]	
(2)	'Tα' (<passive>)	SELECT α TEST L <passive> L: .
	where α is an integer [1,31]	
	and <passive> is any meta-	
	language code not invoking	
	the testing of source symbols.	

To complete our illustration, the assembly form of the compiler machine program for the first example in this section would be:

```
          CALL      group
          STOP

group     CALL      item
          FALSE     A1
          SELECT    1
          TEST      A2
          PRINT     d
A2        CALL      item
A1        RETURN
```

```
item    MATCH       a
        TRUE        B1
        MATCH       b
        FALSE       B2
        MARK        1
B2      TRUE        B1
        MATCH       C
B1      RETURN
```

2.3.5 GENERATING AN ASSEMBLY PROGRAM FOR THE PARSING MACHINE

It will often be desirable to generate as target code a program for the parsing machine itself. This program may take one of two representations, an assembly language form allowing symbolic labels, or a machine language form which has addresses in absolute value and can be executed by the parsing machine. Since the compiler machine explicitly provides the additional facility of an assembler, the best approach is to generate the symbolic assembly language program and let the assembler resolve the symbolic labels and translate into machine language form. The parsing machine program is the only program that the student must provide to the compiler machine. The parsing machine, in executing this program, will construct the "interface" which consists of the combined data and program for the editor machine. Therefore, the parsing machine program may, for application purposes, be thought of as the "program" of the compiler

machine. This section will describe the techniques required to generate an assembly program that will run the parsing machine.

Figure 5 shows the compiler machine extended to include an assembler and indicates the ordered manner in which the machine operates. Initially the compiler machine reads an assembly program (symbolic parsing machine program) into the assembler, assembles it into the machine form, and places it into the sequential memory of the parsing machine. Control is then transferred to the parsing machine. It, under control of the machine program, reads in the source text, parses it, and generates an interface to the editor machine. The editor machine interprets the symbols and edit codes of the interface to produce the target code. This target code, however, may actually be an assembly program for the parsing machine in which case it can optionally be fed back into the assembler to begin the cycle anew. Otherwise the cycle terminates upon production of target code.

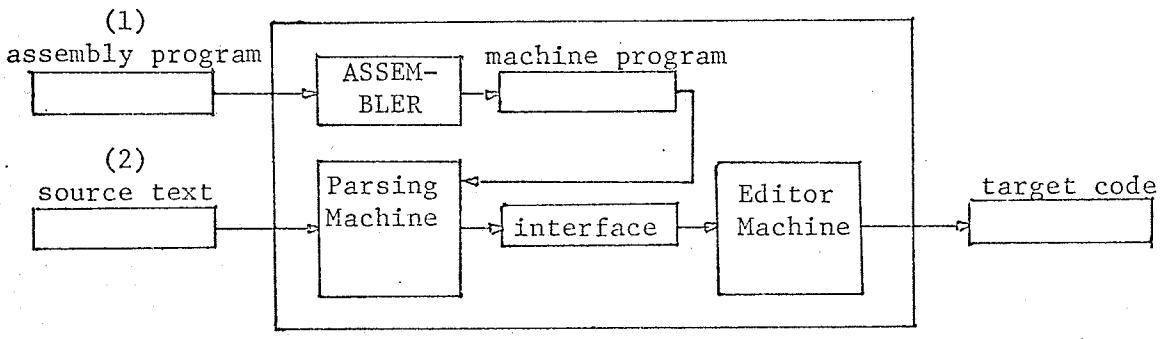


FIG. 5 COMPILER MACHINE

2.3.5.1 SYMBOLIC PARSING MACHINE PROGRAM

Until now all examples of programs for the parsing machine have been written using mnemonics for labels, operation codes and operands. This was done for ease of understanding, but these programs are not in a form that the compiler machine's assembler can directly assemble into the required machine language form. The assembly form of the program which is acceptable is a stream of integers as follows:

```
'BEGIN'    <assembly program>;
<assembly program>::=*<routines>"999";
<routine>::=*<instructions>"0";
<instruction>::=*<label><op.code><operand>;
<label>::=<internal label>|<external label>;
<internal label>::=<integer X,-999≤X≤-1>;
<external label>::=<integer X,     X≤-1000>;
<op.code>::=<integer X, 1≤X≤998>;
<operand>::=<label>|<character symbol>|<positive integer>|<null symbol>;
<character symbol>::=<integer X, 35≤X≤198>;
<positive integer>::=<integer X, X>0>
<null symbol>::="0"
'END'
```

The integers to be used for operation codes are given in Appendix A. Character symbol representation is described in Appendix C.

For purposes of illustration, consider the last example in 2.3.4

```
'BEGIN' <group>; <group>::=<item>'T1' ( (/ d /) ) <item>;
                 <item>::="a"|"b"'M1'|"c"      'END'
```

The assembly program that will compile a <group> is:

Mnemonic Form as before			Correct Symbolic Form		
	CALL	group		1	-1000
	STOP			19	0
				0	
group	CALL	item	-1000	1	-1001
	FALSE	A1		4	-1
	SELECT	1		8	1
	TEST	A2		9	-2
	PRINT	d		16	53
A2	CALL	item	-2	1	-1001
A1	RETURN		-1	2	0
				0	
item	MATCH	a	-1001	11	50
	TRUE	B1		3	-1
	MATCH	b		11	51
	FALSE	B2		4	-2
	MARK	1		7	1
B2	TRUE	B1	-2	3	-1
	MATCH	c		11	52
B1	RETURN		-1	2	0
				0	
				999	

Thus a negative integer represents a label or symbolic address. Labels identifying routines are called external labels and are integers less than or equal to -1000. Labels within routines are internal labels and have values in the range [-999, -1]. Mnemonic operation codes are replaced by positive integers in the range [1, 998]. Operands may be positive or negative, depending upon whether they are a character symbol, format control value, or a label. When the assembly program is assembled into an executable machine program, the symbolic labels are resolved and replaced by

positive absolute addresses, so that a zero denotes a null address in a CALL, RETURN, NOT, or COPY instruction or the end of a routine.

2.3.5.2 INTEGER OUTPUT

The compiler machine must possess the capability to explicitly output integers. This ability is necessary in order for the student to generate the integer stream comprising a parsing machine assembly language. Also, in a more general sense, this facility is required by the parsing machine in order to generate the integer stream comprising the interface:

```
'BEGIN'  <interface>;
         <interface>::=*(<symbol>|<edit code>);
         <symbol>::=<character symbol>|<format control symbol>
                   |<special positive integer>|<null symbol>;
         <character symbol>::=<integer X,35≤X≤198>;
         <format control symbol>::=<integer X,X≥ 600000000>;
         <null symbol>::=500000000;
         <edit code>::=<integer X,    X≤-1>;
         <special positive integer>::=<integer X, X>0>
'END'
```

The mnemonic edit codes are replaced (see Appendix A) by negative integers. The symbols are all positive integers.

Generating integers is a different process from producing character digits, and enters the meta-language in two ways.

(1) Via the Parsing Machine -- the PRINT instruction

The meta-language PRINT symbol may be used to output integers by placing the integer in parentheses "()". Thus (/ (1) /) is compiled as PRINT 1, whereas (/ 1 /) is compiled as PRINT 41 since 41 is the internal character code for the digit 1. The PRINT meta-symbology may now be defined as

"(/"*("("<pos.integer>")"|<character symbol>)"/)"

With this facility operation codes and most operands may be generated.

(2) Via the Editor Machine

Two new meta-language edit codes 'V' and 'N' compile into EDIT V and EDIT N respectively. Their effects when interpreted by the editor are as follows:

'V' assuming that the top list of the editor stack "estore" contains a string of digits, replace it by the corresponding integer.

'N' assuming that the top list of "estore" contains a single integer value, negate it.

The use of the 'V' code is seen when specifying an integer in the meta-language.

<integer>::=<digit>'K'*(<digit>'KC')'V';

As successive digits are read, they are copied and concatinated into a string, and finally the whole string is replaced by the value of the integer.

'N' is required to generate negative integers in the target code. The parsing machine cannot directly generate output integers that are negative, since they would be intercepted by the editor as edit codes. Thus the parser produces the edit code N in the interface and lets the editor create the negative integer.

2.3.6 LABEL GENERATION

The generation of integers for operation codes and for all operands, including negative ones but excluding labels, may be handled by the techniques described in 2.3.2. Labels require special methods for each label must be unique and thus necessitates additional parser and editor instructions. Also, since labels have two integer ranges according to whether they are internal to routines or global, two separate methods are used:

(1) External Label Generation

External labels correspond to the names of syntactic variables in the meta-language and they name the location

of the heads of routines in the parsing machine program. This correspondence is set up via a further edit code, 'L', in the meta-language which compiles as EDIT L.

Before explaining the action caused by L entering the editor machine, it is necessary to extend the editor machine. Independent of the editor stack "estore" is another stack, "name", which is a list of sub-lists (see Figure 6). A sub-list consists of a first element which is the external label (integer $\leq$ -1000), followed by a second element which is a sub-sub-list containing the symbols making up the variable name. Finally the editor contains a register "index" which is initialized to -1000.

The action of the edit code L is as follows:

L Assuming that the top list of the editor stack "estore" is a string of symbols making up a variable name, scan the list "name" for this name. If found, replace the entry in the stack by the corresponding external label value. If not found, enter the name from "estore" in the list "name" with the current value of "index" as the external label value, and replace the "estore" entry as before - also decrement the value of "index" by unity.

The use of the L edit code is seen in the full specification of a variable in the meta-language as shown in Appendix B.

<variable>::="<"'UK'*(⌐">"'KC')">"'L';

(2) Internal Label Generation

For generation of internal references the parser machine is extended to include an additional five instructions and associated hardware, a register "tag" to hold positive integers, and a push down list "label" whose elements are values of "tag". The extra instructions are as follows:

SETLAB	Set "tag" = 1 and clear "label" list.
INCLAB	Push current value of "tag" onto top of "label" list, and increment "tag" by unity.
LABEL	Output top entry in the "label" list, followed by the N edit code.
DECLAB	If "flag" is false, copy the top entry of the "label" list into "tag". Delete the top entry.
RESETLAB	Clear "label" list.

For convenience, these five instructions are actually written with one operation code "reference" and operand of 1, 2, 3, 4, 5 respectively. Correspondingly, the meta-language specifies them as 'R1', 'R2', 'R3', 'R4', and 'R5'. The full specification of the meta-language is in Appendix B.

SETLAB (reference 1) and DECLAB (reference 2) should be specified at the beginning and end of a parent routine that calls other routines which will generate labels. INCLAB and DECLAB should be specified at the beginning and end of a routine whose instruction sequence contains LABEL instructions corresponding to one particular label. A label is output whenever LABEL is obeyed.

Note that the labels are output by the parsing machine as positive integers followed by an N edit code. This is because the label would be interpreted by the editor as an edit code if it was negative in the interface. The N gets around this difficulty.

The five instructions are sufficient to handle all internal labelling and are well illustrated in Appendix B which defines the complete meta-language, both syntax and semantics, in the meta-language itself.

2.4 CYCLING AND COMPILER USAGE

The ability of the compiler machine to cycle was basically described in section 2.3. In this section the cycling process will be restated and its function explained with respect to intended compiler machine usage.

2.4.1 CYCLING AND RECYCLING

The compiler machine "cycle" consists of an "assembly phase" followed by a "compile phase". In the assembly phase, a parsing machine assembler program, describing the syntax and semantics of some source language, is read into the assembler. The assembler resolves all symbolic references, generates the machine form of the program and places it into the sequential memory of the parsing machine (see figure 6). Then, in the compile phase, source text is compiled into target code. That is, the parsing machine under control of its newly acquired machine program inputs some source language text (described originally by the assembler program), parses it and generates the appropriate interface to the editor machine. The editor machine then interprets the interface and produces the final target code.

The target code may be any kind of output the student desires including an assembler program for the parsing machine in which case it may be routed as input to the assembler to initiate the start of a new cycle. This is referred to as "recycling" and may theoretically occur to an arbitrary finite depth.

2.4.2 HOW CYCLING IS APPLIED

Normal usage of the compiler machine consists of feeding it three consecutive inputs:

1) Assembly program defining the meta-language. This is a standard program that need not be written by the student.
2) Meta-language definition of a source language.
3) A text written in the described source language and finally receiving the desired object code.

(1) and (2) form the assembly and source text inputs of a first cycle. (3) forms the source text input of a second cycle. This occurs as follows:

1) Cycle 1.
   a) Assembly Phase. In order to initially provide the parsing machine with a program, an assembly program is read in, assembled into machine language form and placed in the parsing machine's sequential memory. This assembly program defines the syntax and semantics of the meta-language itself.
   b) Compile Phase. The student's source language definition, written in meta-language form, is read in as source text by the parsing

machine under control of its machine program. The target code finally produced is an assembly program for the parsing machine that corresponds to the student's source language definition.

2) Cycle 2.

a) Assembly Phase. A recycle now commences. The assembly language program, corresponding to the student's source language definition and just produced as target code, as now assembled and placed into the parsing machine's sequential memory, over-writing the previous contents.

b) Compile Phase. The parsing machine, under control of its new program, reads in the student's source language text. This is now compiled into the target code desired by the student (or a warning of "faulty syntax" is output).

See Figure 7.

Once the student has made his final source language definition, either in the standard meta-language or his own, he is able to output the target code assembly program for the parsing machine that corresponds to his source definition. Then each time he wishes to compile a program, he need only make one cycle, reading in this assembly program before his source deck.

In addition, if the student, say, described his own meta-language in terms of the standard one, then described a source language written in this new meta-language, and finally read in the source language, there would be three full cycles (a first cycle followed by two recycles). Recycling thus provides a flexable system.

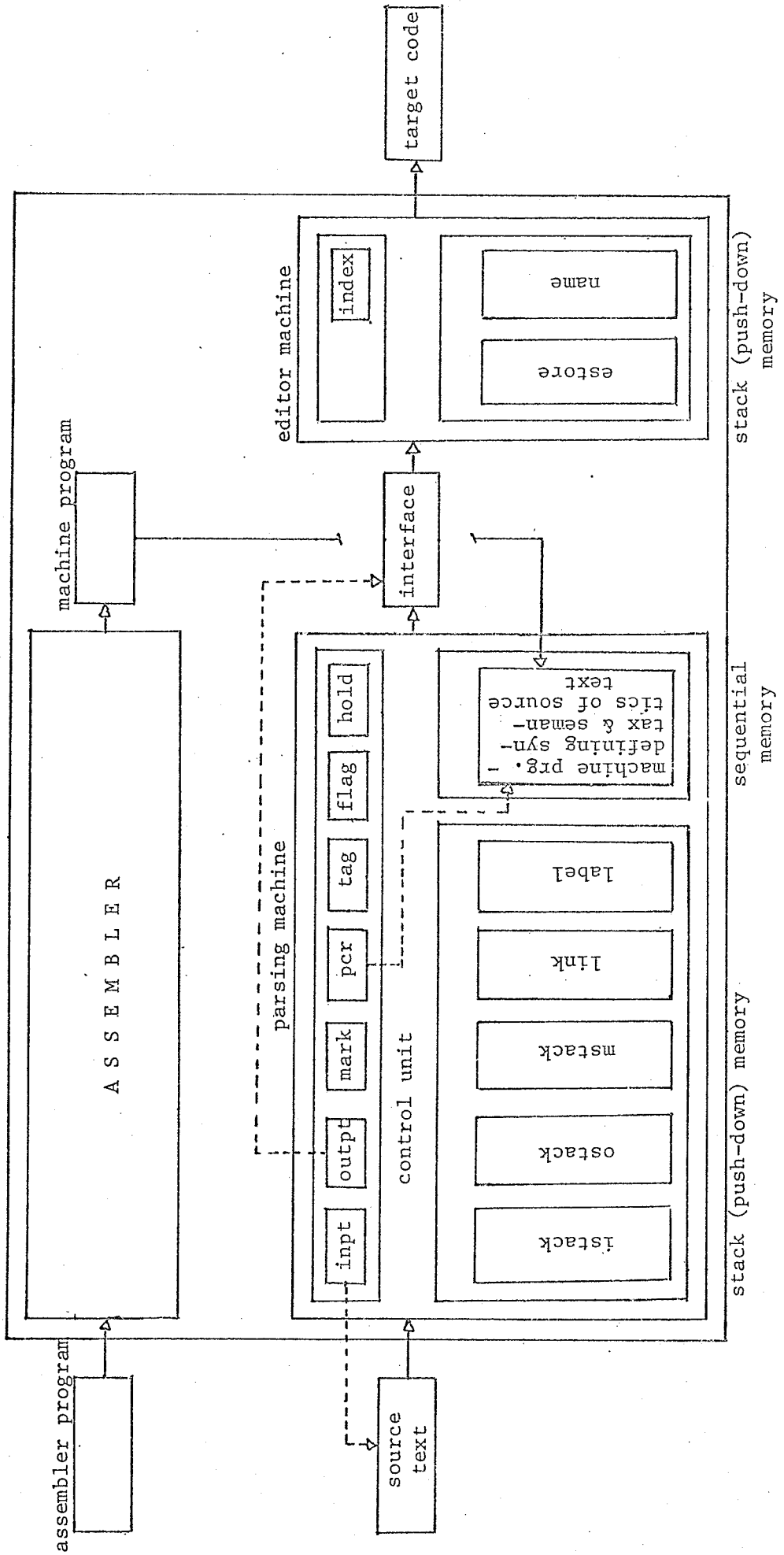


Fig. 6 The Compiler Machine

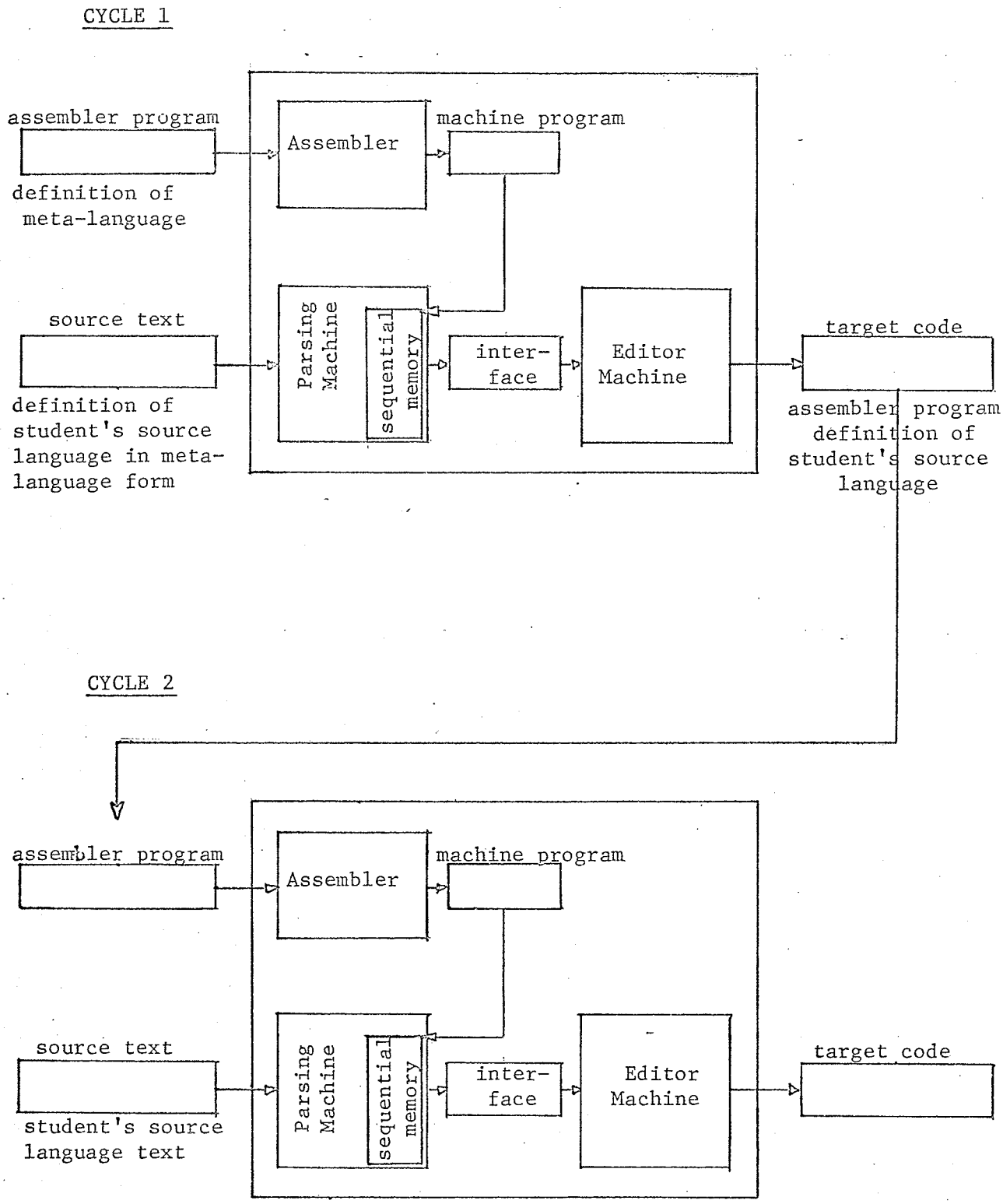


FIG. 7 NORMAL CYCLING FOR STUDENT USAGE

CHAPTER III

AN IMPLEMENTATION IN ALGOL

The compiler machine was implemented as a virtual or pseudo machine, rather than as a hardware entity. That is, it is a simulated machine programmed on a computer.

This chapter will describe the software approach that was followed.

3.1 PROGRAMMING LANGUAGE CONSIDERATIONS

3.1.1 CHOICE OF ALGOL 60

In order to achieve the goal of relative machine independence, the syntax-directed compiler was written in the universally accepted algebraic language ALGOL 60 [3].

3.1.2 ALGOL 60 INPUT-OUTPUT ROUTINES

The inherent problem associated with this choice of languages is that ALGOL 60 was originally defined without I/O routines, leaving this domain to be developed locally by each user installation and creating source language I/O incompatility between computer centers.

A set of standard primitives were recently accepted [5], and based on these, Stanford University suggested a group of standard

I/O transfer procedures [11]. Ignored, however, have been the I/O control procedures (next record, tabulation control, carriage control, etc.)

To resolve this dilemma, a general set of nine I/O control procedures were designed. Stanford's I/O transfer procedures were implemented, and this total package was documented and added to the compiler machine as a global block. (See Appendix D)

Thus, the program may be implemented using any ALGOL translator having Stanford I/O procedures [11], or, more generally, having the primitives defined in [5]. The body of each I/O control procedure, however, will need to be locally defined for all but IBM 360 translators.

3.2 STORAGE MANAGEMENT

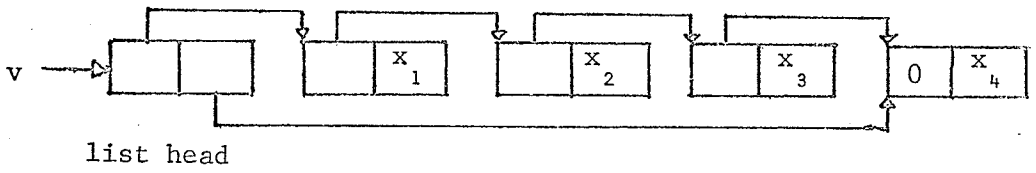
Memory space for all the data structures used by the compiler machine is handled by list processing methods. This approach was necessitated by a need for dynamic reallocation of storage, and makes available the usage of list structures as well as linear strings.

3.2.1 LISTS AND THEIR REPRESENTATION

A list is a connected sequence of elements, each element consisting of two computer words, a "head" and a "tail". These elements are connected through the "head" parts, each of which contains the address of the succeeding element. Since each

element points to its successor, successive list elements need not be, and in general are not, consecutive double words in memory. In fact, one of the attractive features of lists is their ability to utilize arbitrary, disjointed sections of memory.

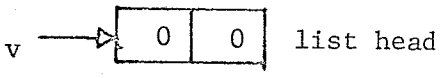
The second word of each element, the "tail", may contain data (a symbol or numerical value) or may point to another list. A list v would have the following structure:



SIMPLE LIST STRUCTURE

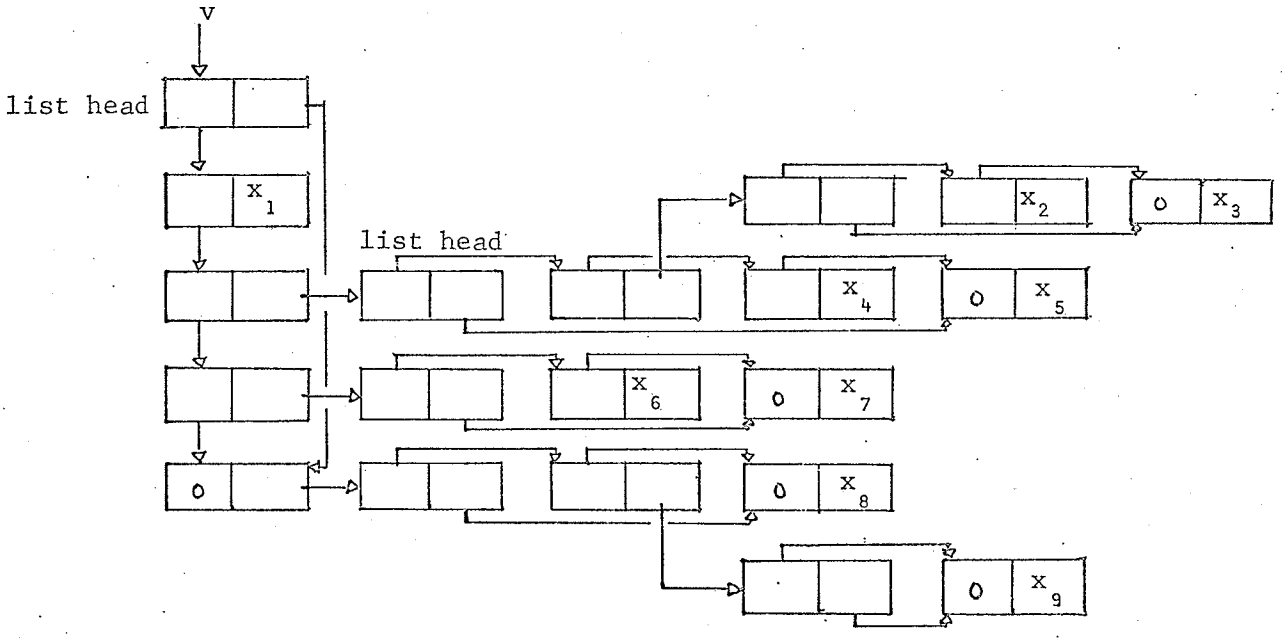
where $x_1,.....,x_n$ in the tail of all elements other than the first represent data.

Each list is provided with a "list head", a known location in memory, through which the programmer can locate the list. This list head points to the first and last element in the list. In addition, the last element is flagged by a pointer of 0. A null list is represented by a list head containing two 0 pointer values.



the null list

Since the tail portion of each list element may point to another list (called a "sub-list"), and so forth to an arbitrary finite depth, very complex configurations called "list structures" may be formed. For example:



AN EXAMPLE OF A LIST STRUCTURE

As can be seen, a list structure contains information in two ways, the first being the actual datum $x_1,\ldots,x_n$, and the second being the structural location of the datum.

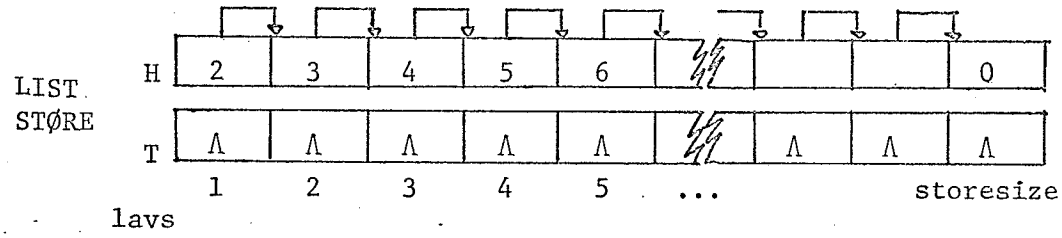
3.2.2 STORAGE ALLOCATION

The storage space required for all lists is taken dynamically from a special list called the "list-of-available-space". Initially, all available memory is assigned to this

list. Then when various lists are created and built, their elements are allocated one at a time from this list-of-available-space. Similarly when a list is no longer needed, its elements are returned and are then available for re-allocation to another list. Thus at any time only the exact amount of needed storage is being used and lists have the ability to begin, increase and decrease in size, and be erased as required at execution time.

This dynamic allocation and reallocation of memory to list structures has been implemented as follows:

a) List Store and List-of-Available-Space



LIST STORE INITIALIZED TO THE LIST-OF-AVAILABLE-SPACE

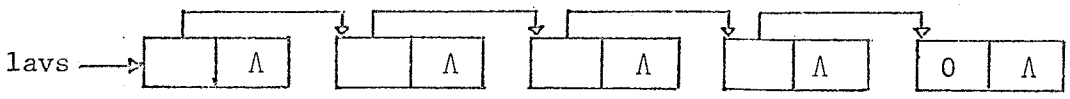
Total storage for lists, the "list store", consists of two integer arrays declared H, T [1:storesize] where "storesize" is the length of each array and is read as an input parameter at the beginning of the program. (A storesize of 2000 is sufficient for most applications.) A list element allocated

from this store will have an address i, $1 \leq i \leq$ storesize and will consist of the pair H[i], T[i] being respectively the head and the tail of the element.

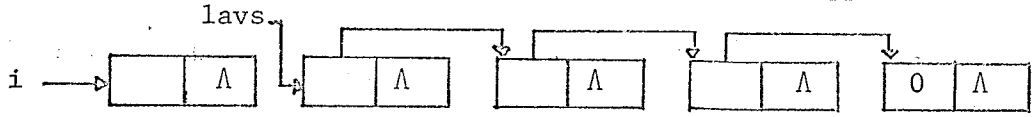
Storage allocation is handled in a conventional manner by means of a list-of-available-space, "lavs", which has no list head. Initially, "lavs" points to the first element in the "list store" (has the value 1) and each element is chained in a linear fashion as shown above. Thus at the beginning of the program, lavs consists of the whole store list. The datum in lavs are set to 500000001, the null value Λ, in order to facilitate monitoring.

b) Allocation of Elements from Lavs

Mechanisms are needed for obtaining "new" elements from and returning unneeded elements to the list-of-available-space lavs. An element H[i], T[i] may be taken from lavs by an ALGOL procedure "take(i)" which assigns the address of the first element in lavs to the integer i. Lavs is changed to H[lavs], that is, lavs now points to the next element in its list. Locations no longer required must be returned to lavs by a procedure "put(i)". This pushes the element H[i], T[i] onto the beginning of lavs by chaining H[i] to lavs, setting T[i] to Λ and then letting lavs equal the address i. For example, if after many operations, the list-of-available-space looked as follows:



a call of the procedure take(i) would cause it to appear



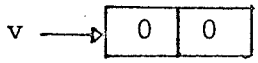
and an immediate put(i) would return it to its original state.

