A LOW-LEVEL LANGUAGE
FOR USE ON THE
MOS 6502 MICROCOMPUTER

By

MARY ANN GERTRUDE LAWRY, B.Sc.

A Project
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University

April 1981

MASTER OF SCIENCE (1981)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: A Low-Level Language For Use On The MOS 6502 Microcomputer

AUTHOR: Mary Ann Gertrude Lawry B.Sc. (Biochemistry, McMaster University)

SUPERVISOR: Dr. N.S. Solntseff

NUMBER OF PAGES: x, 139

ABSTRACT

A low-level language, GRASSHOPPER, was developed for use as a systems programming language on the MOS 6502 microcomputer. GRASSHOPPER was designed as an alternative to assembly language for systems programming, and its use requires some knowledge of the MOS 6502 hardware. To facilitate the writing of correct and readable programs, GRASSHOPPER includes three control structures used in the higher level structured languages, and provides five distinct data types.

ACKNOWLEDGMENTS

I would like to thank my supervisor Dr. Nick Solntseff for suggesting this challenging project, and for his direction over the past two years.

I would like to express my appreciation to Chris Bryce for his frequent assistance with the technical aspects of this project.

I would like to thank my first reader, Dr. Derick Wood for his valuable comments during the writing of this report.

Finally, I would like to thank the members of the McMaster University Unit for Computer Science for a very interesting and educational three years.

T A B L E   O F   C O N T E N T S

T A B L E S

F I G U R E S

CHAPTER 1

INTRODUCTION

1.1 The Project

This project is divided into two parts: the design of a structured, low-level language, GRASSHOPPER, for use on the MOS 6502 microcomputer; and the development of a GRASSHOPPER compiler which produces MOS 6502 assembly language.

The purpose of the development of GRASSHOPPER has been to provide an alternative to assembly language programming on the 6502. Assembly language is still a popular choice for systems programming for micro-computers because of its flexibility and efficiency in comparison to high-level languages. However the disadvantages of assembly language programming are well documented for mini and micro computers, namely, that correctness is more difficult to guarantee and that readability is poor. In particular the writing of structured and intelligible programs is almost impossible to achieve. On the other hand, high-level languages are designed to be readable, to encourage the writing of correct and structured programs, but are usually not easily translated into highly efficient code. In this project I have attempted to design a low-level language that combines the advantages of assembly language and high-level languages while substantially minimizing the disadvantages.

To be attractive to the user there must be a minimal loss of flexibility in programming. For this reason, GRASSHOPPER provides some access to the processor status register, to the accumulator and to the X and Y registers. All core addressing methods used by the MOS 6502 are available to GRASSHOPPER. The ability to embed assembly language code is provided for those cases when the desired code cannot be written in GRASSHOPPER.

Another criterion of my language design is that the writing of readable and correct code should be facilitated. Control statements have therefore been provided which help the reader to understand the logical structure of the GRASSHOPPER programs. These statements have been designed to be understandable with minimal explanation so they can be learned quickly and remembered easily.

Finally, the language design has been strongly influenced by consideration of ease of parsing and compilation.

The overall design of the GRASSHOPPER compiler reflects the goal of not doing anything the assembler already does adequately. The principal design criteria are:

1/ Economy of space. This is of major importance when working with a micro-computer. In designing the structured statements and their translation I have attempted to produce object code with little increase in size over what would have been produced if the source code had been written in assembly language to start with.

2/ Speed of translation. This was a consideration in the design of the symbol tables, but was otherwise of much lower priority then space.

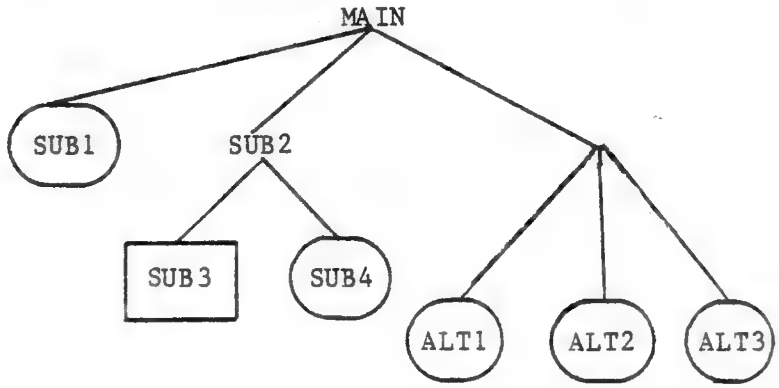
3/ Expandability and adaptability. These are important considerations because it should be possible to adapt GRASSHOPPER to future needs. Where practical, I have applied the principles of modularity and structured design, and have made the assembler source code for the compiler as easy to read as possible. Frequent use has been made of constants to make certain changes easier and to enhance readability.

1.2 The Report

Chapter 2 of this report introduces the language, GRASSHOPPER, discussing its data types and identifier declaration, its statements and constructs, and finally precisely describing its syntax using syntax graphs. In Chapter 3, the semantics of GRASSHOPPER is described in terms of the MOS 6502 assembly language, which is the object code of the compiler. In Chapter 4 the GRASSHOPPER compiler is described in general, including a discussion of error detection and diagnostics. Chapters 5, 6 and 7 discuss three major areas in the implementation: I/O and file management; the lexical scan; and the format and generation of the object code. Finally, Chapter 8 summarizes the testing of the translator, discusses the usability of GRASSHOPPER, and presents some ideas for further language developement.

