MAC-TR-42

DESIGN AND IMPLEMENTATION
OF A
TABLE-DRIVEN COMPILER SYSTEM

by

Chung L. Liu

Gabriel D. Chang

Richard E. Marks

July 1967

Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ABSTRACT

Our goal is to provide users of the table-driven compiler system with an environment within which they can freely design and produce their compilers. The primary design criterion is generality so that the users can define a large class of input languages oriented toward any kind of problem-solving purposes, and can also define a large class of object programs to be executed on different computer systems. Therefore, in our system we do not limit the users to specific ways of doing syntactic analysis, or doing storage allocation, or producing binary programs of a specific format for a particular computer system. What we provide are mechanisms that are general enough for whichever way a user desires to build his compiler.

The table-driven compiler system consists of a base program and two fixed higher-level languages – the Table Declaration and Manipulation Language and the Macro Interpretation Language – together with the corresponding translators which generate the control tables according to the user's specification. A third higher-level language – the Syntax Defining Language – and its corresponding translator are also needed. However, their definitions are left to the users for the reason of providing them with greater flexibility in specifying the method of syntactic analysis. The base program is controlled by the control tables to perform the task of translating source programs into object machine codes. It is a general program which is independent of the particular source language being translated as well as the method of translation. The control tables contain an encodement of the syntax of the source language, an encodement of the method of translation and an encodement of the characteristics of the target machine.

In our design, we emphasize the segmentation of the system so that the functions of each section will be clearly defined and be brought out in evidence. The communication problem between the segments is not a difficult one to handle as illustrated in our design. It should also be pointed out that for the generality and flexibility we try to attain, less consideration is placed on efficiency.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I

INTRODUCTION AND OVERVIEW

The application of digital computers to diverse fields has prompted the design of many problem-oriented programming languages. Although developing a compiler for a special purpose language is no longer a mysterious task, it is still, in most cases, a tedious task that may consume many man-years. The purpose of developing a table-driven compiler system is to allow a language designer to produce and modify a compiler for his special language at a reduction of the time currently required. This facility provides a simulation environment for testing new syntactic constructions and new translation techniques for the source language, and lends itself to the more rapid development of new programming languages, especially in a time-sharing environment.

The notion of a "table-driven compiler" is an extension of the notion of a "syntax-directed compiler" first studied by E. Irons. The difference between a conventional (i.e., not syntax-directed or table-driven compiler) and a syntax-directed compiler is that in a conventional compiler the syntax of the source language is buried in the coding of the compiler itself; the slightest deviation from the original syntax requires tampering with the original coding of the compiler - often, a hopeless task. In a syntax-directed compiler, the encoding of the syntax of the source language is kept in tables separated from the remainder of the compiler. The tables control the recognition of strings in the source language and may be readily changed so that the same processing program may handle source languages of differing syntax.

The idea of using replaceable tables to specify the syntax of a source language to a compiler is extended in this report. In addition to tabular control of syntactic analysis, the system presented here allows the compiler designer to construct tables controlling the allocation of storage space, the method of translation, and the assembly of binary machine code. To design a compiler for a new source language, the designer need only specify these tables. To modify a compiler, he need only change the appropriate entries in the existing tables.

The design philosophy of our "Table-driven Compiler System" is not to provide the user with an all-inclusive set of compiling facilities, but rather to provide him with an *environment* within which he can freely design and produce his own compiler. We wish to allow as large a class of problem-oriented input languages and object (i.e. machine) languages as possible. We try *not* to limit the compiler designer to specific methods for syntactic analysis or storage allocation or to specific binary machine codes.

SECTION II

GENERAL ORGANIZATION

2.1 INTRODUCTION

The table-driven compiler system described here consists of a) a base program and b) a set of control tables for controlling the operation of the base program. The control tables, in turn, are specified by statements in the corresponding control languages. The base program, when supplied with a set of control tables, first translates source programs into an equivalent set of "macro" instructions and then generates the binary machine code for the macro instructions. When interpreted by their bootstrap translators, statements in the control languages are encoded into the control tables needed by the base program to govern the method of syntactic analysis, the allocation of storage space, and the translation of the "macro" instructions.

To provide the base program with a complete set of control tables, the designer must prepare sets of statements in three control languages. In the first of these languages, the "Syntax Defining Language", the designer specifies the control tables for syntactic analysis. Both the Syntax Defining Language and its bootstrap translator must be prepared by the designer. It is expected that eventually two or three syntax defining languages and their bootstrap translators will be held within the system for a general use. In the second of these languages, the "Table Declaration and Manipulation Language", the designer specifies the control tables for allocation of storage space. In the third of these languages, the "Macro Interpretation Language", the designer specifies the control tables for the method of translation of the equivalent "macro" instructions generated by the base program from the source language program. The latter two languages and their bootstrap translators are provided in the system.

As shown in Figure 2-1, the base program can be divided into three parts: the Syntactic Analyzer, the Table Processor, and the Assembler. Each part is controlled by one or more control tables, as shown in Figure 2-2.

2.2 THE SYNTACTIC ANALYZER

The Syntactic Analyzer scans programs written in the source language, recognizes syntactic types, and generates a set of equivalent macro instructions that will later be interpreted by the assembler. The Syntactic Analyzer also transmits storage allocation information to the Table Processor. The Syntactic Analyzer is controlled by three tables: the Lexical Table, the Test Table, and the Action Table. (See Figure 2-2.) The Lexical Table and the Test

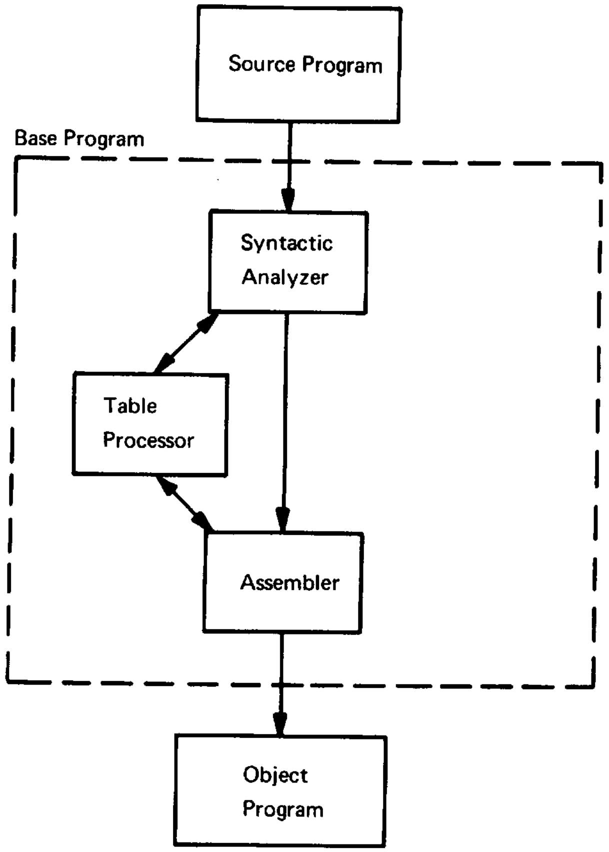
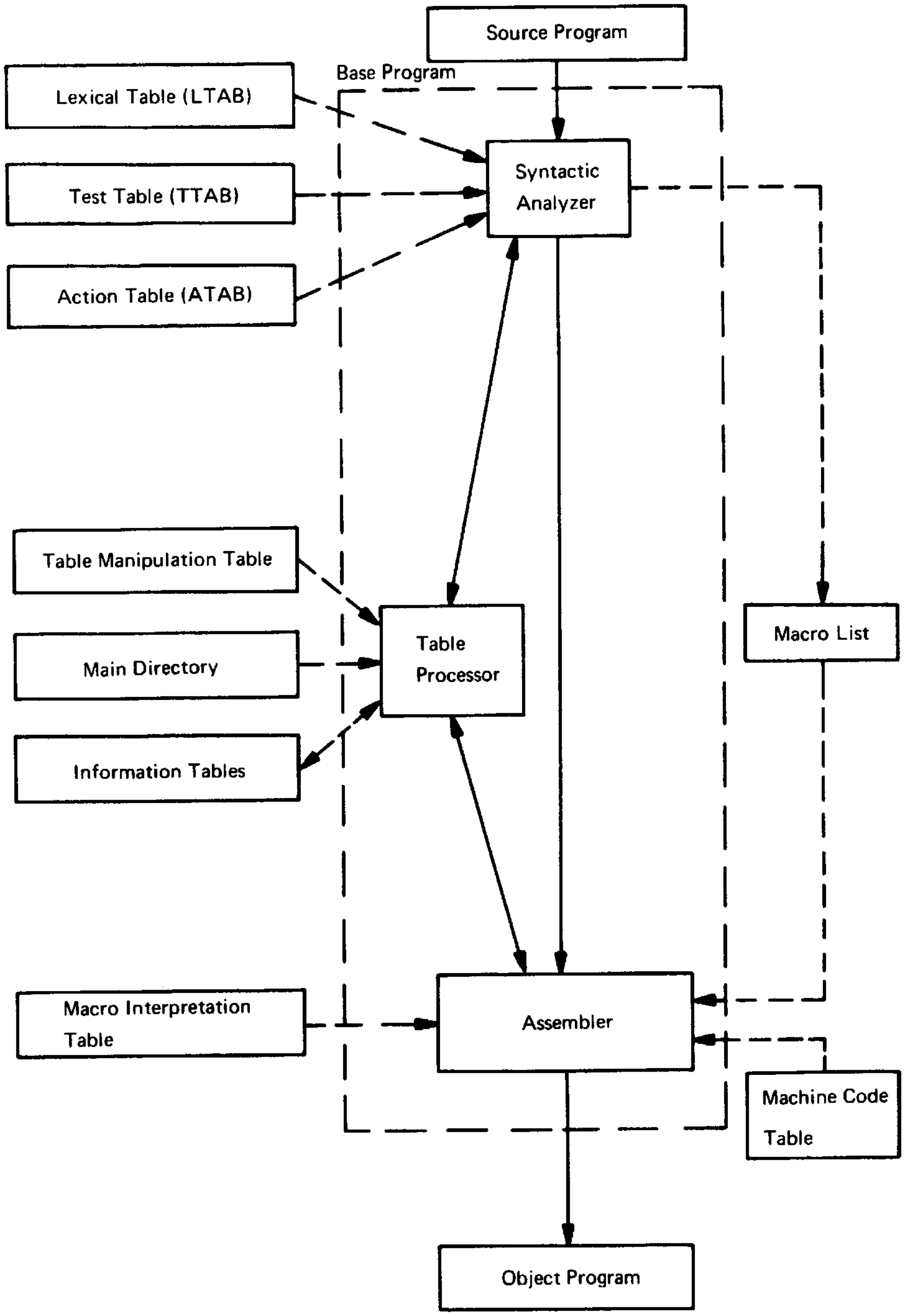


Figure 2-1. Organization of the Base Program

Table control the recognition of syntactic types. The Action Table controls the generation of macros and the passage of related information to the Table Processor.

2.3 THE TABLE PROCESSOR

The Table Processor is divided into two parts. The first part accepts an item of information (e.g., a variable name) from the Syntactic Analyzer, enters it into the appropriate information table (e.g., a symbol table), and returns a pointer to the item (e.g., the pointer to the corresponding entry in the symbol table). The second part, called after the Syntactic Analyzer has completed its analysis, sorts and merges the information tables and assigns addresses to the symbols and literals within the tables. The Table Processor is controlled by two tables: the Main Directory, and the Table Manipulation Table. The Main Directory contains the format specification of the information tables, i.e., the maximum number of entries in each information table, the number of fields in each entry, the packing mask and shift for each field, and a sorting indicator designating whether the table should



Conventions: ——► Solid lines indicate program flow

□ – – – –► Dotted lines *away from* tables designate control tables *used* by the base program

□◄ – – – – Dotted lines *into* tables designate control tables *formed* by the base program

These conventions will apply to all figures in the text

Figure 2-2. Tabular Control of the Base Program

be kept sorted. The Table Manipulation Table designates how the information tables are to be processed after they have been constructed during the syntactic analysis.

2.4 THE ASSEMBLER

The Assembler interprets the list of macro instructions generated by the syntactic analyzer and produces the corresponding machine code. The Assembler is controlled by two tables: the Macro Interpretation Table and the Machine Code Table. The Macro Interpretation Table specifies how each macro is to be translated. The Machine Code Table gives the binary code for each machine instruction. The Assembler frequently calls the Table Processor to extract information collected in the information tables.

To design a compiler for a particular source language, the designer must specify a set of control tables for the source language. (See Figure 2-3.) Using the Syntax Defining Language, he must specify the rules for recognizing source language constructions and the macros to be generated upon the recognition of these constructions. This information must be assimilated by a bootstrap translator and stored in the Lexical Table, Test Table and Action Table. Using the Table Declaration and Manipulation Language, he must declare all information tables to be used by the base program and the way these tables are to be sorted or merged. This information must be assimilated by a second bootstrap translator and stored in the Main Directory and the Table Manipulation Table. Using the Macro Interpretation Language he must specify the machine code translation of the macros generated by the Syntactic Analyzer. This information must be processed by a third bootstrap translator and stored in the Macro Interpretation Table. (An extended example of the use of the control languages is given in Section VI.) The designer must also supply a Machine Code Table and a number of parameters to the compiler system, such as the length of certain temporary storage blocks, the number of machine registers in the computer in which the object program will run, and the identification bits for each instruction.

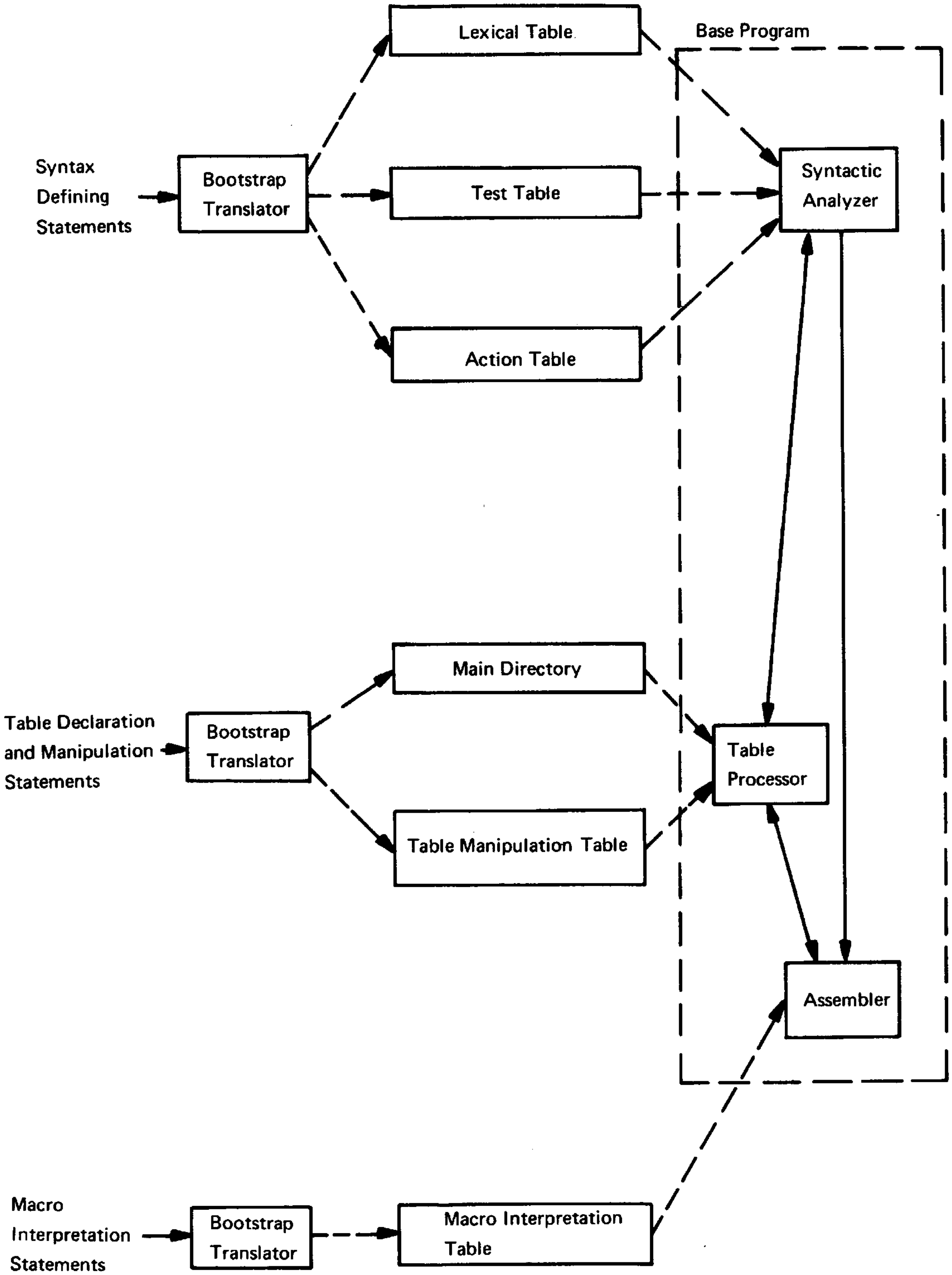


Figure 2-3. The Bootstrap Operation

SECTION III

THE SYNTACTIC ANALYZER

3.1 INTRODUCTION

The purpose of the Syntactic Analyzer is to operate on the source language input strings and produce a list of equivalent macro instructions. The analyzer consists of three routines, called LEXICAL, TEST, and ACTION. These routines have control tables, respectively called LTAB, TTAB, and ATAB. Another table, STAB (pronounced S-TAB), is used to store the results of partial analysis. Figure 3-1 shows the organization of the analyzer.

Routine LEXICAL, as controlled by LTAB, performs the lexical analysis on the basic syntactic types of the input string. When a basic syntactic type is recognized (a variable name or literal), LEXICAL passes this information (via routine ACTION) to the Table Processor for entry into an information table. The Table Processor returns a pointer to the newly formed entry *. This pointer will be stored in the table STAB and control will be given to routine TEST.

Routine TEST, as controlled by TTAB, performs the comparisons between the basic syntactic types associated with the STAB pointers and an encodement of the syntax which is stored in the table TTAB. When a successful sequence of tests are performed (when a designated syntactic pattern is found) control is given to routine ACTION.

Routine ACTION, as controlled by ATAB, produces the desired set of macro instructions for the portion of the input string matched by routine TEST. By manipulating the pointers tested by routine TEST, ACTION also alters the STAB table and performs bookkeeping operations upon the fields of the pointers. For example, when an identifier is used, ACTION calls the table processor to check its tables of used identifiers for consistency with current usage (e.g., to prevent the usage of a label as an indexed array name).

The fields for the control table entries are given in Appendix A and will be discussed in the following sections.

*In general, the pointer returned by the Table Processor does not point directly to the entry created for the new item. Instead, the pointer points to an entry in the Main Pointer Table which, in turn, contains the direct pointer to the item. The additional level of indirectness allows the information table entries to be reordered without requiring that all references to the item be updated; only the pointer in the Main Pointer Table need be updated. For ease of reading, when an Information Table Pointer is mentioned, we will not explicitly state this additional level of indirectness.