Take(i) and Put(i) are the primitives required for other list processing procedures.

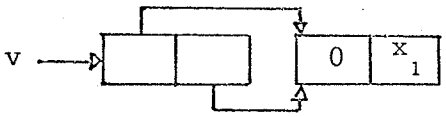
3.2.3 LIST PROCESSING PROCEDURES

In order to build lists, operate upon them, and finally delete them, the following procedures are used:

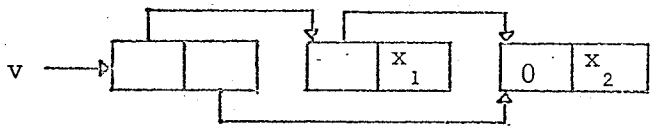
1) open list (V) ~ takes the first free element from lavs, makes it a list head and initializes it to a null list



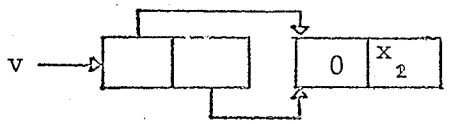
2) push (X_1, V) ~ connects the first free element in lavs to the head of list V and assigns it the value X_1.



3) append (X_2, V) ~ connects the first free element in Lavs to the end of list V and assigns it the value X_2.



4) pull (X, V) ~ assigns to X the value of the first element in list V and returns the element to the list-of-available-storage.



and X: = X_1

5) closelist (V) ~ returns the head and elements of list V to the list-of-available-storage and sets V to 0.

V=0

6) combine (V1, V2) ~ concatenates the elements of list V2 to the end of list V1, creating a combined list V1.

7) exchange (V1, V2) ~ exchanges the lists V1 and V2.

8) reverse (V) ~ reverses the linear order of the elements in list V.

9) behead (i,V) ~ returns to the list-of-available-space any elements chained from the head of list V to the list element with address i.

10) betail (i,V) ~ returns to the list-of-available-space any elements of list V that trail from the list element with address i.

11) equal (V1,V2) ~ compares two lists V1 and V2 to see if they contain the same information.

3.3 IMPLEMENTATION OF THE ASSEMBLER

An assembly language program that is to be assembled consists of an integer stream of instructions, each having the form (* <label> <operation code> <operand>). The instructions are grouped into routines, each routine terminated by "0", and the whole program is terminated by "999". The only negative integers are either labels, or else operands referencing the instructions named by the labels. The complete description was given in 2.3.5. The machine language form resulting from assembly was specified in 3.2. It is output as a list "target" and then transferred to the sequential memory of the parsing machine to become the machine "program".

3.3.1 THE ASSEMBLY CONFIGURATION

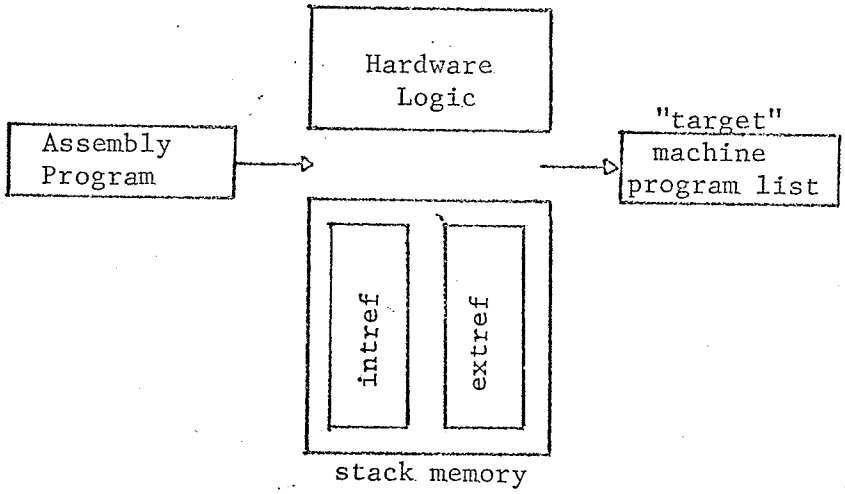
The assembler consists of:

1) an input stream of integers forming the assembly program.

2) an output list, "target", each element of which contains a machine instruction in packed form (see 3.3.2).

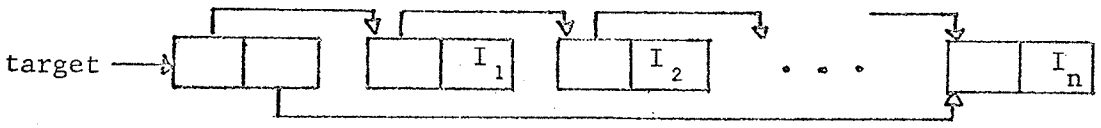
3) memory to contain "intref" and "extref". These are list structures to aid in resolving symbolic address references.

4) logical operations that will assemble the input program an integer at a time.



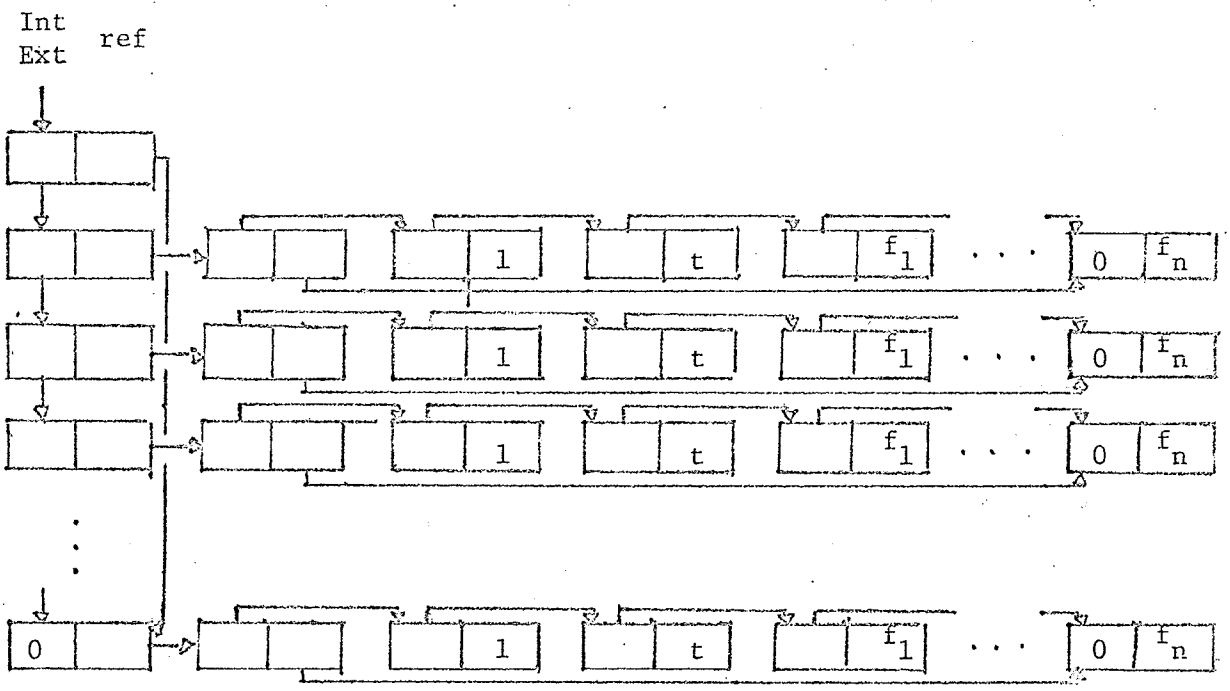
3.3.2 ASSEMBLER DATA STRUCTURES

1) "target" - machine program that is generated



where I is the i^{th} instruction in packed form.

2) "extref" and "intref" - both lists have the following structure:

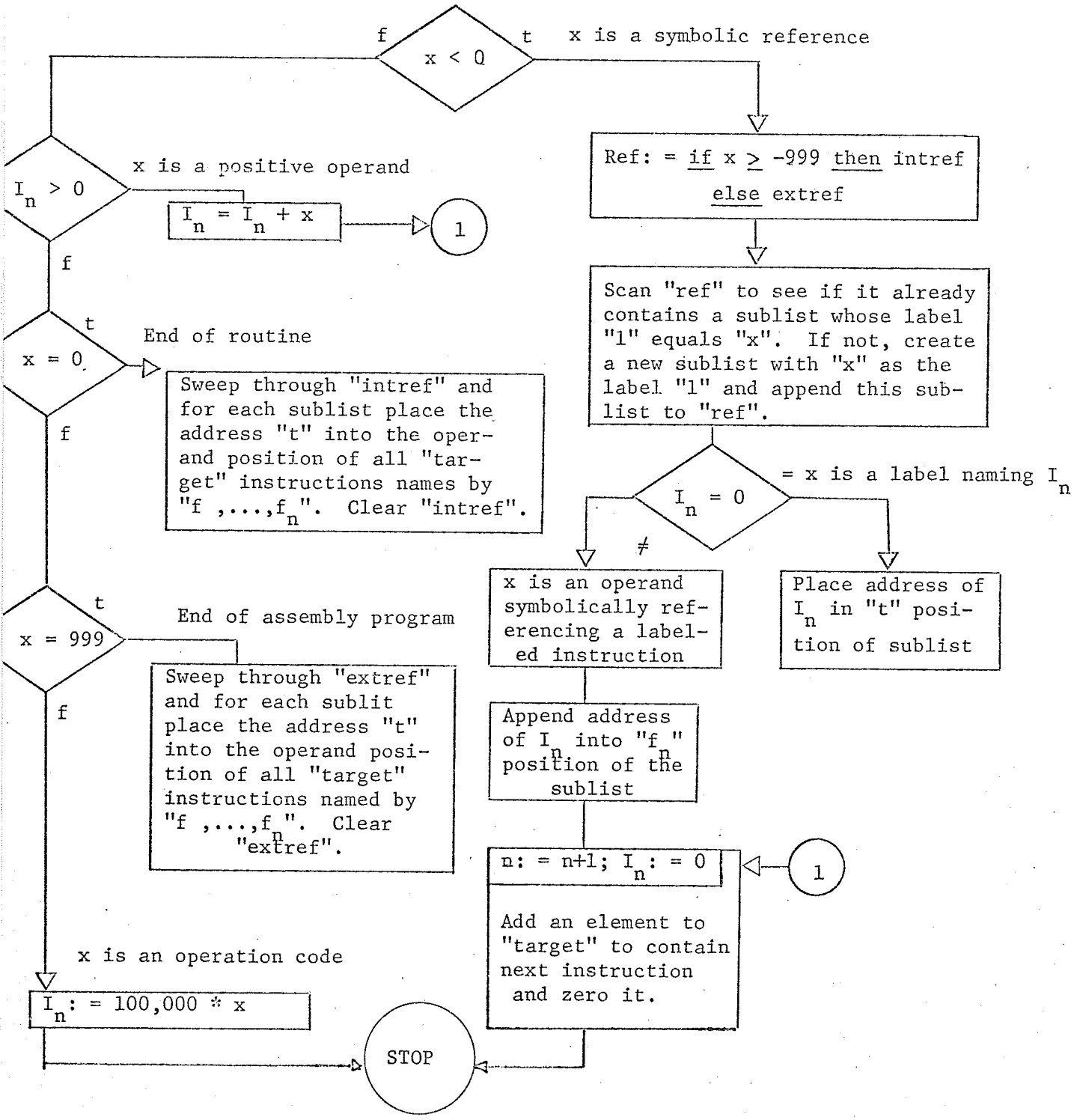


The first element in each sublist contains the label "l" in its negative integer form. The second element contains "t", the address in "target" of the instruction labelled by "l". The remaining data "f_1, f_2, ..., f_n" are the addresses in "target" of all instructions whose operands made a symbolic reference to "l".

"Intref" is the list handling all references within routines. "Extref" handles all external references between routines.

3.3 ASSEMBLER LOGIC

The integers in the assembler program are read in consecutively as the variable "x" and immediately assembled by invoking the procedure "assemble(x)". Assemble(x) proceeds as follows:


x < 0
f
t
x is a symbolic reference
Ref: = if x ≥ -999 then intref else extref
Scan "ref" to see if it already contains a sublist whose label "1" equals "x". If not, create a new sublist with "x" as the label "1" and append this sublist to "ref".
$I_n = 0$
= x is a label naming I_n
≠
x is an operand symbolically referencing a labeled instruction
Place address of I_n in "t" position of sublist
Append address of I_n into "f_n" position of the sublist
n: = n+1; I_n: = 0
1
Add an element to "target" to contain next instruction and zero it.
$I_n > 0$
x is a positive operand
$I_n = I_n + x$
1
x = 0
End of routine
Sweep through "intref" and for each sublist place the address "t" into the operand position of all "target" instructions names by "f ,...,f_n". Clear "intref".
x = 999
End of assembly program
Sweep through "extref" and for each sublit place the address "t" into the operand position of all "target" instructions named by "f ,...,f_n". Clear "extref".
x is an operation code
I_n: = 100,000 * x
STOP

3.4 IMPLEMENTATION OF THE PARSER AND EDITOR MACHINES

Given the complete functional delineation in Chapter II of the parser and editor machines, together with, in 3.2, the procedures for all list/stack handling, the basic programming of parser and editor is quite straightforward. (See Appendix E for the actual coding)

This section will discuss two structures that merit attention -- the implementation of the parsing machine program, and a student language efficiency measure.

3.4.1 STORED FORM OF A PARSING MACHINE PROGRAM

An assembly program is assembled and the resulting "target" code is stored in the sequential memory of the parsing machine (see Figure 6). Here it is called the stored form of the parsing machine program, or, simply, the "machine program". This section will describe its appearance within the ALGOL implementation program and give a brief illustration.

The machine program is held in the form of a simple list, each datum element being a complete instruction. The method of packing an instruction into a single computer word is as follows:

100,000 x <operation code> + <operand>

This saves storage and facilitates monitoring. All symbolic

PARSING MACHINE FOR A <GROUP>

Mnemonic form		Assembly form			index address	Machine form* H	Machine form* T
					"program" →	25	72
	CALL group		1	-1000	25	13	100030
	STOP		19	0	13	30	1900000
			0				
group	CALL item		1	-1001	30	31	100094
	FALSE A1		4	-1	31	33	400052
	SELECT 1		8	1	33	47	800001
	TEXT A2		9	-2	47	26	900093
	PRINT d		16	54	26	52	1600054
A2	CALL item	-1	1	-1001	52	93	100094
A1	RETURN	-2	2	0	93	94	200000
			0				
item	MATCH a	-1001	11	50	94	95	1100050
	TRUE B1		3	-1	95	61	300
	MATCH b		11	51	61	68	1100051
	FALSE B2		4	-2	68	100	400112
	MARK 1		7	1	100	112	700001
B2	TRUE B1	-2	3	-1	112	71	300072
	MATCH c		11	52	71	72	1100052
B1	RETURN	-1	2	0	72	0	200000
			0				
			999				

*Note that the head elements "point" to their successor. The absolute index addresses are, of course, arbitrary and this is an illustration only.

labels have been resolved into absolute address references.

For purposes of illustration, consider once again the parsing machine program for a group where a group was defined:

```
'begin <group>;
       <group>::= <item> 'T1'((/d/)) <item>;
       <item> ::= a|b'M1'|c
'end'
```

(See the previous page)

3.4.2 LANGUAGE DEFINITION PERFORMANCE TEST

Students experimenting in the definition and manipulation of formal languages, might wish to compare the efficiencies of various language definitions used for the compiling of source texts.

Thus, an optional performance test was added to the compiler. It is a count of the number of attempts that were made by the parsing machine to recognize both syntactic constants and variables. This was implemented by incrementing counters each time a match instruction was executed or when a call instruction referenced a syntactic variable subroutine respectively.

3.5 COMPILER MACHINE I/O STREAMS

The input stream (<assembler program>*<source text>) and the output stream (*<target code>) of the compiler machine are extended somewhat to allow flexibility and ease of use for the students. The extensions facilitate:

1. Control of compiler machine storage requirements. The student specifies <storesize>, the number of locations given to "LIST STORE", from which all lists in the machine are taken.
2. Control of I/O devices. Through control parameters, the student may specify
   a) Input/output devices for initial assembler program. Thus, this initialization parsing machine program that defines the meta-language is available from disk as well as cards. Also, it may be optionally copied onto any device.
   b) Output device for target codes. The target code generated at the termination of each cycle may be optionally output on any device. Thus, it may be listed on a printer, as in the case of final object code, or, as in the case of the assembler program definition of the student's meta-language, it may be stored on disk for future compiler machine initialization.

The input source texts are assumed to always come from cards.

3. Formatted printer listings. Whenever the student specifies output on the printer, the compiler machine formats the listing
   a) Assembler programs are listed in both numeric and symbolic form, one instruction to a line.

b) Target texts that are not assembler programs are page aligned.

3.5.1 THE INPUT STREAM

Syntax

```
'begin' <input stream>
<input stream> ::= <store size>
                   <assembler control params><assembler program>
                  *(<source control params><source text>"$");
<store size> ::= <integer>;
<assembler control params> ::= <device><listing?>;
<device> ::= <integer x, 0 ≤ x ≤ 15>;
<listing?> ::= "'TRUE'" <device>|"'FALSE'";
<assembler program> ::= see 2.3.5.1
<source control params> ::= <route><listing?>
                            <performance testing>;
<route>::= "1"|"2"|"3";
<performance testing> ::= "'TRUE'"|"'FALSE'";
<source text> ::= *'U'
'end'
```

Semantics

<store size>, the first entity in the input stream, is a positive integer specifying the number of storage locations available to arrays H and T of LIST STORE.

<assembler control parameters>, control the choice of I/O devices used in the assembly phase of the first cycle. The first <device> specifies the input device to be used for the assembly program. The convention used is 0 for card reader, 1 for printer and 2 - 15 for devices that the student may define via control cards. <listing?> determines if the student wants this assembler program listed ('TRUE', 'FALSE') and if so, on what output <device>.

<assembler program> is the initialization assembler program that defines the meta-language. See 2.3.5.1.

<source control parameters> control recycling and output listing of target code. <route> is an integer which selects the course of the compilation as follows:

1. A source language text is to be input and no recycling after it is processed.
2. A grammar is to be input and no recycling after it is processed.
3. A grammar is to be input and recycling will occur.

<listing?> specifies if the student wants the target code output ('TRUE', 'FALSE') and if so, on what device. The parsing count of terminal and non-terminal structures may be demanded by again stating the Boolean value 'TRUE' or 'FALSE'.

<source text> is the text that will be compiled in each cycle. It can be a meta-grammar, a meta-language, or a source program. A final terminating $ is required.

3.5.2 THE OUTPUT STREAM

Syntax

```
'begin' <output stream>;
<output stream> ::= (<assembler program>|)
                    *(<source control params><source text>"$"
                     (<performance test results>|)(<target code>|));
<assembler program> ::= "METAGRAMMAR"<see 2.3.5.1>;
<source control params> ::= <route><listing?><performance testing>;
<route> ::= "ROUTE:="("1"|"2"|"3");
```

```
<listing> ::= "OBJECT CODE LISTING:="("'FALSE'"|"'TRUE' DEVICE
              FOR OBJECT CODE LISTING:=" <device>)
<performance testing> ::= "PERFORMANCE TESTING:="("'TRUE'"|"'FALSE'");
<source test> ::= *'U';
<performance test results> ::= "NUMBER OF ATTEMPTS TO RECOGNIZE
              A NONTERMINAL TYPE:=" <integer> "NUMBER OF ATTEMPTS
              TO RECOGNIZE A TERMINAL TYPE:=" <integer>;
<target code> ::= <assembler program>|*'U'
'end'
```

Semantics

The initialization assembler program may optionally be listed on any device. Then for this and all succeeding cycles, the <source control parameters> are echoed onto the printer along with the <source text>. <performance test results> may optionally be given and also <target code> output.

It should be noted that when <target code> is an assembler program for the parsing machine (i.e. when <route> ::= 3) and is output, it is listed in numeric and symbolic form, for ease of reading.

CHAPTER IV

STUDENT USER REFERENCE

This chapter is written as an aid to the student wishing to use the compiler machine with minimum knowledge of its internal workings. It is not an independent and totally descriptive chapter, but an outline with cross references to required reading sections in the thesis, and provides an applied example. See Appendix F for a short glossary of any unfamiliar terms.

4.1 INTRODUCTION

The compiler machine provides a facility for practical experimentation in the definition and manipulation of formal languages. There is a standard meta-language in which the student may write a grammar which describes a programming language and, optionally, the object code to which it should be translated. Then he can submit a program in this language, leaving the compiler machine to scan the program for faulty syntax, and map it into the corresponding target language, if this was described in the grammar. For more advanced students, there is the possibility of designing one's own meta-language.

4.2 DESCRIBING ONE'S PROGRAMMING LANGUAGE

Before submitting a program to the compiler machine, it is first necessary to define the language in which the program is written. In this programming language definition there are two distinct but related areas of description - syntax and semantics. Syntax is the written appearance of the language, its allowable phrases, its structure. Semantics refers to the meaning associated with the syntax, and this is usually formalized as the target code for a computer that permits the programmer's objectives to be realized.

Therefore, the standard meta-language provided with the compiler machine to allow the student to describe his programming language may be conceptualized as an amalgam of a meta-syntactic language and a meta-semantic language.

4.2.1 META-SYNTACTIC LANGUAGE

The meta-syntactic language is that half of the meta-language in which the student can define the syntax of his programming language. It is a simple extension of Backus Normal Form, and is well described in section 2.2.1.

As a further example to section 2.2.1, let us define a normal arithmetic expression:

```
'begin' <arith expr>;
<arith expr> ::= <term>*(("+" "-")<term>)
<term> ::= <factor>*(("+" "-")<factor>);
<factor> ::= <primary>*("**"<primary>);
<primary> ::= <variable>|<integer>|"("<arith expr>")";
<variable> ::= <letter>*(<letter>|<digit>);
<integer> ::= <digit>*<digit>;
<letter> ::= "A"|"B"|"C"|.....|"Z";
<digit> ::= "0"|"1"|"2"|.....|"9";
'end'
```

4.2.2 META-SEMANTIC LANGUAGE

The meta-semantic language is the other half of the meta-language and is used to describe the semantics associated with the student's programming language. It describes the target code the compiler machine must generate from any given program in the student's language, and the printed format of this code.

The meta-semantic language consists of a set of symbols which dictate semantic output operations at three distinct, successive levels:

1. Generation of strings of code.
2. Editing (rearranging) operations on those strings yielding final target code.
3. Output initiation and control.

4.2.2.1 GENERATING CODE STRINGS

This is the primary level of the semantics or object code production process. The meta-semantic symbols used to generate

code at various positions and levels in the syntactic description are:

1. String Output (/<string>/)

   where <string> ::= *-"/)"

   Any string of printable characters may be directly output as one code string by bracketing them with (/ , /).

2. Copy Previous Character 'K'

   Any character (terminal character) of the incoming source text may be echoed as a code string to the output stream.

3. Null Character '0'

   This generates a code string consisting of a null unit, that is a code string that will be considered as such in the editing level, but will never in fact be printed. This is often required for generality with respect to the later editing operations.

As a trivial example of 1. and 2., consider the objective of recognizing an integer, and outputting it along with some kind of signal:

```
'begin'<integer>
<integer> ::= <digit>'K'*(<digit>'K') (/NOTE*INTEGER/);
<digit> ::= "0"|"1"|"2"|.....|"9"
'end'
```

Each time an integer was recognized in the source text, its digits would be copied as a sequence of individual code

strings to the output stream, followed by the code string "NOTE*INTEGER"; i.e. d,d,d,....,d,NOTE*INTEGER where d is a digit, and code strings are separated by ",".

4.2.2.2 EDITING THE CODE STRINGS

The linear sequence of code strings generated by (/..../), 'K', '0' may now be reordered or "edited" by the symbols 'c' and 'x'. The conceptual kernel is as follows:

The compiler machine regards the grammar describing a source language as a tree structure, using it as a top-to-bottom, left-to-right parsing algorithm. Each time it recognizes a syntactic type, and semantic output is dictated, it appends this current code string to the linear sequence of previously generated code strings. Each time the edit codes 'c' or 'x' are indicated, the compiler makes the respective operation on the last two code strings.

1. Combine 'c'

   Combine the previous two code strings.

2. Exchange 'x'

   Exchange the previous two code strings.

These operations are quite attractive, from somewhat the dominoe principle. The combine operation can successively accumulate many code strings into a single string of code, and the exchange operation could then be effected on the lot.

Example 1

Consider reversing input text. The grammar would be:

```
'begin' <text>;
<text> ::= '0'*('U''x''c')"$"(/***TEXT*REVERSED/)
'end'
```

Then if the input source text was

```
LIVE $
```

the output stream would be generated as follows:

```
Level 1        ϕ, L'xc', I'xc', V'xc', E'xc', ***TEXT*REVERSED
Level 2   a)        Lϕ, I'xc', V'xc', E'xc', ***TEXT*REVERSED
          b)            ILϕ, V'xc', E'xc', ***TEXT*REVERSED
          c)                 VILϕ, E'xc', ***TEXT*REVERSED
          d)                      EVILϕ, ***TEXT*REVERSED
Level 3                             EVIL ***TEXT*REVERSED
```

Note that in Level 3, the final output phase, the null symbol ϕ does not appear.

Example 2

Consider the problem associated with arithmetic expressions -- translation into Polish post-fix notation. The grammar to do this is found on page 41 (example 2 of 2.3.3).

4.2.2.3 OUTPUT CONTROL

This third and final level of semantic or object code production process, consists of symbols 'W', 'S', 'B' which initiate and format the actual output of the edited output stream.

1. Write 'W'

   Initiates the editing and actual output of the string codes generated up to this point. 'W' must always appear in a grammar involving semantic output, at least once, and at the end.

2. Space 'S<integer>'

   Space <integer> positions on the output record before writing again.

3. Tabulate 'B<integer>'

   Skip <integer> output records before writing again.

The meta-semantic symbols, 'S' and 'B', although controlling output formatting, are handled until this time by the compiler machine as individual code strings and are thus under control of the edit symbols 'x' and 'c'.

4.2.4 MARKING AND TESTING

Marking and testing is an additional feature allowing conditional semantic output at the first level, and economy in description of a source language. It consists of "marking" or flagging the occurrence of a syntactic type and the, at the next higher syntax level, "testing" for the syntactic type's occurrence and conditonally generating a code string.

This can be most easily shown by the illustration of section 2.3.4.1 on page 43.

4.3 JOB SET-UP

Now in order to finally use the compiler machine, the student must submit a deck of cards consisting of a grammar followed by a source program and each preceeded by control information.

This layout is described formally and in full generality in section 3.5.1. This section, however, will describe the set-up for a typical student run consisting of a grammar followed by a source deck.

```
    <job> ::=
a)              <storesize><initiation info>
b)              <grammar control parameters><grammar>$
c)              <source control parameters><source program>$
```

a) <storesize> ::= integer specifying the number of double word memory locations to give to the compiler machine for list handling. Generally, 3,000 has been found satisfactory;

```
<initiation info> ::= <device><listing?>(<initiation data>|)
<device> ::= <integer x | 0≤x≤15>;
<listing?> ::= 'TRUE' <device>|'FALSE';
```

The compiler machine requires initiation data which will normally be kept on random access for ease of student use. <device> here refers to the device on which it is found, and will be given locally by the professor. <listing?> should normally be 'FALSE' to prevent the output of the initiation data, which is an assembler program to drive the compiler machine.

If however the <initiation data> is not on a R.A. device, but is given to the student in card form, this should follow

<device> ::= 0, <listing?> ::= 'FALSE'.

b) <grammar control parameters> ::= <route><listing?>
<performance testing>
<route> ::= "2"|"3"
<listing> ::= 'FALSE'
<performance listing> ::= 'FALSE'

Specifying <route> as 2 determines that the grammar is to be checked syntactically after which the compiler machine is to terminate. <route> = 3 specifies that a <source program> is to follow the <grammar>.

<listing> = 'FALSE' prevents a newly generated internal program of the compiler machine from being listed.

<performance testing> = 'FALSE' is normal.

<grammar> ::= student's grammar (in meta-language form) defining his programming language. This will always be listed for the student on the printer.

c) <source language> ::= <route><listing?><performance testing>

<route> ::= 1
<listing?> ::= 'TRUE' <device>|'FALSE'
<device> ::= 1
<performance testing> ::= 'TRUE'|'FALSE'

Specifying <route> = 1 determines that the <source program> is following and the compiler machine is to terminate after translation.

<listing> = 'TRUE' <device> specifies that the semantics (student's object code) is to be output after compilation of the following <source program> and on device <device>. (Note that <device> ::= 1 is for printer.) If, however, the student specified no semantic output, only program syntax checking in his grammar, <listing?> must be 'FALSE'.

<performance testing> allows conditional output of the total number of syntactic variables and constants that were attempted to be recognized in the analyses of the <source program>.

<source program> ::= a program in the programming language described by the student in his grammar . This will always be listed for the student on a printer, optionally followed by the generated object code.

4.4 A STUDENT EXAMPLE

A) Problem:

Let us now consider a typical computer science problem - defining a simple programming language for a target computer of the Polish notation class (e.g. Burrough's 5500, English Electric KDF9) with a stack-like accumulator.

The target code to be generated is an assembly language as follows:

Syntax

```
'BEGIN' <assembler language>; Target code for a stack computer;
<assembler language> ::= *(<operand>|<operator>",")"*END";
<operand> ::= <identifier>|<number>;
<operator> ::= <declarative operator>|<imperative operator>;
<declarative operator> ::= "*VAR"|"*LAB";
<imperative operator> ::= "*CLA"|"*ADD"|"*SUB"|"*MUL"|"*DIV"|
                          "*EXP"|"*NEG"|"*ABS"|"*STO"|"*TRA"|
                          "*TZE"|"*TPL"|"*TMI"|"*HLT";
<identifier> ::= <letter>*(<letter>|<digit>);
<number> ::= <digit>*<digit>;
<digit> ::= "0"|"1"|"2"|.....|"9";
<letter> ::= "A"|"B"|"C"|......|"Z"
'end'
```

Semantics

a) Assembly Time:- The assembly program will be assembled conventionally placing each imperative operator and each operand of the program in successive locations.

The following three declarative operators are executed by the assembler at assembly time:

*VAR <operand> - allocates a location for the following operand. This instruction must preceed all imperative operators.

*LAB <operator>- defines the current location (pointed to by location counter) symbolically by the following operand.