Throughout this report I have illustrated the structure of sections of the compiler using subroutine maps. Each node of such a map represents a named routine. Each leaf will be enclosed in an ellipse if it can call no other routines, or in a rectangle if it may. In many cases a leaf node appearing on one tree will be a root node in another tree. Appendix B indicates where each referenced routine is described. The text accompanying a subroutine map will indicate under what circumstances each root node routine may be called.

FIGURE 1.1 Example Of A Subroutine Map



In the sample map shown in Figure 1.1 the root routine MAIN may call: SUB1 which calls no more routines; SUB2 which may call SUB3 and SUB4; and ALT1, ALT2 or ALT3 which are called using indirect addressing where the address is stored in a pre-set location. SUB3 may call one or more other routines, not listed here.

All algorithm descriptions are given in pseudo-GRASSHOPPER. The extra features of this notation include; the procedure header and parameter passing capability in subroutine calls; complex expressions as conditions in the IF construct; and the WITH construct.

The WITH construct is similar to that found in Pascal, but is used to access individual bits of a byte, instead of fields of a record. In conjunction with this, the identifiers BIT0, BIT1,..,BIT7 will represent boolean variables. BIT i will be true when the i'th bit of the identified value is 1 and false otherwise. Thus:

```
with CHRFLG do
   if BIT7 then statement list 1 endif;
   if not BIT5 then statement list 2 endif;
endwith
```

may be described as follows:

```
if the seventh bit of CHRFLG = 1
   then  execute statement list 1  endif
if the fifth bit of CHRFLG = 0
   then  execute statement list 2  endif
```

CHAPTER 2

THE LANGUAGE GRASSHOPPER: SYNTAX

GRASSHOPPER has been written as an extension to the existing assembler with little attempt made to add to the primitive operations and data types already provided by the hardware.

Currently the scope rules for a GRASSHOPPER program are those of an assembly language program, that is all identifiers declared in a program are global to that program. Some structure has been introduced in that all identifiers must be declared at the beginning of the program.

Sequence control statements and conditional statements have been provided to aid in designing and understanding the logical structure of a GRASSHOPPER program. Simple one-to-one translatable commands have been included to allow the use of subroutine calls but parameter lists and more elaborate subprogramming capabilities have not been implemented. This is a prime area for extending GRASSHOPPER.

A GRASSHOPPER program consists of:

1/ A header line giving the starting address for the executable code, for example:

program ADDRESS;

where ADDRESS is an integer in the range $0..2^{16}-1$.

2/ A declaration section in which all identifiers used must be listed and assigned a type. Constants must be given values, and variables may be given initial values. Line labels are not declared;

3/ The key word **begin**;

4/ The statement list, which consists of the executable statements of the program;

5/ The key word **end**.

Several general aspects of GRASSHOPPER are briefly commented on here. Declarations and Data Types will be discussed in Section 2.1. Section 2.2 will describe the data manipulation and comparison operations. Sequence control will be discussed in Section 2.3. Statement delimiters will be discussed in Section 2.4 and Section 2.5 describes the use of line labels. Finally a summary of the syntax of GRASSHOPPER will be given in Section 2.6.

Reserved words are not extensively used, instead, key words are delimited by single quotes or are in lower case letters. The letters "A", "X", "Y", "S" and "P" are reserved as variable names by the OSI resident assembler/editor. "S" and "P" are not directly accessible using GRASSHOPPER so their use is illegal. "A", "X" and "Y" represent the accumulator and the X and Y registers respectively. In addition, "X", as the first letter in an identifier with two or more characters, is reserved by the compiler for system line labels, [Sections 3.4 and 7.2].

Identifiers are made up of digits and upper case letters and must start with a letter. Only the first six characters will be used by the compiler so these must uniquely define the identifier.

Numerical literals may be base 2, [illegible] or 16, and each value must be immediately preceded by the character %, @ or $ respectively, to indicate the number base. Thus %1010, @12 and $0A all equal 10, (base ten). Note that the letters used in hexadecimal numbers must be in upper case.

Character and string constants must begin and end with double quotes, ("), for example: "This is a string", and the strings may not exceed 2^8 characters in length.

Statements are separated by statement delimiters or by semi-colons. This will be further discussed in Section 2.3. A single statement may extend over one or more lines, and more than one statement may occur on a line. A single item within a statement, such as an identifier or key word, may not be split between two lines.

Comments begin and end with "!", may extend over several lines, and may be inserted between items in a statement:

```
if ZERO then  ! empty list !  $00 -> FOUND;...
```

Assembly language inserts may be placed anywhere a statement may. Each insert is enclosed in square brackets and is copied unchanged into the compiler output. In the following example, an insert is used to comment the object code:

```
if LENGTH > MAX then  [;        overflow ]...
```

2.1 Data Types, And Identifier Declaration

The smallest addressable piece of information on the MOS 6502 is the 8-bit byte. Addressing one byte requires a 16-bit address, unless it is on the zero page, in which case an 8-bit address is required. The data types available in GRASSHOPPER reflect these facts. In this section I will discuss the declaration of identifiers, and the addressing of data. The kinds of operations which may be performed are discussed in Section 2.2.