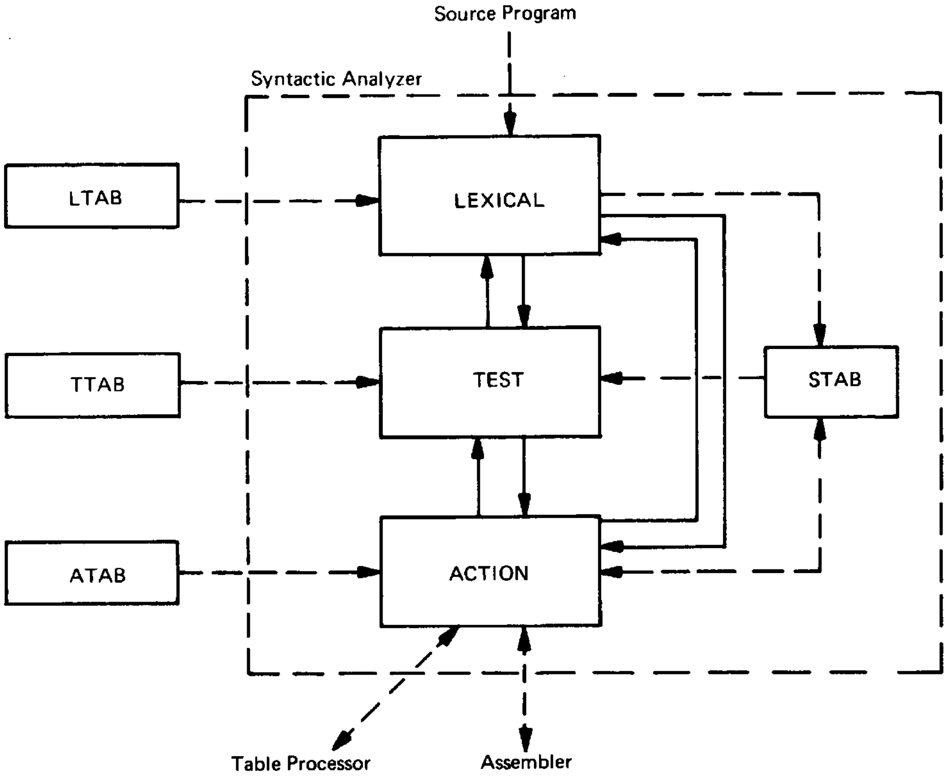


Figure 3-1. The Syntactic Analyzer

3.2 ORGANIZATION OF DATA TABLES

Before discussing the control tables LTAB, TTAB, and ATAB for the Syntactic Analyzer, we will discuss the two major tables affected by the Syntactic Analyzer: the information tables within the Table Processor, and the internal table STAB containing numeric values and pointers to the information tables.

3.2.1 Referencing the Information Tables

The information tables of the Table Processor are used for storing quantities such as variable names and terminal symbols. The Table Processor and its information tables are external to the Syntactic Analyzer. Within the analyzer an entry in an information table is referenced by the entry points issued by the Table Processor. Within the Table Processor there are two values associated with the entry pointer: a table number and an entry number. The table number identifies the information table that the entry is in; the entry number identifies the location of the entry within the table. The table number also gives the location within the Table Processor of the packing information describing the location of each field within an entry. A *field* is the smallest quantity of information considered as an entity. The size of a field may range from one bit to several computer words. When referencing a field, both an entry pointer and a field number must be given. The *field number* identifies which field in the entry is referenced. Table 3-1 lists the Table Processor interface routines called by the analyzer to store and retrieve information to and from the information tables.

Table 3-1. Interface Routines Between Syntactic Analyzer and Table Processor

Routine	Function and Calling Parameters
INVAL:	puts data in tables of the Table Processor. INVAL (value, entry pointer, field number)
OUTVAL:	fetches data out of tables of the Table Processor. OUTVAL (value, entry pointer, field number)
TABNO:	gets table number corresponding to a given entry pointer. TABNO (entry pointer, table number)
INCRM:	gets entry pointer for a new zero entry within given table. INCRM (entry pointer, table number)
SENTER:	(search and enter) searches a given field within all entries of a given table for a given value. If the value is found, SENTER returns the negative of the entry reference number. If the value is not found, SENTER forms a new entry which has the given value in the given field and returns the pointer to the new entry. SENTER (value, table number, field number, entry pointer)

3.2.2 STAB – The Analyzer's Data Table

The STAB table is constructed by the analyzer to store the results of partially analyzed strings. Entries within the STAB may contain two types of fields, numeric values and pointers. The pointers point to entries in the information tables or to other STAB entries. The pointers serve as a common representation for the diverse elements to which they point. An entry in STAB consists of six fields:

NAME	Likely bits	Use
PINTP	2 bits	0 - PPTR is the entry number of another STAB entry 1 - PPTR is a numeric value - PPTRS is its sign 2 - PPTR is a pointer to an entry in an information table
PPTR	15 bits	——
PPTRS	1 bit	——
PADD	15 bits	arbitrary additional information
PFLGS	1 bit	flag for arbitrary use
PFLGF	1 bit	flag for arbitrary use

The last three fields may be used by the compiler designer as will be shown in the example of Section IV. The number of binary bits associated with each field is given only to aid the reader is visualizing the size of a field and is not fixed.

The entries in STAB may be organized by the designer into blocks of pushdowns and arrays. Whether an entry in STAB is referenced as a pushdown entry or an indexed member of an array is determined by the way in which the ATAB and TTAB tables control the accessing of entries. Pushdowns and arrays have similar implementations. For example, consider the lists of entries A and B whose base addresses relative to the origin of STAB are X and Y, i.e., we define loc A(0) = X and loc B(0) = Y. (See Figure 3-2.)

A(5)	PAL	X + 5
A(4)	L	X + 4
A(3)	P	X + 3
A(2)	CD	X + 2
A(1)	AX	X + 1
A(0)	——	X
A(-1)	5	X - 1

(a) An array A of length 5

B(5)	——	Y + 5
B(4)	——	Y + 4
B(3)	Q	Y + 3
B(2)	FOX	Y + 2
B(1)	G	Y + 1
B(0)	3	Y
B(-1)	5	Y - 1

(b) A pushdown stack B of maximum length 5 and of present length 3.

Figure 3-2. Array vs. Pushdown Stack Implementation

If the list A is declared as an array then A(-1), i.e., STAB (X-1), contains the maximum size of the array. If the list B is declared as a pushdown, then B(-1) contains the maximum size of the stack and B(0) contains the current number of entries. When an entry is inserted into the pushdown, B(-1) is incremented and the new entry is placed in Y + B(0). When an entry is removed from the pushdown, B(0) is decremented.

A pushdown stack may be referenced like an array, i.e., without invoking the pushdown mechanism to "manipulate" the most recently inserted entry. For example, to fetch the most recently inserted entry in the list B of Figure 3-2(b) without decrementing the pushdown counter, the reference B(B(0)), rather than the reference STACK (B) *, can be used.

* Stack is a pseudo-function that "pops up" the most recently inserted entry.

3.3 ROUTINE LEXICAL

Routine LEXICAL recognizes *basic syntactic types*. These include key words (GOTO, IF, THEN), terminal symbols (+,-,*), identifiers (ALPHA, B, C), and literals (2,3.14). Each block of LTAB entries governs the recognition for one basic syntactic type. LEXICAL is called with a pointer to a sequence of ATAB entries; each ATAB entry contains a pointer to a block of LTAB entries and a pointer to an information table and field number to be used by the table processor if an acceptable syntactic type is found.

LEXICAL performs the analysis designated by each LTAB entry within the block. LEXICAL uses the information table and field number to place an entry in the information table. If the analysis designated by the block succeeds, the LEXICAL truth value is set to TRUE: the pointer to the information table entry is placed on top of a system stack called PSTK; and control is returned to ACTION. If the analysis designated by the block fails, LEXICAL automatically performs the lexical analysis designated by the block of LTAB entries pointed to by the next ATAB entry. If the block of analysis pointed to by the last ATAB entry fails, the LEXICAL truth value is set to FALSE and control is returned to ACTION.

The LTAB entries specify one of three mechanisms for handling a string that is recognized as a basic syntactic type: a) the string is to be inserted into an information table (a literal or identifier), b) a search is to be made to match the string with an existing entry in the information tables (key words), or c) no action is to be taken, the information table entry number has already been coded into the LTAB entry (a terminal symbol). In any case, LEXICAL returns a truth value indicating whether an acceptable information table entry exists and a pointer is given to the entry.

Besides LTAB there are two important tables used by LEXICAL: CLIST and CPLIST. Table CLIST, which is part of STAB, is used to store the BCD characters of the input string - one BCD character per CLIST entry. The CLIST entries are periodically shifted or deleted to accommodate new characters. Table CPLIST contains the "property" bits associated with each of the sixty-four possible BCD characters. The interpretation of the bits is defined by the control tables. For example, the characters 0 to 9 are likely to have a property bit for "number" set, while the characters 0 to 7 are also likely to have a property bit for "octal number" set, and the character 0 of the special property bit for "zero value" set (the latter for use in eliminating leading zeros in literals). A maximum of 15 property bits can be defined. The testing for the occurrence of a certain class of character, i.e., a character with a certain "property", is basic to lexical analysis.

The system variable BLPROP is used to store the property bits for ignorable characters (blanks). LEXICAL uses BLPROP to scan the characters in the input string until a character that does not have the properties of BLPROP is found, and only the characters that do not have the properties of BLPROP are added to CLIST. This technique provides a quick, non-interpretive scan of ignorable characters.

LEXICAL forms temporary BCD strings which are used by the search routine in calls to the Table Processor. As each character is analyzed, LTAB indicates whether this character should be added to the BCD strings. When the lexical analysis designated by a block of entries is finished, the newly formed BCD string will contain a copy of the accepted string or its compressed equivalent (the string "3.000" is usually compressed to "3").

The basic testing sequence for lexical analysis is outlined in Figure 3-3. The fields within an LTAB entry and their interpretation in controlling lexical analysis are given in Table 3-2.

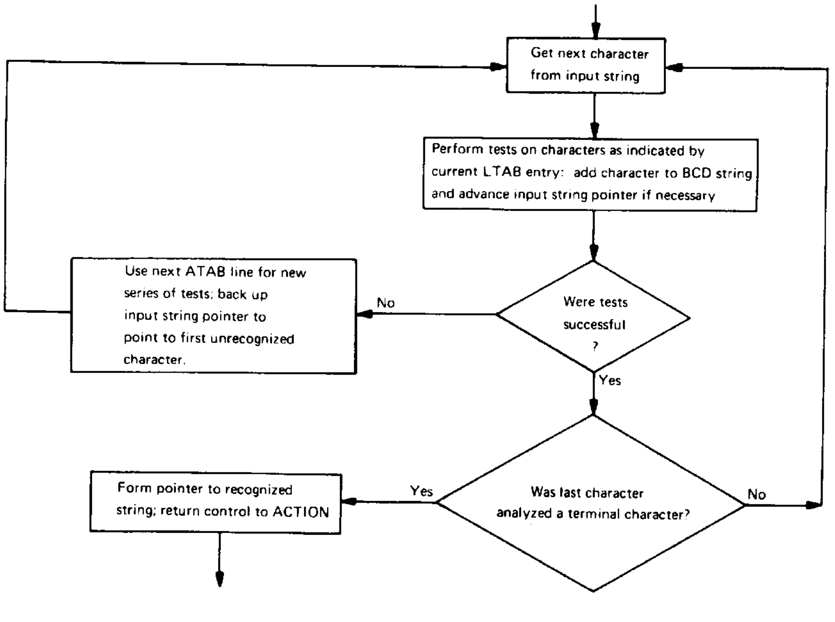

Get next character from input string
Perform tests on characters as indicated by current LTAB entry: add character to BCD string and advance input string pointer if necessary
Use next ATAB line for new series of tests; back up input string pointer to point to first unrecognized character.
No
Were tests successful ?
Yes
Form pointer to recognized string; return control to ACTION
Yes
Was last character analyzed a terminal character?
No

Figure 3-3. Character Testing Sequence for Routine LEXICAL

Example:

To illustrate the LTAB-LEXICAL operation, consider the following Backus-Naur Form specification* for the syntactic type "literal":

digit : : = 0| 1 |2 |3 | 4 | 5 | 6 | 7 | 8 | 9
integer : : = [⟨digit⟩]$_0^\infty$
literal : : = ⟨integer⟩⟨blank⟩ | ⟨integer⟩.⟨blank⟩
.⟨integer⟩⟨blank⟩ | ⟨integer⟩. ⟨integer⟩⟨blank⟩

The table CPLIST might be initialized as:

CPLIST (0) = 1	CPLIST (33)** = 2
CPLIST (1) = 1	CPLIST (60)*** = 4

The LTAB entries for this syntactic type might be as follows (continued top of p.16):

*In addition to the notation used in pure Backus-Naur form, we use the brackets [ ]$_{k_1}^{k_2}$ to designate any number from k_1 through k_2 occurrences of the enclosed expression.

**33 is the octal equivalent for the BCD character "."

***60 is the octal equivalent for the BCD character blank

Table 3-2. Fields for Entries in LTAB Control Table

Field	No. of Bits	Interpretation
(1) Get character		
LCHGXB	1	0 - use input string character previously tested 1 - examine LPOS and LPLMN to get new character from input string
LPOS	5	relative location in CLIST of characters to be tested
LPLMN	1	0 - set X = XC + LPOS (see legend at bottom of table for definition of XC) 1 - set X = XC-LPOS
(2) Perform Test		
LTEST	15	character (LWHAT = 1) or property bits (LWHAT = 2) to be tested for
LWHAT	2	0 - no test, assume test is TRUE 1 - test if "CLISTL(X) = LTEST" 2 - test if "CPLIST (CLIST(X)) .A. LTEST ≠ 0".
(3) If test is TRUE		
LTBCD	2	0 - take no action 1 - add CLIST (X) to BCD string 2 - excise BCD string
LADVN	6	relative location in CLIST of latest characters recognized (LTADVN = 2)
LTADVN	2	0 - take no action 1 - set XC = X 2 - set XC = XC + LADVN 3 - set XC = SXC
LTDONE	15	0 - perform new test from entry LTDONE of LTAB 1 - ATAB test failed; reset XC to SXC, excise BCD string, and start next ATAB test 2 - terminal character found; LTAB (AFALSE) is the table name; LARG (AFALSE) is the field number for the entry 3 - terminal character found; set pointer to LTDONE
(4) If test is FALSE LFBCD, LFADVN, LFDONE, LFALSE	—	similar set of fields for FALSE test result. (The field LADVN is not duplicated.)

System variables used by LEXICAL:

XC - Location in CLIST of last characters analyzed

LXC - Location in CLIST of last character input to CLIST

SXC - Location in CLIST of first character that has not been identified

Entry	Interpretation
LTAB (20)	

LGHGXC = 1, LPOS = 0, LPLMN = 0,
LTEST = 1, LWHAT = 2,
LTBCO = 1, LTDONE = 22, LTRUE = 0,
LFDONE = 24, LFALSE = 0.
	Use current character;
	Test if character has property bit for "digit set", i.e., test if character is a digit
	if TRUE, add character to BCD string and test LTAB (22)
	if FALSE, test LTAB (24).

LTAB (22)	

LCHGXC = 1, LPOS = 1, LPLMN = 0,
LTEST = 1, LWHAT = 2,
LTBCO = 1, LTADVN = 1, LTDONE = 22, LTRUE = 0,
LFDONE = 24, LFALSE = 0.
	Use next character;
	test if character is a digit;
	if TRUE, add character to BCD string and repeat test for next character;
	if FALSE, test LTAB (24).

LTAB (24)	

LCHGXC = 0,
LTEST = 33, LWHAT = 1,
LTBCD = 1, LTADVN = 1, LTDONE = 28, LTRUE = 0
LFDONE = 26, LFALSE = 0.
	Use previous characters;
	test if character is a ".";
	if TRUE, add character to BCD string and test LTAB (28);
	if FALSE, test LTAB (26).

LTAB (26)	

LCHGXC = 0,
LTEST = 2, LWHAT = 2,
LTBCD = 0, LTADVN = 1, LTRUE = 2,
LFALSE = 1.
	Use previous character;
	test if character is a blank;
	if TRUE, search for literal in table processor;
	if FALSE, reset BCD string and try next ATAB test.

LTAB (28)	

LCHGXC = 0, LPOS = 1, LPLMN = 0,
LTEST = 1, LWHAT = 2,
LTBCD = 1, LTADVN = 1, LTDONE = 28, LTRUE = 0
LFDONE = 26, LFALSE = 0.
	Use next character;
	test if character is a digit;
	if TRUE, add character to BCD string and repeat test for next character;
	if FALSE, test LTAB (26).

LTAB (30)	

LCHGXC = 0,
LTEST = 33, LWHAT = 1,
LTBCD = 1, LTDONE = 28,
LFALSE = 1.
	Use previous character;
	test if character is a ".";
	if TRUE, test LTAB (28);
	if FALSE, report ATAB test failed.

3.4 ROUTINE TEST

Routine TEST is called by routine ACTION to make a series of tests on the entries in STAB in order to identify *patterns* of *basic syntactic types*. If a series of tests is successful, control is returned to routine ACTION.

TEST performs the tests designated by a sequence of TTAB entries. The fields for a TTAB entry are given in Table 3-3. The tests allowed in TEST are rather simple; if a more complex test is needed, a special call can be made to ACTION. The special call initiates a routine that operates more slowly than TEST but has a general arithmetic testing facility. The pointer that is accessed during the scan of the first part of the TTAB entry is given as an argument to ACTION; the result of the call is a truth value, which, upon return from ACTION, is used in the same manner as a truth value computed internally within TEST. Ultimately TEST must return control to ACTION.

Table 3-3. Fields for Entries in TTAB Control Table

Field	No. of Bits	Interpretation
(1) Get entry number of an STAB entry (all entry numbers refer to STAB)		
TLOC	15	entry number
TSTPT	2	1 - TLOC is entry number of a stack base (as noted previously, STAB may be organized in blocks of stacks or arrays) 2 - TLOC is entry number of a value 3 - TLOC is entry number of a pointer
TPOS	15	

TPLMN	1	if TSTPT = 1:
		0 - TPOS is added to stack base
		1 - TPOS is subtracted from current stack limit
		if RSTPT = 2:
		0 - TPOS is added to value pointed to
		1 - TPOS is subtracted from value pointed to

(2) Check for indirect ref. (if TSTP = 3)		
TINDR	1	1 - if the pointer in STAB points to another pointer (PINTP = 0), use that pointer

Table 3-3. Fields for Entries in TTAB Control Table (Cont.)

Field	No. of Bits	Interpretation
(3) Extract proper field from entry		
TTABLE	15	entry number of last entry to be checked
TTBTST	1	0 - ignore TTABLE, check only one entry
		1 - test all entries up to TTABLE
TWHAT	3	0 - no test; assume test is TRUE
		1 - a complex test is indicated; pass control to ACTION for test and return to (5) below. (ACTION returns a truth value)
		2 - get PPTR field of STAB entry
		3 - get PADD field of STAB entry
		4 - get PFLGS field of STAB entry
		5 - get PFLGP field of STAB entry
TSFLD	15	field number
TFLDTS	1	0 - ignore TSFLD
		1 - get field TSFLD from table processor using value gotten above as entry number
(4) Perform the following test on the field		
TTEST	15	value to be matched
TMNPRP	1	0 - test if "field = TTEST"
		1 - test if "field .A. TTEST = 0"
(5) If test is TRUE		
TTDONE	15	entry number
TTRUE	1	0 - perform another test from TTAB(TTDONE)
		1 - go to ACTION and perform operations specified by ATAB (TTDONE) and the succeeding entries.
(6) If test is FALSE		
TFDONE	15	entry number
TFALSE	1	0 - perform another test from TTAB (TFDONE)
		1 - go to ACTION and perform operations specified by ATAB (TFDONE) and the succeeding entries

3.5 ROUTINE ACTION

After TEST finds a particular syntactic pattern in the input string, routine ACTION generates the equivalent set of macro instructions and performs bookkeeping operations. When ACTION completes its processing, control is returned to TEST for more pattern testing.

There are two modes in which the controlling ATAB entries may be interpreted. In the normal mode the entries call for access to the information table fields. Operations such as printing or transfers of control, however, use literal arguments so frequently that it would be unduly time-consuming to use the field accessing routines of the Table Processor. For these operations a special mode of interpretation exists in which the required fields are taken directly from ATRUE and AFALSE fields of the ATAB entry. These latter two fields overlay the fields interpreted in the normal mode.

There are two system pushdown stacks used by routine ACTION: VSTK, a pushdown for values, and PSTK, a pushdown for pointers. They are used for the storage of temporary results. The fields within an ATAB entry and their interpretation are given in Table 3-4.