*END - terminates assembly. Thus it is always the final operator in the program.

All operands other than immediately behind a *VAR or *LAB, are assembled into their corresponding location as an address for variables, or as the value of that operand (for numbers).

All imperative operators are assembled into their corresponding locations.

b) Execution Time:- Thus each imperative operator and operand is contained in separate words of the computer. Control steps sequentially through the computer, jumping to a new sequence when commanded by a transfer operator. The appearance of an operand causes the operand to be placed in the computer's

push-down stack. The appearance of an operator causes execution of that operation.

The imperative operators are:

*CLA - replace the variable address in the top entry of stack with that variable value.

*ADD - replace the top two entries of stack with their sum.

*SUB - replace the top two entries of stack with their difference.

*MUL - replace the top two entries of stack with their product.

*DIV - replace the top two entries of stack with their quotient.

*EXP - replace the top two entries of stack with their exponentiation.

*NEG - replace the top entry of stack with its negative.

*ABS - replace the top entry of stack with its absolute values.

*STO - store the value of the top entry of stack in the address specified by the next-to-top entry. Remove both entries from stack.

*TRA - transfer control to the address specified by the top entry of stack and remove it.

*TZE - transfer control to the address specified by the top entry of stack if, and only if, the next-to-top entry contains a zero. Remove both entries.

*TPL - functions as *TZE, but for a positive value.

*TMI - functions as *TZE, but for a minus value.

*HLT - stops execution.

B) Solving:

Assuming the student wishes to design a simple programming language for this computer, and submit an example program in this language, a job deck would be set up as follows:

(Initiation data assumed to be on device 5.)

```
INITIATION:     STORESIZE:=3000;INDEVICE:=5;LISTING:='FALSE';
GRAMMAR:        ROUTE:=3;INTERNAL CODE LISTING:='FALSE';
                PERFORMANCE TESTING:='FALSE';

'begin' <program>; student grammar defining a simple programming
      language;
<program> ::=<declaration list>";"'B1'<statement list>"#"'B1'(/*
     END./)'W';
<declaration list> ::= <declaration>*(";"<declaration>'c');
<declaration> ::="("<variable list>")"    'S8C';
<variable list> ::=<variable>(/*VAR,/)'xc'*(","<variable>(/*VAR,/)'xcc');
```

```
<statement list> ::= <statement>*(";"<statement>'c')(/*HLT,/)'c';
<statement> ::=<label list><basic statement>'cBlc';
<label list> ::='0'*(<label>":"(/*LAB,/)'xccM1')'T1'('Blc');
<label> ::=<identifier>;
<basic statement> ::=<assign stmt>|<go to stmt>|<if stmt>;
<assign stmt> ::=<variable>"="<expression>(/*STO,/)'cc';
<go to stmt> ::= "GO TO"<label>(/*TRA,/)'c';
<if stmt> ::= "IF"<expression>"="<condition>", go to"<label>'xcc';
<condition> ::= "0"(/*TZE,/) | "+"(/*TPL,/) | "-"(/*TMI,/);
<expression> ::=<term>*("+"<term>(/*ADD,/)'cc' | "-"<term>(/*SUB,/)'cc')
     |"+"<expression>|"-"<expression>(/*NEG,/)'c';
<term> ::=<factor>*("*"<factor>(/*MUL,/)'cc'|"/"<factor>(/*DIV,/)'cc');
<factor> ::=<primary>*("**"<primary>(/*EXP,/)'cc');
<primary> ::=<variable>(/*CLA,/)'c'|<number>|"("<expression>")"
     |"/"<expression>"/"(/*ABS,/)'c';
<variable> ::=<identifier>;
<number> ::=(<integer>(<fraction>'c'|)|<fraction>)(/,/)'c';
<identifier> ::=<letter>'K'*(<letter>'Kc'|<digit>'Kc')(/,/'c';
<fraction> ::="."'K'<integer>'c';
<integer> ::=<digit>'K'*(<digit>'Kc');
<digit> ::="0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
<letter> ::="A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"
     |"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"
'end'$
```

```
SOURCE:        ROUTE:=1;OBJECT CODE LISTING:='TRUE';ON DEVICE:=1;
               PERFORMANCE TESTING:='TRUE';

    (A,B,C);
    B = (A + 1)/2;
S1: T = B;
    B = B + (A/B - B)/2;
    IF /B - T/ - 0.0001 = +, GO TO S1#
    $
```

N.B. Note that the example source program computes the square root (B) of a number A by Newton Raphson technique, iterating until the incremental change in the computed square root is less than 0.0001.

c) Output:- Output on the printer (DEVICE=1) would be a copy of the job deck submitted by the student (control parameters, grammar and source program), followed by:

1. Relative Measure of Performance:-

   Number of attempts to recognize a nonterminal type:=353;

   Number of attempts to recognize a terminal type:=1412;

2. Then on a new page would appear the semantic output, that is, the generated assembler program for the stack computer.

```
*VAR,A,*VAR,B,*VAR,T
B,A,*CLA,1,*ADD,2,*DIV,*STO,
*LAB,S1,
T,B,*CLA,*STO,
B,B,*CLA,A,*CLA,B,*CLA,*DIV,B,*CLA,*SUB,2,*DIV,*ADD,*STO,
B,*CLA,T,*CLA,*SUB,*ABS,0.0001,*SUB,S1,*TPL,
*HLT
*END
```

This target program could now be assembled by the target machine and executed to find the square root B of A to a precision of 0.0001.

4.5 EXTENDED USAGE OF THE COMPILER MACHINE

This chapter has described how the student may design a computer language for the compiler machine and compile a program in this language.

The more advanced student may wish to design first his own meta-language to allow more sophisticated grammars. This is very simply done by describing his new grammar in terms of the standard one, and storing the output on disk as the new initiation data. However, this demands a firm knowledge of the compiler machine internals and thus the reading of Chapter II.

C H A P T E R V

CONCLUSIONS

This chapter reviews the compiler machine with respect to its merits, limitations, and future possibilities.

5.1 EVALUATION OF THE COMPILER MACHINE

1. Implementation:- The compiler machine has been implemented as a fully working system.

2. Concept:- Within the global realm of generalized compilers, the basic design principle of the compiler machine is conceptually attractive. For generality, the machine consists of a sequence of two processors, the output of one being the input to the second. Its functioning entails a hierarchical recycling of output back into itself. Thus, the same machine may be used successively to compile the grammar describing a source language and then a program in that language, or, a grammar defining a grammar, the new grammar, and finally the language text. Conceptually sophisticated, functionally simple, yet displaying the generality of a sequential processor.

3. Success as a Compiler:-

a) Scope of Input - The class of languages that the machine is capable of compiling is equivalent to the machine's power of syntactic description and parsing. It was de-

signed to accept the broad base of context free phrase structure languages, and cannot realize parsing economics gained from such specialized classes of languages describable by precedence grammars (13, 15, 19) or LR(k) grammars (17).

b) Scope of Output - The range of output corresponds to the machine's power of semantic description and is the major weakness of the compiler. Although formal semantic theory is not currently at a well-developed level, and the semantics described on this machine appear reasonable, the allowable output of the compiler machine is unsatisfactory for students of Computer Science. The machine is a reasonable editor and can output object code for stack computers, but lacks the ability to generate code for the entire class of non-stack Princeton computers, other than itself.

4. Ease of Use:- Facility to students is provided through:

a) Device Control - Input and output is mostly device independent, the I/O unit being simply designated by the control parameters. This is particularly helpful in generating and utilizing initialization data on disk.

b) Minimal Storage Requirements - Depending on their size of application and possible memory restrictions, students may designate STORESIZE.

c) Few Parameters - The control parameters required before each input text are minimal and easily learned.

d) Free Format Input - Since the I/O is handled by ALGOL routines, the input to the compiler machine is regarded as a stream, blanks ignored. The only rule is that numbers (required for the initial assembly program) need be separated by a non-numeric character, such as ";".

e) Universality - Implemented in ALGOL, the compiler machine is widely acceptable, although the nine I/O control routines may necessitate local modification. The choice of an extended BNF meta-language should, for educational purposes, be representative of the class of context free formal grammars usually dealt with in Computer Science.

f) Educational Tool for Students - The purpose of this thesis was to provide students of Computer Science with a facility for practical experimentation in the definition and manipulation of simple formal languages. Languages described by specialized grammars such as precedence and LR(k) were not considered.

The compiler machine's practical success corresponds to its general success as a compiler. It provides a simple, easy-to-use, but flexible system for syntactic description and efficiency experimentation. With semantics, however, it favours specification simplicity at the expense of output capability, particularly the faculty for generation of object code for computers of the Princeton class, i.e. computers whose accumulators are individually accessed by instructions and not considered as a stack.

As an added educational benefit, it does provide an example for the concepts of meta-grammars, grammars, meta-languages, source languages, machine languages for stack computers, and design of a generalized compiler.

5.2 FUTURE EXTENSIONS

Presently, the student has a translator with editing features, capable of defining a type of transfer function between a pair of context free phrase structure languages. The class of input language could be extended to include context bound languages. The compiler could be tailored, or more likely, augmented to recognize such important language classes as those described by precedence and LR(k) grammars and consideration should be given to extend the allowable class of output languages.

For precedence and LR(k) languages additional parsing facilities are necessary for their recognition and inherent parsing efficiencies. The parsing machine would need alignment to utilize new (e.g. precedence) functions.

For other languages, additional output and editing facilities are necessary. A dictionary entry and look-up facility would be very useful in general programming language translation. The "mark"ing control deserves extension to more than one level. The ability to generate internal symbolic labels

is necessary in most programming languages. Rescanning of the input text would facilitate improved semantics. These and other features can be implemented through the use of additional edit codes and further extensions to the editor. In fact, it might prove useful to have several editors, since many features may be dependent on the target language required. An editor could take the duties of target code assembly into absolute machine form. These approaches would, specifically, allow the highly desirable generation of Princeton computer object code. The inherent cost to a student, however, is a very individual complex meta-semantic language, and for purposes of education this undesirable consequence should be considered.

A P P E N D I X A

INSTRUCTION SET FOR PARSER MACHINE

A P P E N D I X A

INSTRUCTION SET FOR PARSOR MACHINE

OPERATION CODE		OPERAND	MNEMONIC IDENTIFIER	META-SYMBOL
Mnenonic	Internal Code			
CALL	1	address or 0		
RETURN	2	0		
TRUE	3	address		
FALSE	4	address		
FLAG	5	address		
NOT	6	0		
MARK	7	integer[1,32]		
SELECT	8	integer[1,32]		
TEST	9	address		
REFERENCE	10	1	SETLAB	'R1'
		2	INCLAB	'R2'
		3	LABEL	'R3'
		4	DECLAB	'R4'
		5	RESETLAB	'R5'
MATCH	11			
NEXT	12			
COPY	13			'K'
NULL	14			'O'
EDIT	15	1	WRITE	'W'
		2	CONCATENATE	'C'
		3	EXCHANGE	'X'
		4	NEGATE	'N'
		5	VALUE	'V'
PRINT	16	character code		(/ /)
SPACE	17	positive integer		'S' integer
TAB	18	positive integer		'B' integer
STOP	19	0		

NOTE: The code printed by the EDIT instruction is the negated operand part.

A P P E N D I X B

GRAMMAR FOR META-LANGUAGE

```
CONTROL PARAMETERS :
ROUTE := 3; OBJECT CODE LISTING := 'FALSE'; PERFORMANCE TESTING := 'FALSE';

GRAMMAR

'BEGIN' <GRAMMAR>;      GRAMMAR FOR METALANGUAGE : A DEFINITION ;
<GRAMMAR> ::= "'BEGIN'" <SUBJECT>
        *(";" (<DEFINITION>|<COMMENT>)) "'END'" (/(999)/) 'W';
<SUBJECT> ::= (/(1)/) <VARIABLE> (/(19)(0)(0)/) 'W';
<COMMENT> ::= *¬(";" | "'END'");
<DEFINITION> ::= 'R1' <VARIABLE> ":=" <EXPRESSION> (/(2)(0)(0)/) 'WR5';
<VARIABLE> ::= "<" 'JK' *(¬">" 'KC') ">" 'L';
<EXPRESSION> ::= 'R2' ("'R1'" (/(10)(1)/) | "'R2'" (/(10)(2)/) | 'O')
        <ALTERNATIVE>'C' *("|" (/(3)/)'R3CCM1'<ALTERNATIVE>'C') 'T1' ('R3C')
        ("'R4'" (/(10)(4)/)'C' | "'R5'" (/(10)(5)/)'C' | ) 'R4' ;
<ALTERNATIVE> ::= 'R2O' *(<PASSIVE>'C')
        (<ACTIVE>'C' *('O' *(<PASSIVE>'C') <ACTIVE>'C' (/(4)/) 'R3CXCCM1')
        'O' *(<PASSIVE>'CM1M2') 'T2' ((/(4)/)'R3CXC') 'CT1' ('R3C') | ) 'R4';
<ACTIVE> ::= <VARIABLE> (/(1)/)'XC' | <CONSTANT>
        | "(" (/(1)(0)/) <EXPRESSION>'C' ")" (/(2)(0)/) 'C'
        | "¬" (/(1)(0)/) <ACTIVE>'C' (/(6)(0)/)'C';
<PASSIVE> ::= 'R2' "*" 'R3' <ACTIVE>'C'(/(5)/)'R3CC'
    | "(/"( ("("<INTEGER>")"|¬"/)"'K') (/(16)/)'XC'
        *(("("<INTEGER>")"|¬"/)"'K') (/(16)/)'XCC'(/(15)(2)/)'C') |'O') "/)"
    | <CODE> (/(15)/)'XC' | "'O'" (/(14)(0)/) | "'K'" (/(13)(0)/)
    | "'U'"(/(12)(0)/) | "'R3'"(/(10)(3)/) | "'M'"(/(7)/)<PRIME INTEGER>'C'
    | "'T'"(/(8)/)<PRIME INTEGER>'C'(/(9)/)'R3CC' "(" *(<PASSIVE>'C') ")" 'R3C'
        |  ("'S'"(/(17)/) | "'B'"(/(18)/))<PRIME INTEGER>'C' 'R4' ;
<CONSTANT> ::= 'R2'  """ (/(11)/)'UKC'
        *(¬""" (/(4)/)'R3C' (/(11)/)'KCCCM1') """ 'T1'('R3C')  'R4';
<CODE> ::= "'W'" (/(1)/) | "'C'" (/(2)/) | "'X'" (/(3)/) | "'N'" (/(4)/)
        | "'L'" (/(5)/) | "'V'" (/(6)/);
<PRIME INTEGER> ::= <PRIME DIGIT>*(<PRIME DIGIT>'C')'V';
<PRIME DIGIT> ::= "'1'"(/1/) | "'2'"(/2/) | "'3'"(/3/) | "'4'"(/4/) |"'5'"(/5/)
                  "'6'"(/6/) | "'7'"(/7/) | "'8'"(/8/) | "'9'"(/9/) |"'0'"(/0/);
<INTEGER> ::= <DIGIT>'K' *(<DIGIT>'KC') 'V';
<DIGIT> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
'END'  $
```

A P P E N D I X C

CHARACTER CODES

CHARACTER		INTERNAL CODING	CHARACTER		INTERNAL CODING
	blank	ignored	-	minus	80
$	dollar sign	-1	+	plus	81
'	prime	36	_	underscore	82
(	left parenthesis	37	¢		83
/	slash	38	|	or sign	84
)	right parenthesis	39	&	ampersand	85
0	numbers	40	>	right angle bracket	86
1		41	:	colon	87
2		42	;	semi-colon	88
⋮			—	not sign	89
9		49	?	question mark	90
A	letters	50	"	quotation mark	91
B		51	=	equal sign	92
C		52	@	at sign	93
D		53	%	percent sign	94
⋮			*	asterisk	95
Z		75	<	left angle bracket	96
#	number sign	76	(/		97
,	comma	77	/)		98
.	period	79			

Quoted characters:

To any characters appearing within a pair of primes ('), add 100 to its normal internal coding, except '$' which is 178.