There are two primitive data types: the Byte and the Word. The type Byte corresponds to an 8-bit computer byte and contains a subrange $0..2^8-1$ of the integers. ASCII characters are considered to be a subset of Byte, from 00..$7F. The type Word corresponds to 16 bits, or 2 computer bytes, and contains the subrange $0..2^{16}-1$ of the integers.

In addition to these two primitive types there is one structured data type, the Array. Arrays are one dimensional with up to 2^8 elements; the base type is always Byte.

Except for line labels, each identifier in a GRASSHOPPER program must be declared. The declaration section of a program consists of a list of declaration statements, each of which begins with a key word identifying the declaration type followed by a list of identifiers separated by commas and ending with a semi-colon. For example:

byte NAME1, NAME2, NAME3;

There are six declaration types, which can be grouped into three classes, as in Table 2.1

TABLE 2.1: The Three Classes of Declaration Statements

Declaration Class	Declaration Type
Constant	**constant**
Variable	**byte** **word** **array** **zeropage**
Read-Only Variable	**condition**

2.1.1 Constant Identifers

Each constant identifer must be assigned a value when it is declared. The value assigned may either be a literal or a previously defined constant identifier, for example:

constant TRACK = $65, OTHER = TRACK;

Byte and Word constant identifiers may be declared in the same declaration statement, and type assignment will depend on the size of the literal assigned. Table 2.2 gives examples of constant declarations and of the type and values that result from these declarations.

A constant of type Word followed by the selector **lo** or **hi** will specify the least or most significant part of the value respectively.

TABLE 2.2: Examples of Constant Declarations

Declaration	Type	Value
ONE = $12	BYTE	$12
TWO = $0012	WORD	$0012
THREE = $1234	WORD	$1234
FOUR = %11111111	BYTE	$FF
FIVE = %111111111111	WORD	$0FFF
SIX = ONE	BYTE	$12
SEVEN = TWO	WORD	$0012
EIGHT = THREE **hi**	BYTE	$12
NINE = THREE **lo**	BYTE	$34
TEN = "A"	BYTE	$41

Preceding a constant identifier or numeric literal with the key word **loc** in any expression indicates that the value given is to be interpreted as the core address of the operand. For example, assuming that SIZE has been declared as a Byte variable:

loc $5A -> SIZE;

will assign the value found at the address $5A to the variable SIZE, while:

$5A -> SIZE;

will assign the value $5A to this variable. In the first case **loc** $5A is the absolute address of the operand, and in the second case $5A is the immediate operand.

2.1.2 Byte and Array Variables

Byte and Array variables may be either local or external. Local variables are located in the data space which precedes the executable code in the object program, their absolute addresses in core need not be known by the programmer. External variables are declared within the Byte or Array declaration in the following way:

NAME **at** ADDRESS

where NAME is the variable identifier and ADDRESS the absolute address in core of the variable. The address must be given as a numeric literal or a predefined constant. These variables may not be initialized in the declarations, and the dimension of an external array is not declared. Otherwise, there is no difference between the use of an external variable and the use of a local variable of the same type.

A **byte** declaration statement may be used to declare local and external variables of type Byte. Literals or pre-defined Byte constants may be used to initialize the local variables but initialization is not required. Thus:

byte COUNT, MAX = $2A, FLAG **at** $35B0;

declares COUNT and MAX to be local Byte variables and stores the value $2A in MAX. FLAG is declared as an external Byte variable whose absolute address is $35B0.

An **array** declaration is used to declare all local and external variables of type Array. Arrays are one dimensional, are indexed upwards from zero and each element is of the type Byte. The largest possible array has 2^8 elements. The maximum index of a local array is declared as a numerical literal in parentheses immediately after the identifier. If all elements of an array are initialized, declaration of the length is optional, but the parentheses are still required. Items used to initialize an array are enclosed in parentheses:

```
NAME(LENGTH) = (ITEM 1, ITEM 2...)
```

and may include Byte literals, pre-declared Byte constants, and string constants. Thus, in this example:

```
array   ARRAY1 at $4900, ARRAY2($0F),
        ARRAY3($05) = ("HELLO", $05),
        ARRAY4($05) = ($1, $2),
        ARRAY5() = ("STRING");
```

ARRAY1 is external, ARRAY2 is local with $10 elements indexed from $00 to $0F and is not initialized. ARRAY3 and ARRAY4 both have $06 elements, indexed from $00 to $05. ARRAY3 is fully initialized but only the first two elements of ARRAY4 are. ARRAY5 is fully initialized with no declaration of length.

The individual elements of an array are of type Byte and are addressed within the array using the X or Y register:

```
ARRAY1,X -> ARRAY2,Y;
```

2.1.3 Word Variables

Local and external Word variables are declared with a **word** declaration statement. Local Word variables may be given initial values in the declarations using numerical literals or constant identifiers, but not character constants. External Word variables are declared in the same way that external Byte and Array variables are, [Section 2.1.2].