Table 3-4. Fields for Entries in ATAB Control Table

Field	No. of Bits	Interpretation
(1) Initialize value of dummy variable NUMBER		
APTR	15	
ANUM	2	0 - Let NUMBER = value on top of VSTR
		1 - Let NUMBER = APTR
		2 - Let NUMBER = STAB(APTR)
ASTK	3	0 - Let NUMBER = STAB (NUMBER (AVLPTR below must = 1.)
		1 - NUMBER is base of pushdown
		2 - NUMBER is base of pushdown; process next ATAB entry (ASTK of next entry must = 3 or 4.)
		3 - NUMBER is location relative to base of stack previously used
		4 - NUMBER is location relative to current limit of stack previously used
		5 - let PPTR = NUMBER, PINTP = 1. (for fetching only)

Table 3-4. Fields for Entries in ATAB Control Table (Cont.)

Field	No. of Bits	Interpretation
(2) If NUMBER is a value, exit		
AVLPTR	1	0 - NUMBER is a value exit 1 - NUMBER is a pointer
(3) If NUMBER is a pointer, use the following fields		
AUSPTR	2	0 - exit 1 - use only the fields PINTP, PPTRS, PPTR of the entry pointed to 2 - use only the field AFLD of the entry pointed to 3 - use field number AARG from Table Processor. Error if PINTP ≠ 2
AFLD	3	0 - error (use only when AUSPTR = 2) 1 - use PPTR, PPTRS 2 - use PINTP 3 - use PADD 4 - use PFLGS 5 - use PFLGF 6 - use table reference number of entry pointed to. If PINTP ≠ 2, use PINTP
AARG	15	field reference number (use only if AUSPTR = 3)
(4) Perform the ACTION operation indicated by AOPN		
AEOPN	6	Number of ACTION operation to be performed

The use of these fields is illustrated by the following examples:

(1) Initializing Number

VSTKO* = 425	ANUM = 0; NUMBER = 425
APTR = 233	= 1; NUMBER = 233
STAB(233) = 607	= 2; NUMBER = 607

*VSTKO designates the top element of VSTK, VSTK1 the next to top element, etc.

(2) Stack accessing (assume NUMBER is 233).

ASTK = 0: AVLPTR = 0; VSTK0 233
AVLPTR = 1; PSTK0 STAB(233)

= 1: If STAB(233) $\leq$ 0, error;
else fetch STAB (233+STAB(233)).

= 2: STKNUM 233. If ASTK (next line) $\neq$ 3 or 4, error.

= 3: If STAB(STKNUM-1) < 233, error;
else fetch STAB(STKNUM+233).

= 4: If STAB(STKNUM) $\leq$ 233, error;
else fetch STAB(STKNUM+STAB(STKNUM)0 233).

= 5: Fetch pointer with
PINTP = 1 and
PPTR = 233.

A list of the ACTION operations designated by AOPN is given in Appendix B. A maximum of sixty-four operations is allowed, although presently only forty-three have been implemented. The compiler designer may add additional operations to this list.

3.6 RELATED INFORMATION

3.6.1 The Analyzer as a Subroutine

The entire analyzer may be treated as a subroutine. The calling sequence is the equivalent of the MAD statement EXECUTE SYNTAX. (A, B). SYNTAX is the symbolic name of the main entry point for the analyzer. The argument A is used to return the number ERRFLG of an error (or zero if no error). Appendix C contains a table of error numbers, the error comments that are printed if an error occurs, and the probable cause of errors. The error exit may be used to detect source program errors or errors in the design of the control tables. The argument B is used to indicate which of the control tables (LTAB, TTAB, or ATAB) is to be overlaid with sections of other tables if the control tables overflow core space. This feature has not yet been implemented. The analyzer requires 6473_8 or 3387_{10} locations and uses 30010_8 or 12296_{10} locations of common storage.

3.6.2 Recursive Calls

When a predicate call (i.e., a call which returns a truth value) from either TEST, LEXICAL or ACTION is made to ACTION, the entry number (in TTAB, or ATAB) which initiated the call is saved on top of the system stack DOSTK. DOSTK is used to keep track of the level of recursive calls. Two flags (DOPRED and DOLEX) are set in the DOSTK entry to designate the calling routine. When the RETURN operation is invoked, the flags in the entry on top of DOSTK are examined to determine to which routine control should be returned.

All communication between routines ACTION, TEST, and LEXICAL is done through routine CONTRL. CONTRL keeps track of calling sequences and the save stack DOSTK. Because of the complex recursion that may occur between calls, a return from a predicate cannot be a simple function return. The common system variable CONFLG is used to indicate the return sequence for calls to LEXICAL, TEST, and ACTION. The interpretation of CONFLG is as follows:

ROUTINE	Value of CONFLG	MEANING
Exit from ACTION	0	exit from analyzer
	1	call LEXICAL
	2	call TEST
	3	predicate return to TEST or LEXCAL
Entry into TEST or LEXCAL	0	normal entry
	1	predicate return from ACTION
Exit from TEST or LEXCAL	0	done - return to ACTION
	1	(special) predicate call to ACTION

3.6.3 Control Table References

An entry in the control tables may require more than one computer word. To allow all fields of a given entry could be referenced with the same entry index, the macro names LTAB1, TTAB1, TTAB2, TTAB3, and ATAB1 are defined. For example, TTAB1(x) = TTAB(x+1); the fields referenced by the name TTAB1 are equivalent to those referenced by the name TTAB displaced by one computer word. (This technique has the disadvantage that when indexing through the control tables, one must increment by something other than unity.)

3.6.4 BCD Data

BCD strings (routine names and error comments) are handled in two ways. For BCD strings stored in the Table Processor, the "value" of a BCD string is the address of the first computer word in the string. The first six bits of the string are interpreted as an octal number designating the total number of BCD characters in the string. For BCD strings stored in the STAB of the Syntactic Analyzer, the "value" of a BCD string entry is the entry number of that string in a block called BCDTAB. For a string of five or less characters, the entry number is prefixed by plus indicator and the first six bits of the string is an octal number (1 to 5) giving the number of characters in the string. For strings of greater length, the entry number is prefixed by a minus indicator; the decrement field of the entry contains the number of BCD characters in the string and the address field points to the words containing the BCD string. This method, rather than the one used in the Table Processor, is used to allow for very long strings like those used for error comments.

Before the bootstrap operation, BCDTAB contained the names for the system routines and system error comments. After the bootstrap operation, it is expected that the BCD strings for the designer's error comments will be stored in BCDTAB.

3.6.5 ACTION Operations

a.) The PRINT Operation (AOPN = 28):

All output from the analyzer is handled by a single ACTION operation, PRINT. The PRINT Operation is controlled by four system variables - PRNTVL, OUTLNT, PRNTMD and PRNTSP. PRNTVL is the value to be added to the BCD output string. OUTLNT is the maximum number of characters allowed per output line. If the BCD output string being formed becomes greater in length than OUTLNT, the forming process is temporarily halted, the current output string is pointed and expunged, and the forming process is resumed. PRNTMD is an integer indicating the interpretation of PRNTVL:

PRNTMD	INTERPRETATION
0	ignore PRNTVL; load blanks into output line
1	PRNTVL is a signed decimal number
2	PRNTVL is an unsigned octal number
3	PRNTVL is a binary number
4	PRNTVL is a Table Processor BCD string
5	PRNTVL is an internal (BCDTAB) string
6	ignore PRNTVL; print current output string
7	skip line

PRNTSP is the number of characters to use when printing PRNTVL. If the number of characters required to print PRNTVL is less than PRNTSP, the value is printed left adjusted with trailing blanks. If PRNTVL requires more characters than PRNTSP, only the leftmost characters are printed. If PRNTSP is 0, the given string will be printed with no blanks.

b.) The Operation NEWCHR (AOPN = 42):

The ACTION operation NEWCHR allows characters to be read from the input medium. NEWCHR inserts the new characters into CLIST, eliminates fully analyzed characters, rearranges CLIST, and changes the values of SXC and LXC if necessary.

If the compiler is used in time-shared operation from a console, and a line consisting of a single break character is read, NEWCHR will cause the word "INPUT" to be printed. Thus if the user wants to know when the system requires input, he simply hits the break character (the carriage return) and "INPUT" will be typed when input is needed. Two successive break characters will cause an error exit from the analyzer. NEWCHR will not put the break character in CLIST unless the system variable BRKCHR is non-zero.

c.) MOVE Operation (AOPN = 29):

The formation of a macro is usually handled in one stack and later transferred to another stack when completed. The MOVE operation is used to empty the elements for a completed macro from one stack to a second stack. The operation begins by transferring the *first* element entered into one stack onto the second stack and ends by transferring the *last* element of the first stack onto the top of the second stack. To distinguish the name of a macro from its arguments, the leftmost bit of the first element transferred (the name of the macro) is set to 1. The leftmost bit of the remaining elements transferred (the arguments of the macro) is set to 0. Two system pointers, FMOVE and LMOVE, are set by the MOVE operation. FMOVE points to the first element moved into the second stack; LMOVE points to the last element moved onto the second stack. For example, suppose that i) STACKQ is a stack on top of which the code number for the macro PLUS and pointers to the elements "A" and "B" of the input string have been formed, and that ii) the macros resulting from syntactic analysis are put in a stack STACKM. The call MOVE (STACKM, STACKQ, STACKQ0, STACKQ2) would load entries STACKQ0 through STACKQ2 onto STACKM.

d.) The ROUTINE Operation (AOPN = 35):

There may be many complex operations which a user would like to perform but cannot do efficiently in the present system. The ROUTINE operation provides the facility for user supplied external subroutines. These might include routines to evaluate mathematical functions (log x) or routines to convert BCD strings into their intended values. (The BCD string "147" may have to be converted into the integer representation of "147".)

It is assumed that when the control tables are formed, the auxiliary routines may not be available and hence their starting location in core not known. When the control tables are formed, the table RTNTAB must be loaded with the BCDTAB entry numbers for the entries containing the BCD names of the auxiliary routines. When starting execution, the entry routine SYNTAX searches the MOVIE TABLE for these BCD names and replaces them with their starting locations. If a subroutine given in RTNTAB is not found in the MOVIE TABLE, a comment is printed and analysis proceeds. However, if a call is made to an undefined subroutine (one not found in the MOVIE TABLE) a system error results.

SECTION IV

THE TABLE PROCESSOR

4.1 INTRODUCTION

The Syntactic Analyzer encounters information that should be processed and made available for later use by the Assembler. The Table Processor is designed for the collection of this storage information (variable names, label names, array dimensions, and data type information) and the allocation of storage space. It works with the Syntactic Analyzer to enter the information into the information tables, and later processes these tables upon completion of the syntactic analysis. The functions of the Table Processor include a) assigning core locations to symbols, arrays, and literals and b) merging and sorting the information tables so that the Assembler can quickly access the information in them. Unlike the generation of macros, which can be carried out when certain complete syntactic units have been recognized, the allocation of storage can be carried out only at the end of the syntactic analysis. (For example, all source program variable names must be collected and they can be allocated storage words and the referencing machine instructions assembled.)

4.2 INFORMATION TABLES AND MAIN DIRECTORY

The information tables are used to store variable names, label names, literals, integers, character patterns, dimensional information, and other information that the Syntactic Analyzer encounters in the source program. The names and formats for each information table are declared using the Table Declaration and Manipulation Language. The format information for each table is coded into a table called the Main Directory.

An information table has a simple structure. At the base of the table, there is a bookkeeping word containing two pieces of information. The address part of the bookkeeping word contains a pointer to the current top entry in the table; this pointer is used when adding an entry to the table or sorting the table. The decrement part of the bookkeeping word contains a pointer to the entry last processed. Figure 4-1 gives an example of an information table called LITTAB which might be used to store literals. The most recent entry to the table occupies registers 34400-02. The pointer 34427 in the decrement of LITTAB(0) is considered to have been set by the utility routine SEARCH when a match was found in the entry in location 34427. This pointer may be so used to access other fields of the entry without calling the routine SEARCH again.

The format description of each information table is kept in a table called the Main Directory. The system uses the Main Directory to meaningfully access an information table entry. Although an information table is referenced with its symbolic name by the designer, it is referenced internally by the address of the first word of the block of words in the *Main Directory* that contains its format information. The format of each information table

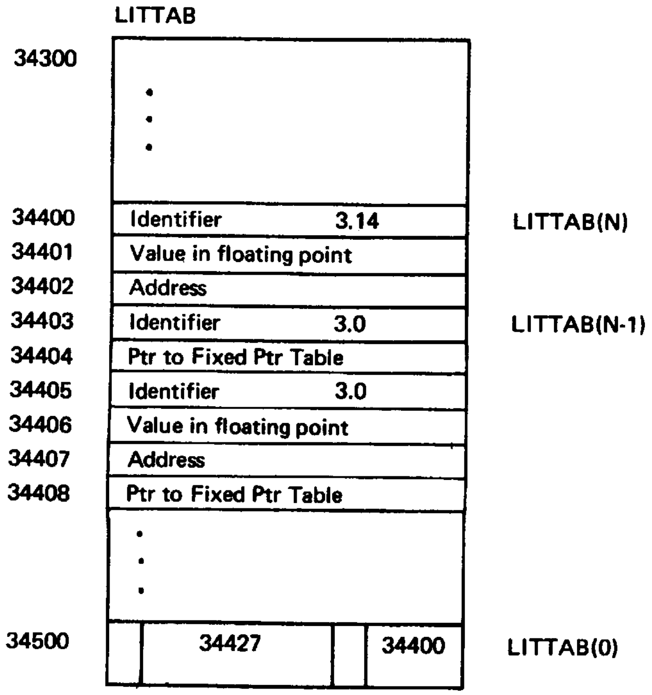


Figure 4-1. Example of an Information Table LITTAB used to Store Literals

is declared using the Table Declaration and Manipulation Language. For each information table the user must declare:

(1) the symbolic name

(2) the maximum number of entries

(3) the sorting option (whether the table should be kept sorted), the field on which the sort is to be based, and the sorting scheme (the order of precedence)

(4) the number of fields in an entry and the packing information for each field (If the designer does not specify how the fields should be packed, the bootstrap translator will specify it for the user.)

The bootstrap translator interprets the declarations, assigns storage space for the declared tables, and inserts the format information into a block of words in the Main Directory.

The format of the blocks in the Main Directory is shown in Figure 4-2. The first two words in each block contain miscellaneous information about the information table. The decrement of the first (top) word gives the address of the last (bottom) word of the information table. The address of the first word gives the size of the information table. The

14000		34500		00200	
14001		00003	0	——	
14002	0	00000	0	00000	format of field 1 (identifier)
14003		00001		00001	
14004	0	00000	0	00000	format of field 2 (value)
14005		00001		00002	
14006	0	00000	0	77777	format of field 3 (address of literal in object program)
14007		00000		00003	
14010	0	00000	0	77777	format of field 4 (ptr to Fixed Ptr Table)
14011		00000		00004	

Figure 4-2. Example Block in the Main Directory for Figure 4-1 Literal Table. All numbers given are in octal.

decrement of the second word gives the number of words occupied by one entry in the information table. The tag of the second register specifies the sorting option for the table:

0 - the table is not to be sorted.

1 - the table is to be sorted according to the standard BCD scheme.

m - the table is to be sorted according to the m-th sorting scheme (m = 2,3,4,5,6).

The address of the second word is the field based on which table is to be sorted.

Each following pair of words in the block gives the format of one of the fields in the information table. The first word contains the mask for the field, i.e., a word containing 1's in the bits occupied by the field and 0's elsewhere. A mask of all 0's designates a field of one or more whole registers. If the mask is non-zero, the decrement of the second word will contain the number of bit positions to be right-shifted when the field is to be right-justified, and the address of the second word will give the location in the entry of the word in which the field is stored. If the mask is zero, the decrement of the second register will contain the number of registers occupied by the field, and the address of the second register will contain the location in the entry of the first of the words occupied by this field. The last pair of words always contains the information on the field for the pointer to the Fixed Pointer Table. (See Section 4.3 for description of Fixed Pointer Table.)

As an example, a possible block in the Main Directory for the literal table LITTAB (see Figure 4-1) is indicated in Figure 4-2. As mentioned earlier, tables are referenced internally by the address of the first word assigned to its format information in the Main Directory; thus LITTAB is referenced by the address *14000.* The Main Directory and storage for the information tables are set up by the bootstrap translator from the declaration statements in the Table Declaration and Manipulation Language. These statements are explained in Section 6.3.

4.3 THE FIXED POINTER TABLE

Besides the information table fields that are declared using the Table Declaration and Manipulation Language, there is an additional field that is provided by the system. This field, known as the *fixed pointer,* contains a pointer to a corresponding entry in a table called the Fixed Pointer Table. The Fixed Pointer Table entry (see Figure 4-3) occupies one word; the address part contains a pointer back to the pointer in the information table and the decrement part contains the internal name (the Main Directory address) of the information table. The Fixed Pointer Table is used to keep track of the locations of all entries in all the information tables when some of the entries are merged or sorted. Figure 4-3 shows an example of the chaining between an entry in the literal table of Figure 4-1 and the Fixed Pointer Table. Note that the literal table name given in the Fixed Pointer Table entry is 14000, the internal name for LITTAB. (See Figure 4-2.)

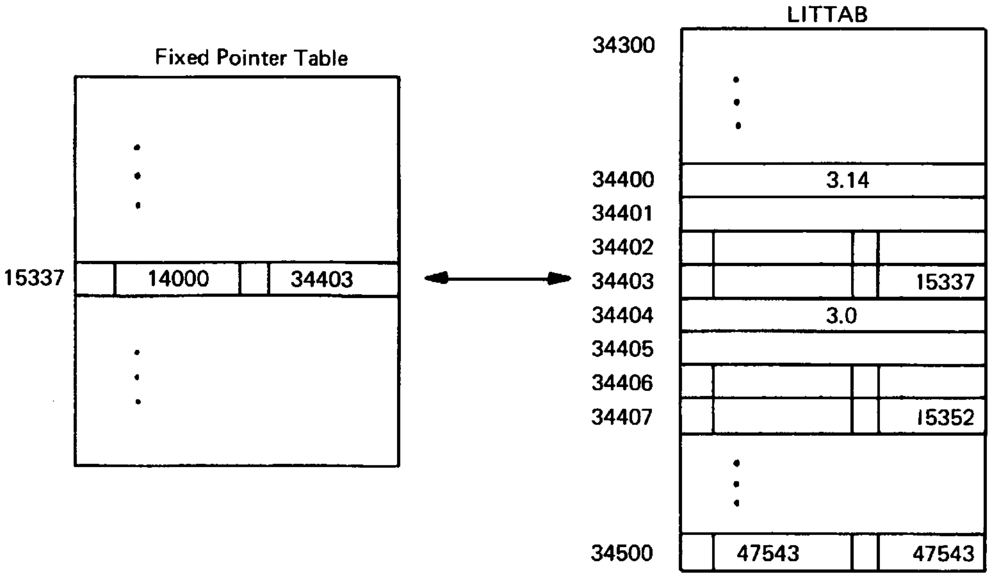


Figure 4-3. Linkage Between an Information Table and the Fixed Pointer Table

When an element is entered into an information table, the control routine will also fill the fixed pointer field and the corresponding entry in the Fixed Pointer Table. The control routine will return the location of the entry in the Fixed Pointer Table.

This location, rather than the location of the information table entry, is returned so that references to the information table entry will not have to be updated if the entry in the information table entry is displaced during a sorting or merging of the tables.

Each time an information table entry is displaced, the fixed pointer field of the entry is traced back to the Fixed Pointer Table entry to update the reverse pointer. When two information tables with matching entries are merged, (See Figure 4-4), both Fixed Pointer Table entries to the matching entries must be updated to point to the single entry in the new merged table. This is effected by setting a flag in one Fixed Pointer Table entry and setting its address field to point to the second Fixed Pointer Table entry. The address field of the second Fixed Pointer Table entry is set to point to the combined information table entry. The fixed pointer field of the combined entry will be set to point to the second Fixed Pointer Table entry.