APPENDIX D

ALGOL I/O ROUTINES

```
 SC     SOURCE STATEMENT

00000   'BEGIN' 'COMMENT'**************************************************************
00000   ******************************************************************************
00000
00000                        A SET OF BASIC INPUT-OUTPUT PROCEDURES
00000                                    FOR ALGOL
00000
00000              IMPLIMENTED BY - K. CARPENTER, UNIVERSITY OF MANITOBA, 1969.
00000   FORWARD :
00000       WITH IBM/360 ALL DATA IS GROUPED INTO 'DATA SETS'.  IN AN ALGOL
00000   PROGRAM THESE ARE UNIQUELY NUMBERED 0-15.  0 AND 1 ARE RESERVED FOR
00000   READER AND PRINTER RESPECTIVELY.  THE PROGRAMMER MAY SPECIFY 3-15 TO
00000   REPRESENT OTHER DATA SETS ,IE: DISK,TAPE,OR CARDS, PROVIDING HE SUPPLIES
00000   A CORRESPONDING 'DD' JOB CONTROL STATEMENT.
00000       EACH OF THE FOLLOWING I/O ROUTINES MUST EXPLICITLY OR IMPLICITLY
00000   REFER TO A DATA SET (REPRESENTED BY "CHANNEL" IN THE DEFINITIONS).  SOME
00000   I/O ROUTINES HAVE "CHANNEL" AS A FORMAL PARAMETER TO BE SUPPLIED BY THE
00000   PROGRAMMER.  THE OTHERS REFER IMPLICITLY TO A DATA SET BY THE CURRENT VALUE
00000   OF "INCHANNEL" OR "OUTCHANNEL"(DEPENDING ON IF THEY ARE INPUT OR OUTPUT
00000   PROCEDURES).  0 AND 1 ARE ASSUMED TO BE THE THE STANDARD VALUES FOR
00000   "INCHANNEL" AND "OUTCHANNEL" BUT THEY MAY BE ASSIGNED NEW VALUES (2-15)
00000   BY THE PROGRAMMER (EG: OUTCHANNEL := 3 ).
00000       THE I/O PROCEDURES ARE CONCATENATED TO THE FRONT OF THE USERS SOURCE
00000   DECK AS A GLOBAL BLOCK TO HIS PROGRAM.  AN EXTRA 'END' IS APPENDED TO THE
00000   USERS SOURCE DECK TO COMPLETE THIS BLOCK.  THE ROUTINES REQUIRE 2.3 SECONDS
00000   OF COMPILE TIME AND 18 K BYTES OF CORE STORAGE.
00000
00000
00000   I  PRIMITIVES :
00000                          IN SYMBOL  (CHANNEL,STRING,I)
00000                          OUT SYMBOL (CHANNEL,STRING,I)
00000                          OUT STRING (CHANNEL,STRING)
00000
00000   II INPUT :
00000
00000      1 *BASIC PROCEDURES :                          ( DEFINED BY IBM FOR 360 )
00000                          IN REAL     (CHANNEL,R)
00000                          IN INTEGER  (CHANNEL,I)
00000                          IN BOOLEAN  (CHANNEL,B)
00000                          IN ARRAY    (CHANNEL,A)
00000                          INTARRAY    (CHANNEL,IA)
00000                          INBARRAY    (CHANNEL,BA)
00000           SEQUENTIALLY SCAN THE SPECIFIED DATA SET UNTIL FINDING THE FIRST
00000           SYNTACTICALLY CORRECT ALGOL NUMBER. THE NUMBER IS THEN DELIMITED
00000           BY 2 OR MORE BLANKS, A RECORD END OR THE FIRST SYNTACTICALLY
00000           INCORRECT CHARACTER. IT'S VALUE IS ASSIGNED TO THE 2ND PARAMETER.
00000
```

```
SC      SOURCE STATEMENT

00000       2 *DERIVED PROCEDURES*                              ( THE BERKLEY STYLE )
00000                         READ R
00000                         READ I
00000                         READ B
00000           ARE FUNCTION DESIGNATORS WITHOUT PARAMETERS WHICH CAN BE USED TO
00000           INPUT RESPECTIVELY A REAL, AN INTEGER OR A BOOLEAN FROM
00000           "IN CHANNEL".  IF "IN CHANNEL" IS NOT THE CARD READER ( 0 ), THEN
00000           THE INTEGER VALUE FOR THE APPROPRIATE INPUT DEVICE MUST BE
00000           EXPLICITLY ASSIGNED TO "IN CHANNEL".
00000
00000   III
00000     OUTPUT :                                              ( DEFINED BY BERKLEY )
00000
00000       1 *BASIC PROCEDURES*
00000                         OUT REAL        (CHANNEL,R)
00000                         OUT INTEGER     (CHANNEL,I)
00000                         OUT BOOLEAN     (CHANNEL,B)
00000           TRANSFER A REAL, AN INTEGER OR A BOOLEAN TO THE OUTPUT DEVICE
00000           "CHANNEL".  CALL OF THESE OUTPUT PROCEDURES MUST BE PRECEDED BY A
00000           CALL TO THEIR CORRESPONDING PROCEDURES -
00000                         REAL FORMAT     (BOOL,W,D)
00000                         INTEGER FORMAT (W)
00000                         BOOLEAN FORMAT (W)
00000                         WHERE    BOOL  - 'TRUE' IF FIXED POINT FORMAT IS DESIRED
00000                                          'FALSE' IF FLOATING POINT FORMAT IS DESIRED
00000                                  W     - FIELD WIDTH
00000                                  D     - NUMBER OF DIGITS AFTER THE DECIMAL POINT.
00000                         ARE FORMAT PARAMETERS.
00000
00000       2 *DERIVED PROCEDURES*
00000           THE FOLLOWING PROCEDURES DO NOT REQUIRE EXPLICIT CALLS OF THE
00000           FORMAT PROCEDURES.  IF THE CARDREADER AND LINE PRINTER ARE NOT THE
00000           INPUT AND OUTPUT DEVICES IN USE, THEN THE APPROPRIATE DEVICE NUMBERS
00000           MUST BE EXPLICITLY ASSIGNED TO "IN CHANNEL" AND "OUT CHANNEL"
00000           RESPECTIVELY.  OTHERWISE I/O IS ASSUMED TO BE ON READER AND PRINTER.
00000
00000                         IO R        (R,'('VAR NAME')',BOOL,W,D)
00000                         IO I        (I,'('VAR NAME')',W)
00000                         IO B        (B,'('VAR NAME')',W)
00000           ARE FUNCTION DESIGNATORS AND
00000                         IO A        (A,L,U,'('ARRAY NAME')',BOOL,W,D)
00000           IS A PROCEDURE STATEMENT.  THEY INPUT RESPECTIVELY A REAL, AN
00000           INTEGER, A BOOLEAN OR A REAL 1-DIM ARRAY AND ECHO THE INPUT TO AN
00000           EQUIVALENT ALGOL OUTPUT STATEMENT.  THE 1ST PARAMETER IS ASSIGNED
00000           THE INPUT VALUE.  THE 2ND PARAMITER IS A STRING TO BE OUTPUTTED.
00000           THE OTHERS GIVE THE FORMAT PARAMETERS.  FOR "IOA", THE 2ND AND 3RD
00000           PARAMATERS ARE THE 1ST AND LAST SUBSCRIPTS OF ELEMENTS IN THE
00000           ARRAY TO BE READ.  ITS LAST PARAMETERS GIVE THE STRING TO BE
```

```
 SC     SOURCE STATEMENT

00000          OUTPUTTED AND THE FORMAT.
00000          EXAMPLES:
00000          IOR (TIME, '('TIME IN MINUTES')', 'TRUE', 5, 2)
00000          IOA (HIPPOPOTOMUS, 1, 3, '('HIPPOPOTOMUS')', 'TRUE', 4, 1)
00000          WOULD OUTPUT WITH APPROPRIATE INPUT:
00000          TIME IN MINUTES := 21.05;                              'COMMENT'
00000          I := 1, 'FOR' HIPPOPOTOMUS (/I/) := 15.1, 6.2, 7.0 'DO' I := I + 1;
00000
00000                          OT I        (I,'('VAR NAME')',W)
00000                          OT R        (R,'('VAR NAME')',BOOL,W,D)
00000                          OT B        (B,'('VAR NAME')',W)
00000                          OT A        (A,L,U,'('ARRAY NAME')',BOOL,W,D)
00000          ARE FOR OUTPUT ONLY. THE FORM OF OUTPUT IS IDENTICAL TO THAT OF
00000          "IOI",IOR","IOB","IOA".
00000
00000                          OUTR        (R,BOOL,W,D)
00000                          OUTI        (I,W)
00000                          OUTB        (B,W)
00000                          OUTA        (A,L,U,BOOL,W,D)
00000                          OUTIA       (IA,L,U,W)
00000                          OUTBA       (BA,L,U,W)
00000          ARE FOR OUTPUT ONLY. THEIR FORMAT INFO IS PROVIDED BY THE PARAMETERS.
00000
00000
00000      3                   OUTST       (STRING)
00000          OUTPUTS A STRING ONTO "OUTCHANNEL".
00000
00000      4                   COPYTEXT
00000          SEQUENTIALLY INPUTS CHARACTERS ON "INCHANNEL" AND COPIES THEM ONTO
00000          "OUTCHANNEL" UNTIL FINDING A SEMICOLON.
00000
00000
00000
00000
00000
00000                              INPUT/OUTPUT CONTROL PROCEDURES
00000                                        FOR ALGOL
00000
00000    FORWARD :    ***
00000        A DATA SET IS OPENED BY "OPEN(CHANNEL)", WHERE "CHANNEL" IS AN INTEGER
00000    X,  0<=X<=15.  ANY SETTING OF RECORD SIZE OR BLOCK SIZE (IE: LINESIZE AND
00000    PAGESIZE ON THE PRINTER) MUST BE DONE BEFORE THIS TIME.  ALL OTHER I/O
00000    COMMANDS MUST FOLLOW AN OPEN STATEMENT.
00000        IF RECORD SIZE AND BLOCK SIZE ARE NOT SPECIFIED, A DATA SET ASSUMES A
00000    STANDARD RECORD SIZE OF 80 AND IS NOT BLOCKED (IE: BLOCKSIZE:=0).  THE
00000    PRINTER (OUTCHANNEL:=1), HOWEVER, IS ASSIGNED A DEFAULT RECORD SIZE OF
00000    120 AND A BLOCKSIZE OF 64.
00000        ALL DATA SETS ARE AUTOMATICALLY CLOSED UPON PROGRAM TERMINATION.
```

```
 SC     SOURCE STATEMENT

00000   I  FORMAL OPENING AND CLOSING OF DATA SETS
00000
00000                        OPEN ( CHANNEL )
00000       THE DATA SET ASSOC. W. "CHANNEL" IS LOGICALLY CONNECTED TO THE ALGOL
00000       PROGRAM AND THE RECORD POINTER POINTS TO THE FIRST RECORD IN IT.
00000
00000                        CLOSE ( CHANNEL )
00000       CLOSES THE DATA SET ASSOC. W. "CHANNEL".  IE: LOGICALLY DISCONNECTS
00000       IT FROM THE ALGOL PROGRAM.
00000
00000   II INPUT CONTROL                              --  ON DATA SET "INCHANNEL"
00000
00000                        NEWCARD
00000       FETCHES THE NEXT INPUT RECORD.  THE CHARACTER POINTER IS SET TO THE
00000       FIRST POSITION.
00000
00000   III  OUTPUT CONTROL                           --  ON DATA SET "OUTCHANNEL"
00000
00000                        LINESIZE ( N )
00000       DEFINES THE RECORDS IN THE DATA SET TO BE "N" CHARACTERS LONG.
00000       MUST BE SPECIFIED WHILE DATA SET "OUTCHANNEL" IS CLOSED.
00000
00000                        PAGESIZE ( N )
00000       SPLITS THE DATA SET INTO BLOCKS OF "N" RECORDS.  IF "OUTCHANNEL":=1,
00000       EACH BLOCK APPEARS UPON A PAGE OF THE PRINTER.   MUST BE SPECIFIED
00000       WHILE DATA SET "OUTCHANNEL" IS CLOSED.
00000
00000                        SPACE ( N )
00000       SKIPS "N" CHARACTER POSITIONS, FILLING THEM WITH BLANKS.  CHARACTER
00000       POSITIONS MAY BE SKIPPED WITHIN THE CURRENT OUTPUT RECORD, ACROSS
00000       RECORD BOUNDARIES, AND ACROSS BLOCKS.  IF "N"<0 THEN "N" IS ASSUMED
00000       TO BE 0.
00000
00000                        TAB ( N)
00000       SKIPS TO COLUMN "N".  MORE SPECIFICALLY :
00000       IF "N">= CHARACTER POINTER, THE CHARACTER POINTER SKIPS TO "N".
00000       IF "N" < CHARACTER POINTER, A NEW RECORD IS FETCHED AND THE CHARACTER
00000       POINTER IS SET TO "N".  ALL SKIPPED SPACES ARE FILLED WITH BLANKS.
00000
00000                        NEWLINE ( N )
00000       SKIPS TO THE NEXT RECORD "N" TIMES.  IF "N"<=0, IT HAS NO EFFECT.
00000
00000                        NEWPAGE
00000       SKIPS TO THE FIRST RECORD IN THE NEXT BLOCK (PAGE).
00000   ***************************************************************************
00000       THIS COMPLETES THE WRITTEN DESCRIPTION OF THE I/O FACILITIES.  THE
00000       ACTUAL ALGOL DEFINITION OF THESE ROUTINES FOLLOWS ON THE NEXT PAGE.
00000   ***************************************************************************
```

```
SC    SOURCE STATEMENT

00000 ************************************************************************
00000
00000                    DEFINITION OF  THE CONTROL  PROCEDURES ;
00000
00000 'PROCEDURE'  OPEN(CHANNEL);
00001 'VALUE' CHANNEL;  'INTEGER' CHANNEL;  SYSACT(CHANNEL, 12, 1);
00004
00004 'PROCEDURE'  CLOSE (CHANNEL);
00005 'VALUE' CHANNEL;  'INTEGER' CHANNEL;  SYSACT(CHANNEL, 12, 0);
00008
00008 'PROCEDURE'  NEW CARD;  SYSACT(INCHANNEL, 14, 1);
00010
00010 'PROCEDURE' LINESIZE(N);
00011 'VALUE' N;  'INTEGER' N;  SYSACT(OUTCHANNEL, 6, N);
00014
00014 'PROCEDURE' PAGESIZE(N);
00015 'VALUE' N;  'INTEGER' N;  SYSACT(OUTCHANNEL, 8, N);
00018
00018 'PROCEDURE' SPACE(NS);  'VALUE' NS;  'INTEGER' NS;
00021 'IF' NS>0 'THEN'
00021 'BEGIN' 'INTEGER' CHPTR, RECLTH, NL, CH;
00022    SYSACT(OUTCHANNEL, 1, CHPTR);  SYSACT(OUTCHANNEL, 5, RECLTH);
00024    CH:= CHPTR+NS ;
00025    'IF' CH<=RECLTH 'THEN' SYSACT(OUTCHANNEL, 2, CH) 'ELSE'
00025    'BEGIN' CH:= CH-1;  NL:= CH'/'RECLTH;  CH:= CH-NL*RECLTH;
00028        NEWLINES(NL);  'IF' CH>0 'THEN' SYSACT(OUTCHANNEL,2, CH)
00029    'END'
00029 'END' OF SPACE ;
00030
00030 'PROCEDURE' TAB(N);  'VALUE' N;  'INTEGER' N;
00033 'BEGIN' 'INTEGER' CHPTR;  SYSACT(OUTCHANNEL, 1, CHPTR);
00035    'IF' N¬=CHPTR 'THEN' SYSACT(OUTCHANNEL, 2, N)
00035 'END' OF TAB;
00036
00036 'PROCEDURE' NEWLINES(NL);  'VALUE' NL;  'INTEGER' NL;
00039 'IF' NL>0 'THEN'
00039 'BEGIN' 'INTEGER' RECPTR, BLKSIZE, NB, REC, K;
00040    SYSACT(OUTCHANNEL, 3, RECPTR);  SYSACT(OUTCHANNEL, 7, BLKSIZE);
00042    'IF' BLKSIZE=0 | RECPTR+NL<=BLKSIZE 'THEN' SYSACT(OUTCHANNEL, 14, NL)
00042    'ELSE'
00042    'BEGIN' REC:=RECPTR+NL-1;  NB:= REC'/'BLKSIZE;  REC:= REC-NB*BLKSIZE;
00045        'FOR' K:=1 'STEP' 1 'UNTIL' NB 'DO' SYSACT(OUTCHANNEL, 15, 1);
00046        'IF' REC>0 'THEN' SYSACT(OUTCHANNEL, 14, REC)
00046    'END'
00046 'END' OF NEWLINES;
00047
00047 'PROCEDURE' NEWPAGE;  SYSACT(OUTCHANNEL, 15, 1);
00049
```

```
SC    SOURCE STATEMENT
00049 'COMMENT'*************************************************************************
00049
00049                        DEFINITION OF THE BERKLEY SYSTEM
00049                                    OF
00049                                INPUT-OUTPUT
00049                        *********************************
00049
00049
00049                                ASSUMED TO BE GIVEN
00049                                *******************
00049
00049                         IN SYMBOL      (CHANNEL, STRING, I)
00049                         OUT SYMBOL     (CHANNEL, STRING, I)
00049                         OUT STRING     (CHANNEL, STRING, I)
00049                         IN REAL        (CHANNEL, R)
00049                         IN INTEGER     (CHANNEL, I)
00049                         IN BOOLEAN     (CHANNEL, B)
00049                         INIARRAY       (CHANNEL, IA)
00049                         INBARRAY       (CHANNEL, BA)
00049                         IN ARRAY       (CHANNEL, A)
00049 *************************************************************************
00049
00049                        NONLOCAL COMMUNICATION PARAMETERS
00049                        *********************************;
00049
00049 'INTEGER' IN CHANNEL, OUT CHANNEL, RFW, RD, IFW, BFW;
00050 'BOOLEAN' RBOOL;
00051
00051 'COMMENT'*************************************************************************
00051
00051                              3  PRIMITIVE PROCEDURES
00051                              ************************;
00051                                                                        ****   1
00051 'PROCEDURE'  ENOUGH ROOM  (CHANNEL, FW);
00052       'VALUE' CHANNEL, FW;  'INTEGER' CHANNEL, FW;
00054 'COMMENT' ENSURES THAT THE OUTPUT DEVICE "CHANNEL", TO WHICH DATA IS
00054   TO BE TRANSMITTED, IS IN THE OPEN STATE AND THAT THERE IS ENOUGH ROOM
00054   LEFT IN THE CURRENT RECORD TO PLACE "FW" (FIELD WIDTH) CHARACTERS. A
00054   NEW RECORD IS FETCHED IF NECESSARY;
00054 'BEGIN'
00054       'INTEGER' CH PTR, REC LTH;
00055       SYSACT(CHANNEL, 12, 1);
00056       SYSACT(CHANNEL, 1, CH PTR);  SYSACT(CHANNEL, 5, REC LTH);
00058       'IF' REC LTH - CH PTR + 1 <  FW 'THEN' SYSACT(CHANNEL, 14, 1)
00058 'END'  ENOUGH ROOM;
00059 'COMMENT' THIS PROCEDURE IS APPROPRIATE TO IBM/360 AND WAS NOT SPECIFIED
00059   IN THE BERKLEY STYLE SPECIFICATIONS;
00059
```

```
 SC     SOURCE STATEMENT

00059                                                                    ****   2
00059   'PROCEDURE'  DCMP INTEGER  (I, NEGATIVE, N OF DIGITS, DIGIT);
00060          'VALUE' I;  'INTEGER' I, N OF DIGITS;  'BOOLEAN' NEGATIVE;
00063          'INTEGER''ARRAY' DIGIT;
00064   'COMMENT' "NEGATIVE" := "I"<0, "N OF DIGITS" := THE NUMBER OF DECIMAL
00064     DIGITS OF "I" (IF "I"=0 THEN "N OF DIGITS" := 0), "DIGIT"(/0:"N OF DIGITS"-1/)
00064     := THE DECIMAL DIGITS OF "I" STARTING FROM THE RIGHT;
00064   'BEGIN'
00064          'INTEGER' J;
00065          'IF' I<0 'THEN'  'BEGIN' NEGATIVE := 'TRUE';  I := -I 'END'
00066                   'ELSE'  NEGATIVE := 'FALSE';
00067          N OF DIGITS := 0;
00068        L:
00068          'IF' I > 0 'THEN'
00068          'BEGIN'
00068                  J := I '/' 10;  DIGIT(/N OF DIGITS/) := I - J * 10;
00070                  N OF DIGITS := N OF DIGITS + 1;  I := J;  'GO TO' L
00072          'END'
00072   'END'  DECOMPOSE INTEGER;
00073
00073                                                                    ****   3
00073   'PROCEDURE'  DCMP REAL (R,MAX N OF DIGITS,NEGATIVE,SIZE,EXPONENT,DIGIT);
00074          'VALUE' R;  'INTEGER' MAX N OF DIGITS,SIZE,EXPONENT;  'REAL' R;
00077          'BOOLEAN' NEGATIVE;  'INTEGER''ARRAY' DIGIT;
00079   'COMMENT' "NEGATIVE" := R<0, "SIZE" := -1 IF "R" IS TOO SMALL, I.E. IS
00079     SUCH THAT WHEN  ABS("R")  IS MULTIPLIED REPEATEDLY BY 10 IT DOES NOT
00079     BECOME EVENTUALLY LARGER THAN ONE, "SIZE" := 0 OTHERWIZE,  "EXPONENT"
00079     := THE POWER OF 10 BY WHICH "R" IS TO BE DEVIDED TO OBTAIN A NUMBER
00079     WHOSE FIRST SIGNIFIGANT DIGIT IS IMMEDIATELY TO THE LEFT OF THE DECIMAL
00079     POINT, "DIGIT"(/0:"MAX N OF DIGITS" - 1/) := THE DECIMAL DIGITS OF "R"
00079     STARTING WITH THE FIRST SIGNIFIGANT DIGIT TO THE LEFT;
00079   'BEGIN'
00079          'INTEGER' I,K,M;
00080          'BOOLEAN' 'PROCEDURE'  TOO SMALL(R);  'REAL' R;
00082          TOO SMALL := ABS(R) < 2.7'-78;
00083          'COMMENT'  THIS PROCEDURE APPROPRIATE TO IBM/360;
00083          NEGATIVE := 'FALSE';
00084          'IF' TOO SMALL (R) 'THEN'
00084          'BEGIN' SIZE := -1;  'GOTO' END DECOMPOSE 'END' 'ELSE' SIZE := 0;
00086          'IF' R < 0 'THEN'  'BEGIN' NEGATIVE := 'TRUE';  R := -R 'END';
00088          'IF' R < 1 'THEN'
00088          'BEGIN'
00088                  EXPONENT := -1;
00089        SCALE UP: R := R * 10;
00090                  'IF' R < 1 'THEN'
00090                  'BEGIN' EXPONENT := EXPONENT - 1;  'GO TO' SCALE UP 'END'
00091          'END'
00091          'ELSE'
```

```
SC      SOURCE STATEMENT

00091           'BEGIN'
00091                   EXPONENT := 0;
00092               TEST:
00092                   'IF' R >= 10 'THEN'
00092                       'BEGIN'
00092                               EXPONENT:= EXPONENT + 1;  R := R * 0.1;
00094                               'GO TO' TEST
00094                       'END';
00095           'END';
00096           M := MAX N OF DIGITS;
00097           R := R + 5 * 0.1** M;
00098           I := ENTIER(R);
00099           'IF' I >= 10 'THEN'
00099               'BEGIN'
00099                       I := 1;  EXPONENT := EXPONENT + 1;
00101                       M := M + 1;  R := R / 10
00102               'END'
00102           'ELSE' 'IF' I = 0 'THEN' I := 1;
00103           DIGIT(/0/) := I;
00104           'FOR' K := 1 'STEP' 1 'UNTIL' M - 1 'DO'
00104               'BEGIN'
00104                       R := (R - I) * 10;  I:= ENTIER(R);
00106                       I := DIGIT(/K/) := 'IF' I <= 0 'THEN' 0 'ELSE'
00106                                          'IF' I = 10 'THEN'9 'ELSE' I
00106               'END';
00107    END DECOMPOSE:
00107   'END'  DECOMPOSE REAL;
00108
00108   'COMMENT'**************************************************************************
00108
00108                               6 BASIC OUTPUT PROCEDURES
00108                               *************************;
00108
00108                                                                          ****   1
00108   'PROCEDURE'  INTEGER FORMAT  (W);  'INTEGER' W;  IFW := W;
00111
00111                                                                          ****   2
00111   'PROCEDURE'  REAL FORMAT  (B,W,D);  'INTEGER' W,D;  'BOOLEAN' B;
00114   'BEGIN' RBOOL := B;  RFW := W;  RD := D 'END';
00117
00117                                                                          ****   3
00117   'PROCEDURE' BOOLEAN FORMAT  (W);  'INTEGER' W;  BFW := W;
00120
00120                                                                          ****   4
00120   'PROCEDURE'  OUT INTEGER  (CHANNEL, I);  'VALUE' CHANNEL, I;
00122           'INTEGER' CHANNEL, I;
00123   'COMMENT' THE BERKLEY STYLE IS CHARACTERIZED BY USE OF A FIELD WIDTH
00123     PARAMETER "FW" AND FOR REAL NUMBERS, BY THE USE OF A PARAMETER "RBOOL"
```

```
SC      SOURCE STATEMENT

00123     WHICH DECIDES IF THE FIXED POINT (VALUE 'TRUE') OR THE FLOATING POINT
00123     REPRESENTATION (VALUE 'FALSE') IS REQUESTED AND BY THE NUMBER OF DIGITS
00123     "RD" AFTER THE DECIMAL POINT. THE SIGN IS OUTPUTED JUST BEFORE THE MOST
00123     SIGNIFIGANT DIGIT, IF THE NUMBER IS NEGATIVE. IN FLOATING POINT FORM
00123     THE FIRST SIGNIFIGANT DIGIT IS IMMEDIATELY TO THE LEFT OF THE DECIMAL
00123     POINT. IF THE FIELD PARAMETER IS LESS THAN REQUIRED, THE FIELD IS FILLED
00123     WITH ASTERISKS. THESE PROCEDURES PAIR WITH THE CORRESPONDING INPUT
00123     REQUIRED;
00123   'BEGIN'
00123       'INTEGER' N OF DIGITS, J, K; 'BOOLEAN' NEGATIVE;
00125       'INTEGER''ARRAY' DIGIT(/0:19/);
00126       ENOUGH ROOM(CHANNEL, IFW);
00127       DCMP INTEGER(I, NEGATIVE, N OF DIGITS, DIGIT);
00128       'IF' N OF DIGITS = 0 'THEN'
00128       'BEGIN' N OF DIGITS := 1;  DIGIT(/0/) := 0 'END';
00130       J := N OF DIGITS + ('IF' NEGATIVE 'THEN' 1 'ELSE' 0);
00131       'IF' J > IFW 'THEN'
00131       'BEGIN' 'FOR' K:= 1 'STEP' 1 'UNTIL' IFW 'DO'
00131                 OUT STRING(CHANNEL, '('*')')  'END'
00131       'ELSE'
00131       'BEGIN'
00131            'FOR' K:= IFW - 1 'STEP' -1 'UNTIL' J 'DO'
00131              OUT STRING(CHANNEL, '(' ')');
00132            'IF' NEGATIVE 'THEN' OUT STRING(CHANNEL, '('-')');
00133            'FOR' K:= N OF DIGITS - 1 'STEP' -1 'UNTIL' 0 'DO'
00133              OUT SYMBOL(CHANNEL, '('0123456789')', DIGIT(/K/) + 1)
00133       'END'
00133   'END'  OUT INTEGER;
00134
00134                                                                    ****   5
00134   'PROCEDURE'  OUT REAL  (CHANNEL, R);  'VALUE' CHANNEL, R;
00136         'INTEGER' CHANNEL;  'REAL' R;
00138   'COMMENT' THIS PROCEDURE OUTPUTS "R" PROPERLY ROUNDED TO  "CHANNEL"
00138     USING THE BERKLEY STYLE. IN THIS VARIANT, THE EXPONENT PART IN THE
00138     FLOATING POINT FORM IS REPLACED BY 4 SPACES IF THE EXPONENT IS ZERO.
00138     THE SIGN OF THE EXPONENT IS ALWAYS OUTPUTED, FOR COMPATIBILITY WITH
00138      "IN REAL". THE EXPONENT IS RESTRICTED TO THE INTERVAL -99 TO 99;
00138   'BEGIN'
00138      'INTEGER' J, K, SIZE, EXPONENT;  'BOOLEAN' NEGATIVE;
00140      'INTEGER''ARRAY' DIGIT(/0 : RD+1+('IF' RBOOL 'THEN'
00140              ENTIER(LN(ABS(R)+1)*0.4343) 'ELSE' 0)/);
00141      'PROCEDURE' OUT DIGIT(D);  'INTEGER' D;
00143      'BEGIN' OUT SYMBOL(CHANNEL, '('0123456789')', D+1) 'END' OUT DIGIT;
00144      'PROCEDURE' ERROR ;
00145      'BEGIN'
00145          'FOR' K:= 1 'STEP' 1 'UNTIL' RFW 'DO' OUTSTRING(CHANNEL,'('*')');
00146          'GO TO' END OUT REAL
00146      'END' ERROR;
```

```
SC      SOURCE STATEMENT

00147       ENOUGH ROOM(CHANNEL, RFW);
00148       'IF' RBOOL 'THEN'
00148       'BEGIN'
00148           DCMP REAL(R, 'IF' RD+EXPONENT<-1 'THEN' 1 'ELSE' 1+
00148               RD + EXPONENT, NEGATIVE, SIZE, EXPONENT, DIGIT);
00149           'IF' SIZE = -1 'THEN'
00149           'BEGIN'
00149               EXPONENT := 'IF' RD = 0 'THEN' 0 'ELSE' -RD - 1;
00150               DIGIT(/0/) := 0
00150           'END'   ZERO
00150           'ELSE' 'IF' RD = 0 & EXPONENT < 0 'THEN'
00150           'BEGIN' EXPONENT := 0;  DIGIT(/0/) := ENTIER(ABS(R) + .5) 'END';
00152           J := ('IF' NEGATIVE 'THEN' 3 'ELSE' 2) +
00152                ('IF' RD = 0 'THEN' -1 'ELSE' RD) +
00152                ('IF' EXPONENT >= 0 'THEN' EXPONENT 'ELSE' -1);
00153           'IF' J > RFW 'THEN' ERROR 'ELSE'
00153           'BEGIN'
00153               'FOR' K := RFW-1 'STEP' -1 'UNTIL' J 'DO'
00153                   OUT STRING(CHANNEL, '(' ')');
00154               'IF' NEGATIVE 'THEN' OUT STRING (CHANNEL, '('-')');
00155               'FOR' K := 0 'STEP' 1 'UNTIL' EXPONENT 'DO'
00155                   OUT DIGIT(DIGIT(/K/));
00156               'IF' RD > 0 'THEN'
00156               'BEGIN'
00156                   OUT STRING(CHANNEL, '('.')');
00157                   'FOR' K := EXPONENT+1 'STEP' 1 'UNTIL' EXPONENT+RD 'DO'
00157                   'IF' K< 0 'THEN' OUT STRING(CHANNEL, '('0')')
00157                             'ELSE' OUT DIGIT(DIGIT(/K/))
00157               'END'   FRACTION
00157           'END'
00157       'END'   FIXED POINT REPRESENTATION
00157       'ELSE'
00157       'BEGIN'
00157           DCMP REAL(R, RD+1, NEGATIVE, SIZE, EXPONENT, DIGIT);
00158           'IF' SIZE = -1 'THEN'
00158           'BEGIN'
00158               EXPONENT := 0;
00159               'FOR' K := 0 'STEP' 1 'UNTIL' RD 'DO' DIGIT(/K/) := 0
00159           'END';
00160           J := 6 + ('IF' RD=0 'THEN' -1 'ELSE' RD) +
00160                ('IF' NEGATIVE 'THEN' 1 'ELSE' 0);
00161           'IF' J > RFW 'THEN' ERROR 'ELSE'
00161           'BEGIN'
00161               'FOR' K := RFW - 1 'STEP' -1 'UNTIL' J 'DO'
00161                   OUT STRING(CHANNEL, '(' ')');
00162               'IF' NEGATIVE 'THEN' OUT STRING(CHANNEL, '('-')');
00163               OUT DIGIT (DIGIT(/0/));
00164               'IF' RD -= 0 'THEN' OUT STRING(CHANNEL, '('.')');
```

```
SC      SOURCE STATEMENT

00165               'FOR'  K := 1 'STEP' 1 'UNTIL' RD 'DO'
00165                   OUT DIGIT(DIGIT(/K/));
00166               'IF' EXPONENT = 0 'THEN' OUT STRING(CHANNEL, '('     ')')
00166               'ELSE'
00166               'BEGIN'
00166                   OUT STRING(CHANNEL, '('''')');
00167                   'IF' EXPONENT >= 0 'THEN' OUT STRING(CHANNEL, '('+')')
00167                   'ELSE'
00167                   'BEGIN'
00167                       OUT STRING(CHANNEL, '('-')');  EXPONENT := -EXPONENT
00168                   'END';
00169                   J := EXPONENT '/' 10;
00170                   'IF' J=0 'THEN' OUT STRING(CHANNEL, '(' ')')
00170                            'ELSE' OUT DIGIT(J);
00171                   OUT DIGIT(EXPONENT-J*10)
00171               'END'
00171           'END'
00171       'END'   FLOATING POINT REPRESENTATION ;
00172    END OUT REAL :
00172   'END'  OUT REAL;
00173
00173                                                                          ****   6
00173   'PROCEDURE'  OUT BOOLEAN  (CHANNEL, B);  'VALUE' CHANNEL;
00175           'INTEGER' CHANNEL;  'BOOLEAN' B;
00177   'COMMENT' THIS PROCEDURE ASSUMES THAT 'TRUE' AND 'FALSE' TAKE RESPECT-
00177     IVELY 6 AND 7 SPACES. HOWEVER, IF THE FIELD WIDTH  IS TOO SMALL THEN
00177     THE CHARACTER "T" OR "F" WILL BE OUTPUT RIGHT JUSTIFIED IN THE FIELD.;
00177   'BEGIN'
00177           'INTEGER' I, J;
00178           ENOUGH ROOM(CHANNEL, BFW);
00179           J := 'IF' B 'THEN' 6 'ELSE' 7;
00180           'IF' J > BFW 'THEN'
00180           'BEGIN'
00180                   'FOR' I := BFW 'STEP' -1 'UNTIL' 2 'DO'
00180                       OUT STRING(CHANNEL, '(' ')');
00181                   OUT SYMBOL(CHANNEL, '('TF')', 'IF' B 'THEN' 1 'ELSE' 2)
00181           'END'
00181           'ELSE'
00181           'BEGIN'
00181                   'FOR' I := BFW -1  'STEP' -1 'UNTIL' J 'DO'
00181                       OUT STRING(CHANNEL, '(' ')');
00182                   'IF' B 'THEN' OUT STRING(CHANNEL, '(''TRUE'')')
00182                          'ELSE' OUT STRING(CHANNEL, '(''FALSE'')')
00182           'END'
00182   'END'  OUT BOOLEAN;
00183   'COMMENT' THE OUTPUTTING OF "T" AND "F" FOR FIELD WIDTHS  < 6 IS NOT A
00183     FEATURE OF THE BERKLEY SPECIFICATIONS
00183   ************************************************************************
```

```
SC      SOURCE STATEMENT

00183                                   DERIVED PROCEDURES
00183                                   ******************;
00183                                                                                   READ
00183                                                                                   ****    I
00183   'INTEGER' 'PROCEDURE'  READ I;
00184   'BEGIN'
00184           'INTEGER' I;
00185           IN INTEGER(IN CHANNEL, I);  READ I := I
00186   'END'   READ I;
00187
00187                                                                                   ****    R
00187   'REAL' 'PROCEDURE'  READ R;
00188   'BEGIN'
00189           'REAL' R;
00189           IN REAL(IN CHANNEL, R);  READ R := R
00190   'END'   READ R;
00191                                                                                   ****    B
00191   'BOOLEAN' 'PROCEDURE' READ B;
00192   'BEGIN'
00192           'BOOLEAN' B;
00193           IN BOOLEAN(IN CHANNEL, B);  READ B := B
00194   'END'   READ B;
00195
00195                                                                                   IO
00195                                                                                   ****    I
00195   'INTEGER' 'PROCEDURE'  IOI (I, S, N);  'STRING' S;  'INTEGER' I,N;
00198   'COMMENT' THIS AND THE NEXT 3 PROCEDURES INPUT RESPECTIVELY AN INTEGER,
00198     A REAL NUMBER, A BOOLEAN OR A ONE DIMENSIONAL ARRAY. THE INPUT IS
00198     ECHOED TO AN EQUIVALENT ALGOL OUTPUT STATEMENT.;
00198   'BEGIN'
00198           OUT STRING(OUT CHANNEL, S);  OUT STRING(OUT CHANNEL, '(' := ')');
00200           IOI := I := READI;
00201           INTEGER FORMAT(N);  OUT INTEGER(OUT CHANNEL, I);
00203           OUTSTRING(OUT CHANNEL,'('; ')')
00203   'END'   IOI;
00204
00204                                                                                   ****    R
00204   'REAL' 'PROCEDURE'  IOR (R, S, B, N, D);
00205           'REAL' R;  'STRING' S;  'BOOLEAN' B;  'INTEGER' N, D;
00209   'BEGIN'
00209           OUT STRING(OUT CHANNEL, S);  OUT STRING(OUT CHANNEL, '(' := ')');
00211           IOR := R := READR;
00212           REAL FORMAT(B, N, D);  OUT REAL(OUT CHANNEL, R);
00214           OUT STRING(OUT CHANNEL, '('; ')')
00214   'END'   IOR;
00215
00215                                                                                   ****    B
00215   'BOOLEAN' 'PROCEDURE' IOB(B, S, N);
```

```
SC     SOURCE STATEMENT
00216          'BOOLEAN' B;  'STRING' S;  'INTEGER' N;
00219  'BEGIN'
00219          OUT STRING(OUT CHANNEL, S);  OUT STRING(OUT CHANNEL, '(' := ')');
00221          IOB := B := READB;
00222          BOOLEAN FORMAT(N);  OUT BOOLEAN(OUT CHANNEL, B);
00224          OUT STRING(OUT CHANNEL,'('; ')')
00224  'END'   IOB;
00225                                                                    ****   A
00225  'PROCEDURE'  IOA (A, L, U, S, B, N, D);
00226          'INTEGER' L, U, N, D;   'ARRAY' A;  'STRING' S;  'BOOLEAN' B;
00230  'BEGIN'
00230          'INTEGER' I;
00231          'IF' L>U 'THEN' 'GO TO' END IOA;
00232          REAL FORMAT(B, N, D);  OTI(L, '('I')', 3);
00234          OUT STRING(OUT CHANNEL, '(' FOR ')');
00235          OUT STRING(OUT CHANNEL, S);
00236          OUT STRING(OUT CHANNEL, '('(/I/) := ')');
00237         'FOR' I := L 'STEP' 1 'UNTIL' U 'DO'
00237          'BEGIN'
00237              A(/I/) := READR;  OUTREAL(OUTCHANNEL, A(/I/));
00239              'IF' I < U 'THEN' OUT STRING(OUT CHANNEL, '('; ')')
00239              'ELSE' OUT STRING(OUT CHANNEL, '(' 'DO' I := I + 1 ; ')')
00239          'END';
00240  END IOA :
00240  'END' IOA;
00241                                                                    OT
00241                                                                    ****   I
00241  'PROCEDURE'  OTI(I, S, N);  'VALUE' I, N;  'INTEGER' I, N;  'STRING' S;
00245  'COMMENT' THIS AND THE FOLLOWING 3 PROCEDURES OUTPUT ALGOL STATEMENTS
00245    COMPATIBLE WITH THOSE OF THE INPUT OUTPUT PROCEDURES IOI, IOR, IOB,
00245    IOA;
00245  'BEGIN'
00245          OUT STRING(OUT CHANNEL, S);  OUT STRING(OUT CHANNEL, '(' := ')');
00247          INTEGER FORMAT(N);  OUT INTEGER(OUT CHANNEL, I);
00249          OUT STRING(OUT CHANNEL, '('; ')')
00249  'END'   OTI;
00250
00250                                                                    ****   R
00250  'PROCEDURE'  OTR (R, S, B, N, D);
00251          'REAL' R;  'STRING' S;  'BOOLEAN' B;  'INTEGER' N, D;
00255  'BEGIN'
00255          OUT STRING(OUT CHANNEL, S);  OUT STRING(OUT CHANNEL, '(' := ')');
00257         REAL FORMAT(B, N, D);  OUT REAL(OUT CHANNEL, R);
00259         OUT STRING(OUT CHANNEL, '('; ')')
00259  'END'   OTR;
00260
00260                                                                    ****   B
00260  'PROCEDURE'  OTB (B, S, N);  'BOOLEAN' B;  'STRING' S;  'INTEGER' N;
```

```
 SC     SOURCE STATEMENT

00264   'BEGIN'
00264         OUT STRING(OUT CHANNEL, S);  OUT STRING(OUT CHANNEL, '(' := ')');
00266         BOOLEAN FORMAT(N);  OUT BOOLEAN(OUT CHANNEL, B);
00268         OUT STRING(OUT CHANNEL, '('; ')')
00268   'END' OTB;
00269
00269                                                                          ****   A
00269   'PROCEDURE'  OTA (A, L, U, S, B, N, D);
00270         'INTEGER' L, U, N, D;  'ARRAY' A;  'STRING' S;  'BOOLEAN' B;
00274   'BEGIN'
00274         'INTEGER' I;
00275         'IF' L > U 'THEN' 'GO TO' END OTA;
00276        REAL FORMAT(B, N, D);  OTI(L,'('I')', 3);
00278         OUT STRING(OUT CHANNEL, '(' 'FOR' ')');
00279         OUT STRING(OUT CHANNEL, S);
00280         OUT STRING(OUT CHANNEL, '('(/I/) := ')');
00281         'FOR' I := L 'STEP' 1 'UNTIL' U 'DO'
00281         'BEGIN'
00281             OUT REAL(OUT CHANNEL, A(/I/));
00282             'IF' I < U 'THEN' OUT STRING(OUT CHANNEL, '('; ')')
00282             'ELSE' OUT STRING(OUT CHANNEL, '(' 'DO' I := I + 1; ')')
00282         'END';
00283    END OTA:
00283   'END' OTA;
00284                                                                        OUT
00284                                                                        ****   I
00284   'PROCEDURE'  OUTI (I, N);  'INTEGER' I, N;
00286   'COMMENT' THIS AND THE 3 FOLLOWING PROCEDURES OUTPUT INTEGERS, REAL
00286     NUMBERS, BOOLEANS OR ONE DIMENSIONAL ARRAYS USING FORMAT AS INDICATED
00286     IN "OUT INTEGER";
00286   'BEGIN'
00286         INTEGER FORMAT(N);
00287         OUT INTEGER(OUT CHANNEL, I)
00287   'END'  OUTI;
00288
00288                                                                          ****   R
00288   'PROCEDURE'  OUTR(R, B, N, D);  'REAL' R;  'BOOLEAN' B;  'INTEGER' N, D;
00292   'BEGIN'
00292         REAL FORMAT(B, N, D);
00293         OUT REAL(OUT CHANNEL, R)
00293   'END'  OUTR;
00294
00294                                                                          ****   B
00294   'PROCEDURE'  OUTB (B, N);  'BOOLEAN' B;  'INTEGER' N;
00297   'BEGIN'
00297         BOOLEAN FORMAT(N);
00298         OUT BOOLEAN(OUT CHANNEL, B)
00298   'END'  OUTB;
```

```
SC      SOURCE STATEMENT

00299
00299                                                                   ****   A
00299   'PROCEDURE'  OUTA (A, L, U, B, N, D);  'INTEGER' L, U, N, D;  'ARRAY' A;
00302          'BOOLEAN' B;
00303   'BEGIN'
00303   'INTEGER' I;
00304          'IF' L > U 'THEN' 'GO TO' END OUTA;
00305          REAL FORMAT(B, N, D);
00306          'FOR' I:= L 'STEP' 1 'UNTIL' U 'DO' OUT REAL(OUT CHANNEL,A(/I/));
00307    END OUTA;
00307   'END'  OUTA;
00308                                                                   ****  IA
00308   'PROCEDURE'  OUTIA (IA, L, U, N);  'INTEGER' L, U, N;
00310          'INTEGER' 'ARRAY' IA;
00311   'BEGIN'
00311          'INTEGER' I;
00312          'IF' L > U 'THEN' 'GO TO' END OUTIA;
00313          INTEGER FORMAT(N);
00314          'FOR' I := L 'STEP' 1 'UNTIL' U 'DO'
00314              OUT INTEGER(OUT CHANNEL,IA(/I/));
00315    END OUTIA :
00315   'END'  OUTIA;
00316
00316                                                                   ****  BA
00316   'PROCEDURE'  OUTBA (BA, L, U, N);  'INTEGER' L, U, N;
00318          'BOOLEAN' 'ARRAY' BA;
00319   'BEGIN'
00319          'INTEGER' I;
00320          'IF' L > U 'THEN' 'GO TO' END OUTBA;
00321          BOOLEAN FORMAT(N);
00322          'FOR' I:= L 'STEP' 1 'UNTIL' U 'DO'
00322              OUT BOOLEAN(OUT CHANNEL, BA(/I/));
00323    END OUTBA :
00323   'END'  OUTBA;
00324
00324                                                                   OUTST
00324   'PROCEDURE' OUTST (S);  'STRING' S;  OUTSTRING(OUTCHANNEL, S);
00327
00327                                                                   COPY TXT
00327   'PROCEDURE'  COPY TEXT ;
00328   'COMMENT' THIS PROCEDURE INPUTS CHARACTERS ON "INCHANNEL" AND COPIES
00328   THEM ONTO "OUTCHANNEL" UNTIL IT COPIES A SEMICOLON ;
00328   'BEGIN' 'INTEGER' X;
00329     L :  INSYMBOL(INCHANNEL, '(';0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ @%*<
00329   -/#,$.+_)  |&>:¬'?"= (')', X);
00330         OUTSYMBOL(OUTCHANNEL,'(';0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ @%*<
00330   -/#,$.+_)  |&>:¬'?"= (')', X);    'IF' X¬=1 'THEN' 'GOTO' L
00331   'END'  COPY TEXT;
```

```
00332
00332
00332
00332
00332
00332
00332
00332
00332
00332
00332
00332   'COMMENT'*****************************************************************************
00332
00332                FOR COMPATABILITY WITH THE OVER-RIDDEN IBM PROCEDURES
00332          OUTREAL(CHANNEL, R), OUTINTEGER(CHANNEL, I), OUTBOOLEAN(CHANNEL, B)
00332                DEFAULT VALUES ARE GIVEN TO THE FORMAT PARAMETERS .
00332                      ************************************** ;
00332
00332      REAL FORMAT('FALSE', 15, 6);  INTEGER FORMAT(13);  BOOLEAN FORMAT(9);
00335
00335   'COMMENT'    THE STANDARD INPUT AND OUTPUT DEVICES ARE ASSUMED TO BE
00335                           THE CARD READER AND LINE PRINTER
00335                      ************************************** ;
00335
00335                      IN CHANNEL := 0;  OUT CHANNEL := 1;
00337
00337   'COMMENT'         THE PRINTER HAS A DEFAULT LINESIZE AND PAGESIZE
00337                      ************************************** ;
00337
00337                           LINESIZE(120);  PAGESIZE(64);
00339
00339   'COMMENT'************  USER PROGRAM TO BEGIN ON NEXT PAGE  ************;*********
00339
00339
00339
00339
00339
00339
00339
00339
00339
00339
00339
00339
00339
00339
00339
```