In the current implementation of GRASSHOPPER, the operations which are outlined in Section 2.2.2 may not be used on operands of type Word. Instead, the least and most significant bytes of Word variables and constants may be addressed by using the selectors **lo** and **hi**, respectively. The high or low part of a Word variable may be used anywhere a Byte variable may. Thus, given the following declarations:

word NAME1, NAME2 = $4000; **byte** NAME3;

then one can write, for example:

$50 -> NAME2 **hi**; NAME3 -> NAME2 **lo**;

but not:

NAME1 -> NAME2; NAME3 -> NAME2;

2.1.4 <u>Zeropage Variables</u>

Zeropage variables are special variables located on the zero page, which are used for indirect addressing. These variables are declared as follows:

zeropage at $40, ZNAME1;

zeropage at $50, ZNAME2, ZNAME3, ZNAME4;

The value immediately following the key word **at** is the address of the first variable in the declaration statement. The address of each succeeding variable is obtained by incrementing the value of the address of the previous one by two. For example, the second declaration above is analogous to:

```
word  ZNAME2 at $50,  ZNAME3 at $52,
        ZNAME4 at $54;
```

There are two types of indirect addressing modes used for accessing data: indexed indirect which uses the X index register and indirect indexed which uses the Y index register. In both cases the key word **ind** is used as a prefix, for example:

```
ind ZNAME1,X  ->  ind ZNAME2,Y;
```

Both of these methods require locations on the zero-page in which sixteen bit addresses may be stored. The address is always stored with the low order byte first, followed by the high order byte.

For indexed indirect addressing, the Zeropage variable is an implied external array, located on the zero page. It contains a series of addresses, such that the n′th address begins at the displacement 2(n-1) in the array. The values stored in this array may be accessed using either the X or Y index register for absolute indexed addressing. Thus, given the above declarations:

```
$04 -> X;
$00 -> ZNAME1,X;     inc X;
$45 -> ZNAME1,X;
```

assigns the value $4500 as the third address stored in the Zeropage array ZNAME1. Then an operand whose address is stored in this array may be accessed using the X index register for indexed indirect addressing, Thus:

```
$04 -> X;
ind ZNAME1,X -> A;
```

results in the value found at address $4500 being loaded into the accumulator. In this example the effective address was stored in zero page locations $44 and $45.

For indirect indexed addressing of data, the Zeropage variable is a external word variable, located on the zero page. The value stored in this variable is accessed by absolute addressing, using the selectors **lo** and **hi** to address the least and most significant bytes, respectively. This value is the address in core of an implied array of operands. Thus, an operand within this implied array is accessed using the Y index register for indirect indexed addressing. For example, given the above declarations:

```
$04 -> Y;
$00 -> ZNAME2 lo;
$45 -> ZNAME2 hi;
ind ZNAME2,Y -> A;
```

results in the value found at address $4504 being loaded into the accumulator. In this example, the effective address was found by adding the Y register to the address stored in zero page locations $50 and $51.

Table 2.3 summarizes the addressing modes in which Zeropage variables are used.

TABLE 2.3: Uses of the Zeropage Addressing Constants

Operand	Addressing Mode
ZEROPAGE **hi**	Absolute Addressing
ZEROPAGE lo	Absolute Addressing
ZEROPAGE,X	Absolute Indexed Addressing
ZEROPAGE,Y	Absolute Indexed Addressing
ind ZEROPAGE,X	Indexed Indirect Addressing
ind ZEROPAGE,Y	Indirect Indexed Addressing

2.1.5 Condition Identifiers

These are special identifiers used to access the individual bits of the processor status register. Each bit of this register is used to indicate the status of a particular condition in the processor.

TABLE 2.4: Processor Status Register

Bit Number	Name	Significance
0	Carry	1 = True
1	Zero	1 = Zero Result
2	Interrupt	1 = Disable
3	Decimal Mode	1 = True
4	Break Command	1 = A BRK has been executed
5	-	None
6	Overflow	1 = True
7	Negative	1 = Negative Result

Four of the processor status bits or flags shown in Table 2.4 are accessible through GRASSHOPPER:

> The Carry flag, which is adjusted during each arithmetic operation. During addition it is set to one if there is a carry, and cleared to zero if there is not. During subtraction it is set for result greater than or equal to zero, and cleared otherwise, indicating a borrow.
>
> The Zero flag, which is set when any data transfer or calculation operation results in a zero, otherwise it is cleared.
>
> The Overflow flag, which is important during signed number arithmetic and is set whenever the result is outside the range of -127 to +127 decimal.
>
> The Negative flag will always be equal to the seventh bit of the result of any data transfer or calculation operation. This is important during signed number arithmetic since the seventh bit gives the sign.

The Condition identifiers represent Boolean variables which give information on the state of specific bits of the status register. Each must be declared to equal, or not equal one of the bits. For example:

```
condition  CARRY = $0,     NOTCARRY /= $0,
           ZERO  = $1,     NOTZERO  /= $1,
           OVER  = $6,     NOTOVER  /= $6,
           NEG   = $7,     NOTNEG   /= $7;
```

Thus, CARRY, which has been declared to be equal to bit zero, will be true when bit zero is set to one, and false when bit zero is cleared.

Conversely, NOTCARRY will be false when bit zero is set and true when bit zero is cleared. The use of these variables will be discussed in Section 2.3.1.

2.1.6 The Registers

The Accumulator and the X and Y index registers, referred to as A, X and Y respectively, are 8-bit registers in the MOS 6502 microcomputer. The accumulator will be involved in most data-manipulation operations even if it is not specifically referenced in the GRASSHOPPER code, and will be altered in almost all arithmetic, Boolean and comparison operations. The index registers are used in three modes of addressing: absolute indexed; indexed indirect and indirect indexed. These modes have already been discussed in Sections 2.1.2 and 2.1.4.