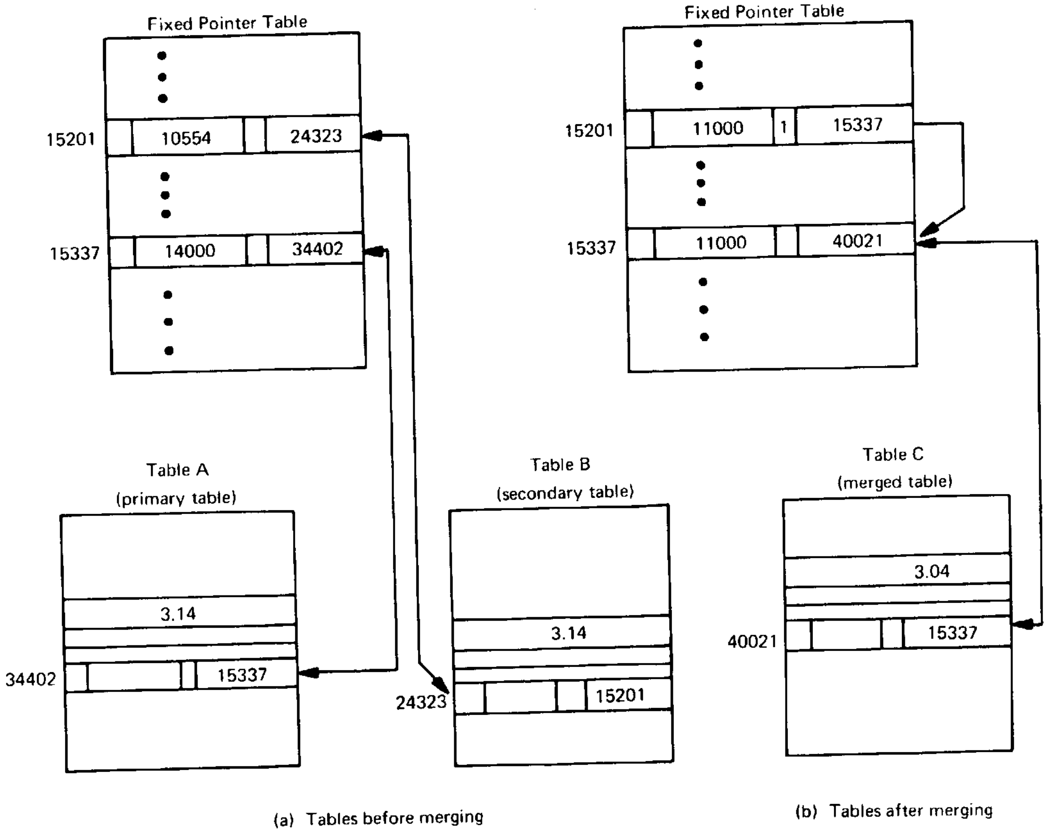


Figure 4-4. Example of the Merge Operation

4.4 TABLE MANIPULATION TABLE

The Table Manipulation Table is used to control the processing of the information tables. The table is set up by the bootstrap translator from statements given by the designer using the Table Declaration and Manipulation Language.

The entries in the Table Manipulation Table are grouped into blocks. Each block specifies a set of operations to be performed on one or more tables. The fields within a Table Manipulation Table entry and their interpretation are given in Table 4-1. Example statements in the Table Declaration and Manipulation Language are given with each example entry; these statements are explained in Section VI.

4.5 OPERATION OF THE TABLE PROCESSOR

When control is passed to the Table Processor upon completion of syntactic analysis, the Table Processor operates in the following manner. The entries in the Table Manipulation Table are scanned and the first block of entries is found. The routine PROCESS is then called to process the block of entries. Routine PROCESS scans each entry in the block, identifies the function that each represents, and calls the associated utility routines (SORT, INSERT). Since the utility routines are written in MAD, the call to a utility routine is processed through a FAP subroutine that converts the control table information into the appropriate parameters and generates the suitable calling sequence. After processing each entry in the block, routine PROCESS returns control to the main portion of the Table Processor for further action until all blocks of entries are processed.

Table 4-1. Fields for Entries in Table Manipulation Table

Operation	Prefix-tag of first word in encoded entry	Entry Format	Example Entry	Interpretation*	(Corresponding Statement in Table Declaration and Manipulation Language)
BEGIN BLOCK	1 · 0	1 loc j 0 table p	55000 1 55033 0 16600	Loc j (55033) is the last word in the table manipulation block: table p (16000 or TABLE-A) is the primary table for the block.	PROCESS TABLE-A, TABLE-B, TABLE-C
SORT	1 · n (n = 1 thru 6)	1 table p n field m	55001 1 16600 1 16602	Table p (16600 or TABLE-A) is the table to be sorted; field m (16600) is the field upon which the sort is to be based. (In the Main Directory, the format specification of the first field of TABLE-A occupies locations 16602 and 16603. n = 1 specifies sorting scheme 1, the standard BCD sorting scheme is to be used.)	SORT TABLE-A'(1,1)
INSERT	6 · 0	6 table p 0 field m table q field n	55002 6 14423 0 14427 55003 16600 16606	Field m of table p (field 2 of TABLE-B) is to be matched against all occurrences of field n of table q (field 3 of TABLE-A). If a match is not found, a new entry corresponding to field n of table p will be inserted into table q.	INSERT (TABLE-B,2) INTO (0,3)
TEST	4 · n (n = 1,2,3, or 4)	4 table p n field m test quantity	55004 4 17700 4 17702 55005 0 000000 212223	Field m of table p (the first field of TABLE-C) is tested if equal to the test quantity (the BCD string ABC). If test is true, the accumulator will be set to 1; otherwise it will be set to 0. n = 1,2,3, or 4 depending on whether the test quantity is another field, a variable, an integer, or a BCD string.	TEST (TABLE-C,1) AGAINST (ABC)
SEARCH	5 · 0	5 table p 0 field m table q field n	55006 5 16600 0 16602 55007 17700 17702	All instances of field m of table p are searched for field n in current entry of table q; if a match is found, the accumulator will be set to 1; otherwise it will be set to zero.	SEARCH (0,1) FOR (2,1)

*All table addresses refer to the corresponding addresses in the Main Directory.

Assume the following Main Directory Table addresses: TABLE-A = 16600 TABLE-B = 14423 TABLE-C = 17700

Table 4-1. Fields for Entries in Table Manipulation Table (Cont.)

Operation	Prefix-tag of first word in encoded entry	Entry Format	Example Entry	Interpretation	(Corresponding Statement in Table Declaration and Manipulation Language)
TRANSFER	2 · n (n = 0 or 1)	2 loc 1 n loc 2	55010 2 55015 1 55004	For an unconditioned transfer (n = 0), a transfer is made to examine the entry in loc 2 (55004) in the Table Manipulation block. For a conditioned transfer (n = 1), if the accumulator contains a one, the entry in loc 1 (55015) of the Table Manipulation Table is examined next. If the accumulator contains a 0, the entry in loc 2 (55004) is examined next.	$ 3, · 2 $
INDEX	2 · n (n = 2 or 3)	2 table p 3 field m integer (or) 2 2 adv. of var integer	55011 2 17700 3 17004 55012 4 2	The field (or variable for n = 2) specified in the first word (field 2 of TABLE-C) is incremented by the integer (·2) in the second word.	INDEX (2,2) ·2
ASSIGNMENT	3 · n (n = 1,2, or 3)	3 table p n field n quantity	55013 3 17000 2 17706 55014 5	Field n of table p (the 3rd field of TABLE-B) is set equal to the quantity specified by the second word. n = 1,2, or 3 specifies that the quantity in the second word is a field, an integer, or a variable.	(2,3) = 5
ARITHMETIC	7 · 7	7 —— 7 n 1st instruction 2nd instruction • • • nth instruction	55015 7 —— 7 00006 55016 TSX 4 50332 55017 17700 17704 55020 MPY 20313 55021 XCA 55022 ADD 26401 55023 STO 26401 20313 5 5 26401 3 A	The Arithmetic entry is used to set the value of a variable. The n words after the first are interpreted as machine instructions. (The TSX instruction is a call to a routine beginning at 50332. This routine gets the field value of the field given below the TSX instruction and inserts the value into the accumulator.)	A = A + (2,2)* 5
PRINT	1 · 7	1 table p 7 n 1st word 2nd word • • • nth word	55024 1 16600 7 7 55025 23 1 55026 30 4 55027 60606060 0000 55030 46 2 55031 30 3 55032 60606060 0000 55033 46 3	The Print entry is used to specify the printing of information table fields. The n words after the first specify the format of the fields to be printed from table p. 23 (octal for C) indicates the field designated by the address is to be printed in BCD format. 30 (octal for I) indicates that the following word is to be printed in decimal integer format; the address gives the number of character in the string. 46 (octal for O) indicates that the field designated by the address is to be printed in octal integer format.	PRINT TABLE-A(C/1, 4H ,I/2,4H ,I/3)

SECTION V

THE ASSEMBLER

5.1 INTRODUCTION

The Assembler accepts the list of macros generated by the Syntactic Analyzer and uses the information tables furnished by the Table Processor to generate binary machine code from the list of macros. The Assembler is controlled by the Macro Interpretation Table. This table contains information regarding the interpretation of the macros. The Assembler also uses another table, the Machine Code Table, which contains the binary representation of the machine instructions.

5.2 MACRO LIST

A macro generated by the Syntactic Analyzer contains the following information:

1. The macro name, which is the address in the Macro Interpretation Table of the block of words specifying the interpretation of the macro;
2. A count, specifying the number of times the result of the macro is referenced by other macros;
3. A list of arguments, each of which is one of the three types:

   type 0 - the fixed pointer to an information table entry,

   type 1 - a pointer to another macro,

   type 2 - a fifteen bit number..

Type 0 arguments are used to reference values obtained during syntactic analysis. Type 1 arguments are used to reference results of other macros. Type 2 arguments designate fixed values.

The format of a macro is shown in Figure 5-1. The prefix 4 in the first word of the figure designates the first of a block of words representing a macro; the decrement and address of the first word contain the macro name and count. The second word is a word reserved to store the value of the macro after is is processed by the assembler. The remaining words in the block contain the arguments of the macro; the prefix of these words contains the type number.

5.3 TEMPORARY STORAGE POOL

When algebraic or boolean expressions are evaluated, it is often necessary to store temporary results. The temporary storage words needed by the object program are drawn from a common pool of words. The compiler keeps track of the number and location of these temporary storage words.

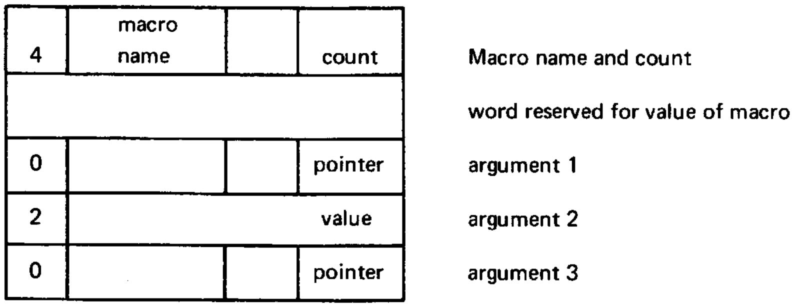
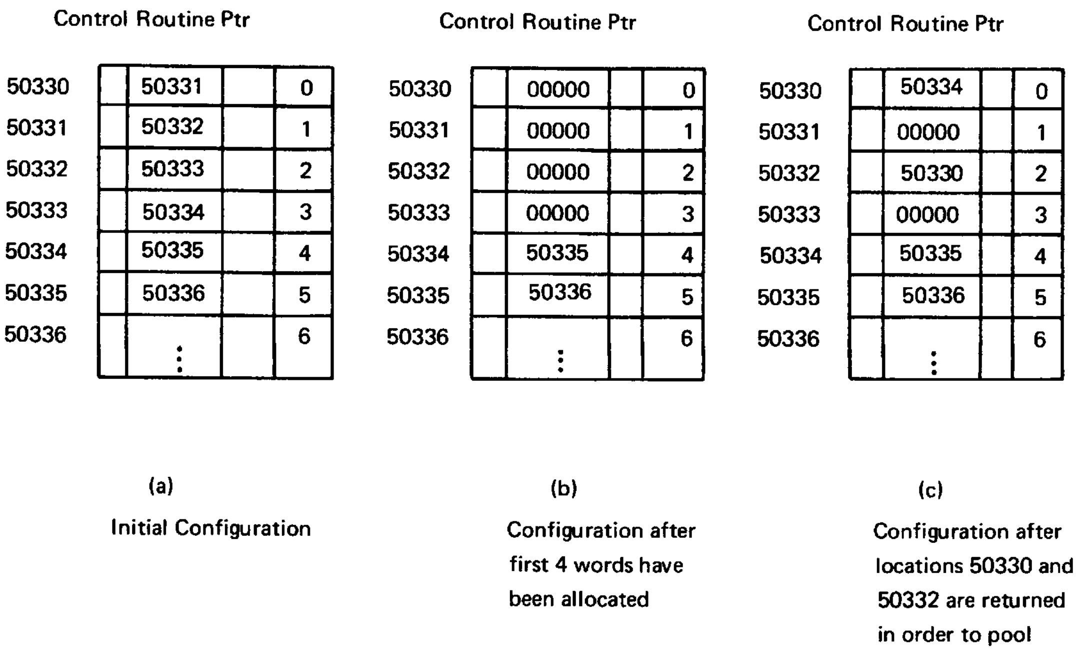


Figure 5-1. Format of a Macro with Three Arguments

We distinguish between two kinds of temporary storage. One kind is used only by the machine instructions corresponding to a single macro. The temporary storage words used by these instructions should be returned to the common temporary storage pool after the instructions are executed. The other kind is used to store the result of the set of machine instructions of a macro so that the result can be referenced by the machine instructions of other macros. These temporary storages should be returned to the pool when there is no further reference to these results.

For each temporary storage word used by a macro, a word is reserved in the Macro Interpretation Table for a pointer to the temporary storage word. Since there is no way of knowing the length of a program or the maximum number of registers needed from the temporary storage pool until the entire object program has been generated, locations of the words to be used as temporary storage cannot be assigned until all machine instructions are generated. During the course of compilation, the instructions which address a word from the temporary storage pool will use the address of the temporary storage word relative to the beginning of the temporary storage pool. The temporary storage words are thus addressed as 00000, 00001, 00002, etc., when they are assigned. In order to identify these temporary addresses later, an "identification bit" is attached to each address. After all macros are processed, the "program break" (the location of the first word in the temporary storage pool) is added to addresses containing an identification bit.

The temporary storage pool is organized as a chained list whose size is initialized by the user. The initial organization of a temporary storage list is shown in Figure 5-2(a). Here the decrement of each word contains the pointer to the next available location and the address of each word gives the location of the word relative to the origin of the temporary storage list. The temporary storage control routine of the compiler maintains a pointer to the first available word in the temporary storage list. When a new temporary storage word is requested, the control routine allocates the word pointed to by the pointer, updates its pointer



Control Routine Ptr

50330		50331		0
50331		50332		1
50332		50333		2
50333		50334		3
50334		50335		4
50335		50336		5
50336		⋮		6

(a)

Initial Configuration

Control Routine Ptr

50330		00000		0
50331		00000		1
50332		00000		2
50333		00000		3
50334		50335		4
50335		50336		5
50336		⋮		6

(b)

Configuration after first 4 words have been allocated

Control Routine Ptr

50330		50334		0
50331		00000		1
50332		50330		2
50333		00000		3
50334		50335		4
50335		50336		5
50336		⋮		6

(c)

Configuration after locations 50330 and 50332 are returned in order to pool

Figure 5-2. Use of a Temporary Storage Pool

to point to the next available word in the list, and sets the decrement of the allocated word to zero. Figure 5-2(b) shows how the example list appears after the first four words have been allocated. When a temporary storage word is released, the control routine sets the decrement of the released word equal to the control routine pointer and updates the control routine pointer to point to the word just released. Figure 5-2(c) shows the organization of the example list after the words in locations 50330 and 50332 have returned to the pool, in that order.

If the count (number of references to the macro) is greater than zero, the result of the macro is stored in a temporary storage word and the address of the temporary storage word is stored in the address portion of the second register of the block of words representing the macro. When this result is accessed by another macro, the count is decremented by one. This count is tested every time it is reduced, and if the count becomes zero, the temporary storage word will be returned to the pool and the address of the second word in the macro will be set to zero.

5.4 USE OF MACHINE REGISTERS

The execution of a machine instruction usually involves the use of machine registers, such as the accumulator, the multiplier-quotient register, or an index register. By keeping track of the contents of these registers, the assembler can generate more efficient binary code. For example, if the value of a variable is in the accumulator, there is no need to reload the variable from core storage. To keep track of the contents of the machine registers a word for each machine register is reserved in a list called the Register Association List. If the execution of a macro leaves the result of the macro in a machine register, a two-way pointer will be set up between the Register Association List and the macro block. The word corresponding to the machine register in the Register Association List will contain a pointer to the second word in the macro block, and the decrement of the second word in the macro block will contain a pointer back to the word in the Register Association List. The pointers are used to determine the location of the result when the result is referenced by another macro. When the machine register is used by another computation, the two-way pointer will be erased.

To use this feature, it is necessary to declare the machine registers that each macro uses. The Register List statement of the Macro Interpretation Language (see Section 6.4) is used for this purpose.

5.5 MACRO INTERPRETATION TABLE

The Macro Interpretation Table is used to specify the interpretation of the macros. Each macro is defined by a block of entries in the Macro Interpretation Table. There are several types of macro interpretation entries. These entries and their fields are given in Table 5-1.

The example operands and comparands given in Table 5-1 are of a limited type. In general, operands and comparands may be of a more varied form. Table 5-2 gives a list of the possible types of operand. The binary operators +, -, *, .A. and .X. and unary operators .L. and .R. designate the operations of addition, subtraction, multiplication, logical "and", logical "exclusive OR", logical "left shift", and logical "right shift" respectively. These operators may be used to combine the constituents of an or-segment. (The logical "or" operator is not defined since all or segments are eventually combined in a logical "or".)

5.6 OPERATION OF THE ASSEMBLER

The first action the assembler takes is to initialize the temporary storage areas using the subroutine INTEMP. (See Appendix E, Figure E-1.) The instruction counter is set equal to the address of the first location not used for temporary storage. The macros are then processed one at a time. The routine GETMAC is used to get the starting location of the block of registers in the Macro Interpretation Table that describes how the macro should be interpreted. Next the arguments of the macro are tested. For a type 0 argument (a table entry pointer), the pointer to the main pointer table will be replaced by the address of the entry in an information table. The count of the macro is then tested. If temporary storage is needed,

Table 5-1. Entries and Fields for Macro Interpretation Table

Entry Type	Prefix-Tag of Encoded Entry	Entry Format	Example Entry	Interpretation	(Corresponding statement in Macro Interpretation Language)
TEMPORARY STORAGE	0-0	loc.buf. n	52000 52030 3	This entry must be the first in each macro block. It specifies the number n of temporary storage words used by this macro; loc. buf. gives the location of the first of the n contiguous words which will contain pointers to the words taken from the temporary storage pool	TEMP 3
REGISTER LIST	3 - 2	3 adr1 2 n1 n2 • • •	52001 3 52004 2 3 52002 5 52003 6	Adr1 (52004) is the address of the first word in the next entry; n1, n2, . . . , give the machine registers used in the Macro Interpretation block; n1 is the register in which the result of the macro will be left	RL R3, R5, R6 RL R1 (R1 is accummulator)
TRANSFER	2 - 0	2 0 0 entry1	52004 2 0 0 52022	The control is passed to entry1 (52021)	GO +3
ERROR	2 - 0	2 n 0 0	52005 2 7 0 0	The error message associated with the n (7) is to be printed (note: the Error entry is distinguished from a Transfer entry by the non-zero decrement)	ERR 7
CONDITIONAL	1 - 0	1 0 comparand1 comparand2 loc1 loc2 loc3 • • •	52006 1 0 66637 A 52007 64216 B 52010 52013 52011 52015 52012 52020 52013 52015 52021 52014 2 0 52022 GO +1 52015 52021 52021 52016 2 5 0 0 ERR 5 52017 2 0 52004 GO -2 52020 52021 52021 52021 2 0 0 0 END 66637 A 64216 B	If comparand1 (A) is less than, equal to, or greater than comparand2 (B), then the set of entries beginning in loc1, loc2, or loc3 respectively is examined. The first word in these sets of entries contain a) a pointer to the next set of entries in their decrement, and b) a pointer to the last word of the conditional entry block in their address; the first word is followed by an encoding of the entries to be scanned if control was passed to this point	IF A=B (GO +1. ERR 5, GO -1. END)

Table 5-1. Entries and Fields for Macro Interpretation Table (Cont.)