A P P E N D I X E

PROGRAM LISTING

```
SC      SOURCE STATEMENT

00339   'BEGIN'                                                          10
00339       'INTEGER' STORESIZE, RDR, PNTR, SYSDA, INITDV, OTDV;         20
00340       STORESIZE:= READI;
00341       RDR:=INCHANNEL:= 0;  PNTR:=OUTCHANNEL:= 1;  SYSDA:= 15;
00344   'BEGIN'
00344       'INTEGER' 'ARRAY' H, T(/1:STORESIZE/), COL(/1:5/);
00345       'INTEGER' LAVS, TARGET, INTREF, EXTREF, PROGRAM, BUFFER, INTERFACE,
00345            LABEL, MSTACK, ISTACK, OSTACK, LINK, ESTORE, NAME, FF, TRANS,
00345            W, INPT, OUTPT, TAG, MARK, HOLD, PCR, F, A, INDEX, ROUTE,
00345            N NONTERMS, N TERMS, NEXTCHAR;
00346       'SWITCH' FCN1:= CALL, RETURN, TRUE, FALSE, FLAG1, NOT, MARK1, SELECT
00346            , TEST, REFERENCE(/A/), MATCH, NEXT, COPY, NULL, EDIT, PRINT ;
00347       'SWITCH' FCN2:= SPACE1, TAB1 ;
00348       'BOOLEAN' PIP,FLAG,PRIME,LST,RTN,PERFORMANCE TEST, INITL CODE ;
00349       'SWITCH' REFERENCE := SETLAB, INCLAB, LABEL1, DECLAB, RESETLAB;
00350
00350
00350       'PROCEDURE' INITIALIZE;
00351       'BEGIN'
00351            'INTEGER' I;
00352            LAVS := 1;
00353            'FOR' I := 1 'STEP' 1 'UNTIL' STORESIZE 'DO'
00353            'BEGIN' H(/I/) := I + 1;  T(/I/) := 500 000 001 'END';
00355            H(/STORESIZE/) := 0
00355       'END' OF INITIALIZE;
00356
00356
00356       'PROCEDURE'  TAKE (I);
00357       'INTEGER' I;
00358       'BEGIN'
00358            'IF' LAVS = 0 'THEN' WARN('(' LIST STORE FULL ')');
00359            I := LAVS;  LAVS := H(/LAVS/)
00360       'END' OF TAKE;
00361
00361       'PROCEDURE'  PUT (I);
00362       'VALUE' I;  'INTEGER' I;
00364       'BEGIN'
00364            H(/I/) := LAVS;  T(/I/) := 500 000 001;
00366            LAVS := I
00366       'END' OF PUT;
00367
00367
00367       'PROCEDURE'  OPENLIST (V);
00368       'INTEGER' V;
00369       'BEGIN'
00369            'INTEGER' I;  TAKE(I);
00371            V := I;  H(/I/) := T(/I/) := 0
00372       'END' OF OPENLIST;
```

```
SC      SOURCE STATEMENT
00373
00373
00373       'PROCEDURE'  CLOSELIST (V);
00374       'INTEGER' V;
00375       'BEGIN'
00375            'INTEGER' I, J;  I := V;  V := 0;
00378            'FOR' J := I, I 'WHILE' I ¬= 0 'DO'
00378            'BEGIN' I := H(/I/);  PUT(J) 'END'
00379       'END' OF CLOSELIST;
00380
00380
00380       'PROCEDURE'  PUSH (X, V);
00381       'VALUE' X, V;  'INTEGER' X, V;
00383       'BEGIN'
00383            'INTEGER' I;  TAKE(I);
00385            H(/I/) := H(/V/);  T(/I/) := X;
00387            H(/V/) := I;  'IF' T(/V/) = 0 'THEN' T(/V/) := I
00388       'END' OF PUSH;
00389
00389
00389       'PROCEDURE'  PULL (X, V);
00390       'VALUE' V;  'INTEGER' X, V;
00392       'BEGIN'
00392            'INTEGER' I;  I := H(/V/);
00394            'IF' I = 0 'THEN' WARN('(' INVALID PULL ')');
00395            H(/V/) := H(/I/);  X := T(/I/);  PUT(I);
00398            'IF' H(/V/) = 0 'THEN' T(/V/) := 0
00398       'END' OF PULL;
00399
00399
00399       'PROCEDURE'  APPEND (X, V);
00400       'VALUE' X, V;  'INTEGER' X, V;
00402       'BEGIN'
00402            'INTEGER' I, J;
00403            TAKE(I);  H(/I/) := 0;  T(/I/) := X;  J := T(/V/);
00407            'IF' J = 0 'THEN' H(/V/) := T(/V/) := I
00407            'ELSE' H(/J/) := T(/V/) := I
00407       'END' OF APPEND;
00408
00408
00408       'PROCEDURE'  COMBINE(V1, V2);
00409       'VALUE' V1;  'INTEGER' V1, V2;
00411       'BEGIN'
00411            'IF' H(/V1/) = 0
00411            'THEN' 'BEGIN' PUT(V1);  V1 := V2 'END'
00412            'ELSE'
00412            'BEGIN'
00412                 'IF' H(/V2/) ¬= 0 'THEN'
```

```
SC      SOURCE STATEMENT
00412                'BEGIN' H(/T(/V1/)/) := H(/V2/);  T(/V1/) := T(/V2/) 'END'
00413               ;PUT(V2)
00414           'END';
00415           V2 := 0
00415       'END' OF COMBINE;
00416
00416
00416       'PROCEDURE'  EXCHANGE (V1, V2);
00417       'INTEGER' V1, V2;
00418       'BEGIN'
00418            'INTEGER' I;
00419            I:= V1;  V1:= V2;  V2:=I
00421       'END' OF EXCHANGE;
00422
00422
00422       'PROCEDURE'  REVERSE (V);
00423       'INTEGER' V;
00424       'BEGIN'
00424            'INTEGER' I, A, B;  OPENLIST(A);  B := V;
00427            'FOR' I := H(/B/) 'WHILE' I¬=0 'DO'
00427            'BEGIN' PULL(I, B);  PUSH(I, A) 'END';
00429           CLOSELIST(B);  V := A
00430       'END' OF REVERSE;
00431
00431
00431       'PROCEDURE'  BEHEAD (I, V);
00432       'VALUE' I, V;  'INTEGER' I, V;
00434       'BEGIN'
00434            'INTEGER' J, K;  K:= H(/V/);  H(/V/) := I;
00437            'FOR' J := K 'WHILE' K¬=I 'DO'
00437            'BEGIN' K:= H(/K/);  PUT(J) 'END'
00438       'END' OF BEHEAD;
00439
00439
00439       'PROCEDURE'  BETAIL (I, V);
00440       'VALUE' I, V;  'INTEGER' I, V;
00442       'BEGIN'
00442            'INTEGER' J, K;
00443           J := H(/I/);  T(/V/) := I;  H(/I/) := 0;
00446            'FOR' K := J 'WHILE' J¬=0 'DO'
00446            'BEGIN' J := H(/J/);  PUT(K) 'END'
00447       'END' OF BETAIL;
00448
00448
00448       'BOOLEAN' 'PROCEDURE'  EQUAL(V1, V2);
00449       'VALUE' V1, V2;  'INTEGER' V1, V2;
00451       'BEGIN'
00451            'INTEGER' I, J, K;  I := H(/V1/);  J := H(/V2/);
```

```
00454           'FOR' K := 1 'WHILE' H(/I/) ¬= 0 & H(/J/) ¬= 0 'DO'
00454                'IF' T(/I/) = T(/J/) 'THEN'
00454                'BEGIN' I := H(/I/);  J := H(/J/) 'END'
00455                'ELSE'
00455                'BEGIN' EQUAL := 'FALSE';  'GO TO' OUT 'END';
00457           EQUAL := H(/I/)=H(/J/) & T(/I/)=T(/J/);
00458      OUT :
00458      'END' OF EQUAL;
00459
00459
00459      'PROCEDURE'  STEP INPT;
00460      'BEGIN'
00460           'IF' H(/H(/INPT/)/) = 0 'THEN' APPEND(CHARIN, BUFFER);
00461           INPT := H(/INPT/)
00461      'END' OF STEP INPT;
00462
00462
00462      'PROCEDURE'  FACE (X);
00463      'VALUE' X;  'INTEGER' X;
00465      'BEGIN'
00465           T(/OUTPT/) := X;  APPEND(0, INTERFACE);  OUTPT := H(/OUTPT/)
00467      'END' OF FACE;
00468
00468
00468      'PROCEDURE'  ASSEMBLE (X);
00469      'VALUE' X;  'INTEGER' X;
00471      'BEGIN'
00471           'PROCEDURE' LOCATE(REF);
00472           'INTEGER' REF;
00473           'BEGIN'
00473                'INTEGER' A, I, AT, F;
00474                'FOR' I := H(/REF/) 'WHILE' I¬=0 'DO'
00474                'BEGIN'
00474                     PULL(A, REF);  PULL(I, A);  PULL(AT, A);
00477                     'FOR' I := H(/A/) 'WHILE' I¬=0 'DO'
00477                     'BEGIN' PULL(F, A);  T(/F/) := T(/F/) + AT 'END';
00479                     CLOSELIST(A)
00479                'END';
00480                CLOSELIST(REF)
00480           'END' OF LOCATE;
00481           'INTEGER' REF, SUB, K;
00482           'IF' X<0 'THEN'
00482           'BEGIN'
00482                SUB := REF := 'IF' X>-999 'THEN' INTREF 'ELSE' EXTREF;
00483                'FOR' SUB := H(/SUB/) 'WHILE' SUB¬=0 'DO'
00483                'IF' T(/H(/T(/SUB/)/)/) = X 'THEN'
00483                'BEGIN' SUB := T(/SUB/);  'GO TO' FOUND 'END';
00485                OPENLIST(SUB);  PUSH(X, SUB);  PUSH(SUB, REF);
```

```
SC    SOURCE STATEMENT

00488         FOUND :  'IF' FF=0 'THEN'
00488                  'BEGIN'
00488                       PULL(K, SUB);  PUSH(TRANS, SUB);  PUSH(K, SUB)
00490                  'END'
00490                  'ELSE'
00490                  'BEGIN'APPEND(TRANS, SUB);  'GO TO' LOAD 'END'
00491              'END'
00491              'ELSE' 'IF' FF>0 'THEN'
00491              'BEGIN'
00491                   FF := FF+ X;
00492            LOAD : PUT(TRANS);  APPEND(FF, TARGET);  TAKE(TRANS);  FF := 0
00495              'END'
00495              'ELSE' 'IF' X=0 'THEN'
00495              'BEGIN' LOCATE(INTREF);  OPENLIST(INTREF) 'END'
00496              'ELSE' 'IF' X>=999 'THEN' FF:= 100 000 * X
00496              'ELSE'
00496              'BEGIN' CLOSELIST(INTREF);  LOCATE(EXTREF);  PUT(TRANS) 'END'
00498        'END' OF ASSEMBLE;
00499
00499
00499        'INTEGER''PROCEDURE'  CHARIN;
00500        'BEGIN'
00500             'INTEGER' X;
00501             'INTEGER''PROCEDURE' IOCHAR;
00502             'BEGIN'
00502                  'INTEGER' I, COL;
00503                  INSYMBOL(RDR,'(' '(/)0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ#
00503   ,$.-+_ |&>:;¬?"=@%*<')', I);
00504                  OUTSYMBOL(PNTR,'(' '(/)0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ#
00504   ,$.-+_ |&>:;¬?"=@%*<')', I);
00505                  SYSACT(RDR, 1, COL);  'IF' COL=1 'THEN' NEWLINE(1);
00507                  IOCHAR := I
00507             'END'  IOCHAR;
00508             'IF' NEXTCHAR>0 'THEN'
00508             'BEGIN' X := NEXTCHAR;  NEXTCHAR := 0 'END'
00509             'ELSE' L : X := IOCHAR;
00510             'IF' X=0|X=1 'THEN' 'GO TO' L;
00511             'IF' X=2 'THEN'
00511             'BEGIN'
00511                  PRIME := 'IF' PRIME 'THEN' 'FALSE' 'ELSE' 'TRUE';
00512                  'GO TO' L
00512             'END';
00513             'IF' X=3 'THEN'
00513             'BEGIN'
00513                  NEXTCHAR := IOCHAR;
00514                  'IF' NEXTCHAR=4 'THEN'
00514                  'BEGIN' X:= 63;  NEXTCHAR := 0 'END'
00515             'END'
```

```
SC      SOURCE STATEMENT

00515           'ELSE'
00515           'IF' X=4 'THEN'
00515           'BEGIN'
00515                NEXTCHAR := IOCHAR;
00516                'IF' NEXTCHAR=5 'THEN'
00516                'BEGIN' X := 64;  NEXTCHAR := 0 'END'
00517           'END';
00518           X := X+34+( 'IF' PRIME 'THEN' 100 'ELSE' 0 );
00519           CHARIN := 'IF' X=78 'THEN' -1 'ELSE' X
00519      'END' OF CHARIN;
00520
00520
00520      'PROCEDURE' CHAROUT(X);
00521           'VALUE' X;  'INTEGER' X;
00523      'BEGIN'
00523           'BOOLEAN' PRIME;
00524           X := X-34;
00525           'IF' X<101 'THEN' PRIME := 'FALSE' 'ELSE'
00525           'BEGIN' PRIME := 'TRUE';  X := X-100 'END';
00527           'IF' X<1 | X>64 'THEN' 'GO TO' OUT;
00528           'IF' X=63 'THEN' OUTSTRING(OTDV, '('(/')') 'ELSE'
00528           'IF' X=64 'THEN' OUTSTRING(OTDV, '('/')') 'ELSE'
00528           'BEGIN'
00528                'IF' PRIME 'THEN' OUTSTRING(OTDV, '(''')');
00529                OUTSYMBOL(OTDV,'(' '(/)0123456789ABCDEFGHIJKLMNOPQRSTUVWXY
00529   Z#,$.-+_ |&>:;¬?"=@%*<')', X);
00530                'IF' PRIME 'THEN' OUTSTRING(OTDV, '(''')')
00530           'END';
00531      OUT:
00531      'END' OF CHAROUT;
00532
00532
00532      'INTEGER' 'PROCEDURE' DIGIT (X);
00533           'VALUE' X;  'INTEGER' X;
00535      'BEGIN'
00535           'IF' X<40 | X>49 'THEN' WARN('(' DIGIT ERROR ')');
00536           DIGIT := X-40
00536      'END' OF DIGIT;
00537
00537
00537      'PROCEDURE'  PROGOUT (X);
00538           'VALUE' X;  'INTEGER' X;
00540      'BEGIN'
00540           'INTEGER' I;
00541           'IF' PIP 'THEN'
00541           'BEGIN'
00541                'IF' X<0 'THEN'
00541                'BEGIN'
```

```
SC      SOURCE STATEMENT
00541                       'IF' RTN &¬INITL CODE 'THEN'
00541                       'BEGIN' VAR NAME(X);  RTN:='FALSE' 'END' ;
00543                       COL(/1/) := COL(/2/);  COL(/2/) := COL(/3/);
00545                       COL(/3/) := X
00545                  'END'
00545                  'ELSE'
00545                  'IF' X=0 'THEN'
00545                  'BEGIN' NEWLINE(1);  OUTI(0,31);  NEWLINE(2);  RTN:='TRUE'
00548                  'END' 'ELSE'
00548                  'IF' X=999 'THEN' 'BEGIN' NEWLINE(1);  OUTI(999, 31) 'END'
00549                  'ELSE'
00549                  'BEGIN' COL(/4/) := X;  PIP := 'FALSE' 'END'
00550             'END'
00550             'ELSE'
00550             'BEGIN'
00550                  COL(/5/) := X;  NEWLINE(1);
00552                  'FOR' I := 1 'STEP' 1 'UNTIL' 5 'DO'
00552                  'BEGIN'
00552                       'IF' I<=3 & COL(/I/)=0 'THEN' SPACE(8)
00552                       'ELSE'
00552                       'BEGIN' OUTI(COL(/I/), 7);  OUTSTRING(OTDV,'(';')');
00554                       'END'
00554                  'END';
00555                  'IF' OTDV=1 'THEN' SYMBOLIC CODE;
00556                  'FOR' I := 1 'STEP' 1 'UNTIL' 5 'DO' COL(/I/) := 0;
00557                  PIP := 'TRUE'
00557             'END'
00557        'END' OF PROGOUT;
00558
00558
00558        'PROCEDURE' VAR NAME (X);  'VALUE' X;  'INTEGER' X;
00561        'BEGIN' 'INTEGER' I;  I:= NAME;
00563             'FOR' I:=H(/I/) 'WHILE' I¬=0 'DO'
00563             'IF' X=T(/H(/T(/I/)/)/) 'THEN'
00563             'BEGIN' I:=T(/T(/T(/I/)/)/);  OUTSTRING(OTDV,'('<')');
00565                  'FOR' I:=H(/I/) 'WHILE' I¬=0 'DO' CHAROUT(T(/I/));
00566                  OUTSTRING(OTDV, '('>')');  'GO TO' OUT
00567             'END' ;
00568        OUT :
00568        'END' VAR NAME ;
00569
00569
00569        'PROCEDURE'  SYMBOLIC CODE;
00570        'BEGIN'
00570             'INTEGER' F,A,I;
00571             'PROCEDURE' W(S);  'STRING' S;  OUTSTRING(PNTR, S);
00574             'PROCEDURE' MNEMONIC FCN NAME (FN);
00575                  'VALUE' FN;  'INTEGER' FN;
```

```
SC      SOURCE STATEMENT
00577           'BEGIN'
00577           'PROCEDURE' SHOW(S); 'STRING' S; 'BEGIN' W(S); 'GOTO'OUT 'END';
00581                'SWITCH' FCN NAME := CALL, RETURN, TRUE, FALSE, FLAG, NOT,
00581                     MARK, SELECT, TEST, REFERENCE, MATCH, NEXT, COPY,
00581                     NULL, EDIT, PRINT;
00582                'IF' FN<1|FN>17 'THEN' SHOW('(' ****')');
00583                'GO TO' 'IF' FN=17 'THEN' STOP 'ELSE' FCN NAME(/FN/);
00584                CALL : SHOW('('CALL  ')');  RETURN : SHOW('('RETURN')');
00586                TRUE : SHOW('('TRUE  ')');  FALSE : SHOW('('FALSE ')');
00588                FLAG : SHOW('('FLAG  ')');  NOT : SHOW('('NOT   ')');
00590                MARK : SHOW('('MARK  ')');  SELECT : SHOW('('SELECT')');
00592                TEST : SHOW('('TEST  ')');REFERENCE:SHOW('('REFERENCE')');
00594                MATCH : SHOW('('MATCH ')');  NEXT : SHOW('('NEXT  ')');
00596                COPY : SHOW('('COPY  ')');  NULL : SHOW('('NULL  ')');
00598                EDIT : SHOW('('EDIT  ')');  PRINT : SHOW('('PRINT ')');
00600                STOP : SHOW('('STOP  ')');
00601           OUT :
00601           'END' OF MNEMONIC FCN NAME;
00602           'PROCEDURE' REFERENCE(X);  'VALUE' X;  'INTEGER' X;
00605           'BEGIN'
00605           'PROCEDURE' SHOW(S); 'STRING' S; 'BEGIN' W(S); 'GOTO'OUT 'END';
00609                'SWITCH' REF := SETLAB, INCLAB, LABEL, DECLAB, RESETLAB;
00610                'GO TO' 'IF' X>0 & X<6 'THEN' REF(/X/) 'ELSE' OUT;
00611                SETLAB: SHOW('(' SETLAB')');  INCLAB: SHOW('(' INCLAB')');
00613                LABEL: SHOW('(' LABEL')');  DECLAB: SHOW('(' DECLAB')');
00615                RESETLAB: SHOW('(' RESETLAB')');   OUT :
00616           'END'  REFERENCE;
00617           'PROCEDURE' EDIT(I);  'VALUE' I;  'INTEGER' I;
00620           'BEGIN' 'SWITCH' ED := EW, C, X, N, L, V;
00621           'PROCEDURE' SHOW(S); 'STRING' S; 'BEGIN' W(S); 'GOTO'OUT 'END';
00625                'GO TO' 'IF' I>0 & I<7 'THEN' ED(/I/) 'ELSE' OUT;
00626               EW: SHOW('('W')');  C: SHOW('('C')');  X: SHOW('('X')');
00629                N: SHOW('('N')');  L: SHOW('('L')');  V:  SHOW('('V')');
00632           OUT :
00632           'END'  EDIT;
00633           SPACE(5);
00634           'FOR' I := 1 'STEP' 1 'UNTIL' 3 'DO'
00634           'IF' COL(/I/)=0 'THEN' SPACE(8) 'ELSE' OUTI(COL(/I/), 8);
00635           F := COL(/4/);  A := COL(/5/);
00637           SPACE(2);  MNEMONIC FCN NAME(F);
00639           'IF' F=1 'THEN'
00639           'BEGIN' OUTI(A,10);  'IF' A<=-1000 & ¬INITL CODE 'THEN'
00640                'BEGIN' SPACE(2); VAR NAME(A) 'END'
00641           'END' 'ELSE'
00641           'IF' F=10 'THEN' REFERENCE(A) 'ELSE'
00641           'IF' F=11 'THEN' CHAROUT(A) 'ELSE'
00641           'IF' F=15 'THEN' 'BEGIN' SPACE(9);  EDIT(A) 'END' 'ELSE'
00642           'IF' F=16 'THEN'
```

```
SC      SOURCE STATEMENT
00642           'BEGIN'  'COMMENT' F := PRINT;
00642                'IF' A>34 & A<198 'THEN' CHAROUT(A) 'ELSE'
00642                'BEGIN' OUTI(A, 10);
00643                     'IF' A>0 & A<18 'THEN'
00643                     'BEGIN'  W('(' (')'); MNEMONIC FCN NAME(A);
00645                          'IF' A<6 'THEN'
00645                          'BEGIN' W('('|')'); REFERENCE(A);
00647                               'IF' A<5 'THEN'
00647                               'BEGIN' W('(' | ')'); EDIT(A) 'END'
00648                          'END';  W('(' )')')
00649                     'END'
00649                'END'
00649           'END'
00649           'ELSE' OUTI(A, 10);
00650      'END'  OF SYMBOLIC CODE;
00651
00651
00651
00651      'PROCEDURE' WRITE TEXT (S);
00652      'STRING' S;
00653      'BEGIN'
00653           NEW LINE(2);  SPACE(20);  OUT STRING(PNTR, S);  NEW LINE(1)
00656      'END'  OF WRITE TEXT;
00657
00657
00657      'PROCEDURE'  WARN (STRING);
00658      'STRING' STRING;
00659      'BEGIN' WRITETEXT(STRING);  LST OBJ CODE;  MONITOR;
00662           'GO TO' FAIL  'END' ;
00663
00663
00663      'PROCEDURE'  MONITOR;
00664      'BEGIN'
00664           'PROCEDURE'  SHOW (VAR, NAME);
00665           'VALUE' VAR;  'INTEGER' VAR;  'STRING' NAME;
00668           'BEGIN'
00668                NEW LINE(1);  OUTI(VAR, 16);  SPACE(2);
00671                OUT STRING(PNTR,NAME)
00671           'END'  OF SHOW;
00672           'INTEGER' I,J;  'INTEGER''ARRAY' II,IH,IT(/1:10/) ;
00674           NEW PAGE; WRITETEXT('('BEGIN MONITOR')');
00676           SHOW(TARGET, '('TARGET')');  SHOW(INTREF, '('INTREF')');
00678           SHOW(EXTREF, '('EXTREF')');  SHOW(PROGRAM, '('PROGRAM')');
00680           SHOW(BUFFER,'('BUFFER')');  SHOW(INTERFACE, '('INTERFACE')');
00682           SHOW(LABEL, '('LABEL')');  SHOW(MSTACK, '('MSTACK')');
00684           SHOW(ISTACK, '('ISTACK')');  SHOW(OSTACK, '('OSTACK')');
00686           SHOW(LINK, '('LINK')');  SHOW(ESTORE, '('ESTORE')');
00688           SHOW(NAME, '('NAME')');  NEW LINE(1);
```

```
SC    SOURCE STATEMENT
00690           SHOW(FF, '('FF')'); SHOW(TRANS, '('TRANS')');
00692           SHOW(W, '('W')'); SHOW(INPT, '('INPT')');
00694           SHOW(OUTPT, '('OUTPT')'); SHOW(TAG, '('TAG')');
00696           SHOW(MARK, '('MARK')'); SHOW(HOLD, '('HOLD')');
00698           SHOW(PCR, '('PCR')'); SHOW(F,'('F')');
00700           SHOW(A, '('A')'); SHOW(INDEX, '('INDEX')');
00702           NEWLINE(1); OUTB(PIP, 16); OUTSTRING(PNTR, '(' PIP')');
00705           NEWLINE(1); OUTB(FLAG, 16); OUTSTRING(PNTR, '(' FLAG')');
00708           NEWLINE(3); OUTSTRING(PNTR,'(' LIST STORE')');
00710           I:= 1;
00711           'FOR' J:=0 'WHILE' I<=STORESIZE 'DO'
00711           'BEGIN'
00711                'FOR' I:=I,I+1 'WHILE' J<10&I<=STORESIZE 'DO'
00711                'IF' T(/I/)¬=500 000 001 'THEN'
00711                'BEGIN' J:=J+1;
00712                     II(/J/):=I; IH(/J/):=H(/I/); IT(/J/):=T(/I/)
00714                'END' ;
00715                NEWLINE(1); OUTSTRING(PNTR,'('I')'); OUTIA(II,1,J,11);
00718                NEWLINE(1); OUTSTRING(PNTR,'('H')'); OUTIA(IH,1,J,11);
00721                NEWLINE(1); OUTSTRING(PNTR,'('T')'); OUTIA(IT,1,J,11)
00723           'END' ;
00724           WRITETEXT('(' END MONITOR ')')
00724      'END' OF MONITOR;
00725
00725
00725      'PROCEDURE' LST OBJ CODE :
00726      'BEGIN'
00726           'INTEGER' X ;
00727           INCHANNEL:=OUTCHANNEL:=SYSDA ;
00728           OTI(500 000 000,'('EODATA')',12); OUTCHANNEL:=OTDV;
00730           CLOSE(SYSDA); OPEN(SYSDA); CLOSE(OTDV); OPEN(OTDV);
00734           OUTSTRING(OTDV,'('OBJECT CODE')'); NEWLINE(3);
00736           'FOR' X:=READI 'WHILE' X¬= 500 000 000 'DO'
00736           'BEGIN'
00736                'IF' X>600 000 000 'THEN'
00736                'BEGIN' 'IF' X<700 000 000 'THEN' SPACE(X-600 000 000)
00736                     'ELSE' TAB(X-700 000 000)
00736                'END' 'ELSE'
00736                'IF' ROUTE=1 'THEN' CHAROUT(X) 'ELSE' PROGOUT(X);
00737           'END';
00738           INCHANNEL:= RDR; OUTCHANNEL:= PNTR
00739      'END' ;
00740
00740
00740  INITL INPUT OF PARSING PROGRAM :
00740
00740      OPEN(RDR); OPEN(PNTR);
00742      INITDV:= READI; LST:=READB;
```

```
SC     SOURCE STATEMENT
00744       OTDV:= 'IF' LST 'THEN' READI 'ELSE' PNTR;  NEWCARD;
00746       'IF' INITDV¬=RDR 'THEN'
00746       'BEGIN' OPEN(INITDV);  INCHANNEL:=INITDV 'END';
00748       INITIALIZE;  INITL CODE:= 'TRUE';
00750       TARGET := INTREF := EXTREF := PROGRAM := BUFFER := INTERFACE :=
00750            LABEL := MSTACK := ISTACK := OSTACK := LINK := ESTORE :=
00750            NAME := FF := TRANS := INPT := OUTPT := TAG := MARK :=
00750            N NONTERMS := N TERMS := NEXTCHAR := 0;
00751       OPENLIST(TARGET);  OPENLIST(INTREF);  OPENLIST(EXTREF); TAKE(TRANS);
00755       'IF' LST 'THEN'
00755       'BEGIN' OUTSTRING(PNTR,'('INITIAL METAGRAMMAR LISTING')');
00756            'IF' OTDV¬=PNTR 'THEN''BEGIN' OPEN(OTDV);
00757            OUTSTRING(OTDV,'('RELOCATABLE METAGRAMMAR')') 'END';
00758            PIP:= 'TRUE'; 'FOR' W:= 1 'STEP' 1 'UNTIL' 5 'DO' COL(/W/):= 0;
00760            'FOR' W:= READI 'WHILE' FF¬=0|W¬=999 'DO'
00760            'BEGIN' ASSEMBLE(W);  PROGOUT(W) 'END' ;  PROGOUT(999)
00762       'END' 'ELSE'
00762       'FOR' W := READI  'WHILE' FF¬= 0 'OR' W ¬= 999 'DO' ASSEMBLE(W) ;
00763       ASSEMBLE(999);  INITL CODE:='FALSE' ;
00765       'IF' INITDV¬=RDR 'THEN'
00765       'BEGIN' CLOSE(INITDV);  INCHANNEL:= RDR 'END';
00767
00767    RECYCLE :
00767
00767       OUTSTRING(SYSDA,'('EODATA;')');
00768       PROGRAM := TARGET;  TARGET:= 0;  CLOSE(SYSDA); OPEN(SYSDA);
00772       NEWPAGE;  OUTSTRING(PNTR,'('CONTROL PARAMETERS :')');  NEWLINE(1);
00775       IOI(ROUTE,'('ROUTE')',1);  IOB(LST,'('OBJECT CODE LISTING')', 7);
00777       'IF'LST 'THEN' IOI(OTDV,'('DEVICE FOR OBJECT CODE LISTING')',1)
00777      'ELSE' OTDV:=PNTR; IOB(PERFORMANCE TEST,'('PERFORMANCE TESTING')',7);
00779       NEWCARD;  NEWLINE(2);  RTN:='FALSE';
00782       'IF' ROUTE=1 'THEN' OUTSTRING(PNTR,'('SOURCE LANGUAGE TEXT')')'ELSE'
00782            OUTSTRING(PNTR, '('GRAMMAR')');  NEWLINE(2);
00784       'IF' ROUTE=3 'THEN'
00784       'BEGIN'
00784            OPENLIST(TARGET);  OPENLIST(INTREF);  OPENLIST(EXTREF) ;
00787            FF :=0;  TAKE(TRANS)
00788       'END';
00789       N TERMS := N NONTERMS := NEXTCHAR := 0;  PRIME := 'FALSE';
00791       OPENLIST(BUFFER);  PUSH(CHARIN, BUFFER);  INPT := BUFFER;
00794       OPENLIST(INTERFACE);  PUSH(0, INTERFACE);  OUTPT := H(/INTERFACE/);
00797       OPENLIST(MSTACK);  MARK := 0;
00799       OPENLIST(ISTACK);  OPENLIST(OSTACK);  OPENLIST(LINK);
00802       OPENLIST(NAME);  INDEX := -1000;
00804       'FOR' W := 1 'STEP' 1 'UNTIL' 5 'DO' COL(/W/) := 0;
00805       PCR := H(/PROGRAM/);  FLAG := PIP := 'TRUE';
00807
00807    DECODE :
```

```
SC      SOURCE STATEMENT
00807
00807       W := T(/PCR/);  F := W '/' 100 000;  A := W - 100 000 * F;
00810       PCR := H(/PCR/);
00811       'GO TO' 'IF' F<=16 'THEN' FCN1(/F/) 'ELSE'
00811               'IF' F<=18 'THEN' FCN2(/F-16/) 'ELSE' STOP;
00812
00812
00812    CALL:
00812       PUSH(MARK, MSTACK); MARK := 0;
00814       PUSH(INPT, ISTACK);  PUSH(OUTPT, OSTACK);
00816       PUSH('IF' A=0 'THEN' 0 'ELSE' PCR, LINK);
00817       'IF' A¬=0 'THEN' PCR := A;  FLAG := 'TRUE';
00819       N NONTERMS := N NONTERMS+1;  'GOTO' DECODE;
00821
00821    RETURN :
00821       'BEGIN'
00821            'INTEGER' A, B;
00822            PULL(MARK, MSTACK);  PULL(A, ISTACK);  PULL(B, OSTACK);
00825            'IF' ¬ FLAG 'THEN'
00825            'BEGIN' INPT := A;  OUTPT := B;  BETAIL(OUTPT,INTERFACE) 'END';
00828            PULL(A, LINK);  'IF' A¬=0 'THEN' PCR := A
00829       'END';  'GO TO' DECODE;
00831
00831    TRUE :
00831       'IF' FLAG 'THEN' PCR := A 'ELSE'
00831       'BEGIN'
00831            FLAG := 'TRUE';  MARK := 0;  INPT := T(/H(/ISTACK/)/);
00834            OUTPT := T(/H(/OSTACK/)/);  BETAIL(OUTPT, INTERFACE)
00835       'END';  'GO TO' DECODE;
00837
00837    FALSE :
00837       'IF' ¬ FLAG 'THEN' PCR := A;  'GO TO' DECODE;
00839
00839    FLAG1 :
00839       'IF' FLAG 'THEN' PCR := A;  FLAG := 'TRUE';  'GOTO' DECODE;
00842
00842    NOT :
00842       'BEGIN'
00842            'INTEGER' A;
00843            FLAG := ¬ FLAG;  PULL(MARK, MSTACK);  PULL(INPT, ISTACK);
00846            'IF' FLAG 'THEN' STEPINPT;
00847            PULL(OUTPT, OSTACK);  BETAIL(OUTPT, INTERFACE);
00849            PULL(A, LINK)
00849       'END';  'GO TO' DECODE;
00851
00851    MARK1 :
00851       'BEGIN'
00851            'INTEGER' W, U, V;
```

```
SC     SOURCE STATEMENT
00852            'IF' A>31 'THEN' WARN('(' INVALID MARKER ')');
00853            PULL (W, MSTACK);  U := 2**(A-1);   V := W '/' U;
00856            V := U*(V-2*(V'/'2));  PUSH(W-V+U, MSTACK);
00858       'END';  'GO TO' DECODE;
00860
00860    SELECT: HOLD := A;  'GO TO' DECODE;
00862
00862    TEST :
00862       'BEGIN'
00862            'INTEGER' V;
00863            V := 2**(HOLD-1);  V := MARK'/'V;
00865            'IF' V - 2 * (V '/' 2) = 0 'THEN' PCR := A;
00866       'END';  'GO TO' DECODE;
00868
00868    SETLAB :
00868       TAG := 1;  OPENLIST(LABEL);  'GO TO' DECODE;
00871
00871    INCLAB :
00871       PUSH(TAG, LABEL);  TAG := TAG+1;  'GO TO' DECODE;
00874
00874    LABEL1 :
00874       FACE(T(/H(/LABEL/)/));  FACE(-4);  'GO TO' DECODE;
00877
00877    DECLAB :
00877       'BEGIN'
00877            'INTEGER' A;  PULL(A, LABEL);
00879            'IF' ¬ FLAG 'THEN' TAG := A;
00880       'END';  'GO TO' DECODE;
00882
00882    RESETLAB :
00882       CLOSELIST(LABEL);  'GO TO' DECODE;
00884
00884    MATCH :
00884       'BEGIN'
00884            'INTEGER' B;  B := T(/H(/INPT/)/);
00886            'IF' B<0 'THEN' WARN('(' WANTS MORE TEXT')');
00887            FLAG := B = A;  'IF' FLAG 'THEN' STEPINPT
00888       'END';
00889       N TERMS := N TERMS+1;  'GOTO' DECODE;
00891
00891    NEXT :
00891       'IF' T(/H(/INPT/)/) < 0 'THEN' WARN('('WANTS MORE TEXT')');
00892       FLAG := 'TRUE';  STEPINPT;  'GO TO' DECODE;
00895
00895    COPY :
00895       'IF' INPT=BUFFER 'THEN' WARN('(' INVALID COPY')');
00896       FACE(T(/INPT/));  'GO TO ' DECODE;
00898
```

```
SC    SOURCE STATEMENT
00898     NULL :
00898        FACE(500 000 000);  'GO TO' DECODE;
00900
00900     EDIT :
00900        FACE(-A);  'GO TO' 'IF' A=1 'THEN' EDITOR 'ELSE' DECODE;
00902
00902     PRINT :
00902        FACE(A);  'GO TO' DECODE;
00904
00904
00904    SPACE1 :
00904        FACE(A+600 000 000);  'GO TO' DECODE;
00906
00906    TAB1 :
00906        FACE(A+700 000 000);  'GO TO' DECODE;
00908
00908     STOP :
00908        'IF' ¬ FLAG 'THEN' WARN('(' FAULTY SYNTAX ')');  'GO TO' EXIT;
00910
00910     EDITOR .:
00910
00910        'BEGIN'
00910             'INTEGER' CODE;
00911             'SWITCH' CONTROL := W, C, X, N, L, V;
00912             OPENLIST(ESTORE);
00913        SCAN:
00913             PULL (CODE, INTERFACE);
00914             'IF' CODE<0 'THEN' 'GO TO' CONTROL(/-CODE/);
00915
00915
00915        SYMBOL :
00915             'BEGIN'
00915                  'INTEGER' A;
00916                  OPENLIST(A);  PUSH(CODE, A);  PUSH(A, ESTORE)
00918             'END';  'GO TO' SCAN;
00920
00920
00920        W:   'BEGIN'
00920                  'INTEGER' A, SUB, X;
00921                  REVERSE(ESTORE);  OUTCHANNEL:= SYSDA;
00923                  'FOR' A := H(/ESTORE/) 'WHILE' A¬=0 'DO'
00923                  'BEGIN'
00923                       PULL(SUB, ESTORE);
00924                       'FOR' A := H(/SUB/) 'WHILE' A¬=0 'DO'
00924                       'BEGIN'
00924                            PULL(X, SUB);
00925                            'IF' X¬=500 000 000 'THEN'
00925                            'BEGIN'
```

```
SC      SOURCE STATEMENT
00925                                   'IF' LST 'THEN'
00925                                   'BEGIN' OUTI(X, 11);
00926                                     OUTSTRING(SYSDA,'('';'')') 'END' ;
00927                                   'IF' ROUTE=3 'THEN' ASSEMBLE(X);
00928                             'END'
00928                       'END';  CLOSELIST(SUB)
00929                 'END';  CLOSELIST(ESTORE); OUTCHANNEL:= PNTR ;
00932                 BEHEAD(H(/INPT/), BUFFER);  INPT := BUFFER
00933           'END';  'GO TO' DECODE;
00935
00935
00935       C :  'BEGIN'
00935                 PULL(A, ESTORE);  COMBINE(T(/H(/ESTORE/)/), A)
00936           'END';  'GO TO' SCAN;
00938
00938
00938       X :  EXCHANGE(T(/H(/ESTORE/)/), T(/H(/H(/ESTORE/)/)/));' GOTO' SCAN;
00940
00940
00940       N :  'BEGIN'
00940                 'INTEGER' A;
00941                 A := H(/T(/H(/ESTORE/)/)/);
00942                 'IF' H(/A/)¬=0 'THEN' WARN('(' N ARGUMENT ILLFORMED ')');
00943                 T(/A/) := -T(/A/)
00943           'END';  'GO TO' SCAN;
00945
00945
00945       L :  'BEGIN'
00945                 'INTEGER' A, B, C;  A:= NAME;  B := T(/H(/ESTORE/)/);
00948                 'FOR' A := H(/A/) 'WHILE' A¬=0 'DO'
00948                 'IF' EQUAL(B, T(/T(/T(/A/)/)/)) 'THEN'
00948                 'BEGIN'
00948                       PULL (C, ESTORE);  CLOSELIST(C);  OPENLIST(C);
00951                       PUSH(T(/H(/T(/A/)/)/), C);  PUSH(C,ESTORE);
00953                       'GO TO' SCAN
00953                 'END';
00954                 PULL(C, ESTORE);  OPENLIST(A); PUSH(INDEX, A) ;
00957                 PUSH(A, ESTORE);  OPENLIST(A);  PUSH(C, A);
00960                 PUSH(INDEX, A);  PUSH(A, NAME);  INDEX := INDEX-1
00962           'END';  'GO TO' SCAN;
00964
00964
00964       V :  'BEGIN' 'INTEGER' N, A, D, I;  N := 0;  PULL(A, ESTORE);
00967                 'FOR' I := H(/A/) 'WHILE' I¬=0 'DO'
00967                 'BEGIN' PULL(D, A);  N := 10*N + DIGIT(D) 'END';
00969                 PUSH(N,A);  PUSH(A,ESTORE)
00970           'END';  'GO TO' SCAN;
00972
```

```
SC      SOURCE STATEMENT

00972
00972       'END' OF EDITOR;
00973
00973
00973    EXIT :
00973
00973       'IF' PERFORMANCE TEST 'THEN'
00973       'BEGIN'   NEWLINES(5);
00974            OUTSTRING(PNTR,'(' RELATIVE MEASURE OF PREFORMANCE')');
00975            NEWLINE(2); OTI(N NONTERMS,'('NUMBER OF ATTEMPTS TO RECOGNIZE A
00976    NON TERMINAL TYPE')',10);  NEWLINE(1);  OTI(N TERMS,'('NUMBER OF ATTEMP
00978   TS TO RECOGNIZE A TERMINAL TYPE')',14)
00978       'END';
00979       'IF' LST 'THEN' LST OBJ CODE;
00980       CLOSELIST(BUFFER);  CLOSELIST(INTERFACE);  CLOSELIST(MSTACK);
00983       CLOSELIST(ISTACK);  CLOSELIST(OSTACK);   CLOSELIST(LINK);
00986       CLOSELIST(PROGRAM);   W := NAME;
00988       'FOR' W := H(/W/) 'WHILE' W¬=0 'DO'
00988       'BEGIN' CLOSELIST(T(/T(/T(/W/)/)/));  CLOSELIST(T(/W/)) 'END';
00990       CLOSELIST(NAME);
00991       'IF' ROUTE=3 'THEN' 'GO TO' RECYCLE;
00992    FAIL :
00992       CLOSE(RDR);  CLOSE(PNTR);  CLOSE(SYSDA);
00995       'IF' OTDV¬=PNTR 'THEN' CLOSE(OTDV)
00995
00995   'END'
00995
00995   'END'
00995
00995   'END' OF ALGOL PROGRAM
```