The registers may also be used as explicit operands in the same way that any byte variable may be with two restrictions which will be discussed in the next section.

2.2 Data Manipulation

2.2.1 The Operands

The preceding discussion has concentrated on the declaration of identifiers and their data types. For the remainder of this chapter an understanding of their use is more important, and for this purpose the word term will refer to any possible operand which represents one byte of information. Terms may be subdivided further into Byte Variables and Byte Constants.

Byte Variables	Byte Constants
loc CONST1	CONST1
loc CONST2	CONST2 **hi**
loc Numerical Literal <= $FFFF	CONST2 **lo**
VAR8	Numerical Literal <= $FF
ARRAY,X	Literal Character
ARRAY,Y	
VAR16 **hi**	
VAR16 **lo**	
ZEROPG,X	
ZEROPG,Y	
ind ZEROPG,X	
ind ZEROPG,Y	
ZEROPG **hi**	
ZEROPG **lo**	
Registers A, X and Y	

The registers A, X and Y have been included under Variables with the following stipulations. They may appear anywhere a declared byte variable may except that there may be no more then one register as operand on the left hand side of any assignment statement and that one register may not be directly compared to another. The reason for this is that in both cases translation would require the use of an extra holding variable since there are no assembler code instructions for performing these operations directly. Assigning to and using A, X and Y will be further discussed in the next section.

2.2.2 The Operations

The operators which may act on a term may be divided into the three classes: arithmetic operators; relational operators and prefix operators.

The arithmetic operators are: (+) plus; (-) minus; **and**; **or** and **eor**. These are hardware implemented operations, the operations of multiplication and division, which would require software implementation, are not available in GRASSHOPPER. These operators are used in the assignment statement, which is in this form:

Term	Arithmetic Operator	Term	->	Byte Variable;

The right assignment form was chosen since it is easier to translate than the left assignment form. There is only one operation allowed per statement. If several are required then as many assignment statements must be written with assignment to the accumulator in each of the intermediate steps.

The relational operators are: (=) equal to; (/=) not equal to; (<) less then; (<=) less then or equal to; (>) greater then; and (>=) greater then or equal to. These operators are used in the relational expression which is in this form:

Term	Relational Operator	Term

This expression may be used as the condition in the IF construct which will be described in Section 2.3.1.

The prefix operators are a special class of operators which only have one operand, and which proceed that operand. They are the increment, **inc** and the decrement, **dec**, which will increase or decrease the value of the operand by one, respectively. The statements in which they are used are of this form:

inc Byte Variable;

dec Byte Variable;

One additional data-manipulation statement is the comparison statement:

Term : Term;

which loads the first factor into the accumulator and compares it to the second. The important effect of this operation is on the status register. The use of this statement is further discussed in Section 2.3.1.

2.3 Sequence Control

There are four simple sequence control statements in GRASSHOPPER:

1/ **goto** Destination; which transfers control to the address in core indicated by Destination.

2/ **gosub** Destination; which calls the code at Destination as a subroutine.

3/ **return**; which causes a return from the subroutine.

4/ **exitloop**; which is used to exit the loop construct. This will be discussed in Section 2.3.3.

The destination of the GOTO and GOSUB statements may be given as: a word constant or literal representing the absolute address of the destination; or as an identifier which is used as a statement label elsewhere in the program. Statement labels are discussed in Section 2.5.

In addition to these there are three control structures: IF; CASE and LOOP, each of which has associated with it a specific terminating delimiter: **endif**; **endcase** and **endloop** respectively. This format was chosen over the **begin** ... **end** compound statement found in Pascal because the latter leads to a confusing multiplicity of **end**'s when statements are nested.

2.3.1 The IF Construct

The complete GRASSHOPPER IF construct is of the form:

if condition **then** statement list
orif condition **then** statement list
orif condition **then** statement list
\- - - - - - - - - - - - - -
else statement list **endif**

where a condition is either a relational expression as described in Section 2.2.2, or a Condition identifier, Section 2.1.5. No more then one statement list will be executed in an IF construct. If a condition is found to be true, its accompanying statement list will be executed, then control will be transferred to the next executable statement after the **endif**, no succeeding condition in the construct will be tested. The presence of one or more ORIF portions is optional. The ELSE portion is also optional and when it is present its statement list is executed only if none of the previously tested conditions is true. Thus, the simplest form of the IF construct is:

if condition **then** statement list **endif**

in which the statement list is executed if the condition is true, and nothing is executed if the condition is false.

Given the following declarations:

```
condition  NLESSTHAN = $00,    LESSTHAN /= $00,
           EQUAL     = $01,    NEQUAL   /= $01;
byte  AA, BB;
```

Table 2.5 shows equivalent conditions using relational expressions and Condition variables. The first is more explicit, but the second is preferred if more then one condition is to be tested on the same comparison.