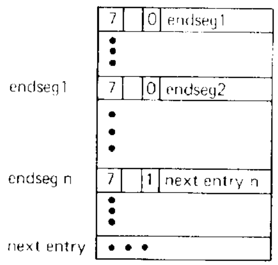
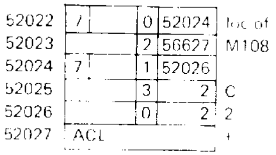
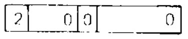
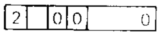
Entry Type	Prefix Tag of Encoded Entry	Entry Format	Example Entry	Interpretation	(Corresponding statement in Macro Interpretation Language)
GENERATE	7 0 or 2	 7 0 endseg1 endseg1 7 0 endseg2 endseg n 7 1 next entry n next entry • • •	 52022 7 0 52024 loc of 52023 2 56627 M108 52024 7 1 52026 52025 3 2 C 52026 0 2 2 52027 ACL +	The Generate entry specifies the generation of one word of binary machine code. Each segment of words specifies the machine code for one or segment, all or segments for a single entry are "or"ed together to obtain the word of binary machine code. Endseg1 (52023) gives the location of the last word for the enclosed or segment. The remaining words are a list of operands and operators, written in post-fix form. For example a tag of 3 and address of 2 specifies C (the instruction locat on counter) as an operand, a tag of 0 specifies an integer, and the letters ACL specify the operation + to be made on the preceding operands	GEN (M108)(C+2)
END MACRO	2 0		52030 	The End Macro entry terminates the block of entries for a macro. (note: The End Macro entry is distinguished from the Error and Transfer entries by having both a zero decrement and zero address.)	END

Table 5-2. Operands and Comparands for Macro Interpretation Table

Operand Type	Tag of encoded word	Example Entry	Interpretation	Corresponding reference in Macro Interpretation Language statement
DECLARED VARIABLE	0	[ ][ ][23302]	23302 is the location of a variable quantity	ALPHA
INTEGER	0	[ ][ ][0][14]	14 is octal for the decimal integer 12	12
		[ ][ ][0][10]	10 is octal for the octal integer 10	10
A-TYPE or	6	[ ][2][6][3]	Use the 2nd field of the information table entry pointed to by the 3rd argument of the macro being evaluated. An S-TYPE entry (tag = 7) means the field is to be right shifted.	A2.3
S-TYPE	7	[ ][2][7][3]		S2.3
P-TYPE	5	[ ][ ][5][3]	Use the 3rd argument of the macro. If the argument is type 0, (pointer) its value is the table name of the table pointed to; if the argument is type 1, its value is the result of a previous macro; if the argument is type 2, (integer) its value is the integer	P3
M-TYPE	2	[ ][ ][2][56627]	56627 is the address of a word in the Machine Code Table	M108
T-TYPE	3	[ ][7][3][1]	Use the contents of the 7th word in the temporary storage pool	T7

Table 5-2. Operands and Comparands for Macro Interpretation Table (Cont.)

Operand Type	Tag of encoded word	Example Entry	Interpretation	Corresponding reference in Macro Interpretation Language statement
R-TYPE	3	[ ] [0] [3] [3]	Use relative address in temporary storage pool of result of current macro. (note: the address of 3 differentiates an R-TYPE entry from A-TYPE entry)	R
RL-TYPE	4	[ ] [ ] [4] [2]	Use the contents of the 2nd register in the hardware register list	RL2
TY-TYPE	1	[ ] [ ] [1] [3]	Use the 3rd argument of the macro	TY3
C-TYPE	3	[ ] [ ] [3] [2]	Use contents of instruction location counter (note: the address 2 distinguishes a C-TYPE entry from a T-TYPE or R-TYPE entry.)	C

the subroutine TEMPTK is called to assign a word of temporary storage to the macro and TEMPTK will return the address of the released temporary storage word and this address will be inserted into the second word of the block of words for the macro. The block of entries in Macro Interpretation Table is then examined for the interpretation of the macro. The first entry in the block (a Temporary Storage entry) specifies the maximum number and starting location of the internal temporary storage words that the macro will use. The subroutine TEMPTK is called again, and the temporary storage words will be assigned accordingly. Subroutine GENPRO is then called to process the macro.

At the end of the processing of a macro, all internal temporary words used by the macro will be returned to the temporary storage pool and the Assembler will examine the next macro in the list of macros. After all the macros are processed and the binary program generated, the identification bits will be scanned and the corresponding addresses in the binary program will be modified by adding the program break.

The subroutine GENPRO (See Appendix E, Figure E-2) processes the entries in the Macro Interpretation Table. A pointer is kept in index register 7 pointing to the current entry.

Subroutine GEN is used to process a generate entry. (See Appendix E, Figure E-3.) The or-segments in each generate statement are processed according to their operator-operand pair. The operand in each pair is converted by the subroutine CONVCN. An indicator is set to denote the type of the operand (an integer, a table-entry, a machine instruction code, a temporary storage). The result of the execution of an or-segment is or-ed to the sense indicators. At the end of the or-segment, the type of integer, as well as the number of bits to be left shifted, are collected in a list called the relocation bit information list. This list is useful in the generation of relocation bits. After the result of the last segment is or-ed to the sense indicators, the values of the sense indicators, which constitute a binary word in the object program, will be saved in the buffer pool that constitutes the object program. After a binary word is generated, the subroutine RELCBT is called. This subroutine will examine the relocation bit information list and generate the appropriate relocation bits.

The subroutine CONVER is called to get the value of a comparand when a conditional statement is processed. There is no need to find out the type of a comparand. The subroutine CONVCN is called during the generation of identification bits if it is necessary to determine the type of operand. When the operand has a 6 in the tag field (Si.j. type), a subroutine GET in the table processor is called to get its value and to determine the type of the operand. When the operand has a 7 in the tag field (Ai.j. type), then in addition to the information to be obtained from a S-type, the number of bits to be right-shifted to make it right-justified is also needed. This is done by the same subroutine in the Table Processor. The flow chart of the subroutines is shown in Appendix E, Figure E-4. When a register list entry is encountered, the existing two way pointer for the specified registers are erased and a new set of two-way pointers corresponding to the macro defined in the new entry is set. When a conditional entry is encountered, the conditional level counter* is examined to determine the level of conditional testing. The arguments in the first and second words are converted into their respective values by the subroutine CONVER; the values of the arguments are compared and the pointer (index register 7) is updated according to the result of comparision.

*The conditional level counter (in index register 5) is set to zero at the beginning of the interpretation of of the macro, incremented by one each time a conditional entry is encountered, and decremented by one each time an exit from a conditional entry is made.

SECTION VI

THE CONTROL LANGUAGES

6.1 INTRODUCTION

For the control tables to be properly filled, the designer must prepare statements in the three control languages. The statements in these languages are interpreted by the bootstrap translators and encoded into the control tables. (See Figure 2-3.) Two of the control languages, the Table Declaration and Manipulation Language and the Macro Interpretation Language, and their bootstrap translators are fixed within the system. The third control language, the Syntax Defining Language, and its bootstrap translator must be prepared by the designer.

The following sections describe the two fixed languages and present an example Syntax Defining Language*. The BNF syntax of a sample source language and the statements needed to completely design a compiler for the sample source language are given in Appendix F.

6.2 AN EXAMPLE SYNTAX DEFINING LANGUAGE

In this section we present an example of a Syntax Defining Language, Markstran - named after its designer R. E. Marks. Markstran resembles the language described in reference 6 . The statements in Markstran define the method of syntactic analysis to be used by the Syntactic Analyzer. We first present the syntax of Markstran, and then give the Markstran program for syntactic analysis of the sample source language.

The Markstran language has five basic statement types, Lexical declarations, Test declarations, Stack declarations, ATAB statements and TTAB statements. These statement types are translated (by the Markstran bootstrap translator) into entries in the LTAB, TTAB, STAB, and ATAB tables, respectively.

6.2.1 Lexical Declarations

Lexical declarations have the following form:

⟨Lexical declaration⟩ : := ⟨lexical block name⟩ = ⟨LEXICAL right part list⟩
⟨lexical block name⟩ : := ⟨identifier⟩
⟨LEXICAL right part list⟩ : := ⟨lexical symbol⟩ [/ ⟨lexical symbol⟩]$_0^\infty$
/ [/ ⟨termination symbol⟩]$_0^\infty$

* The example Syntax Defining Language and its bootstrap translator have been implemented and may be used by the compiler designer.

⟨lexical symbol⟩ : := ⟨bcd string⟩ | ⟨keyword⟩
⟨termination symbol⟩ : := ⟨bcd string⟩ | ⟨keyword⟩
⟨keyword⟩ : := ALPHABETIC | INTEGER | OINTEGER | BLANK |Λ*

A keyword is the symbolic name for a class of characters. Each character in a class will have the bit representing its class set to 1 in the character's entry in CPLIST. The keywords ALPHABETIC, INTEGER, OINTEGER and BLANK represent strings of alphabetic characters, digits, octal digits, and blanks respectively. For example, the lexical declaration

LIT = INTEGER / .INTEGER / INTEGER. / INTEGER.INTEGER // BLANK

defines the lexical block LIT, which will match decimal literals followed by a blank. The declaration

TS = +/-/*///EQ/NE/GT/LT/LE/GE// BLANK

defines the lexical block TS which will match any of the terminal symbols +,-, ... ,GE followed by a blank.

In the above examples TS and LIT must be declared as information tables. MARKSTRAN assumes that there is only one table reference number for each Lexical declaration and that each table referenced is declared in the Table Processor declarations. Corresponding to each list of lexical or terminal symbols in a Lexical declaration, an information table must be declared to contain the symbols.

The Lexical declarations are encoded into LTAB entries. An example encoding for the lexical block for LIT is given in Section 3.3. There is no simple one-to-one mapping between lexical declarations and entries in LTAB. However, for any given lexical declaration, the mapping is unambiguous. Rapid lexical analysis is important; therefore, it is usually desirable to hand code the lexical tables rather than to use the possibly less optimum ones that could be encoded by the bootstrap translator.

6.2.2 Test Declarations

There are two kinds of Test declarations: the VALUE TEST declaration and the PROPERTY TEST declaration.

⟨Test declaration⟩ : := VALUE TEST ⟨test primary⟩ | PROPERTY TEST ⟨test primary⟩
⟨test primary⟩ : := ⟨field name⟩ = ⟨TEST right part list⟩
⟨field name⟩ : := ⟨STAB pointer field name⟩ ⟨information table field name⟩

* "Λ" stands for the null character

⟨STAB pointer field name⟩ : := PFLGF | PFLES | PADD | PINTP | PPTRS | PPTR
⟨information table field name⟩ : := ⟨identifier⟩
⟨TEST right part list⟩ : := [⟨test value⟩ | ⟨test value⟩ (⟨test symbol list⟩)]$^{1}_{1}$
[,⟨test value⟩ | , ⟨test value⟩ (⟨test symbol list⟩)]$^{\infty}_{0}$
⟨test value⟩ : := ⟨identifier⟩
⟨test symbol list⟩ : := ⟨test symbol⟩ [, ⟨test symbol ⟩]$^{\infty}_{0}$
⟨test symbol⟩ : := ⟨bcd string⟩

An information table field name must be the symbolic name of an information table field. A test symbol must be the symbolic name for a symbol defined by a lexical declaration.

Example: Consider the declarations

VALUE TEST PFLGF = OFF, ON
VALUE TEST PADD = TERM, FACT, AEXP

The first declaration defines a test for one of the two values OFF or ON on the PFLGF field of a pointer. The second declaration defines a test for one of the three values TERM, FACT, AEXP on the PADD field of a pointer. Example: Within the analyzer a "property" is represented by the occurrence of a single bit. For example, if 001_8, 002_8, and 004_8 represent three properties, the field with value 003_8 has two properties 001_8 and 002_8. A "property test" on a field is a check for the existence of one or more property bits in the field. If the Lexical declaration

TS = +/-/*///EO/NE/GT/LT/LE/GE//BLANK

has been made. The test declaration

PROPERTY TEST TPROP = ADDOP (+,-), MULOP(*,/), RELOP (EQ,NE,GT,LT,GE,LE)
will, a) assign the values 1_8, 2_8 and 4_8 to the symbolic names ADDOP, MULOP, and RELOP respectively, b) initialize the entries + and - of the TS table to the value 1_8, the entries * and / to 2_8, and EQ,...,LE to 4_8, and c) assign the value 7_8 to the symbolic name TPROP. The tests ADDOP(P), RELOP(P), and TPROP(P), where P is a pointer to the EQ entry in the TS information table, would have the values FALSE, TRUE, and TRUE respectively.

6.2.3 Stack Declarations

Stack declarations declare symbolic names and maximum sizes of the stacks,

⟨stack declaration⟩ : := STACKS ⟨stack list⟩
⟨stack list⟩ : := ⟨stack name⟩ (⟨integer⟩) [, ⟨stack name⟩ (⟨integer⟩)]$^{\infty}_{0}$
⟨stack name⟩ : := ⟨identifier⟩

The declared stacks are set up as blocks of STAB entries.

Example: The declaration

STACKS STACKL(20)

defines a stack of maximum length 20, associates the symbolic name STACKL with the base address of the stack, and initializes STACKL(-1) to 20.

6.2.4 ATAB Statements

The ATAB statements are used to specify the sequence of actions for syntactic analysis

⟨ATAB statement⟩ : := [⟨label⟩ $\$]_0^1$ [⟨command statement⟩ | ⟨IF statement⟩
| ⟨assignment statement⟩ $]_1^1$ | ⟨predicate definition⟩

⟨command statement⟩ : := ⟨command name⟩ $[(\langle\text{argument list}\rangle)]_0^1$

⟨IF statement⟩ : := IF ⟨boolean expression⟩ THEN [⟨ATAB statement⟩] END|
IF ⟨boolean expression⟩ THEN [⟨unlabeled ATAB statement⟩] ELSE
[⟨unlabeled ATAB statement⟩] END

⟨assignment statement⟩ : := ⟨identifier⟩ = ⟨value⟩

⟨predicate definition⟩ : := PREDICATE START ⟨predicate name⟩ [⟨ATAB statement⟩$]_0^\infty$
STOP

⟨command name⟩ : := ⟨basic ATAB-TTAB operation⟩ | ⟨macro command name⟩

⟨argument list⟩ : := ⟨argument⟩ [, ⟨argument⟩]

⟨argument⟩ : := ⟨label⟩ | ⟨value⟩ | ⟨bcd string⟩ | ⟨stack name⟩

⟨value⟩ : := ⟨integer⟩ | ⟨test value⟩

⟨predicate name⟩ : := ⟨identifier⟩

A command statement results in the formation of an ATAB entry. The basic ATAB operations are given in Appendix B. In addition to the basic ATAB operations, several special macro commands are defined within Markstran. A macro command and its arguments, like a FAP macro call, may be used in place of a sequence of commands. The macro commands are given in Table 6-1. An IF statement results in the generation of several ATAB entries. The first entry is a conditional transfer; the remaining entries are those designated by the statements in theTHEN-ELSE and ELSE-END blocks. An assignment statement generates a STORE entry in the ATAB table. A predicate definition is used to define names for predicate macro definitions of one argument. For example, the definition

```
PREDICATE START    EQ2

    IF PREDVR EQ 2 THEN SETTRUE (1).  RETURN.  ELSE

    SETTRUE ( ).  RETURN.

    STOP
```

defines the predicate EQ2. The system variable PREDVR is used in all macro definitions and refers to the argument passed in the macro definition. The call EQ2(N) will result in

Table 6-1. Markstran Macro Definitions

Macro name	Argument	Equivalent ATAB operation	Encoded ATAB entries APON	APTR	ANUM	ASTK	AFLD	AVLPTR	AUSPTR	Meaning
LOAD		GET CURSYM	15	1000	1	0	—	1	0	Put next input symbol on top of STACKQ
			16	1002	1	1	—	1	0	
PRINT COMM	((bcd string))	GET (bcd string)	15	0042	1	0	0 0	0	0	Print the bcd string given as its argument
		PRINT (bcd mode no)	28	—	AFALSE = 5 ATRUE = 0					
		PRINT (output mode)	28	—	AFALSE = 6					
EXCISE	(n)	GET STACKQ(0)	15	1002	2	0	0	0	0	Remove top n elements from STACKQ
		MINUS n	3	n	1	0	0	0	0	
		PUT STACKQ(0)	16	1002	2	0	0	0	0	
SET	(n)	GET n	15	n	1	0	0	0	0	Place the value n in the PADD field of the points on top of STACKQ
		PUT PAD (STACKQ)	16	1002	1	1	3	1	2	
STACKTOP	(n)	GET		1002	1	2	0	0	0	Access the n th element from the top of STACKQ
				n	1	4	-	1	0	
POINT	(pntr)	—	16	pointer				1	1	The PINTP, PPTR and PPTRS fields of a pointer are used for forming a pointer

The following values are assumed: loc of CURSYM = STAB(1000)
base of STACKQ = STAB(1002)
loc of bcd comment = 0042

the formation of the ATAB entries for the predicate defined above, where N will be used in place of the identifier PREDVR.

6.2.5 TTAB Statements

The following two system variables are defined within Markstran:

STACKQ	The symbolic name for the stack of pointers upon which syntactic analysis is performed.
CURSYM	The symbolic name for the location into which the result of lexical analysis is put.

The occurrence of the special label INITIAL designates the first ATAB statement.

The TTAB statements are used to control routine TEST:

⟨TTAB statement⟩ : := [⟨label⟩ $]$_0^1$ ⟨unlabeled TTAB statement⟩
⟨unlabeled TTAB statement⟩ : := /[⟨test name⟩]$_1^\infty$ // | // ⟨test name⟩ /// |
/[⟨test name⟩]$_1^\infty$ // ⟨test name⟩ ///

The statements have the following meaning (a_n, ... a_o and b are test names):

TTAB statement form	Meaning
/ a_n . . . a_o //	a_n specifies a test on STACKQ(n), ... , a_o specifies a test on STACKQ(0).
// b ///	b specifies a test on CURSYM
/ a_n ... a_o // b ///	a_n specifies a test on STACKQ(n), ... , a_o specifies a test on STACKQ(0), and b specifies a test on CURSYM

Following each TTAB statement must be one or more ATAB statements. A test is specified by giving the name of a defined table entry, the name of a table, or the name of a value or property test. If the stack element being tested has all of the values and properties associated with the given name, then that test is TRUE. If all the tests in a test sequence are TRUE, the ATAB statements that follow are executed; if any test in the sequence is FALSE, then the next test sequence is performed. There is a pseudo-test, named OTHERWISE, which is always TRUE. When the name of a stack is mentioned in a statement, that stack is to be used as a pushdown.

Example: Consider the statements:

```
   X1$    // START /// LOAD.  DO(SCAN).  TEST(S1)
         OTHERWISE ERR$  PRINT  COMM(ILLEGAL PROGRAM).
                                ERROR EXIT.

S1$ // . . .
```

If CURSYM points to the symbol START, then a) the macro command load is executed, b) control is passed to the statement labeled SCAN for performing more lexical analysis, and c) if SCAN returns without error, control is passed to the TTAB statement labeled S1. Otherwise, an error comment is printed and an error exit is taken.

Example: Assume that the TS and LIT declarations have been given as in the previous examples and that:

ADDOP	= 1	TS table reference number	= 32000
MULOP	= 2	TPROP field number	= 270
RELOP	= 4	+ reference number	= 13000
OFF	= 0	Next ATAB entry to be checked	= ATAB(40)
ON	= 1	Current TTAB entry to be generated	= TTAB(20)
CURSYM	= STAB(1000)		
STACKQ(0)	= STAB(1002)		

The following TTAB statements would cause the corresponding TTAB entries to be generated.