PBN	SC	PBN SURR	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN
001	00000	000	BFW	I		02C	BOOLEA	P	01	08C	CLOSE	P	01	074
			COPYTE	P	00	134	DCMPIN	P	04	098	DCMPRE	P	06	0A0
			ENOUGH	P	02	094	IFW	I		028	INCHAN	I		018
			INTEGE	P	01	084	IOA	P	07	0F0	IOB	B P	03	0EC
			IOI	I P	03	0E4	IOR	R P	05	0E8	LINESI	P	01	07C
			NEWCAR	P	00	078	NEWLIN	P	01	08C	NEWPAG	P	00	090
			OPEN	P	01	070	OTA	P	07	104	OTB	P	03	100
			OTI	P	03	0F8	OTR	P	05	0FC	OUTA	P	06	118
			OUTB	P	02	114	OUTBA	P	04	128	OUTBOO	P	02	094
			OUTCHA	I		01C	OUTI	P	02	10C	OUTIA	P	04	120
			OUTINT	P	02	0C0	OUTR	P	04	110	OUTREA	P	02	0C4
			OUTST	P	01	130	PAGESI	P	01	080	RBOOL	B		030
			RD	I		024	READB	B P	00	0E0	READI	I P	00	0D8
			READR	R P	00	0DC	REALFO	P	03	0B8	RFW	I		020
			SPACE	P	01	084	TAB	P	01	088				
002	00000	001	CHANNE	I V		018								
003	00004	001	CHANNE	I V		018								
004	00008	001												
005	00010	001	N	I V		018								
006	00014	001	N	I V		018								
007	00018	001	NS	I V		018								
008	00021	007	CH	I		024	CHPTR	I		018	NL	I		020
			RECLTH	I		01C								
009	00030	001	CHPTR	I		020	N	I V		018				
010	00036	001	NL	I V		018								
011	00039	010	BLKSIZ	I		01C	K	I		028	NB	I		020
			REC	I		024	RECPTR	I		018				
012	00047	001												
013	00051	001	CHANNE	I V		018	CHPTR	I		028	FW	I V		020
			RECLTH	I		02C								
014	00059	001	DIGIT	I AN		030	I	I V		018	J	I		038
			L	L		09C	NEGATI	B N		020	NOFDIG	I N		028

PBN	SC	PBN SURR	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN
015	00073	001	DIGIT	I AN		040	ENDDEC	L		0B0	EXPONE	I N		038
			I	I		048	K	I		04C	M	I		050
			MAXNOF	I N		020	NEGATI	B N		028	R	R V		018
			SCALEU	L		0A8	SIZE	I N		030	TEST	L		0AC
			TOOSMA	B P	01	0A4								
016	00080	015	R	R N		020	TOOSMA	B P	01	0A4				
017	00108	001	W	I N		018								
018	00111	001	B	B N		018	D	I N		028	W	I N		020
019	00117	001	W	I N		018								
020	00120	001	CHANNE	I V		018	DIGIT	I A	01	038	I	I V		020
			J	I		02C	K	I		030	NEGATI	B		034
			NOFDIG	I		028								
021	00134	001	CHANNE	I V		018	DIGIT	I A	01	03C	ENDOUT	L		0D0
			ERROR	P	00	0CC	EXPONE	I		034	J	I		028
			K	I		02C	NEGATI	B		038	OUTDIG	P	01	0C8
			R	R V		020	SIZE	I		030				
022	00141	021	D	I N		018								
023	00144	021												
024	00173	001	B	B N		020	CHANNE	I V		018	I	I		028
			J	I		02C								
025	00183	001	I	I		020	READI	I P	00	0D8				
026	00187	001	R	R		020	READR	R P	00	0DC				
027	00191	001	B	B		020	READB	B P	00	0E0				
028	00195	001	I	I N		020	IOI	I P	03	0E4	N	I N		030
			S	T N		028								
029	00204	001	B	B N		030	D	I N		040	IOR	R P	05	0E8
			N	I N		038	R	R N		020	S	T N		028
030	00215	001	B	B N		020	IOB	B P	03	0EC	N	I N		030
			S	T N		028								

PBN	SC	PBN SURR	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN
031	00225	001	A	R AN		018	B	B N		038	D	I N		048
			ENDIOA	L		0F4	I	I		050	L	I N		020
			N	I N		040	S	T N		030	U	I N		028
032	00241	001	I	I V		018	N	I V		028	S	T N		020
033	00250	001	B	B N		028	D	I N		038	N	I N		030
			R	R N		018	S	T N		020				
034	00260	001	B	B N		018	N	I N		028	S	T N		020
035	00269	001	A	R AN		018	B	B N		038	D	I N		048
			ENDOTA	L		108	I	I		050	L	I N		020
			N	I N		040	S	T N		030	U	I N		028
036	00284	001	I	I N		018	N	I N		020				
037	00288	001	B	B N		020	D	I N		030	N	I N		028
			R	R N		018								
038	00294	001	B	B N		018	N	I N		020				
039	00299	001	A	R AN		018	B	B N		030	D	I N		040
			ENDOUT	L		11C	I	I		048	L	I N		020
			N	I N		038	U	I N		028				
040	00308	001	ENDOUT	L		124	I	I		038	IA	I AN		018
			L	I N		020	N	I N		030	U	I N		028
041	00316	001	BA	B AN		018	ENDOUT	L		12C	I	I		038
			L	I N		020	N	I N		030	U	I N		028
042	00324	001	S	T N		018								
043	00327	001	L	L		138	X	I		018				
044	00339	001	INITDV	I		028	OTDV	I		02C	PNTR	I		020
			RDR	I		01C	STORES	I		018	SYSDA	I		024
045	00344	044	A	I		0C0	APPEND	P	02	164	ASSEMB	P	01	18C
			BEHEAD	P	02	174	BETAIL	P	02	178	BUFFER	I		074
			CALL	L		288	CHARIN	I P	00	19C	CHAROU	P	01	1A8
			CLOSEL	P	01	158	COL	I A	01	048	COMBIN	P	02	168
			COPY	L		2C8	DECLAB	L		2B8	DECODE	L		284
			DIGIT	I P	01	1B0	EDIT	L		2D0	EDITOR	L		2E4
			EQUAL	B P	02	17C	ESTORE	I		090	EXCHAN	P	02	16C

NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN
EXIT	L		30C	EXTREF	I		06C	F	I		08C
FACE	P	01	188	FAIL	L		310	FALSE	L		294
FCN1	S	16	13C	FCN2	S	02	140	FF	I		098
FLAG	B		0D9	FLAG1	L		298	H	I A	01	018
HOLD	I		084	INCLAB	L		280	INDEX	I		0C4
INITIA	P	00	148	INITLC	B		0DE	INITLI	L		27C
INPT	I		0A4	INTERF	I		078	INTREF	I		068
ISTACK	I		084	LABEL	I		07C	LABEL1	L		2B4
LAVS	I		060	LINK	I		08C	LST	B		0D8
LSTOBJ	P	00	278	MARK	I		080	MARK1	L		2A0
MATCH	L		2C0	MONITO	P	00	270	MSTACK	I		080
NAME	I		094	NEXT	L		2C4	NEXTCH	I		0D4
NMONTE	I		0CC	NOT	L		29C	NTERMS	I		0D0
NULL	L		2CC	OPENLI	P	01	154	OSTACK	I		088
OUTPT	I		0A8	PCR	I		0B8	PERFOR	B		0DD
PIP	B		0D8	PRIME	B		0DA	PRINT	L		2D4
PROGOU	P	01	1B4	PROGRA	I		070	PULL	P	02	160
PUSH	P	02	15C	PUT	P	01	150	RECYCL	L		280
REFERE	S	05	144	RESETL	L		28C	RETURN	L		28C
REVERS	P	01	170	ROUTE	I		0C8	RTN	B		0DC
SELECT	L		2A4	SETLAB	L		2AC	SPACE1	L		2D8
STEPIN	P	00	184	STOP	L		2E0	SYMBOL	P	00	1C0
T	I A	01	030	TAB1	L		2DC	TAG	I		0AC
TAKE	P	01	14C	TARGET	I		064	TEST	L		2A8
TRANS	I		09C	TRUE	L		290	VARNAM	P	01	1B8
W	I		0A0	WARN	P	01	26C	WRITET	P	01	268

046 00350 045	I	I		018							
047 00356 045	I	I	N	018							
048 00361 045	I	I	V	018							
049 00367 045	I	I		020	V	I	N	018			
050 00373 045	I	I		020	J	I		024	V	I	N 018
051 00380 045	I	I		028	V	I	V	020	X	I	V 018
052 00389 045	I	I		028	V	I	V	020	X	I	N 018
053 00399 045	I	I		028	J	I		02C	V	I	V 020
	X	I	V	018							
054 00408 045	V1	I	V	018	V2	I	N	020			

PBN	SC	PBN SURR	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN
055	00416	045	I	I		028	V1	I N		018	V2	I N		020
056	00422	C45	A	I		024	B	I		028	I	I		020
			V	I N		018								
057	00431	045	I	I V		018	J	I		028	K	I		02C
			V	I V		020								
058	00439	045	I	I V		018	J	I		028	K	I		02C
			V	I V		020								
059	00448	045	EQUAL	B P	02	17C	I	I		030	J	I		034
			K	I		038	OUT	L		180	V1	I V		020
			V2	I V		028								
060	00459	045												
061	00462	045	X	I V		018								
062	00468	045	FOUND	L		194	K	I		028	LOAD	L		198
			LOCATE	P	01	190	REF	I		020	SUB	I		024
			X	I V		018								
063	00471	062	A	I		020	AT	I		028	F	I		02C
			I	I		024	REF	I N		018				
064	00499	045	CHARIN	I P	00	19C	IOCHAR	I P	00	1A0	L	L		1A4
			X	I		020								
065	00501	064	COL	I		024	I	I		020	IOCHAR	I P	00	1A0
066	00520	045	OUT	L		1AC	PRIME	B		020	X	I V		018
067	00532	045	DIGIT	I P	01	1B0	X	I V		020				
068	00537	045	I	I		020	X	I V		018				
069	00558	045	I	I		020	OUT	L		1BC	X	I V		018
070	00569	045	A	I		01C	EDIT	P	01	240	F	I		018
			I	I		020	MNEMON	P	01	1C8	REFERE	P	01	21C
			W	P	01	1C4								
071	00571	C70	S	T N		018								
072	00574	070	CALL	L		1D4	COPY	L		204	EDIT	L		20C

PBN	SC	PBN SURR	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN
			FALSE	L		1E0	FCNNAM	S	16	1D0	FLAG	L		1E4
			FN	I V		018	MARK	L		1EC	MATCH	L		1FC
			NEXT	L		200	NOT	L		1E8	NULL	L		208
			OUT	L		218	PRINT	L		210	REFERE	L		1F8
			RETURN	L		1D8	SELECT	L		1F0	SHOW	P	01	1CC
			STOP	L		214	TEST	L		1F4	TRUE	L		1DC
073	00577	072	S	T N		018								
074	00602	070	DECLAB	L		234	INCLAB	L		22C	LABEL	L		230
			OUT	L		23C	REF	S	05	224	RESETL	L		238
			SETLAB	L		228	SHOW	P	01	220	X	I V		018
075	00605	074	S	T N		018								
076	00617	070	C	L		250	ED	S	06	244	EW	L		24C
			I	I V		018	L	L		25C	N	L		258
			OUT	L		264	SHOW	P	01	248	V	L		260
			X	L		254								
077	00621	076	S	T N		018								
078	00651	045	S	T N		018								
079	00657	045	STRING	T N		018								
080	00663	045	I	I		018	IH	I A	01	038	II	I A	01	020
			IT	I A	01	050	J	I		01C	SHOW	P	02	274
081	00664	080	NAME	T N		020	VAR	I V		018				
082	00725	045	X	I		018								
083	00821	045	A	I		018	B	I		01C				
084	00842	045	A	I		018								
085	00851	045	U	I		01C	V	I		020	W	I		018
086	00862	045	V	I		018								
087	00877	045	A	I		018								
088	00884	045	B	I		018								
089	00910	045	C	L		2F8	CODE	I		018	CONTRO	S	06	2E8

PBN	SC	PBN SURR	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN	NAME	TYPE	DM PR	DSP LN
			L	L		304	N	L		300	SCAN	L		2EC
			SYMBOL	L		2F0	V	L		308	W	L		2F4
			X	L		2FC								
090	00915	089	A	I		018								
091	00920	089	A	I		018	SUB	I		01C	X	I		020
092	00940	089	A	I		018								
093	00945	089	A	I		018	B	I		01C	C	I		020
094	00964	089	A	I		01C	D	I		020	I	I		024
			N	I		018								

OBJECT MODULE SIZE 56908 BYTES.