TABLE 2.5: Equivalent Conditions; Relational Expressions and Condition variables where the IF is proceeded by the statement: AA : BB;

Relational Expression	condition variable
if AA = BB **then**	**if** EQUAL **then**
if AA /= BB **then**	**if** NEQUAL **then**
if AA < BB **then**	**if** LESSTHAN **then**
if AA >= BB **then**	**if** NLESSTHAN **then**
if AA > BB **then**	**if** NEQUAL **then** **if** NLESSTHAN **then**

Thus the following two segments of GRASSHOPPER code will be logically equivalent:

```
if AA = BB then gosub TRANSFER endif
```

and

```
AA : BB;

if EQUAL then gosub TRANSFER endif
```

and the same object code will be generated in both cases. This will be discussed further in Section 3.5.1.

The case of AA <= BB is not included in this table because there is no straight forward equivalent using Condition variables. If GRASSHOPPER is ever extended to allow more complex conditions for the IF statement, then:

```
if AA<= BB then.....
```

and

```
if EQUAL or LESSTHAN then.....
```

will be equivalent.

2.3.2 The CASE Construct

The CASE construct names a selector, which must be a Byte variable, followed by a series of statement lists each of which is guarded with one or more terms. The selector will be compared to each guard, if a match is found the statement list following that guard will be executed. Control will then be transferred to the next statement after the CASE construct. If more then one guard is equal to the selector, only the first one encountered will be matched, so only one statement list may be selected for execution. The CASE statement is of this form:

```
case  Byte Variable
   of Term (, Term): statement list
   of Term (, Term): statement list
          - - - - - - -
   other statement list
endcase
```

The OTHER portion is optional; if it is absent and there is no match made then there is no action. The translation of this construct results in the selector being loaded into the accumulator and compared to each guard. Since the X and Y registers may not be compared to the accumulator, these registers are not legal as case guards, however, they are legal as the selector. The accumulator may never be a case guard.

2.3.3 The LOOP Construct

There is only one looping constuct available in GRASSHOPPER, and it was designed to be as simple as possible. This allows the programmer to use one construct to write the iterative or conditional loop required for the problem. The format of the LOOP construct is:

loop statement list **endloop**

The EXITLOOP statement is used to exit the loop to the next executable statement after **endloop**. Generally, **exitloop** is part of a statement list in an IF or CASE construct. For example; where CURRNT and LENGTH are Byte variables:

```
loop
   if CURRNT = LENGTH then exitloop endif
   gosub TRANSFER;
endloop
```

2.4 Statement Delimiters

The structured statements: CASE, LOOP and IF each have a leading and a terminating delimiter. The IF and CASE statements also have intermediate delimiters which separate statement lists. The two other sets of delimiters which are important to the logical structure of a GRASSHOPPER program are: the square brackets which enclose assembler code inserts; and the key words **begin** and **end** which precede and terminate the complete statement list of a GRASSHOPPER program. Table 2.6 summarizes these statement delimiters.

TABLE 2.6: Statement Delimination in GRASSHOPPER

Structure	Leading Delimiter	Intermediate Delimiters	Terminating Delimiter
Verb list	**begin**		**end**
LOOP statement	**loop**		**endloop**
IF statement	**if**	**orif, else**	**endif**
CASE statement	**case**	**of, other**	**endcase**
Assembler code Insert	[		]

The simple statement types in GRASSHOPPER have been described in the preceding sections, they are:

1/ The Assignment Statement;

2/ The Comparison Statement;

3/ The Sequence Control Statements.

An intermediate or terminating statement delimiter must occur after each simple statement. Where the end of a statement list has been reached, the appropriate intermediate or terminating delimiter from table 2.6 is used. In all other cases the semi-colon is used as the terminating statement delimiter. An example of a GRASSHOPPER program is given in figure 2.1 which should clarify this. Note that a semi-colon which occurs at the end of a statement list and before one of the intermediate or termination delimiters in table 2.6 is ignored. This means that a semi-colon preceding an **else** is not illegal, just unnecessary. Similarily, a semi-colon following an a terminating delimiter such as **endif** is ignored.

2.5 Statement Labels

A statement label is a special identifier which is not listed in the declarations and which is used to reference an executable statement. The label and the statement are separated with a comma. Any executable statement in a GRASSHOPPER program may be prefixed by a label, except for the first statement in a statement list of a LOOP, IF or CASE control structure. For example, where COUNT is a Byte variable and LABEL a statement label:

```
if COUNT = $FF then exitloop endif
LABEL1, inc COUNT;
```

is legal, but:

```
if COUNT = $FF then exitloop
else LABEL1, inc COUNT endif
```

is illegal.

A statement label may be used for the destination part of a GOTO or GOSUB statement. Figure 2.1 illustrates the use of statement labels in a GRASSHOPPER program. Note that a section of code in the program has been labeled and used as a subroutine by another part of the program.