Part of statement	Encoding for TTAB entry
// + ///	TLOC = 1000, TSTPT = 3, TWHAT = 2, TTEST = 13000, TMNPRP = 0, TTRUE = 1, TTDONE = 40, TFALSE = 0, TFDONE = 21
// ADDOP ///	TLOC = 1000, TSTPT = 3, TTABLE = 32000, TTBTEST = 1, TWHAT = 2, TSFLD = 270, TFLDTS = 1, TTEST = 1, TMNPRP = 1, TTDONE = 40, TTRUE = 2, TFDONE = 24, TFALSE = 0
/ IF //	TSTPT = 1, TLOC = 1002, TPLMN = 1, TPOS = 0, TWHAT = 2, TMNPRP = 0, TTEST = 14677, TRUE = 1, TTDONE = 236, TFALSE = 0, TFDONE = 26
// EQREL ///	TLOC = 1000, TSTPT = 3, TTABLE = 32764, TTBTST = 1, TWHAT = 2, TSFLD = 271, TFLDTS = 1, TTEST = 1, TMNPRP = 1, TTDONE = 236, TTRUE = 1, TFDONE = 27, TFALSE = 0
/ OFF //	TLOC = 1002, TSTPT = 1, TPOS = 0, TPLMN = 1, TWHAT = 5, TMNPRP = 0, TTEST = 0, TTRUE = 1, TTDONE = 236, TFALSE = 0, TFDONE = 26
/ a b //	TLOC = 1002, TSTPT = 1, TPOS = 1, TPLMN = 1, TWHAT = 2, TFLDTS = 1, TSFLD = 0, TMNPRP = 0, TTEST = 4 -test for "a" assume TSFLD = 0-, TTRUE = 0, TTDONE = 26, TFALSE = 0, TFDONE = 30

TLOC = 1002, TSTPT = 1, TPOS = 0, TWHAT = 2, TFLDTS = 1,
TSFLD = 1, TMNPRP = 0, TTEST = 1, -test for "b" assume
TSFLD = 1 - TTRUE = 1, TTDONE = 236, TFALSE = 0,
TFDONE = 30

6.3 TABLE DECLARATION AND MANIPULATION LANGUAGE

The information tables formed during syntactic analysis must be defined using the Table Declaration and Manipulation Language. The language consists of two classes of statements, table declaration statements and table processing statements. The table declarations allow the designer to specify the name, the maximum number of entries, the fields, and the sorting option for each information table. The table processing statements allow the designer to specify how the information tables are to be processed by the Table Processor upon completion of the syntactic analysis.

6.3.1 Table Declaration Statements

There are two types of table declaration statements: type A, in which the bootstrap translator assigns the packing of fields, and type B, in which the designer specifies the packing of fields. These statements are of the form:

⟨table declaration⟩ : := ⟨type A declaration⟩ | ⟨type B declaration⟩
⟨type A declaration⟩ : := ⟨table name⟩⟨integer⟩⟨sorting option⟩⟨field list⟩
⟨type B declaration⟩ : := ⟨table name⟩⟨integer⟩⟨sorting option⟩⟨field and packing list⟩
⟨sorting option⟩ : := NO SORT | SORT | SORT (⟨integer⟩ , ⟨integer⟩)
⟨field list⟩ : := ⟨integer⟩ [, ⟨integer⟩]$_0^\infty$
⟨field and packing list⟩ : := ⟨integer⟩ (⟨integer⟩ , ⟨integer⟩) [, ⟨integer⟩
(⟨integer⟩ , ⟨integer⟩)]$_0^\infty$

The integer following the table name gives the maximum number of entries in the table named. The sorting option NO SORT designates a table not to be kept sorted during syntactic analysis; the sorting option SORT designates a table to be kept alphabetically sorted on the basis of the first (BCD) field in each entry; the sorting option SORT(integer,integer) designates a table to be kept sorted, where the first integer gives the index of the field used for sorting and the second integer identifies a designer-specified sorting scheme (see sorting scheme statement below). Each integer in a field list designates the number of bits in a field; the number of integers in the bit list is the number of fields in an entry. Each set of three integers in a field packing list designates a field and the way it is to be packed; the first integer is the number of bits in the field, the second is the index in the entry of the word that the first bit is to occupy, and the third is the position of the first bit in the word.

In addition to the table declaration statement, an additional statement is used to specify sorting schemes. This statement has the form:

⟨sorting scheme statement⟩ : := ⟨integer⟩ [, ⟨integer⟩]$_1^{63}$

The first integer identifies the sorting scheme to be defined; it must have a value between two and six (the standard BCD sorting scheme is referred to as scheme one). The remaining integers gives the indices of the fields upon which the sorting is to be based and the order of precedence of the fields; the order of the integers is the order of precedence and each integer must have a value between one and six.

6.3.2 Table Manipulation Statements

Table manipulation statements are used to specify the processing of tables upon completion of syntactic analysis. The table manipulation statements are grouped into blocks of statements, each block specifying certain actions for each of the entries in an information table. Each block of statements consists of a process statement, memory initialization statements, and action statements:

⟨statement block⟩ : := ⟨process statement⟩ [⟨memory initialization statement⟩ $]_0^\infty$
[⟨action statement⟩ $]_0^\infty$
⟨process statement⟩ : := PROCESS ⟨table name⟩ [, ⟨table name⟩ $]_0^\infty$
⟨memory initialization statement⟩ : := MEMORY INITIALIZATION
⟨variable name⟩ = ⟨integer⟩

In the process statement for a statement block, the first table named is referred to as the primary table in the block (table 0), and the remaining tables are referred to as the secondary tables in the block (tables 1, 2, 3, etc.).

Example statements	Encoded entries in table manipulation table *				
PROCESS TABLE-A, TABLE-B, TABLE-C	55000	1	55033	0	16600
PROCESS LITTAB	65000	1		0	14000

In the first process statement above, TABLE-A is declared as a primary table (table 0) and tables TABLE-B and TABLE-C are declared as secondary tables (tables 1 and 2 respectively).

Memory initialization statements are used to initialize the values of variables. For example, the statement

MEMORY INITIALIZATION RELOCA = 144

will reserve a word for RELOCA and initialize its contents to 144.

The action statements specify the processing to be performed on the tables named in the process statement. Upon completion of syntactic analysis, each action statement will be executed once for each entry in the primary table. An action statement can be one of the following types:

* The encoded entries are explained in Section 4.4

⟨action statement⟩ : := ⟨sort statement⟩ | ⟨insert statements⟩ | ⟨test statement⟩
⟨search statement⟩ | ⟨transfer statement⟩ | ⟨index statement⟩
⟨assignment statement⟩ | ⟨arithmetic statement⟩ | ⟨print statement⟩

⟨sort statement⟩ : := SORT ⟨table name⟩ (⟨integer⟩ , ⟨integer⟩)

⟨insert statement⟩ : := INSERT ⟨field⟩ INTO ⟨field⟩

⟨test statement⟩ : := TEST ⟨field⟩ AGAINST [⟨field⟩ | ⟨variable⟩ | ⟨integer⟩ | (bcd string)]$_1^\infty$

⟨search statement⟩ : : = SEARCH ⟨field⟩ FOR ⟨field⟩

⟨transfer statement⟩ : := $ ⟨integer⟩ $ | $ ⟨integer⟩ , ⟨integer⟩ $

⟨index statement⟩ : := INDEX [⟨field⟩ | ⟨variable⟩]$_1^1$ ⟨signed integer⟩

⟨assignment statement⟩ : := ⟨field⟩ = [⟨field⟩ | ⟨variable⟩ | ⟨integer⟩]$_1^1$

⟨arithmetic statement⟩ : := ⟨variable⟩ = ⟨arithmetic expression⟩

⟨print statement⟩ : := PRINT ⟨table name⟩ (⟨format specification⟩)

⟨format specification⟩ : := ⟨unit⟩ [, ⟨unit⟩]$_0^\infty$

⟨unit⟩ : := O/⟨integer⟩ | C/⟨integer⟩ | I/⟨integer⟩ | ⟨integer⟩ H/⟨bcd string⟩

Fields are specified by giving the table name and field number of the field referenced. Since action statements are executed once for each entry in the specified table, no entry number need be given. The table number implied by the location of the table name in a process statement may be substituted for the table name. For example, with respect to the process statement:

PROCESS TABLE-A, TABLE-B, TABLE-C

the field specification (TABLE-C, 2) refers to the second field of a TABLE-C entry, (1, 2) refers to the second field of a TABLE-B entry, and (0, 3) refers to the third field of a TABLE-A entry.

The sort statement specifies the name, the sorting field and sorting option for the table to be sorted.

Example statement	Coded entry in Table Manipulation table
SORT TABLE-A (1, 1)	55001 $\boxed{1}\boxed{16600}\boxed{1}\boxed{16602}$

The name (here TABLE-A) gives the name of the table to be sorted. The first integer gives the number (1) of the sorting scheme to be used. The second integer (1) gives the index of the field on which the sort is to be based.

The insert statement is used to insert an entry into another table.

Example statement	Coded entry in Table Manipulation table				
INSERT (TABLE-B,2) INTO (0,3)	55002	6	14423	0	14427
	55003		16600		16606

The test statement is used to test whether a field matches another field, variable, integer, or BCD string of six or less characters. If the match is successful, the accumulator will be set to 1; otherwise it will be set to zero.

Example statements	Coded entries in Table Manipulation table				
TEST (TABLE-C,1) AGAINST (ABC)	55004	4	17700	4	17702
	55005		000000		212223
TEST (1,1) AGAINST 0		4	14423	3	14425
					0

A search statement is used to search all instances of a field in one table for a match to a field in the current entry in another table. If a match is found, the accumulator will be set to 1; otherwise the accumulator will be set to zero.

Example statement	Coded entry in Table Manipulation table				
SEARCH (0,1) FOR (2,1)	55006	5	16600	0	16602
	55007		17700		17702

The transfer statement is used to make conditional and unconditional transfers to other statements in the statement block.

Example statements	Coded entries in Table Manipulation table				
$ 3,-2 $	55010	2	55015	1	55004
$ 3 $		2		0	55014
$ -2 $		2		0	55004

In the first example a transfer to the third following statement is indicated. In the second example a transfer to the second previous statement is indicated. In the third example, if the accumulator contains a 1, a transfer to the third following statement is indicated, if the accumulator contains a 0, a transfer to the second previous statement is indicated.

The index statement increments the field or variable named by the value of the integer.

Example statement	Coded entry in Table Manipulation table				
INDEX (2,2) -2	55011	2	17700	3	17704
	55012	4			2

The assignment statement is used to fill a field of the current entry in a table

Example statements	Coded entry in Table Manipulation table				
(0,2) = (2,3)		3	16600	1	16604
			17700		17706
(2,3) = 5	55013	3	17700	2	17706
	55014				5
(LITTAB,3) = RELOCA		3	14000	3	14006
					52321

In the first example, the second field of TABLE-A is filled with the third field of TABLE-C. In the second example, the third field of TABLE-C is filled with "5". In the third example, the third field of LITTAB is filled with the value of the variable RELOCA. All variables (except the reserved variables A0,A1, ..., A9 used in the Table Processor for indexing) must be declared by a Memory Initialization statement.

The arithmetic statement is used to set the value of a variable.

Example statement	Coded entry in Table Manipulation table				
A = A + (2,2)*5	50015	7		7	00006
	55016	TSX		4	50332
	55017		17700		17704
	55020	MPY			20313
	55021	XCA			
	55022	ADD			26401
	55023	STO			26401

Coded entry in control table

20313		5	5
26401		3	A

The Print statement is used to print the contents of an information table. Each unit specifies a field or BCD string to be printed. The letters O, C, and I designate octal, BCD, and integer fields; the integer following these letters is the field number of the field to be printed. The letter "H/" designates the printing of a BCD string; the integer before the H is the number of characters in the string.

Example statement	Coded entry in control table				
PRINT TABLE-A (C/1,4H ,I/2,4H ,I/3)	55024	1	14423	7	7
	55025	23			1
	55026	30			4
	55027	60606060			0000
	55030	46			2
	55031	30			4
	55032	60606060			0000
	55033	46			3

The example statement above will cause the printing of three fields in each TABLE-A entry. The first field will be printed in BCD format, the second and third in integer format; the fields will be separated by four blanks.

6.4′ MACRO INTERPRETATION LANGUAGE

The Macro Interpretation Language is used to specify the interpretation of the macros. Each macro is defined by a block of statements in the Macro Interpretation Language:

⟨macro interpretation program⟩ : : = [⟨macro interpretation block⟩ $]_0^\infty$

⟨macro interpretation block⟩ : : = [⟨macro interpretation statement⟩ $]_0^\infty$

⟨macro interpretation statement⟩ : : = ⟨register list statement⟩ | ⟨temporary storage⟩
⟨define statement⟩ | ⟨transfer statement⟩
⟨error statement⟩ | ⟨end macro statement⟩
⟨conditional statement⟩ | ⟨generate statement⟩

⟨register list statement⟩ : : = RL ⟨hardware register name⟩ [, ⟨hardware register name⟩]

⟨temporary storage⟩ : : = TEMP ⟨integer⟩

⟨define statement⟩ : : = ⟨declared variable⟩ = [⟨signed integer⟩ | ⟨hardware register name⟩ $]_1^1$

⟨transfer statement⟩ : : = GO ⟨signed integer⟩

⟨error statement⟩ : : = ERR ⟨integer⟩

⟨end Macro statement⟩ : : = END

⟨conditional statement⟩ : : = IF ⟨comparand⟩ = ⟨comparand⟩ (⟨unconditional statement list⟩.
⟨unconditional statement list⟩ . ⟨unconditional statement list⟩)

⟨generate statement⟩ : : = GEN [(⟨or segment⟩) $]_1^\infty$

⟨unconditional statement list⟩ : : = ⟨unconditional statement⟩ [⟨unconditional statement⟩ $]_0^\infty$

⟨unconditional statement⟩ : : = ⟨register list statement⟩ | ⟨transfer statement⟩ |
⟨error statement⟩ | ⟨end macro statement⟩ | ⟨generate statement⟩

⟨hardware register name⟩ : : = R0 | R1 | R2 | . . . | R10

⟨or segment⟩ : : = ⟨operand⟩ [⟨operator⟩ ⟨operand⟩ | ⟨shift operator⟩ ⟨integer⟩

⟨operand⟩ : : = ⟨declared variable⟩ | ⟨integer⟩ | ⟨octal integer⟩ | A⟨integer⟩.
⟨integer⟩ | S ⟨integer⟩.⟨integer⟩ P ⟨integer⟩ |
M ⟨integer⟩ | T ⟨integer⟩ | R | C

⟨comparand⟩ : : = ⟨declared variable⟩ | A ⟨integer⟩ .⟨integer⟩ |
S⟨integer⟩.⟨integer⟩ | P ⟨integer⟩ |
RL ⟨integer⟩ | TY ⟨integer⟩

⟨operator⟩ : : = + | - | * | .N. | .X.

⟨shift operator⟩ : : = .L. | . R.

The temporary storage statement is used to specify the maximum number of temporary storage words to be taken from the temporary storage pool for the macro. For example,

Example statement	Encoding in Macro Interpretation Table					
TEMP 3	52000		52030		3	(beginning of macro block)

will reserve three temporary storage words for each occurrence of the defined macro.

The Register List statement is used to identify the machine registers used by the macro. The first register identified is taken to be the register in which the result of the macro will be left. The keyword identifiers R1, R2, etc., are used to denote machine registers.

Example statement

RL R3, R5, R6

Encoding in Macro Interpretation Table

52001	3	52004	2	3
52002				5
52003				6

In the above example, registers R3, R5, and R6 will be used by the macro and the result of the macro will be left in R3.

The define statement is used to give mnemonic names to integers or machine registers,

Example statement

BASIS	=	32800	(a decimal integer)
END	=	5570K4	(an octal integer)
AC	=	R1	(a machine register)

The transfer statement is used to pass control to another statement. For example,

Example statement

GO -1

Encoding in Macro Interpretation Table

52004	2		0	52001

passes control to the preceding statement the statement and

GO +3

52004	2		0	52022

passes control to the third following statement.

The error statement is used to call an error printing routine which prints out the error message associated with the given integer. For example,

Example statement

ERR 7

Encoding in Macro Interpretation Table

52005	2	7	0	0

calls for a printing of the message associated with error number 7.

The conditional statement is used to execute one of three unconditional statement lists:

Example statement	Encoding in Macro Interpretation Table					
If A = B (GO +1. ERR 5, GO -2. END)	52006	1		0	66637	A
	52007				64216	B
	52010				52013	
	52011				52015	
	52012				52020	
	52013		52015		52021	
	52014	2			52022	GO 1
	52015		52020		52021	
	52016	2	5		0	ERR 5
	52017	2			52004	GO -2
	52020		52021		52021	
	52021	2	0		0	END

This example has the following meaning. If A is less than B, then go to the next statement; if A is equal to B, then print error message 5 and go to the preceding statement; if A is greater than B, then terminate the interpretation of the macro.

The generate statement is used to handle the generation of machine code. Each generate statement generates one word of binary machine code. The binary code for each or-segment within the generate statement is combined in a logical "or" operation to form the binary word for the statement. The formation of binary code for an or-segment depends on the operators and operands given in the or-segment.

Example statement		Encoding in Macro Interpretation Table				
GEN (M108) (C+2)	52022	7		0	52024	
	52023			2	56627	loc of M108
	52024	7		1	52026	
	52025			3	2	C (instruction loc. counter)
	52026			0	2	2
	52027	ACL				encoding of +

The execution of the example generate statement results in the following sequence of events:

1. The indicators will be set to 0.

2a. The machine code for the 108th entry in the Machine Code Table will be loaded into the accumulator.

b. The accumulator will be OR'd to the indicators.

3a. The contents of the instruction location counter will be loaded into the accumulator.

b. 2 will be added to the accumulator.

c. The accumulator will be OR'd to the indicators.

The indicators will contain the generated binary machine word.

The End Macro statement is used to terminate the block of statements for a macro:

Example statement		Encoding in Macro Interpretation Table			
END	52030	2	0	0	0

SECTION VII

CONCLUSION

The emphasis in our design has been the segmentation of the system to show separately the functions of each segment. Thus, the replacement or modification of a segment is eased. We feel that the communication between the segments is not difficult in our system. We have placed more emphasis on the generality and flexibility of our system than on its efficiency. We feel that this is an unavoidable trade-off. For example, consider the general structure of the information tables in the Table Processor. Because of the flexibility the system provides for the user to have information tables of arbitrary formats, looking up all the references in the Main Directory becomes a necessary intermediate step in accessing the information tables.

Besides providing the users with an environment in which they can write their own compilers, it is hoped that the experience of designing such a system will lead to the further understanding of the general theory of compiler structure and the general technique of compiler writing. The basic idea in designing such a system is to separate the common features of most compilers from the peculiarities of each individual compiler. We also see the possibility of using the system as a classroom instruction tool to demonstrate the functions of a compiler and to provide the students with opportunities for designing segments of a compiler.