DATA STORAGE AREA SIZES

PBN	BYTES	PBN	BYTES	PBN	BYTES	PBN	BYTES	PBN	BYTES
001	52	002	44	003	44	004	36	005	44
006	44	007	36	008	52	009	48	010	36
011	64	012	36	013	60	014	68	015	104
016	52	017	32	018	48	019	32	020	104
021	132	022	48	023	44	024	68	025	48
026	48	027	48	028	72	029	88	030	72
031	108	032	60	033	76	034	60	035	104
036	40	037	56	038	40	039	84	040	68
041	68	042	44	043	40	044	48	045	244
046	64	047	36	048	32	049	44	050	48
051	52	052	56	053	60	054	52	055	52
056	52	057	56	058	56	059	72	060	32
061	32	062	60	063	64	064	44	065	52
066	48	067	48	068	64	069	52	070	48
071	44	072	40	073	32	074	40	075	32
076	40	077	32	078	44	079	32	080	124
081	52	082	40	083	36	084	32	085	52
086	44	087	32	088	36	089	40	090	28
091	56	092	40	093	52	094	48		

```
F88-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED MAP,LIST,LET
          VARIABLE OPTIONS USED -  SIZE=(92160,8192)-          DEFAULT OPTION(S) USED

                                   MODULE MAP

CONTROL SECTION

   NAME      ORIGIN   LENGTH

                00      DE4C
IHIFRIXP*      DE50      84
IHIFSARA*      DE08      E4A
IHIFSARB*      ED28      688
IHIIBOOL*      F380      230
IHIIDECM*      F5E0      648
IHIISYMB*      FC28      138
IHIOSTRG*      FD60      128
IHIOSYMB*      FE88      120
IHISLOGM*      FFA8       CC
IHISYSCT*     10078      760
IHIIORTN*     107D8      BFC
IHIPTTAB*     113D8      108

ENTRY
   NAME     LOCATION     NAME     LOCATION     NAME     LOCATION     NAME     LOCATI
IHIDSTAB      DBFC      IHIENTIF     DE40
IHIFRI        DE58
IHIFSAIN      ECD4
IHIIBOAR      F3D0
IHIIDEAI      F5E0      IHIIDEII     F5F4      IHIIDEIR     F60A
IHISLO        FF80
IHIIOROQ      107D8     IHIIOROP     1086C     IHIIORNX     10C0A     IHIIORCL     10E2
IHIIORCP      10FA8     IHIIORGP     11088     IHIIORCN     1108C     IHIIOREN     110C
IHIIOREV      1111E     IHIIORED     11190     IHIIORCI     11240     IHIIORER     112C

ENTRY ADDRESS     ECD4
TOTAL LENGTH     114E0

****GO          DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
```

```
                        1;  -1000;
                       19;      0;
                        0

                -1000;  11;    151;
                         4;     -3;
                        11;    154;
                         4;     -3;
                        11;    156;
                         4;     -3;
                        11;    158;
                         4;     -3;
                        11;    163;
                   -3;   4;     -2;
                         1;  -1001;
                         4;     -2;
                   -4;   1;      0;
                        11;     88;
                         4;     -6;
                         1;      0;
                         1;  -1002;
                         3;     -8;
                         1;  -1003;
                   -8;   2;      0;
                   -6;   2;      0;
                         5;     -4;
                        11;    154;
                         4;    -11;
                        11;    163;
                         4;    -11;
                        11;    153;
                  -11;   4;     -2;
                        16;    999;
                        15;      1;
                   -2;   2;      0;
                         0

                -1001;  16;      1;
                         1;  -1004;
                         4;     -2;
                        16;     19;
                        16;      0;
                        15;      2;
                        16;      0;
                        15;      2;
                        15;      1;
                   -2;   2;      0;
                         0

        -1003;     -3;   1;      0;
                         1;      0;
                        11;     88;
                         3;     -4;
                        11;    154;
                         4;     -8;
                        11;    163;
                         4;     -8;
                        11;    153;
              -8;     -4;   2;      0;
                         6;      0;
```

```
                        CALL        -1000
                          ****          0


        -1000           MATCH 'B'
                        FALSE          -3
                        MATCH 'E'
                        FALSE          -3
                        MATCH 'G'
                        FALSE          -3
                        MATCH 'I'
                        FALSE          -3
                        MATCH 'N'
            -3          FALSE          -2
                        CALL        -1001
                        FALSE          -2
            -4          CALL            0
                        MATCH ;
                        FALSE          -6
                        CALL            0
                        CALL        -1002
                        TRUE           -8
                        CALL        -1003
            -8          RETURN          0
            -6          RETURN          0
                        FLAG           -4
                        MATCH 'E'
                        FALSE         -11
                        MATCH 'N'
                        FALSE         -11
                        MATCH 'D'
           -11          FALSE          -2
                        PRINT         999
                        EDIT            W
            -2          RETURN          0


        -1001           PRINT           1   (CALL  | SETLAB | W )
                        CALL        -1004
                        FALSE          -2
                        PRINT          19
                        PRINT           0
                        EDIT            C
                        PRINT           0
                        EDIT            C
                        EDIT            W
            -2          RETURN          0


        -1003      -3   CALL            0
                        CALL            0
                        MATCH ;
                        TRUE           -4
                        MATCH 'E'
                        FALSE          -8
                        MATCH 'N'
                        FALSE          -8
                        MATCH 'D'
            -8      -4  RETURN          0
                        NOT             0
```

```
                5;      -3;
                2;       0;
                0

-1002;         10;       1;
                1;   -1004;
                4;      -2;
               11;      87;
                4;      -3;
               11;      87;
                4;      -3;
               11;      92;
    -3;         4;      -2;
                1;   -1005;
                4;      -2;
               16;       2;
               16;       0;
               15;       2;
               16;       0;
               15;       2;
               15;       1;
    -2;        10;       5;
                2;       0;
                0

-1004;         11;      96;
                4;      -2;
               12;       0;
               13;       0;
    -6;         1;       0;
                1;       0;
               11;      86;
                6;       0;
                4;      -8;
               13;       0;
               15;       2;
    -8;         2;       0;
                5;      -6;
               11;      86;
                4;      -2;
               15;       5;
    -2;         2;       0;
                0

-1005;         10;       2;
                1;       0;
               11;     167;
                4;      -5;
               11;     141;
    -5;         4;      -4;
               16;      10;
               16;       1;
               15;       2;
    -4;         3;      -3;
               11;     167;
                4;      -9;
               11;     142;
    -9;         4;      -8;
               16;      10;
               16;       2;
               15;       2;
```

```
                FLAG            -3
                RETURN           0

-1002   REFERENCE SETLAB
                CALL         -1004
                FALSE           -2
                MATCH :
                FALSE           -3
                MATCH :
                FALSE           -3
                MATCH =
    -3  FALSE           -2
                CALL         -1005
                FALSE           -2
                PRINT            2  (RETURN| INCLAB | C )
                PRINT            0
                EDIT             C
                PRINT            0
                EDIT             C
                EDIT             W
    -2  REFERENCE RESETLAB
                RETURN           0

-1004   MATCH <
                FALSE           -2
                NEXT             0
                COPY             0
    -6  CALL             0
                CALL             0
                MATCH >
                NOT              0
                FALSE           -8
                COPY             0
                EDIT             C
    -8  RETURN           0
                FLAG            -6
                MATCH >
                FALSE           -2
                EDIT             L
    -2  RETURN           0

-1005   REFERENCE INCLAB
                CALL             0
                MATCH 'R'
                FALSE           -5
                MATCH '1'
    -5  FALSE           -4
                PRINT           10  (REFERENCE )
                PRINT            1  (CALL  | SETLAB | W )
                EDIT             C
    -4  TRUE            -3
                MATCH 'R'
                FALSE           -9
                MATCH '2'
    -9  FALSE           -8
                PRINT           10  (REFERENCE )
                PRINT            2  (RETURN| INCLAB | C )
                EDIT             C
```

```
   -8;      3;     -3;
            14;      0;
   -3;      2;       0;
             4;     -2;
             1;  -1006;
             4;     -2;
            15;      2;
  -15;       1;      0;
            11;     84;
             4;    -17;
            16;      3;
            10;      3;
            15;      2;
            15;      2;
             7;      1;
             1;  -1006;
             4;    -17;
            15;      2;
  -17;       2;      0;
             5;    -15;
             8;      1;
             9;    -26;
            10;      3;
            15;      2;
  -26;       1;      0;
            11;    167;
             4;    -31;
            11;    144;
  -31;       4;    -30;
            16;     10;
            16;      4;
            15;      2;
            15;      2;
  -30;       3;    -29;
            11;    167;
             4;    -37;
            11;    145;
  -37;       4;    -36;
            16;     10;
            16;      5;
            15;      2;
            15;      2;
  -36;       3;    -29;
  -29;       2;      0;
   -2;      10;      4;
             2;      0;
             0
```

```
   -8  TRUE          -3
       NULL           0
   -3  RETURN         0
       FALSE         -2
       CALL       -1006
       FALSE         -2
       EDIT           C
  -15  CALL           0
       MATCH !
       FALSE        -17
       PRINT          3  (TRUE  |  LABEL | X )
       REFERENCE  LABEL
       EDIT           C
       EDIT           C
       MARK           1
       CALL       -1006
       FALSE        -17
       EDIT           C
  -17  RETURN         0
       FLAG         -15
       SELECT         1
       TEST         -26
       REFERENCE  LABEL
       EDIT           C
  -26  CALL           0
       MATCH 'R'
       FALSE        -31
       MATCH '4'
  -31  FALSE        -30
       PRINT         10  (REFERENCE )
       PRINT          4  (FALSE | DECLAB | N )
       EDIT           C
       EDIT           C
  -30  TRUE         -29
       MATCH 'R'
       FALSE        -37
       MATCH '5'
  -37  FALSE        -36
       PRINT         10  (REFERENCE )
       PRINT          5  (FLAG  | RESETLAB )
       EDIT           C
       EDIT           C
  -36  TRUE         -29
  -29  RETURN         0
   -2  REFERENCE DECLAB
       RETURN         0
```

```
-1006;      10;      2;
            14;      0;
   -4;       1;      0;
             1;  -1007;
             4;     -6;
            15;      2;
   -6;       2;      0;
             5;     -4;
             1;      0;
             1;  -1008;
             4;    -10;
            15;      2;
  -58;       1;      0;
            14;      0;
  -62;       1;      0;
```

```
-1006  REFERENCE INCLAB
       NULL           0
   -4  CALL           0
       CALL       -1007
       FALSE         -6
       EDIT           C
   -6  RETURN         0
       FLAG          -4
       CALL           0
       CALL       -1008
       FALSE        -10
       EDIT           C
  -58  CALL           0
       NULL           0
  -62  CALL           0
```

```
                    1;  -1007;                  CALL        -1007
                    4;    -64;                  FALSE         -64
                   15;      2;                  EDIT            C
          -64;      2;      0;          -64     RETURN          0
                    5;    -62;                  FLAG          -62
                    1;  -1008;                  CALL        -1008
                    4;    -60;                  FALSE         -60
                   15;      2;                  EDIT            C
                   16;      4;                  PRINT           4   (FALSE | DECLAB | N )
                   10;      3;                  REFERENCE   LABEL
                   15;      2;                  EDIT            C
                   15;      3;                  EDIT            X
                   15;      2;                  EDIT            C
                   15;      2;                  EDIT            C
                    7;      1;                  MARK            1
          -60;      2;      0;          -60     RETURN          0
                    5;    -58;                  FLAG          -58
                   14;      0;                  NULL            0
          -84;      1;      0;          -84     CALL            0
                    1;  -1007;                  CALL        -1007
                    4;    -86;                  FALSE         -86
                   15;      2;                  EDIT            C
                    7;      1;                  MARK            1
                    7;      2;                  MARK            2
          -86;      2;      0;          -86     RETURN          0
                    5;    -84;                  FLAG          -84
                    8;      2;                  SELECT          2
                    9;    -93;                  TEST          -93
                   16;      4;                  PRINT           4   (FALSE | DECLAB | N )
                   10;      3;                  REFERENCE   LABEL
                   15;      2;                  EDIT            C
                   15;      3;                  EDIT            X
                   15;      2;                  EDIT            C
          -93;     15;      2;          -93     EDIT            C
                    8;      1;                  SELECT          1
                    9;   -100;                  TEST         -100
                   10;      3;                  REFERENCE   LABEL
                   15;      2;                  EDIT            C
-100;     -10;      3;     -9;          -100    TRUE           -9
           -9;      2;      0;           -9     RETURN          0
                   10;      4;                  REFERENCE   DECLAB
                    2;      0;                  RETURN          0
                    0

         -1008;     1;  -1004;          -1008   CALL        -1004
                    4;     -2;                  FALSE          -2
                   16;      1;                  PRINT           1   (CALL  | SETLAB | W )
                   15;      3;                  EDIT            X
                   15;      2;                  EDIT            C
           -2;      3;     -1;           -2     TRUE           -1
                    1;  -1009;                  CALL        -1009
                    3;     -1;                  TRUE           -1
                   11;     37;                  MATCH (
                    4;    -10;                  FALSE         -10
                   16;      1;                  PRINT           1   (CALL  | SETLAB | W )
                   16;      0;                  PRINT           0
                   15;      2;                  EDIT            C
                    1;  -1005;                  CALL        -1005
                    4;    -10;                  FALSE         -10
                   15;      2;                  EDIT            C
                   11;     39;                  MATCH )
                    4;    -10;                  FALSE         -10
                   16;      2;                  PRINT           2   (RETURN| INCLAB | C )
```

```
                    16;      0;
                    15;      2;
                    15;      2;
              -10;   3;     -1;
                    11;     89;
                     4;    -19;
                    16;      1;
                    16;      0;
                    15;      2;
                     1;  -1008;
                     4;    -19;
                    15;      2;
                    16;      6;
                    16;      0;
                    15;      2;
                    15;      2;
        -19;  -1;    2;      0;
                     0
```

```
                  PRINT          0
                  EDIT           C
                  EDIT           C
           -10    TRUE          -1
                  MATCH ¬
                  FALSE        -19
                  PRINT          1  (CALL  | SETLAB | W )
                  PRINT          0
                  EDIT           C
                  CALL       -1008
                  FALSE        -19
                  EDIT           C
                  PRINT          6  (NOT    )
                  PRINT          0
                  EDIT           C
                  EDIT           C
-19       -1      RETURN         0
```

```
              -1007;    10;      2;
                        11;     95;
                         4;     -2;
                        10;      3;
                         1;  -1008;
                         4;     -2;
                        15;      2;
                        16;      5;
                        10;      3;
                        15;      2;
                        15;      2;
                   -2;   3;     -1;
                        11;     97;
                         4;    -15;
                        14;      0;
                         1;      0;
                         1;      0;
                        11;     37;
                         4;    -21;
                         1;  -1010;
                         4;    -21;
                        11;     39;
                   -21;  3;    -20;
                         1;      0;
                        11;     98;
                         6;      0;
                         4;    -24;
                        13;      0;
        -24;  -20;       2;      0;
                         4;    -19;
                        16;     16;
                        15;      3;
                        15;      2;
                        15;      2;
                   -59;  1;      0;
                         1;      0;
                        11;     37;
                         4;    -63;
                         1;  -1010;
                         4;    -63;
                        11;     39;
                   -63;  3;    -62;
                         1;      0;
                        11;     98;
```

```
          -1007   REFERENCE  INCLAB
                  MATCH *
                  FALSE         -2
                  REFERENCE  LABEL
                  CALL       -1008
                  FALSE         -2
                  EDIT           C
                  PRINT          5  (FLAG  | RESETLAB )
                  REFERENCE  LABEL
                  EDIT           C
                  EDIT           C
            -2    TRUE          -1
                  MATCH (/
                  FALSE        -15
                  NULL           0
                  CALL           0
                  CALL           0
                  MATCH (
                  FALSE        -21
                  CALL       -1010
                  FALSE        -21
                  MATCH )
           -21    TRUE         -20
                  CALL           0
                  MATCH /)
                  NOT            0
                  FALSE        -24
                  COPY           0
-24       -20     RETURN         0
                  FALSE        -19
                  PRINT         16  (PRINT  )
                  EDIT           X
                  EDIT           C
                  EDIT           C
           -59    CALL           0
                  CALL           0
                  MATCH (
                  FALSE        -63
                  CALL       -1010
                  FALSE        -63
                  MATCH )
           -63    TRUE         -62
                  CALL           0
                  MATCH /)
```

```
                6;      0;
                4;    -66;
               13;      0;
-66;    -62;    2;      0;
                4;    -61;
               16;     16;
               15;      3;
               15;      2;
               15;      2;
               16;     15;
               16;      2;
               15;      2;
               15;      2;
        -61;    2;      0;
                5;    -59;
        -19;    3;    -18;
        -18;    2;      0;
                4;    -15;
               11;     98;
        -15;    3;     -1;
                1;  -1011;
                4;    -84;
               16;     15;
               15;      3;
               15;      2;
        -84;    3;     -1;
               11;    164;
                4;    -91;
               16;     14;
               16;      0;
               15;      2;
        -91;    3;     -1;
               11;    160;
                4;    -95;
               16;     13;
               16;      0;
               15;      2;
        -95;    3;     -1;
               11;    170;
                4;    -99;
               16;     12;
               16;      0;
               15;      2;
        -99;    3;     -1;
               11;    167;
                4;   -104;
               11;    143;
       -104;    4;   -103;
               16;     10;
               16;      3;
               15;      2;
       -103;    3;     -1;
               11;    162;
                4;   -107;
               16;      7;
                1;  -1012;
                4;   -107;
               15;      2;
       -107;    3;     -1;
               11;    169;
                4;   -112;
               16;      8;
                1;  -1012;
                4;   -112;
```

```
                NOT             0
                FALSE         -66
                COPY            0
-66     -62     RETURN          0
                FALSE         -61
                PRINT          16   (PRINT   )
                EDIT            X
                EDIT            C
                EDIT            C
                PRINT          15   (EDIT    )
                PRINT           2   (RETURN| INCLAB | C )
                EDIT            C
                EDIT            C
        -61     RETURN          0
                FLAG          -59
        -19     TRUE          -18
        -18     RETURN          0
                FALSE         -15
                MATCH /)
        -15     TRUE           -1
                CALL        -1011
                FALSE         -84
                PRINT          15   (EDIT    )
                EDIT            X
                EDIT            C
        -84     TRUE           -1
                MATCH 'O'
                FALSE         -91
                PRINT          14   (NULL    )
                PRINT           0
                EDIT            C
        -91     TRUE           -1
                MATCH 'K'
                FALSE         -95
                PRINT          13   (COPY    )
                PRINT           0
                EDIT            C
        -95     TRUE           -1
                MATCH 'U'
                FALSE         -99
                PRINT          12   (NEXT    )
                PRINT           0
                EDIT            C
        -99     TRUE           -1
                MATCH 'R'
                FALSE        -104
                MATCH '3'
       -104     FALSE        -103
                PRINT          10   (REFERENCE )
                PRINT           3   (TRUE  |  LABEL | X )
                EDIT            C
       -103     TRUE           -1
                MATCH 'M'
                FALSE        -107
                PRINT           7   (MARK    )
                CALL        -1012
                FALSE        -107
                EDIT            C
       -107     TRUE           -1
                MATCH 'T'
                FALSE        -112
                PRINT           8   (SELECT  )
                CALL        -1012
                FALSE        -112
```

```
                    15;      2;                    EDIT          C
                    16;      9;                    PRINT         9  (TEST    )
                    10;      3;                    REFERENCE  LABEL
                    15;      2;                    EDIT          C
                    15;      2;                    EDIT          C
                    11;     37;                    MATCH (
                     4;   -112;                    FALSE      -112
          -121;      1;      0;             -121   CALL          0
                     1;  -1007;                    CALL       -1007
                     4;   -123;                    FALSE      -123
                    15;      2;                    EDIT          C
          -123;      2;      0;             -123   RETURN        0
                     5;   -121;                    FLAG       -121
                    11;     39;                   'MATCH )
                     4;   -112;                    FALSE      -112
                    10;      3;                    REFERENCE  LABEL
                    15;      2;                    EDIT          C
-112;      -1;      10;      4;      -112     -1   REFERENCE DECLAB
                     2;      0;                    RETURN        0
                     0

          -1009;     10;      2;            -1009   REFERENCE INCLAB
                     11;     91;                    MATCH "
                      4;     -2;                    FALSE        -2
                     16;     11;                    PRINT        11  (MATCH   )
                     12;      0;                    NEXT          0
                     13;      0;                    COPY          0
                     15;      2;                    EDIT          C
             -8;      1;      0;               -8   CALL          0
                      1;      0;                    CALL          0
                     11;     91;                    MATCH "
                      6;      0;                    NOT           0
                      4;    -10;                    FALSE       -10
                     16;      4;                    PRINT         4  (FALSE | DECLAB | N )
                     10;      3;                    REFERENCE  LABEL
                     15;      2;                    EDIT          C
                     16;     11;                    PRINT        11  (MATCH   )
                     13;      0;                    COPY          0
                     15;      2;                    EDIT          C
                     15;      2;                    EDIT          C
                     15;      2;                    EDIT          C
                      7;      1;                    MARK          1
          -10;       2;      0;              -10   RETURN        0
                      5;     -8;                    FLAG         -8
                     11;     91;                    MATCH "
                      4;     -2;                    FALSE        -2
                      8;      1;                    SELECT        1
                      9;    -34;                    TEST       -34
                     10;      3;                    REFERENCE  LABEL
                     15;      2;                    EDIT          C
-34;       -2;      10;      4;      -34      -2   REFERENCE DECLAB
                      2;      0;                    RETURN        0
                      0

          -1011;     11;    172;            -1011   MATCH 'W'
                      4;     -2;                    FALSE        -2
                     16;      1;                    PRINT         1  (CALL   | SETLAB | W )
             -2;      3;     -1;               -2   TRUE          -1
                     11;    152;                    MATCH 'C'
                      4;     -6;                    FALSE        -6
                     16;      2;                    PRINT         2  (RETURN| INCLAB | C )
             -6;      3;     -1;               -6   TRUE          -1
```

```
                    11;    173;
                     4;    -10;
                    16;      3;
          -10;       3;     -1;
                    11;    163;
                     4;    -14;
                    16;      4;
          -14;       3;     -1;
                    11;    161;
                     4;    -18;
                    16;      5;
          -18;       3;     -1;
                    11;    171;
                     4;    -22;
                    16;      6;
-22;      -1;       2;      0;
                     0
```

```
                    MATCH 'X'
                    FALSE        -10
                    PRINT          3  (TRUE  | LABEL | X )
              -10   TRUE          -1
                    MATCH 'N'
                    FALSE        -14
                    PRINT          4  (FALSE | DECLAB | N )
              -14   TRUE          -1
                    MATCH 'L'
                    FALSE        -18
                    PRINT          5  (FLAG  | RESETLAB )
              -18   TRUE          -1
                    MATCH 'V'
                    FALSE        -22
                    PRINT          6  (NOT     )
-22           -1    RETURN         0
```

```
          -1012;      1;  -1013;
                      4;     -2;
             -9;      1;      0;
                      1;  -1013;
                      4;    -11;
                     15;      2;
            -11;      2;      0;
                      5;     -9;
                     15;      6;
             -2;      2;      0;
                      0
```

```
          -1012   CALL        -1013
                  FALSE          -2
             -9   CALL            0
                  CALL        -1013
                  FALSE         -11
                  EDIT            C
            -11   RETURN          0
                  FLAG           -9
                  EDIT            V
             -2   RETURN          0
```

```
          -1013;     11;    141;
                      4;     -2;
                     16;     41;
             -2;      3;     -1;
                     11;    142;
                      4;     -6;
                     16;     42;
             -6;      3;     -1;
                     11;    143;
                      4;    -10;
                     16;     43;
            -10;      3;     -1;
                     11;    144;
                      4;    -14;
                     16;     44;
            -14;      3;     -1;
                     11;    145;
                      4;    -18;
                     16;     45;
                     11;    146;
                      4;    -18;
                     16;     46;
            -18;      3;     -1;
                     11;    147;
                      4;    -24;
                     16;     47;
            -24;      3;     -1;
                     11;    148;
                      4;    -28;
                     16;     48;
            -28;      3;     -1;
                     11;    149;
```

```
          -1013   MATCH '1'
                  FALSE          -2
                  PRINT 1
             -2   TRUE           -1
                  MATCH '2'
                  FALSE          -6
                  PRINT 2
             -6   TRUE           -1
                  MATCH '3'
                  FALSE         -10
                  PRINT 3
            -10   TRUE           -1
                  MATCH '4'
                  FALSE         -14
                  PRINT 4
            -14   TRUE           -1
                  MATCH '5'
                  FALSE         -18
                  PRINT 5
                  MATCH '6'
                  FALSE         -18
                  PRINT 6
            -18   TRUE           -1
                  MATCH '7'
                  FALSE         -24
                  PRINT 7
            -24   TRUE           -1
                  MATCH '8'
                  FALSE         -28
                  PRINT 8
            -28   TRUE           -1
                  MATCH '9'
```

```
                    4;    -32;
                   16;     49;
-32;      -1;       2;      0;
                    0


          -1010;    1;  -1014;
                    4;     -2;
                   13;      0;
             -13;   1;      0;
                    1;  -1014;
                    4;    -15;
                   13;      0;
                   15;      2;
            -15;    2;      0;
                    5;    -13;
                   15;      6;
              -2;   2;      0;
                    0


          -1014;   11;     40;
                    3;     -1;
                   11;     41;
                    3;     -1;
                   11;     42;
                    3;     -1;
                   11;     43;
                    3;     -1;
                   11;     44;
                    3;     -1;
                   11;     45;
                    3;     -1;
                   11;     46;
                    3;     -1;
                   11;     47;
                    3;     -1;
                   11;     48;
                    3;     -1;
                   11;     49;
              -1;   2;      0;
                    0

                  999
```

```
                    FALSE       -32
                    PRINT 9
-32       -1        RETURN        0


-1010               CALL        -1014
                    FALSE          -2
                    COPY            0
             -13    CALL            0
                    CALL        -1014
                    FALSE         -15
                    COPY            0
                    EDIT            C
            -15     RETURN          0
                    FLAG          -13
                    EDIT            V
              -2    RETURN          0


-1014               MATCH 0
                    TRUE           -1
                    MATCH 1
                    TRUE           -1
                    MATCH 2
                    TRUE           -1
                    MATCH 3
                    TRUE           -1
                    MATCH 4
                    TRUE           -1
                    MATCH 5
                    TRUE           -1
                    MATCH 6
                    TRUE           -1
                    MATCH 7
                    TRUE           -1
                    MATCH 8
                    TRUE           -1
                    MATCH 9
              -1    RETURN          0
```

```
CONTROL PARAMETERS :
ROUTE := 3; OBJECT CODE LISTING :=  'TRUE'; DEVICE FOR OBJECT CODE LISTING := 1; PERFORMANCE TESTING :=  'TRUE';

GRAMMAR

'BEGIN' <PROGRAM>;  GRAMMAR FOR A SIMPLE PROGRAMMING LANGUAGE.
   THE TARGET COMPUTER IS OF THE POLISH NOTATION CLASS WITH A STACK ACCUMULATOR;
<PROGRAM>::=<DECLARATION LIST>";"'B1'<STATEMENT LIST>"#"'B1'(/*END./)'W' ;
<DECLARATION LIST>::= <DECLARATION>*(";"<DECLARATION>'C') ;
<DECLARATION>::= "("<VARIABLE LIST>")"   'S8C';
<VARIABLE LIST>::= <VARIABLE>(/*VAR,/)'XC'*(","<VARIABLE>(/*VAR,/)'XCC') ;
<STATEMENT LIST>::= <STATEMENT>*(";"<STATEMENT>'C')(/*HLT,/)'C' ;
<STATEMENT>::=<LABEL LIST><BASIC STATEMENT>'CB1C' ;
<LABEL LIST>::= '0'*(<LABEL>":"(/*LAB,/)'XCCM1') 'T1'('B1C') ;
<LABEL>::= <IDENTIFIER> ;
<BASIC STATEMENT> ::= <ASSIGN STMT>|<GO TO STMT>|<IF STMT> ;
<ASSIGN STMT>::= <VARIABLE>"="<EXPRESSION>(/*STO,/)'CC' ;
<GOTO STMT>::= "GO TO"<LABEL>(/*TRA,/)'C' ;
<IF STMT>::= "IF"<EXPRESSION>"="<CONDITION>", GO TO"<LABEL>'XCC' ;
<CONDITION>::= "0"(/*TZE,/) | "+"(/*TPL,/) | "-"(/*TMI,/) ;
<EXPRESSION>::= <TERM>*("+"<TERM>(/*ADD,/)'CC' | "-"<TERM>(/*SUB,/)'CC')
         |"+"<EXPRESSION>|"-"<EXPRESSION>(/*NEG,/)'C' ;
<TERM>::= <FACTOR>*("*"<FACTOR>(/*MUL,/)'CC'|"/"<FACTOR>(/*DIV,/)'CC') ;
<FACTOR>::= <PRIMARY>*("**"<PRIMARY>(/*EXP,/)'CC') ;
<PRIMARY>::= <VARIABLE>(/*CLA,/)'C'|<NUMBER>|"("<EXPRESSION>")"
         | "/"<EXPRESSION>"/"(/*ABS,/)'C' ;
<VARIABLE>::= <IDENTIFIER> ;
<NUMBER>::= (<INTEGER>(<FRACTION>'C'|)|<FRACTION>) (/,/)'C' ;
<IDENTIFIER>::= <LETTER>'K'*(<LETTER>'KC'|<DIGIT>'KC')(/,/)'C' ;
<FRACTION>::= "."'K'<INTEGER>'C' ;
<INTEGER>::= <DIGIT>'K'*(<DIGIT>'KC') ;
<DIGIT>::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" ;
<LETTER>::= "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"
         |"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"
'END'$



 RELATIVE MEASURE OF PREFORMANCE

NUMBER OF ATTEMPTS TO RECOGNIZE A NON TERMINAL TYPE :=       5784;
NUMBER OF ATTEMPTS TO RECOGNIZE A TERMINAL TYPE :=           8949;
```

OBJECT CODE

```
                        1;  -1000;
                       19;      0;
                        0

<PROGRAM>
              -1000;    1;  -1001;
                        4;     -2;
                       11;     88;
                        4;     -2;
                       18;      1;
                        1;  -1002;
                        4;     -2;
                       11;     76;
                        4;     -2;
                       18;      1;
                       16;     95;
                       16;     54;
                       15;      2;
                       16;     63;
                       15;      2;
                       16;     53;
                       15;      2;
                       16;     79;
                       15;      2;
                       15;      1;
                   -2;     2;      0;
                        0

<DECLARATIONLIST>
              -1001;    1;  -1003;
                        4;     -2;
                   -9;     1;      0;
                       11;     88;
                        4;    -11;
                        1;  -1003;
                        4;    -11;
                       15;      2;
                  -11;     2;      0;
                        5;     -9;
                   -2;     2;      0;
                        0

<DECLARATION>
              -1003;   11;     37;
                        4;     -2;
                        1;  -1004;
                        4;     -2;
                       11;     39;
                        4;     -2;
                       17;      8;
```

```
                CALL      -1000  <PROGRAM>
                  ****        0

-1000           CALL      -1001  <DECLARATIONLIST>
                FALSE        -2
                MATCH ;
                FALSE        -2
                  ****        1
                CALL      -1002  <STATEMENTLIST>
                FALSE        -2
                MATCH #
                FALSE        -2
                  ****        1
                PRINT *
                PRINT E
                EDIT         C
                PRINT N
                EDIT         C
                PRINT D
                EDIT         C
                PRINT .
                EDIT         C
                EDIT         W
   -2           RETURN       0

-1001           CALL      -1003  <DECLARATION>
                FALSE        -2
    -9          CALL         0
                MATCH ;
                FALSE       -11
                CALL      -1003  <DECLARATION>
                FALSE       -11
                EDIT         C
   -11          RETURN       0
                FLAG        -9
    -2          RETURN       0

-1003           MATCH (
                FALSE        -2
                CALL      -1004  <VARIABLELIST>
                FALSE        -2
                MATCH )
                FALSE        -2
                STOP         8
```

```
                16;     50;
                15;      2;
                16;     67;
                15;      2;
                16;     77;
                15;      2;
                15;      3;
                15;      2;
        -21;     1;      0;
                11;     77;
                 4;    -23;
                 1;  -1005;
                 4;    -23;
                16;     95;
                16;     71;
                15;      2;
                16;     50;
                15;      2;
                16;     67;
                15;      2;
                16;     77;
                15;      2;
                15;      3;
                15;      2;
                15;      2;
        -23;     2;      0;
                 5;    -21;
         -2;     2;      0;
                 0
```

```
                PRINT A
                EDIT            C
                PRINT R
                EDIT            C
                PRINT ,
                EDIT            C
                EDIT            X
                EDIT            C
        -21     CALL            0
                MATCH ,
                FALSE          -23
                CALL        -1005   <VARIABLE>
                FALSE          -23
                PRINT *
                PRINT V
                EDIT            C
                PRINT A
                EDIT            C
                PRINT R
                EDIT            C
                PRINT ,
                EDIT            C
                EDIT            X
                EDIT            C
                EDIT            C
        -23     RETURN          0
                FLAG           -21
         -2     RETURN          0
```