FIGURE 2.1: Example Of A GRASSHOPPER Program

```
'PROGRAM' GRSHOP $4000;
!            This is a simplified version of the program
       I used to store and retrieve the assembled version
       of GRASSHOPPER.                                   !

'CONSTANT'          DOS    = $2A51,      INWEKO = $2340,
                    OUTSTR = $2D73,      SEEKA  = $26BC,
                    LDREAD = $2B1A,      SAVE   = $2C3A,

                    CR = $0D,   LF = $0A,   TOTAL = $02;

'BYTE'          DSRNO 'AT' $265E,  DSRLEN 'AT' $265F,  SAVX;

'ARRAY'         ADDRESS() = ($91, $9D, $A9),
                TRACK  () = ($16, $18, $20);

'WORD'          ZADDRESS 'AT' $FF;

'BEGIN' $00 -> SAVX;

 ASK,    'GOSUB' OUTSTR;
[        .BYTE CR,LF,'1/LOAD       2/UPDATE ?',$00 ]
         'GOSUB' INWEKO;

         'CASE' A
         'OF'  $1: 'LOOP'           ! retrieve from disk !
                        'GOSUB' NEXT;
                        'GOSUB' LDREAD;
                    'ENDLOOP'

         'OF'  $2: 'LOOP'           ! save on disk !
                        'GOSUB' NEXT;
                         $0C -> DSRLEN;     'GOSUB' SAVE;
                    'ENDLOOP'

         'OTHER'   'GOTO' ASK;
         'ENDCASE'

```

```
! The following code is called as a subroutine to prepare
 for either a read from, or a write to disk            !

 NEXT,  SAVX -> X;
        'IF' X = TOTAL 'THEN'  'GOTO' DOS   ! end of run !
        'ELSE'
           ADDRESS,X -> ZADDRESS 'HI'; ! address in core !
           $00  ->  ZADDRESS 'LO';
           $01  ->  DSRNO;                 ! sector number !
           X + $1 -> SAVX;
           TRACK,X  ->  A;
           'GOSUB' SEEKA;     ! finds track, number in A !
        'ENDIF'
        'RETURN';
'END'.
```

2.6 Summary

This section presents a description of the syntax of GRASSHOPPER. The basic symbols will be defined first, followed by syntax graphs which will describe the constructs of GRASSHOPPER. The syntax graph notation used here is based on that used by Wirth: ([Wirth 76], pp 288-295). Each terminal symbol in a graph is enclosed in an ellipse, and each non terminal is enclosed in a rectangle.

Letter = Upper case letter or Lower case letter.

Upper case letter = A, B, C, D, E, F, G, H, I, J, K, L, M,
O, P, Q, R, S, T, U, V, W, X, Y, Z.

Lower case letter = a, b, c, d, e, f, g, h, i, j, k, l, m,
o, p, q, r, s, t, u, v, w, x, y, z.

Binary Digit = 0, 1

Octal Digit = 0, 1, 2, 3, 4, 5, 6, 7.

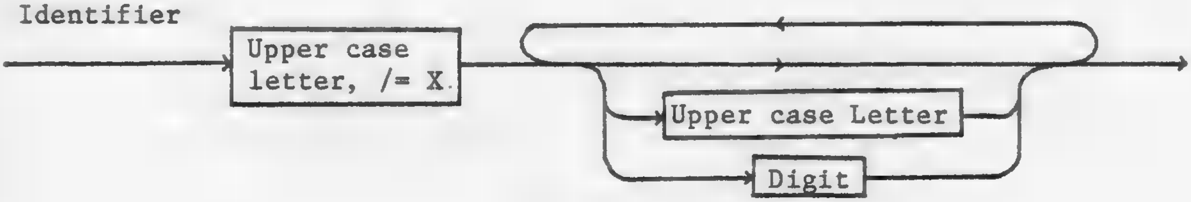
Hex Digit = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D,
E, F.

Arithmetic Operators = +, -, **and**, **or** and **eor**.

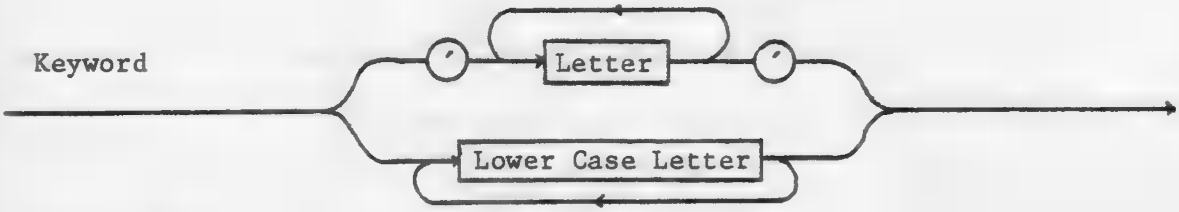
Relational Operators = <, <=, >, >=, =, /=.

Prefix Operators = **inc**, **dec**.