BIBLIOGRAPHY

1. Irons, E.T., *The Structure and Use of the Syntax-Directed Compiler*, **Annual Review in Automatic Programming**, Vol. 3, 1963

2. Irons, E.T., *A Syntax-Directed Compiler for Algol-60*, **Communications of the ACM**, Vol. 4, 1961

3. Cheatham, T.E., Jr. and K. Sattley, *Syntax-Directed Compiling*, **Proceedings of the AFIPS SJCC**, Spartan Books, 1964

4. Warshall, S. and R.M. Shapiro, *A General-Purpose Table-Driven Compiler*, **Proceedings of the AFIPS SJCC**, Spartan Books, 1964

5. Chang, G.D., **A Table-Driven Compiler Generator System**, S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1966

6. Marks, R.E., **A Table-Driven Syntactic Analyzer**, S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1966

APPENDICES

APPENDIX A

DESCRIPTION OF STAB, LTAB, TTAB, AND ATAB FIELDS

Field Name	Mnemonic	No. of Bits	First Bit	Last Bit	Table Reference	Mask	Shift
STAB fields							
PFLGF	p-flag-first	1	0	-	STAB	1	35
PFLGS	p-flag-second	1	1	-	STAB	1	34
PADD	p-additional	15	3	17	STAB	77777	18
PINTP	p-interpret	2	18	19	STAB	3	16
PPTRS	p-pointer-sign	1	20	-	STAB	1	15
PPTR	p-pointer	15	21	35	STAB	77777	0
LTAB fields							
LTABL	L-table	15	3	17	LTAB	77777	18
LARG	L-argument	15	21	35	LTAB	77777	0
LPLMN	L-plus-or-minus	1	0	-	LTAB	4K11	35
LCHGXC	L-change-XC	1	1	-	LTAB	2K11	34
LPOS	L-position	5	2	6	LTAB	37	29
LTEST	L-test	15	7	21	LTAB	77777	14
LTBCD	L-true-BCD	2	22	23	LTAB	3	12
LFBCD	L-false-BCD	2	24	25	LTAB	3	10
LTADVN	L-true-advance-XC	2	26	27	LTAB	3	8
LFADVN	L-false-advance-XC	2	28	29	LTAB	3	6
LADVN	L-advance	6	30	35	LTAB	77	0
LWHAT	L-what	2	0	1	LTAB1	3	34 (2nd word)
LTRUE	L-true	2	2	3	LTAB1	3	32 (2nd word)
LTDONE	L-true-done	15	4	18	LTAB1	77777	17 (2nd word)
LFALSE	L-false	2	19	20	LTAB1	3	15 (2nd word)
LFDONE	L-false-done	15	21	35	LTAB1	77777	0 (2nd word)
TTAB fields							
TSTPT	T-test-pointer	2	0	1	TTAB	3	34
TPLMN	T-plus-or-minus	1	2	-	TTAB	1K11	33 (2nd word)
TLOC	T-location	15	3	17	TTAB	77777	18
TWHAT	T-what	3	18	20	TTAB	7	15
TPOS	T-position	15	21	35	TTAB	77777	0
TTBTST	T-table-test	1	2	-	TTAB1	1K11	33 (2nd word)
TINDR	T-indirect	1	1	-	TTAB1	2K11	34 (2nd word)

Field Name	Mnemonic	No. of Bits	First Bit	Last Bit	Table Reference	Mask	Shift
TTABLE	T-table	15	3	17	TTAB1	77777	18
TFLDTS	T-field-test	1	18	-	TTAB1	4K5	17 (2nd word)
TMNPRP	T-number-or-property	1	19	-	TTAB1	2K5	16 (2nd word)
TTEST	T-test	15	21	35	TTAB1	77777	0
TTRUE	T-true	1	1	-	TTAB2	2K11	34 (2nd word)
TTDONE	T-true-done	15	3	17	TTAB2	77777	18
TFALSE	T-false	1	19	-	TTAB2	2K5	16 (2nd word)
TFDONE	T-false-done	15	21	35	TTAB2	77777	0
TSFLD	T-processor-field	15	21	35	TTAB3	77777	0
ATAB fields							
AOPN	A-operation	6	0	5	ATAB	77	30
AVLPTR	A-value-or-pointer	1	10	-	ATAB	1	25
ANUM	A-number	2	11	12	ATAB	3	23
ASTK	A-stack	3	13	15	ATAB	7	20
AUSPTR	A-use-of-pointer	2	16	17	ATAB	3	18
AFLD	A-field	3	18	20	ATAB	7	15
APTR	A-point	15	21	35	ATAB	77777	0
AFALSE	A-false	15	6	20	ATAB	77777	15
ATRUE	A-true	15	21	35	ATAB	77777	0
AARG	A-argument	15	21	35	ATAB1	77777	0 (2nd word)

APPENDIX B

LIST OF ACTION OPERATIONS

The following is a list of operations performed by routine ACTION. The operations are grouped by function rather than by numeric order. No mention is made of removing processed elements from the stack, although elements are removed when necessary. VSTKO refers to the top element of VSTK, VSTK1 the next to the top element etc. Note the REVERSE operation which is used to process sequences as C-(A*B-D) without temporary storage or a complex analysis algorithm.

Value of AOPN	Mnemonic	Interpretation
Arithmetic Operations		
1	UNARY MINUS	-VSTKO
2	PLUS	VSTK1 + VSTKO
3	MINUS	VSTK1 - VSTKO
4	TIMES	VSTK1 * VSTKO
5	DIVIDE	VSTK1 / VSTKO (integer division)
6	REVERSE	interchange values of VSTK1 and VSTKO
30	LOGICAL AND	bit by bit logical "and" of VSTK1 and VSTKO
31	LOGICAL OR	bit by bit logical "or" of VSTK1 and VSTKO
33	ABS	VSTKO
34	SIGN	If VSTKO 0 then 1 else 0
38	TALLY	VSTKO + 1
Relational Operations		

7	LESS THAN	IF VSTK1 "relation" VSTKO
		true TRUE = 1
		else TRUE = 0
		(TRUE is the symbolic name of a system variable)

8	GREATER THAN	
9	EQUAL	
10	NOT EQUAL	
11	LESS OR EQUAL	
12	GREATER OR EQUAL	
13	EQUAL POINTER	IF PSTK1 "relation" PSTKO then TRUE = 1 else TRUE = 0
14	NOT EQUAL POINTER	
Control Operations		
17	CONDITIONAL TRANSFER	IF TRUE ≠ 0 the go to ATAB(ATRUE) else go to ATAB(AFALSE)
18	TRANSFER	go to ATAB(ATRUE)
19	COMPUTED TRANSFER	go to ATAB(VSTKO) (must use FETCH routine)

Value of AOPN	Mnemonic	Interpretation
20	DO	save current ATAB line number and go to ATAB(ATRUE)
21	COMPUTED DO	same as DO but use VSTK0
22	TEST	go to TTAB(ATRUE)
23	COMPUTED TEST	same as TEST but use VSTK0
24	RETURN	return to caller from predicate in TTAB or LTAB or from DO in ATAB
26	LEXICAL	see later explanation:
27	NEW TABLES	exit from analyzer giving a parameter which requests that the analyzers tables be overlaid with new tables. This is a machine extension operation.
0	HALT	normal exit from analyzer when done with analysis
36	ERROR EXIT	prints error comment and exits analyzer
Other Operations		
15	GET	call interpretive fetch routine to load stack
16	PUT	call interpretive store routine to load storage from stack

28	PRINT	PRNTSP = ATRUE
		PRNTMD = AFALSE
		PRNTVL = VSTK0 (if necessary)
		see explanation in text, section 3.6.5

29	MOVE	see explanation in text, section 3.6.5
32	NOP	dummy - no operation
35	ROUTINE	see explanation in text, section 3.6.5
37	NEW ENTRY	using subroutine call to table processor creates a pointer to a new-blank - entry in the table processor with table reference number ATRUE
39	ZERO	creates new VSTK entry of 0
40	OUTBCD	creates duplicate of table processor BCD string, represented by VSTK0, and places it in the analyzer's BCDTAB - VSTK0 is changed to represent this duplicate string
41	INBCD	creates duplicate of analyzer BCD string in the table processor format - VSTK0 represents this new string
25	SET TRUE	TRUE = VSTK0 - used to set truth value before predicate return TRUE = 0 is FALSE truth value
42	NEWCHR	calls routine to read input string (this is also done automatically by LEXICAL) See text, section 3.6.5
43-63	no defined operation	

APPENDIX C

ERROR COMMENTS

The system table ERRTAB controls the printing of error comments. The entries in this table are referenced by the system variable ERRFLG. The call ERRTAB(ERRFLG) is interpreted by the system as calling for a printing of an error comment in the following format:

```
ERROR FOUND IN ROUTINE routine-name
error-comment          sub-error-code
ERROR EXIT
```

The "routine-name" is the name of the system routine in which the error was found (ACTION or SYNTAX); the "error-comment" is the text comment describing the error; the sub-error-code gives further details about the error.

The table on the following page lists the possible errors.

Value of ERRFLG	Comment Printed	Sub-error-code	Probable cause of error
1	STACK OVERFLOW	Base address of offending stack	Reference made to an index in a stack which was greater than max. allowable value
2	STACK UNDERFLOW	Base address of offending stack	Use of stack index less than zero or fetch from an empty stack
3	ILLEGAL LEXICAL TEST	none	LEXICAL attempts to test a character with a CLIST index less than SXC
4	BCD TABLE OVERFLOW	1(CLIST overflow), 2(LABCD overflow), 3(BCDTAB overflow), or 4(XBCD overflow) (LABCD and XBCD are system tables)	
5	ILLEGAL TABLE FORMAT	1(ATAB), 2(TTAB) or 3(RTNTAB)	Undefined field value, e.g., ASTK = 7
6	ILLEGAL BCD STRING	"value" of string	First six bits in "positive" BCDTAB entry have value greater than 6
7	ILLEGAL LEXICAL PREDICATE	none	Lexical is called while evaluating a lexical predicate
8	NO BCD INPUT	none	Two consecutive break characters in input string
9	INCORRECT USE OF MOVE ACTION	none	Both VSTK and PSTK contain entries after the stack base value has been removed from VSTK
10	INCORRECT USE OF SYSTEM STACKS	66 (base of VSTK) or 84 (base of PSTK)	Either VSTK or PSTK not empty on exit from ACTION
11	CANNOT ACCESS POINTER ARGUMENT	Entry number of STAB entry containing pointer	Pointer used does not point to table processor
12	REFERENCE TO UNDEFINED SUBROUTINE	none or number of BCDTAB entry to subroutine name	If no sub-error-code, then ROUTINE operation attempts to use RTNTAB index greater than RTNMAX

APPENDIX D

BNF SPECIFICATION OF TERMINAL SYMBOLS AND BASIC SYNTACTIC TYPES

⟨letter⟩	: : =	A | B |C . . . | Y | Z
⟨digit⟩	: : =	0 | 1 | 2. . . | 8 | 9
⟨octal digit⟩	: : =	0 | 1 | 2. . . | 6 | 7
⟨integer⟩	: : =	[⟨digit⟩]$_1^\infty$
⟨octal integer⟩	: : =	[⟨octal digit⟩]$_1^\infty$
⟨symbol⟩	: : =	+ | - | * | / |(|) | . | $ | = | '
⟨signed integer⟩	: : =	⟨integer⟩ | + ⟨integer⟩ | - ⟨integer⟩
⟨bcd string⟩	: : =	[⟨character⟩]$_1^\infty$
⟨alphanumeric string⟩	: : =	[⟨letter⟩ | ⟨digit⟩]$_1^\infty$
⟨character⟩	: : =	⟨letter⟩ | ⟨digit⟩ | ⟨symbol⟩
⟨identifier⟩	: : =	⟨letter⟩ [⟨letter⟩ | ⟨digit⟩]$_0^\infty$
⟨defined name⟩	: : =	[⟨letter⟩]$_1^\infty$
⟨table name⟩	: : =	⟨identifier⟩
⟨variable name⟩	: : =	⟨identifier⟩
⟨arithmetic expression⟩	: : =	⟨term⟩ [⟨addition operator⟩ ⟨term⟩]$_1^\infty$
⟨term⟩	: : =	⟨factor⟩ [* ⟨factor⟩]$_1^\infty$
⟨factor⟩	: : =	⟨variable⟩ | ⟨field⟩ | ⟨integer⟩ | ⟨arithmetic expression⟩
⟨addition operator⟩	: : =	+ | -

APPENDIX E

FLOWCHARTS FOR THE ASSEMBLER

The following four flowcharts are presented to show graphically how the assembler and its various routines work together in the compiler system.