<STATEMENTLIST>

```
        -1002;     1;  -1006;
                   4;     -2;
         -11;      1;      0;
                   11;    88;
                   4;    -13;
                   1;  -1006;
                   4;    -13;
                   15;     2;
         -13;      2;      0;
                   5;    -11;
                   16;    95;
                   16;    57;
                   15;     2;
                   16;    61;
                   15;     2;
                   16;    69;
                   15;     2;
                   16;    77;
                   15;     2;
                   15;     2;
          -2;      2;      0;
                   0
```

```
        -1002   CALL        -1006   <STATEMENT>
                FALSE          -2
         -11    CALL            0
                MATCH ;
                FALSE          -13
                CALL        -1006   <STATEMENT>
                FALSE          -13
                EDIT            C
         -13    RETURN          0
                FLAG           -11
                PRINT *
                PRINT H
                EDIT            C
                PRINT L
                EDIT            C
                PRINT T
                EDIT            C
                PRINT ,
                EDIT            C
                EDIT            C
          -2    RETURN          0
```

```
<LABELLIST>
                -1007;      14;       0;
                   -4;       1;       0;
                             1;   -1009;
                             4;      -6;
                            11;      87;
                             4;      -6;
                            16;      95;
                            16;      61;
                            15;       2;
                            16;      50;
                            15;       2;
                            16;      51;
                            15;       2;
                            16;      77;
                            15;       2;
                            15;       3;
                            15;       2;
                            15;       2;
                             7;       1;
                    -6;      2;       0;
                             5;      -4;
                             8;       1;
                             9;     -18;
                            18;       1;
                            15;       2;
                   -18;      2;       0;
                             0

<LABEL>
                -1009;       1;   -1010;
                             2;       0;
                             0

<BASICSTATEMENT>
                -1008;       1;   -1011;
                             3;      -1;
                             1;   -1012;
                             3;      -1;
                             1;   -1013;
                    -1;      2;       0;
                             0

<ASSIGNSTMT>
                -1011;       1;   -1005;
                             4;      -2;
                            11;      92;
                             4;      -2;
                             1;   -1014;
                             4;      -2;
                            16;      95;
                            16;      68;
                            15;       2;
```

```
-1007  NULL          0
   -4  CALL          0
       CALL      -1009  <LABEL>
       FALSE        -6
       MATCH :
       FALSE        -6
       PRINT *
       PRINT L
       EDIT          C
       PRINT A
       EDIT          C
       PRINT B
       EDIT          C
       PRINT ,
       EDIT          C
       EDIT          X
       EDIT          C
       EDIT          C
       MARK          1
   -6  RETURN        0
       FLAG         -4
       SELECT        1
       TEST        -18
         ****        1
       EDIT          C
  -18  RETURN        0

-1009  CALL      -1010  <IDENTIFIER>
       RETURN        0

-1008  CALL      -1011  <ASSIGNSTMT>
       TRUE         -1
       CALL      -1012  <GOTOSTMT>
       TRUE         -1
       CALL      -1013  <IFSTMT>
   -1  RETURN        0

-1011  CALL      -1005  <VARIABLE>
       FALSE        -2
       MATCH =
       FALSE        -2
       CALL      -1014  <EXPRESSION>
       FALSE        -2
       PRINT *
       PRINT S
       EDIT          C
```

```
<GOTOSTMT>
                -1012;      11;      56;
                             4;      -3;
                            11;      64;
                             4;      -3;
                            11;      69;
                             4;      -3;
                            11;      64;
                 -3;         4;      -2;
                             1;   -1009;
                             4;      -2;
                            16;      95;
                            16;      69;
                            15;       2;
                            16;      67;
                            15;       2;
                            16;      50;
                            15;       2;
                            16;      77;
                            15;       2;
                            15;       2;
                 -2;         2;       0;
                             0

-1012  MATCH G
       FALSE          -3
       MATCH O
       FALSE          -3
       MATCH T
       FALSE          -3
       MATCH O
 -3    FALSE          -2
       CALL        -1009  <LABEL>
       FALSE          -2
       PRINT *
       PRINT T
       EDIT            C
       PRINT R
       EDIT            C
       PRINT A
       EDIT            C
       PRINT ,
       EDIT            C
       EDIT            C
 -2    RETURN          0

<IFSTMT>
                -1013;      11;      58;
                             4;      -3;
                            11;      55;
                 -3;         4;      -2;
                             1;   -1014;
                             4;      -2;
                            11;      92;
                             4;      -2;
                             1;   -1015;
                             4;      -2;
                            11;      77;
                             4;      -5;
                            11;      56;
                             4;      -5;
                            11;      64;
                             4;      -5;
                            11;      69;
                             4;      -5;
                            11;      64;
                 -5;         4;      -2;
                             1;   -1009;
                             4;      -2;
                            15;       3;
                            15;       2;
                            15;       2;
                 -2;         2;       0;
                             0

-1013  MATCH I
       FALSE          -3
       MATCH F
 -3    FALSE          -2
       CALL        -1014  <EXPRESSION>
       FALSE          -2
       MATCH =
       FALSE          -2
       CALL        -1015  <CONDITION>
       FALSE          -2
       MATCH ,
       FALSE          -5
       MATCH G
       FALSE          -5
       MATCH O
       FALSE          -5
       MATCH T
       FALSE          -5
       MATCH O
 -5    FALSE          -2
       CALL        -1009  <LABEL>
       FALSE          -2
       EDIT            X
       EDIT            C
       EDIT            C
 -2    RETURN          0
```

```
                    16;     77;
                    15;      2;
              -2;    3;     -1;
                    11;     81;
                     4;     -6;
                    16;     95;
                    16;     69;
                    15;      2;
                    16;     65;
                    15;      2;
                    16;     61;
                    15;      2;
                    16;     77;
                    15;      2;
              -6;    3;     -1;
                    11;     80;
                     4;    -10;
                    16;     95;
                    16;     69;
                    15;      2;
                    16;     62;
                    15;      2;
                    16;     58;
                    15;      2;
                    16;     77;
                    15;      2;
        -10;  -1;    2;      0;
                     0
```

```
                    PRINT ,
                    EDIT          C
              -2    TRUE         -1
                    MATCH +
                    FALSE        -6
                    PRINT *
                    PRINT T
                    EDIT          C
                    PRINT P
                    EDIT          C
                    PRINT L
                    EDIT          C
                    PRINT ,
                    EDIT          C
              -6    TRUE         -1
                    MATCH -
                    FALSE       -10
                    PRINT *
                    PRINT T
                    EDIT          C
                    PRINT M
                    EDIT          C
                    PRINT I
                    EDIT          C
                    PRINT ,
                    EDIT          C
-10           -1    RETURN        0
```

<EXPRESSION>

```
                -1014;    1;  -1016;
                           4;     -2;
                  -21;     1;      0;
                          11;     81;
                           4;    -23;
                           1;  -1016;
                           4;    -23;
                          16;     95;
                          16;     50;
                          15;      2;
                          16;     53;
                          15;      2;
                          16;     53;
                          15;      2;
                          16;     77;
                          15;      2;
                          15;      2;
                          15;      2;
                  -23;    3;    -22;
                          11;     80;
                           4;    -31;
                           1;  -1016;
                           4;    -31;
                          16;     95;
```

```
-1014         CALL        -1016   <TERM>
                    FALSE      -2
              -21   CALL        0
                    MATCH +
                    FALSE      -23
                    CALL       -1016   <TERM>
                    FALSE      -23
                    PRINT *
                    PRINT A
                    EDIT          C
                    PRINT D
                    EDIT          C
                    PRINT D
                    EDIT          C
                    PRINT ,
                    EDIT          C
                    EDIT          C
                    EDIT          C
              -23   TRUE       -22
                    MATCH -
                    FALSE      -31
                    CALL       -1016   <TERM>
                    FALSE      -31
                    PRINT *
```

```
-31;    -22;     2;      0;
                 5;   -21;
          -2;    3;    -1;
                11;     81;
                 4;    -39;
                 1;  -1014;
         -39;    3;    -1;
                11;     80;
                 4;    -41;
                 1;  -1014;
                 4;    -41;
                16;     95;
                16;     63;
                15;      2;
                16;     54;
                15;      2;
                16;     56;
                15;      2;
                16;     77;
                15;      2;
                15;      2;
-41;     -1;    2;      0;
                0
```

```
-31     -22  RETURN        0
             FLAG        -21
          -2  TRUE        -1
              MATCH +
              FALSE       -39
              CALL      -1014  <EXPRESSION>
         -39  TRUE        -1
              MATCH -
              FALSE       -41
              CALL      -1014  <EXPRESSION>
              FALSE       -41
              PRINT *
              PRINT N
              EDIT          C
              PRINT E
              EDIT          C
              PRINT G
              EDIT          C
              PRINT ,
              EDIT          C
              EDIT          C
-41       -1  RETURN        0
```

<TERM>

```
          -1016;     1;  -1017;
                     4;     -2;
           -21;      1;      0;
                    11;     95;
                     4;    -23;
                     1;  -1017;
                     4;    -23;
                    16;     95;
                    16;     62;
                    15;      2;
                    16;     70;
                    15;      2;
                    16;     61;
                    15;      2;
                    16;     77;
                    15;      2;
                    15;      2;
                    15;      2;
           -23;     3;    -22;
                    11;     38;
                     4;    -31;
                     1;  -1017;
                     4;    -31;
                    16;     95;
                    16;     53;
                    15;      2;
                    16;     58;
                    15;      2;
                    16;     71;
```

```
-1016  CALL      -1017  <FACTOR>
       FALSE        -2
 -21   CALL          0
       MATCH *
       FALSE       -23
       CALL      -1017  <FACTOR>
       FALSE       -23
       PRINT *
       PRINT M
       EDIT          C
       PRINT U
       EDIT          C
       PRINT L
       EDIT          C
       PRINT ,
       EDIT          C
       EDIT          C
       EDIT          C
 -23   TRUE        -22
       MATCH /
       FALSE       -31
       CALL      -1017  <FACTOR>
       FALSE       -31
       PRINT *
       PRINT D
       EDIT          C
       PRINT I
       EDIT          C
       PRINT V
```

```
<FACTOR>
     -1017;      1; -1018;          -1017  CALL      -1018  <PRIMARY>
                 4;    -2;                 FALSE       -2
       -13;      1;     0;           -13  CALL        0
                11;    95;                 MATCH *
                 4;   -16;                 FALSE      -16
                11;    95;                 MATCH *
       -16;      4;   -15;           -16  FALSE      -15
                 1; -1018;                 CALL      -1018  <PRIMARY>
                 4;   -15;                 FALSE      -15
                16;    95;                 PRINT *
                16;    54;                 PRINT E
                15;     2;                 EDIT         C
                16;    73;                 PRINT X
                15;     2;                 EDIT         C
                16;    65;                 PRINT P
                15;     2;                 EDIT         C
                16;    77;                 PRINT ,
                15;     2;                 EDIT         C
                15;     2;                 EDIT         C
                15;     2;                 EDIT         C
       -15;      2;     0;           -15  RETURN       0
                 5;   -13;                 FLAG       -13
        -2;      2;     0;            -2  RETURN       0
                 0

<PRIMARY>
     -1018;      1; -1005;          -1018  CALL      -1005  <VARIABLE>
                 4;    -2;                 FALSE       -2
                16;    95;                 PRINT *
                16;    52;                 PRINT C
                15;     2;                 EDIT         C
                16;    61;                 PRINT L
                15;     2;                 EDIT         C
                16;    50;                 PRINT A
                15;     2;                 EDIT         C
                16;    77;                 PRINT ,
                15;     2;                 EDIT         C
                15;     2;                 EDIT         C
        -2;      3;    -1;            -2  TRUE        -1
                 1; -1019;                 CALL      -1019  <NUMBER>
                 3;    -1;                 TRUE        -1
                11;    37;                 MATCH (
                 4;    -8;                 FALSE       -8
                 1; -1014;                 CALL      -1014  <EXPRESSION>
                 4;    -8;                 FALSE       -8
                11;    39;                 MATCH )
        -8;      3;    -1;            -8  TRUE        -1
                11;    38;                 MATCH /
                 4;   -11;                 FALSE      -11
                 1; -1014;                 CALL      -1014  <EXPRESSION>
                 4;   -11;                 FALSE      -11
                11;    38;                 MATCH /
                 4;   -11;                 FALSE      -11
```

```
                -11;      -1;       2;       0;
                                    0
```

```
<VARIABLE>
                          -1005;    1;   -1010;
                                    2;       0;
                                    0
```

```
<NUMBER>
                          -1019;    1;       0;
                                    1;   -1020;
                                    4;      -4;
                                    1;       0;
                                    1;   -1021;
                                    4;      -6;
                                   15;       2;
                               -6;  3;      -5;
                               -5;  2;       0;
                               -4;  3;      -3;
                                    1;   -1021;
                               -3;  2;       0;
                                    4;      -2;
                                   16;      77;
                                   15;       2;
                               -2;  2;       0;
                                    0
```

```
<IDENTIFIER>
                          -1010;    1;   -1022;
                                    4;      -2;
                                   13;       0;
                               -19; 1;       0;
                                    1;   -1022;
                                    4;      -21;
                                   13;       0;
                                   15;       2;
                               -21; 3;      -20;
                                    1;   -1023;
                                    4;      -26;
                                   13;       0;
                                   15;       2;
                    -26;       -20; 2;       0;
                                    5;      -19;
                                   16;      77;
                                   15;       2;
                               -2;  2;       0;
                                    0
```

```
<FRACTION>
                          -1021;   11;      79;
                                    4;      -2;
                                   13;       0;
                                    1;   -1020;
                                    4;      -2;
```

```
-11     -1  RETURN          0
```

```
        -1005  CALL        -1010  <IDENTIFIER>
               RETURN          0
```

```
        -1019  CALL            0
               CALL        -1020  <INTEGER>
               FALSE          -4
               CALL            0
               CALL        -1021  <FRACTION>
               FALSE          -6
               EDIT            C
           -6  TRUE           -5
           -5  RETURN          0
           -4  TRUE           -3
               CALL        -1021  <FRACTION>
           -3  RETURN          0
               FALSE          -2
               PRINT           ,
               EDIT            C
           -2  RETURN          0
```

```
        -1010  CALL        -1022  <LETTER>
               FALSE          -2
               COPY            0
           -19 CALL            0
               CALL        -1022  <LETTER>
               FALSE         -21
               COPY            0
               EDIT            C
           -21 TRUE          -20
               CALL        -1023  <DIGIT>
               FALSE         -26
               COPY            0
               EDIT            C
-26        -20 RETURN          0
               FLAG          -19
               PRINT           ,
               EDIT            C
           -2  RETURN          0
```

```
        -1021  MATCH           .
               FALSE          -2
               COPY            0
               CALL        -1020  <INTEGER>
               FALSE          -2
```

```
                    4;      -14;                FALSE       -14
                   13;        0;                COPY          0
                   15;        2;                EDIT          C
          -14;      2;        0;       -14      RETURN        0
                    5;      -12;                FLAG        -12
           -2;      2;        0;        -2      RETURN        0
                    0

<DIGIT>
        -1023;     11;       40;      -1023     MATCH 0
                    3;       -1;                TRUE         -1
                   11;       41;                MATCH 1
                    3;       -1;                TRUE         -1
                   11;       42;                MATCH 2
                    3;       -1;                TRUE         -1
                   11;       43;                MATCH 3
                    3;       -1;                TRUE         -1
                   11;       44;                MATCH 4
                    3;       -1;                TRUE         -1
                   11;       45;                MATCH 5
                    3;       -1;                TRUE         -1
                   11;       46;                MATCH 6
                    3;       -1;                TRUE         -1
                   11;       47;                MATCH 7
                    3;       -1;                TRUE         -1
                   11;       48;                MATCH 8
                    3;       -1;                TRUE         -1
                   11;       49;                MATCH 9
           -1;      2;        0;        -1      RETURN        0
                    0

<LETTER>
        -1022;     11;       50;      -1022     MATCH A
                    3;       -1;                TRUE         -1
                   11;       51;                MATCH B
                    3;       -1;                TRUE         -1
                   11;       52;                MATCH C
                    3;       -1;                TRUE         -1
                   11;       53;                MATCH D
                    3;       -1;                TRUE         -1
                   11;       54;                MATCH E
                    3;       -1;                TRUE         -1
                   11;       55;                MATCH F
                    3;       -1;                TRUE         -1
                   11;       56;                MATCH G
                    3;       -1;                TRUE         -1
                   11;       57;                MATCH H
                    3;       -1;                TRUE         -1
                   11;       58;                MATCH I
                    3;       -1;                TRUE         -1
                   11;       59;                MATCH J
                    3;       -1;                TRUE         -1
                   11;       60;                MATCH K
                    3;       -1;                TRUE         -1
```

```
                11;      66;
                 3;      -1;
                11;      67;
                 3;      -1;
                11;      68;
                 3;      -1;
                11;      69;
                 3;      -1;
                11;      70;
                 3;      -1;
                11;      71;
                 3;      -1;
                11;      72;
                 3;      -1;
                11;      73;
                 3;      -1;
                11;      74;
                 3;      -1;
                11;      75;
-1;              2;       0;
                 0

              999
```

```
                MATCH Q
                TRUE            -1
                MATCH R
                TRUE            -1
                MATCH S
                TRUE            -1
                MATCH T
                TRUE            -1
                MATCH U
                TRUE            -1
                MATCH V
                TRUE            -1
                MATCH W
                TRUE            -1
                MATCH X
                TRUE            -1
                MATCH Y
                TRUE            -1
                MATCH Z
-1              RETURN           0
```

```
CONTROL PARAMETERS :
ROUTE := 1; OBJECT CODE LISTING :=  'TRUE'; DEVICE FOR OBJECT CODE LISTING := 1; PERFORMANCE TESTING :=  'TRUE';

SOURCE LANGUAGE TEXT

   (A,B,T);
    B=(A+1)/2 ;
S1: T=B;
    B=B+(A/B-B)/2 ;
    IF /B-T/-0.0001=+, GO TO S1#
$



 RELATIVE MEASURE OF PREFORMANCE

NUMBER OF ATTEMPTS TO RECOGNIZE A NON TERMINAL TYPE :=        353;
NUMBER OF ATTEMPTS TO RECOGNIZE A TERMINAL TYPE :=           1412;
```

OBJECT CODE

```
*VAR,A,*VAR,B,*VAR,T,
B,A,*CLA,1,*ADD,2,*DIV,*STO,
*LAB,S1,
T,B,*CLA,*STO,
B,B,*CLA,A,*CLA,B,*CLA,*DIV,B,*CLA,*SUB,2,*DIV,*ADD,*STO,
B,*CLA,T,*CLA,*SUB,*ABS,0.0001,*SUB,S1,*TPL,
*HLT,
*END.
```

APPENDIX F

LIST OF DEFINITIONS

LIST OF DEFINITIONS

assembler - a compiler that translates an assembly language into machine code.

assembly language - a computer language that employs mnemonic operation codes and symbolic addressing and has a 1 - 1 correspondence with machine language instructions.

Backus Normal Form - a type of meta-language designed by John Backus of IBM and Peter Naur as a tool to define programming languages. It was used in the official definition of ALGOL 60 and attracted attention to the possibility of syntax-directed compilation.

binary tree - an extension of the idea of an ordered pair, with the further provision that each of the two elements of the pair may itself be an ordered pair, and so forth to an arbitrary finite depth.

bottom-up parsing - examines the ultimate terminal types (syntactic constants) and matches them against the definition in the grammar. The resulting syntactic variables are matched again against the definitions, and so forth until the root constituent is reached.

compiler - a program to translate a source language into a target language. Also called "translator".

direct compiler - a compiler using ad hoc or direct techniques for compilation. The syntax and semantics of the source language are buried in the encoding of the compiler. These translators are generally very efficient but restricted to one language and not easily modified.

formal grammar - a complete set of grammatical (syntactic) rules for a language, written in a meta-language such as BNF.

formal language - a language definable by a formal grammar (e.g. ALGOL).

grammar - a complete set of grammatical (syntactic) rules for a language.

language - a finite linear sequence of admissable strings (i.e. syntax) and the meaning (semantics) attached to them.

list - general term referring to any sequence of elements. In our context, it refers to a linear sequence of elements.

list processing - refers to a type of computer programming directed toward symbol-manipulation. There are many variants but most share the following features:

1) data representation - strings, list structures, or binary trees. These are used to create complex data structures, the data containing symbolic and/or numeric information. Information is also carried by the relational structure.

2) storage allocation - A "list of available space" is maintained. Locations are assigned to data-structures dynamically as required, and are chained together by pointers rather than occupying sequential location.

A location or list of locations may be returned to the "list of available space" when no longer needed.

3) push down stores - all use push down stores or stacks.

4) recursive operations - are often necessary for constructing or processing list structures of arbitrary complexity.

list structure - a list of elements, where these elements may again be lists, structures, and so forth to an arbitrary finite depth.

LR(k) grammar - a type of grammar such that the syntax symbols of the described language can be unambiguously recognized in context of everything that has already been scanned plus k symbols to the right, i.e. left-to-right scan using all text to the left and k terminal symbols to the right.

meta-grammar - term used by Reeves to signify a grammar defining a meta-language, i.e. in our context, it would be the initialization assembler program.

meta-language - a language with which to define languages; in our context, it emcompasses both syntax and semantics of a language.

meta-semantic language - a language with which to define the semantics (meaning attached to syntactic parts) of a language.

meta-syntactic language - a language with which to define the syntax (structure) of languages.

object language - the language (code) that is output by a compiler. Also called a target language.

parsing - the analysis of a language in order to discover its particular syntax (grammatical structure). More specifically, parsing means to find the hierarchy of syntactic rules which defines the language text.

phrase-structure grammar - formal grammar for a phrase-structured language. For our purposes, it is synonomous with formal grammar.

phrase-structure language - a language consisting of a hierarchical system of admissable phrases. It is definable by a phrase-structure grammar.

precedence grammar - a type of grammar such that unique relations or "precedences" hold between all or certain sets of syntactic units. This allows parsing efficiencies during translation.

Princeton-type computer - computer whose accumulators are individually accessed by instructions and not considered a stack (e.g. IBM 360, CDC 6600).

semantics - the meaning attached to the admissable phrases (strings) of a language. In the sense of a programming language, this means the execution of the program as hoped for by the programmer. From a compiler viewpoint, it refers to the object code generated.

sequential computer - a computer which executes each instruction in the program sequentially. This sequence can be broken and

control transferred to a new instruction sequence only by a "go to" instruction.

source language - the language which a compiler inputs and translates to object code.

stack - a sequential set of memory locations that operates in the fashion of the common cafeteria tray stack. Placing a tray on top of the stack "pushes down" the previous tray; removing a tray "pops up" the previous tray to the top of the stack. A stack always works on a "last in - first out" principle.

stack computer - a computer with "stack" accumulators; that is, accumulators which are accessed by instructions as a "stack" in a push-pull manner.

Instructions are fetched sequentially from memory, individual instructions generally being either an operator or an operand. Operands, when fetched, are immediately placed in the "top" accumulator in the accumulator stack, pushing previous operand entries down one. Operators, when fetched, are executed on the top one or two accumulators in the accumulator stack (e.g. Burroughs B5000, English KDF9).

string - a linear sequence of storage locations. The number of elements in the string can vary, however, during a computer run.

syntax - set of rules defining admissable constructs (strings) in a language, and how to combine simple constructs into more complex strings.

top-down parsing - is an approach to parsing, which establishes goals and sub-goals and attempts to relate its environment to these goals by manipulating it in a trial-and-error fashion. It searches down the grammar tree beginning at the root definition, establishing each constituent as a sub-goal, until matching the ultimate terminal types (syntactic constants) against the source text.

APPENDIX G

A DETAILED EXAMPLE OF PARSING

APPENDIX G

A DETAILED EXAMPLE OF PARSING

In section 2.2.3.2, an example of the program for the following grammar was given.

```
'BEGIN' <sentence>;
<sentence>::=<subject><predicate>;
<subject>::=<noun phrase>;
<noun phrase>::=<article><noun>;
<predicate>::=<verb><object>;
<object>::=<noun phrase>;
<article>::= "The"|"A";
<noun>::= "Boy"|"Tree";
<verb>::= "Sees"
'END'
```

When the parser program was presented with the sample sentence "The boy sees a tree", a trace on the execution of this program would appear as follows on page 183 where the trace of each step is printed before execution.

Input stream:

	T	H	E		B	O	Y		S	E	E	S		A		T	R	E	E
Location	1	2	3		4	5	6		7	8	9	10		11		12	13	14	15

NOTE: Blanks are ignored.

"pcr"	FLAG SETTING	"LINK" STACK	input STACK "iSTACK"	"input"	SYMBOLIC INSTRUCTION	
1	....	....	....	1	CALL	sentence
3	true	2	1	1	CALL	subject
7	true	2,4	1,1	1	CALL	noun phrase
9	true	2,4,8	1,1,1	1	CALL	article
19	true	2,4,8,10	1,1,1,1	1	MATCH	T
20	true	2,4,8,10	1,1,1,1	2	FALSE	L4
21	true	2,4,8,10	1,1,1,1	2	MATCH	1+
22	true	2,4,8,10	1,1,1,1	3	FALSE	L4
23	true	2,4,8,10	1,1,1,1	3	MATCH	E
24	true	2,4,8,10	1,1,1,1	4	TRUE	L5
26	true	2,4,8,10	1,1,1,1	4	RETURN	
10	true	2,4,8	1,1,1	4	FALSE	L2
11	true	2,4,8	1,1,1	4	CALL	noun
27	true	2,4,8,12	1,1,1,4	4	MATCH	B
28	true	2,4,8,12	1,1,1,4	5	FALSE	L6
29	true	2,4,8,12	1,1,1,4	5	MATCH	0
30	true	2,4,8,12	1,1,1,4	6	FALSE	L6
31	true	2,4,8,12	1,1,1,4	6	MATCH	Y
32	true	2,4,8,12	1,1,1,4	7	TRUE	L7
40	true	2,4,8,12	1,1,1,4	7	RETURN	
12	true	2,4,8	1,1,1	7	RETURN	
8	true	2,4	1,1	7	RETURN	
4	true	2	1	7	FALSE	L1
5	true	2	1	7	CALL	predicate
13	true	2,6	1,7	7	CALL	verb
41	true	2,6,14	1,7,7	7	MATCH	S
42	true	2,6,14	1,7,7	8	FALSE	L8
43	true	2,6,14	1,7,7	8	MATCH	E
44	true	2,6,14	1,7,7	9	FALSE	L8
45	true	2,6,14	1,7,7	9	MATCH	E
46	true	2,6,14	1,7,7	10	FALSE	L8
47	true	2,6,14	1,7,7	10	MATCH	S
48	true	2,6,4	1,7,7	11	RETURN	
14	true	2,6	1,7	11	FALSE	L3
15	true	2,6	1,7	11	CALL	object
17	true	2,6,16	1,7,11	11	CALL	noun phrase
9	true	2,6,16,18	1,7,11,11	11	CALL	article
19	true	2,6,16,18,10	1,7,11,11,11	11	MATCH	T
20	false	2,6,16,18,10	1,7,11,11,11	11	FALSE	L4
24	false	2,6,16,18,10	1,7,11,11,11	11	TRUE	L5
25	true	2,6,16,18,10	1,7,11,11,11	11	MATCH	A

"pcr"	FLAG SETTING	"LINK" STACK	input STACK "iSTACK"	"input"	SYMBOLIC INSTRUCTION	
26	true	2,6,16,18,10	1,7,11,11,11	12	RETURN	
10	true	2,6,16,18	1,7,11,11	12	FALSE	L2
11	true	2,6,16,18	1,7,11,11	12	CALL	noun
27	true	2,6,16,18,12	1,7,11,11,12	12	MATCH	B
28	false	2,6,16,18,12	1,7,11,11,12	12	FALSE	L6
32	false	2,6,16,18,12	1,7,11,11,12	12	TRUE	L7
33	true	2,6,16,18,12	1,7,11,11,12	12	MATCH	T
34	true	2,6,16,18,12	1,7,11,11,12	13	FALSE	L7
35	true	2,6,16,18,12	1,7,11,11,12	13	MATCH	R
36	true	2,6,16,18,12	1,7,11,11,12	14	FALSE	L7
37	true	2,6,16,18,12	1,7,11,11,12	14	MATCH	E
38	true	2,6,16,18,12	1,7,11,11,12	15	FALSE	L7
39	true	2,6,16,18,12	1,7,11,11,12	15	MATCH	E
40	true	2,6,16,18,12	1,7,11,11,12	16	RETURN	
12	true	2,6,16,18	1,7,11,11	16	RETURN	
18	true	2,6,16	1,7,11	16	RETURN	
16	true	2,6	1,7	16	RETURN	
6	true	2	1	16	RETURN	
2	true	....	....	16	STOP	

REFERENCES

1. Reeves, C. M., Description of a Syntax-Directed Translator. Computer Journal, Vol. , pp. 244, 1968.

2. Metcalfe, H. H., A Parameterized Compiler Based on Mechanical Linguistics. Annual Review, Automatic Programming, Vol. 4, pp. 125, (R. Goodman, Editor). Pergammon Press, 1964.

3. Backus, J. W., et al, Revised Report on the Algorithmic Language ALGOL 60. Computer Journal, Vol. 5, pp. 349, 1963.

4. Chomsky, Noam, On Certain Formal Properties of Grammars, Inf. and Control, Vol. 2, pp. 137 - 167, June, 1959.

5. IFIP/WG2.1, Report on Input-Output Procedures for ALGOL 60. CACM, Vol. 7, No. 10, pp. 628, 1964.

6. Irons, E. T., A Syntax-Directed Compiler for ALGOL 60. CACM, Vol. 4, pp. 51, January, 1961.

7. Warshall, S. and Shapiro, R. M., A General Purpose Table Driven Compiler. Proc. E.J.C.C., AFIPS, Vol. 25, pp. 59, 1964.

8. Brooker, R. A., Morris, D., McCullum, I. R. and Rohl, J. S., The Compiler Compiler. Annual Review in Automatic Programming, Vol. 3, pp. 229 - 275, 1963.

9. Kanner, H., Kosinski, P. and Robinson, C. L., The Structure of Yet Another ALGOL Compiler. CACM, Vol. 8, pp. 427, July, 1965.

10. KDF9 ALGOL, Duncan, F. G., Input and Output for ALGOL 60 on KDF9, Computer Journal, Vol. 5, pp. 341 - 4.

11. DeVogelaere, R., A Set of Basic I/O Procedures. CACM, Vol. 11, No. 8, pp. 567 - 573, 1968.

12. McKeeman, W., Horning, J., Wortman, D., A Compiler Generator, pp. 125 - 164, 1967.

13. Floyd, R. W., The Syntax of Programming Languages - A Survey. IEEE Transactions on Electronic Computers, Vol. EC-131, August, 1964.

14. Feldman, J. and Gries, ., Translator Writing Systems, CACM, Vol. 11, No. 2, February, 1968.

15. Wirth, N. and Weber, H., Euler: A Generalization of ALGOL and Its Formal Definition: Part 1. CACM, Vol. 9, No. 1, January, 1966.

16. Foxley, E. and King, P., The Implementation of Syntax Analysis Using ALGOL, and Some Mathematical Applications. Computer Journal, Vol. 10, No. 4, February, 1968.

17. De Remer, F. L., Practical Translators for LR(k) Languages. Ph.D. Thesis, M.I.T., October, 1969.

18. Early, J., An Effecient Context-free Parsing Algorithm. CACM, Vol. 13, No. 2, February, 1970.

19. Foulkes, Bruce, A Translator Writing System for Simple Precedence Grammars. Thesis, University of Manitoba, March, 1971.

20. Cohen, D. J. and Gotlieb, C. C., A List Structure of Grammars for Syntactic Analysis, Computing Surveys, Vol. 2, No. 1, 1970.