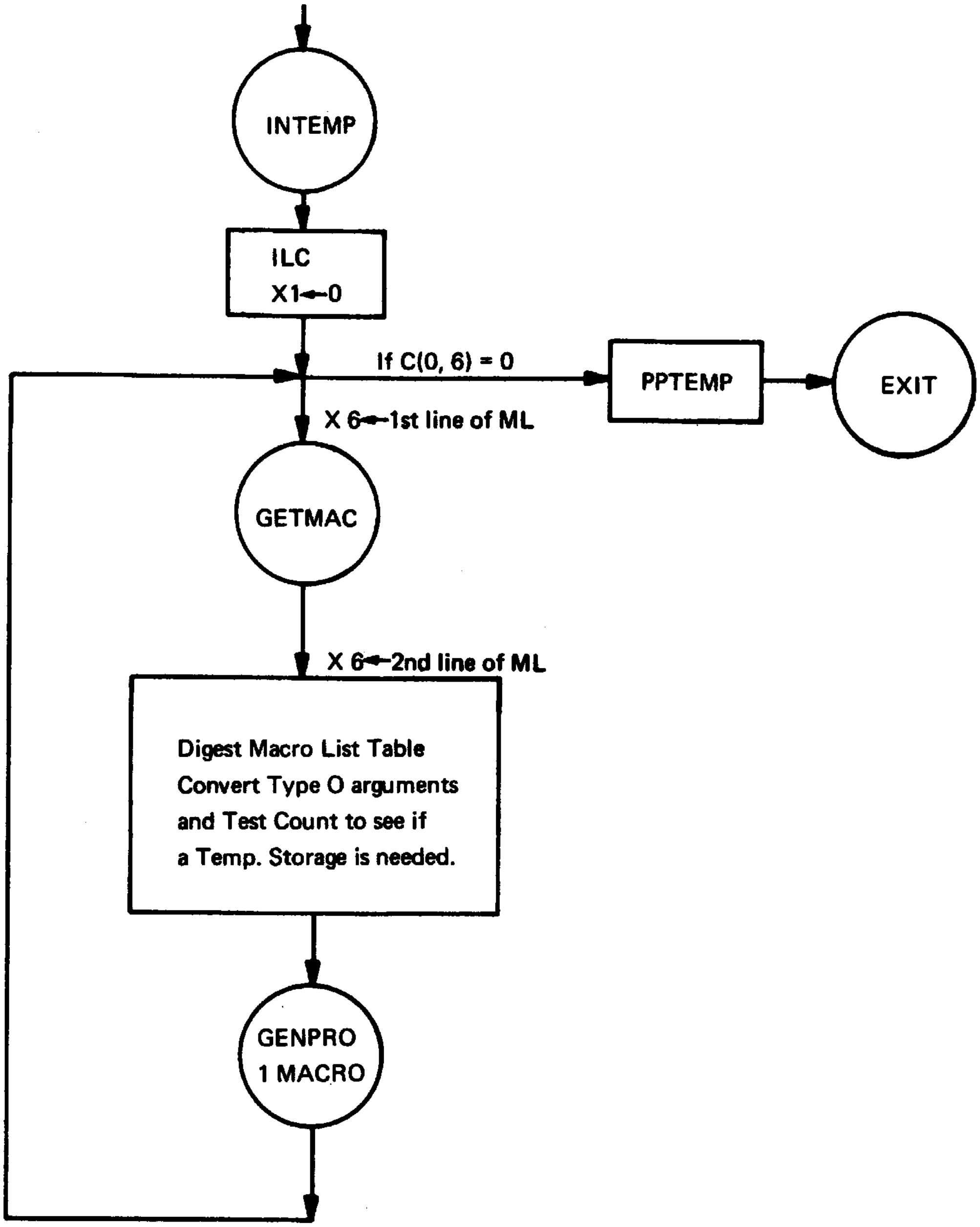


Figure E-1. Overall Flowchart for the Assembler

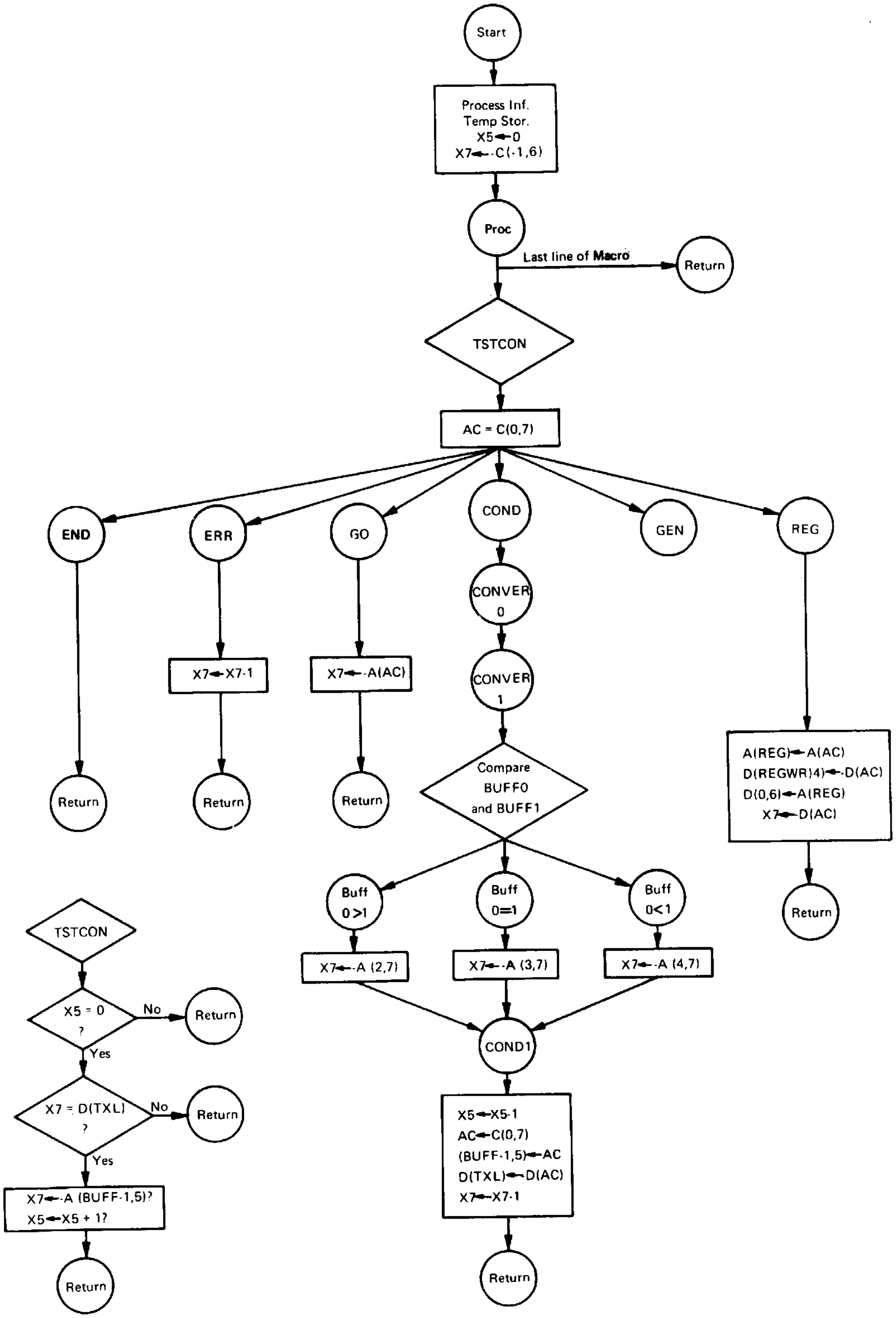


Figure E-2. Flowchart for Routine GENPRO

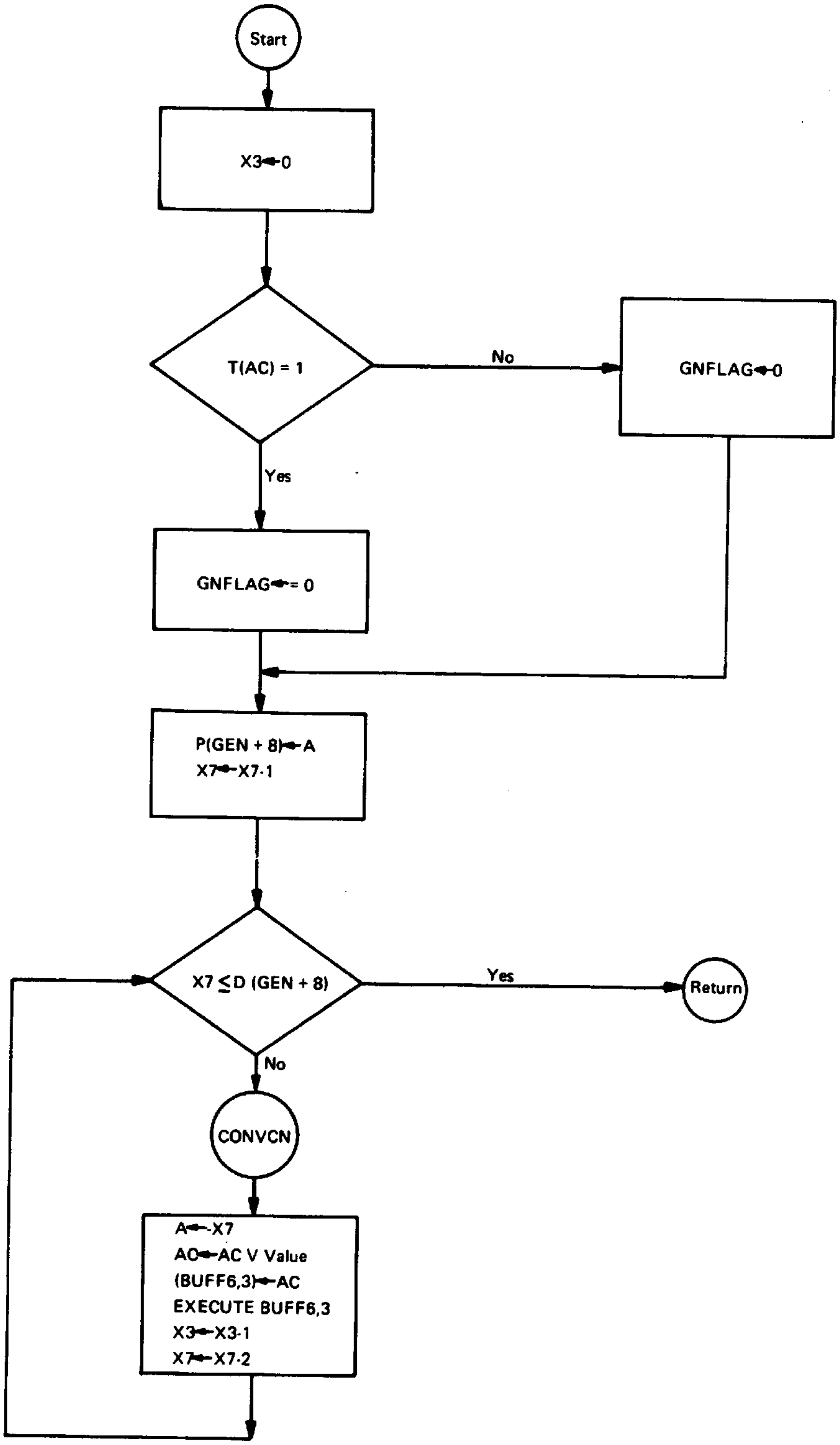


Figure E-3. Flowchart for Routine GEN

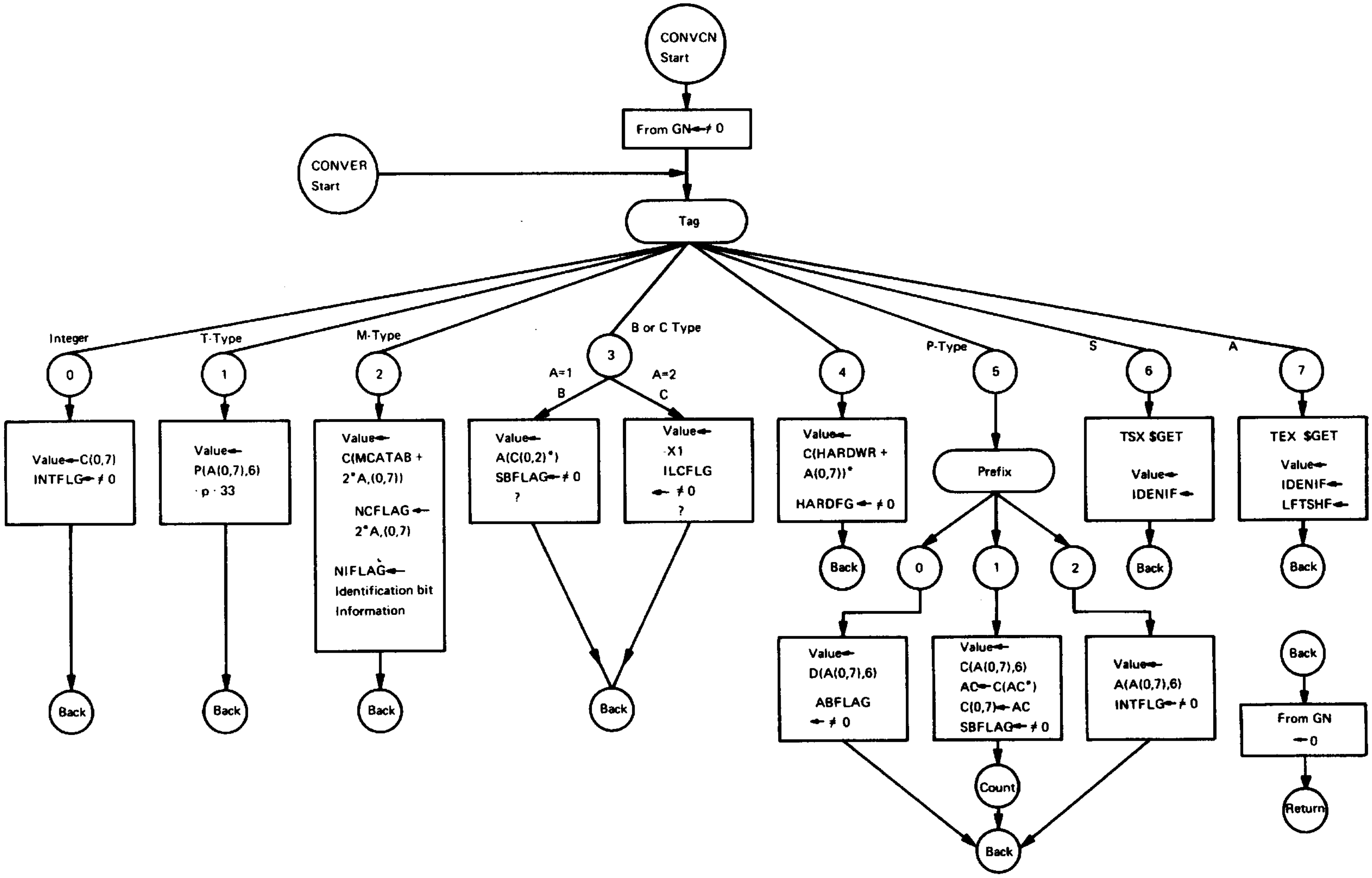

CONVCN Start
From GN←≠0
CONVER Start
Tag
Integer
T-Type
M-Type
B or C Type
P-Type
S
A
0
1
2
3
4
5
6
7
Value←C(0,7) INTFLG←≠0
Value← P(A(0,7),6) ·p·33
Value← C(MCATAB + 2*A,(0,7)) NCFLAG← 2*A,(0,7) NIFLAG← Identification bit Information
A=1 B
A=2 C
Value← A(C(0,2)*) SBFLAG←≠0 ?
Value← X1 ILCFLG ←≠0 ?
Value← C(HARDWR + A(0,7))* HARDFG←≠0
Prefix
TSX $GET Value← IDENIF←
TEX $GET Value← IDENIF← LFTSHF←
Back
Value← D(A(0,7),6) ABFLAG ←≠0
Value← C(A(0,7),6) AC←C(AC*) C(0,7)←AC SBFLAG←≠0
Value← A(A(0,7),6) INTFLG←≠0
Count
From GN ←0
Return

Figure E-4. Flowchart for Routine CONVER and CONVCN

APPENDIX F

SAMPLE SOURCE LANGUAGE AND CONTROL LANGUAGE COMPILATION STATEMENTS

F.1 BNF SYNTAX OF SAMPLE SOURCE LANGUAGE

⟨program⟩	: : =	START [⟨statement⟩]$_0^\infty$ STOP
⟨statement⟩	: : =	[⟨label⟩\$]$_0^1$ [⟨assignment statement⟩ |⟨transfer statement⟩ | ⟨conditional statement⟩]$_1^1$
⟨assignment statement⟩	: : =	⟨identifier⟩ = ⟨arithmetic expression⟩
⟨transfer statement⟩	: : =	GOTO ⟨identifier⟩

⟨conditional statement⟩	: : =	IF ⟨arith. exp.⟩ ⟨rel. op.⟩ ⟨arith. exp.⟩
		THEN [⟨statement⟩]$_0^\infty$.

⟨arith. exp.⟩	: : =	⟨factor⟩[⟨add. op.⟩⟨factor⟩]$_0^\infty$
⟨factor⟩	: : =	⟨term⟩ [⟨mult. op.⟩ ⟨term⟩]$_0^\infty$
⟨term⟩	: : =	⟨integer⟩|⟨identifier⟩|(⟨arith. exp.⟩
⟨identifier⟩	: : =	[⟨characters⟩]$_1^\infty$
⟨integer⟩	: : =	[⟨digit⟩]$_1^\infty$
⟨character⟩	: : =	A | B | . . . | Z
⟨digit⟩	: : =	0 | 1 | . . . | 9
⟨rel. op.⟩	: : =	EQ | NE | GT | LT | LE | GE
⟨mult. op.⟩	: : =	* | /
⟨add. op.⟩	: : =	+ |—

F.2 THE OVERALL SYNTAX OF MARKSTRAN

⟨Markstran program⟩	: : =	MARKSTRAN START ⟨declaration statement block⟩ ⟨syntactic analysis block⟩ MARKSTRAN STOP
⟨declaration statement block⟩	: : =	LEXICAL DECLARE START [⟨LEXICAL declaration⟩]$_0^\infty$ STOP TEST DECLARE START [⟨TEST declaration⟩]$_0^\infty$ STOP STACK DECLARE START [⟨STACK declaration⟩]$_0^\infty$ STOP

⟨syntactic analysis block⟩ : : = [⟨ATAB statement⟩]$_0^\infty$ | ⟨TTAB statement⟩ [⟨ATAB statement⟩]$_0^\infty$

F.3 THE MARKSTRAN PROGRAM FOR SYNTACTIC ANALYSIS OF THE SAMPLE SOURCE LANGUAGE

```
MARKSTRAN START
LEXICAL DECLARE START
  TS = +/-/*/–/ EQ/NE/ST/LT/LE/GE/ = /IF/THEN/./$/START/STOP/GOTO/(/)//BLANK
  IDEN = ALPHABETIC // BLANK
  LIT = INTEGER/.INTEGER/INTEGER./INTEGER.INTEGER//BLANK
STOP
TEST DECLARE START
  PROPERTY TEST   TPROP = ADDOP(+,-), MULOP(*,-), RELOP(E0,NE,GT,LT,LE,GE)
  VALUE TEST      PADD = TERM, FACT, AEXP
STOP
STACK DECLARE START
  STACKS  STACKL(20), STACKM(2000)
STOP
PREDICATE START  EQ2
  IF PREOVR EQ-2  THEN SETTRLE(1).  RETURN.  ELSE SETTRUE(0).,  RETURN.END
STOP
INITIAL$  DO(SCAN).  TEST  (X1)
SCAN$       IF LEXICAL (TS), LEXICAL(LIT), LEXICAL(IDEN)
            THEN   STORE(CURSYM).   RETURN
            ELSE  PRINT  COMM(ILLEGAL LEXICAL TEST).  ERROR EXIT.  END
X1$         // START /// LOAD.  DO(SCAN).   TEST (S1)
            OTHERWISE  ERR$  PRINT  COMM(ILLEGAL PROGRAM)
                                     ERROR EXIT
S1$         // IF ///    LOAD.   DO(AE).   TEST(IF1).
            // TO ///  DO(SCAN).   TEST(TO1).
            // STOP ///      TEST(X2).
            // IDENT ///    LOAD.   DO(SCAN).    TEST(ID1).
            OTHERWISE    TO(ERR).
```

```
X2$   / START //      PRINT  COMM(SYNTACTIC ANALYSIS FINISHED).
                      EXCISE(1).   HALT.
     OTHERWISE        TO(ERR).
ID1$   // $ ///       X = SUSE(STACKTOP(0)).
                      IF    X EQ 0    THEN
                           SUSE(STACKTOP(0)) = 1.
                           SLABEL(STACKTOP(0)) = 1.
                           TO(ID1A).
                      END
                      IF    EQ2(X)     THEN
IDERR$                PRINT COMM(ILLEGAL USE OF IDENT).
                      ERROR EXIT.
                      END
ID1A$                 MOVE(STACKM,LABEL,STACKQ).
                      DO(SCAN).    TEST(S1).
// = ///              IF SUSE(STACKTOP(0)) EQ 1  THEN  TO(IDERR).    END
                      SUSE(STACKTOP(0)) = 2.
                      LOAD.    DO(AE).
                      MOVE(STACKM,=,STACKTOP(2),  STACKTOP(0)).
                      EXCISE(3).    TEST(END1).
      OTHERWISE       TO(ERR).
TO1$  // IDENT ///    IF   EQ2(SUSE(CURSYM))     THEN   TO(IDERR).     END
                      SUSE(CURSYM) = 1.
                      MOVE(STACKM,TO,CURSYM).
                      DO(SCAN).    TEST(END1).
     OTHERWISE        TO(ERR).
IF1$  // RELOP ///    LOAD.    DO(AE).
                      MOVE(STACKM,STACKTOP(1),STACKTOP(2),STACKTOP(0)).
                      STACKL = PPTR(LMOVE).    EXCISE(3).    TEST(IF2)
     OTHERWISE        TO(ERR).
IF2$   / IF // THEN ///
                      DO(SCAN).    TEST(S1).
     OTHERWISE        TO(ERR).
```

```
END1$  // . ///        DO(SCAN).    TEST(END2).
      OTHERWISE        PRINT COMM(ILLEGAL STATEMENT TERMINATION).
                       ERROR EXIT.
END2$  / IF  //.///    PPTR(STAB(STACKL)) = PPTR(LMOVE)+1.
                       EXCISE(1).   DO(SCAN).
                       TEST(END2).
      OTHERWISE        TEST(S1).
AE$                    DO(SCAN).    TEST(AE1).
AE1$  // + ///         DO(SCAN).    TEST(AE1A).
        // - ///       STACKQ = UN-.
                       DO(SCAN).    TEST(AE1A).
AE1A$  // IDENT ///
          // NUM ///   LOAD.     SET(TERM).
                       DO(SCAN).    TEST(AE2).
          // ( ///     LOAD.   SCAN.    TEST(AE1).
      OTHERWISE        TO(ERR).
AE2$  // MULOP ///     LOAD.   SCAN.    TEST(AE1A).
AE2X$  /TERM MULOP TERM//
                       MOVE(STACKM,STACKTOP(1),STACKTOP(2),STACKTOP(0)).
                       EXCISE(3).     STACKQ = POINT(FMOVE).
                       SET(TERM);     TEST(AE2X).
        / TERM //      SET(FACT).      TEST(AE3).
         OTHERWISE     TO(ERR).
AE3$  // ADDOP ///
                       LOAD.     SCAN.     TEST(AE1A).
AE3X$  / FACT ADDOP FACT  //
                       MOVE(STACKM,STACKTOP(1),STACKTOP(2),STACKTOP(0)).
                       EXCISE(3).    STACKQ = POINT(FMOVE).
                       SET(FACT).    TEST(AE3X).
          // UN-FACT //
                       MOVE(STACKM=STACKTOP(1),STACKTOP(0)).
                       EXCISE(2).    STACKQ = POINT(FMOVE).
                       SET(AEXP).    TEST(AE4).
```

```
AE30$   / FACT //    SET(AEXP).    TEST(AE4).
    OTHERWISE        TO(ERR).
AE4$   / ( AEXP // ) ///
    PAREN$           STACKTOP(1) = STACKQ.
    PAREN1$          SET(TERM).    TEST(AE2).
    / ( AEXP //      PRINT COMM(MISSING RIGHT PARENTHESIS).
                     TO(PARENT1).
    / AEXP // ) ///  PRINT COMM(MISSING LEFT PARENTHESIS
                     TO(PAREN1).
    / AEXP //        RETURN.
    OTHERWISE        TO(ERR).
```

F.3 COMMENTS ON THE MARKSTRAN PROGRAM

It is assumed that SUSE and SLABEL have been declared in the Table Processor to be symbolic names for fields of table IDENT. The values in these fields are interpreted in the following way by the MARKSTRAN program:

SUSE field - 0 - ident not yet used
1 - used as label
2 - used as variable

SLABEL field - examined only if SUSE = 1
0 - location not yet defined
1 - location defined

Note that the result macro-strings are placed in STACKM.

If a pointer P points to another pointer (i.e., PINTP(P)= 0), the STAB(PPTR(P)) accesses this pointer.

F.4 TABLE DECLARATION AND MANIPULATION STATEMENTS FOR THE SAMPLE SOURCE LANGUAGE

```
LITTAB  (100)  NOSORT      36(1,0), 36(2,0), 15(3,21)
SYMTAB (200)   SORT  (1,1) 36, 3, 15, 15, 15
PROCESS LITTAB
MEMORY INITIALIZATION RELOCA = 144
```

```
(LITTAB, 3) = RELOCA
INDEX RELOCA 1
PROCESS SYMTAB
(SYMTAB, 5) = RELOCA
INDEX RELOCA 1
```

F.5 EXAMPLE MACRO INTERPRETATION STATEMENTS FOR THE SAMPLE SOURCE LANGUAGE

Consider the sample source language statement:

```
IF X EQ 3.14 THEN GOTO ALPHA
```

The macro instructions for this statement might be as follows:

```
EQ (X,3.14)
TO (ALPHA)
TO (C + 1)
```

Here EQ is a macro that causes a transfer of control to 1) the following macro if its arguments are equal or 2) the second following macro if its arguments are not equal. TO is a macro for unconditional transfer of control. The machine code for these macros might be as follows:

```
CLA   X
SUB   3.14          machine code for EQ (X  3.14)
TNZ   * + 2
TRA   ALPHA         machine code for TO (ALPHA)
TRA   * + 1         machine code for TO (next macro)
```

The macro interpretation statements for the macros EQ and TO might be as follows (assume M101, M106, M94, and M32 designate the machine code table entries for CLA, SUB, TNZ, and TRA respectively):

```
TEMP 0              -no temporary storage is needed for EQ
RL   1              -result of macro is left in AC
GEN  (M101) (ARG1)  -generates code for CLAARG1
GEN  (M106) (ARG2)  -generates code for SUBARG2
GEN  (M94) (C + 2)  -generates code for TNZ *+2; (C is the instruction
END                  location counter)
TEMP 0              -no temporary storage is needed for TO
GEN  (M32) (ARG1)   -generate code for TRA    ARG1
END
```

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE

Design and Implementation of a Table-Driven Compiler System

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical Report, Electrical Engineering, September 1965 to April 1967

5. AUTHOR(S) *(Last name, first name, initial)*

Liu, Chung L., Gabriel D. Chang, and Richard E. Marks

6. REPORT DATE
July 1967

7a. TOTAL NO. OF PAGES
90

7b. NO. OF REFS
6

8a. CONTRACT OR GRANT NO.
Office of Naval Research, Nonr-4102(01)

b. PROJECT NO.
NR 048-189

c.
RR 003-09-01

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
MAC-TR-42

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES
None

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency
3D-200 Pentagon
Washington, D. C. 20301

13. ABSTRACT Our goal is to provide users of the table-driven compiler system with an environment within which they can freely design and produce compilers. The primary design criterion is generality so that the users can define a large class of input languages oriented toward any kind of problem-solving purposes, and can also define a large class of object programs to be executed on different computer systems. Therefore, in our system we do not limit the users to specific ways of doing syntactic analysis, or doing storage allocation, or producing binary programs of a specific format for a particular computer system. What we provide are mechanisms that are general enough for whichever way a user desires to build his compiler. The table-driven compiler system consists of a base program and two fixed higher-level languages -- the Table Declaration and Manipulation Language and the Macro Interpretation Language -- together with corresponding translators to generate control tables according to user specifications. A third higher-level language -- the Syntax Defining Language -- and its corresponding translator are also needed. For the generality and flexibility we try to attain, less consideration is placed on efficiency.

14. KEY WORDS

Compiler generators
Computer
Machine-aided cognition
Multiple-access computers
On-line computers
Real-time computers
Syntax-directed compilers
Table-driven compilers
Time-shared computers

UNCLASSIFIED
Security Classification