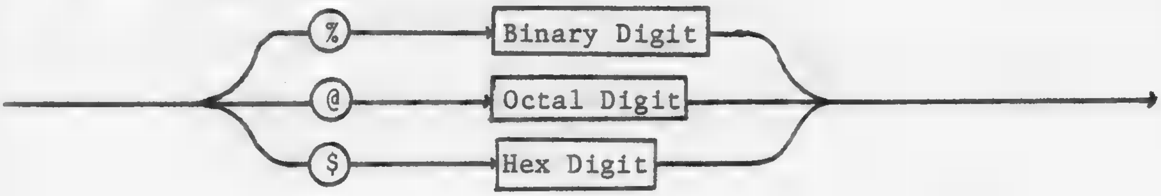
Identifier



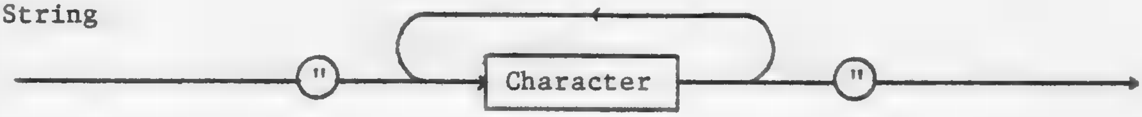
Keyword



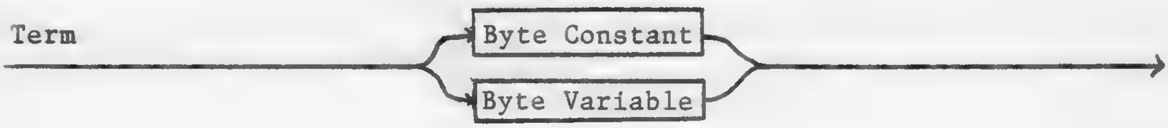
Number



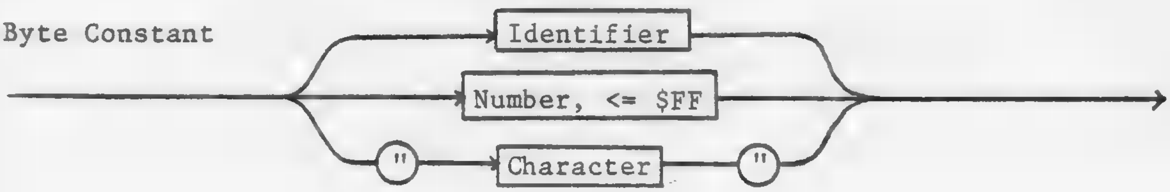
String



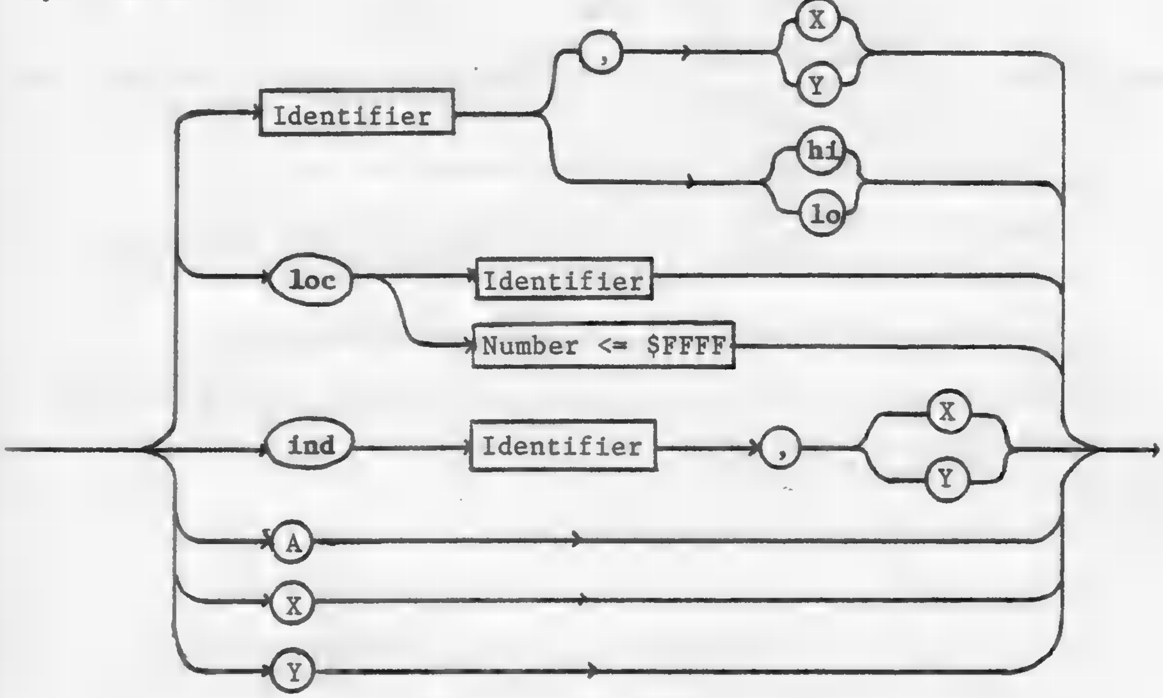
Term



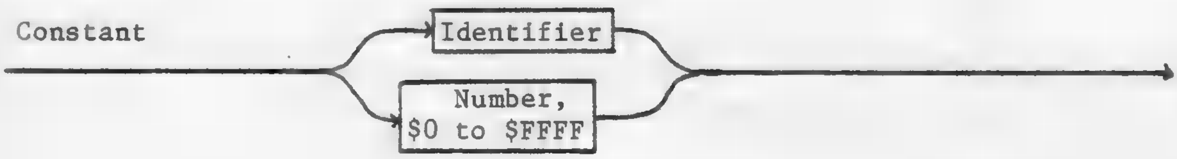
Byte Constant



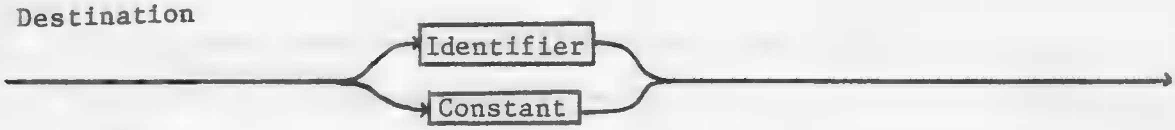
Byte Variable



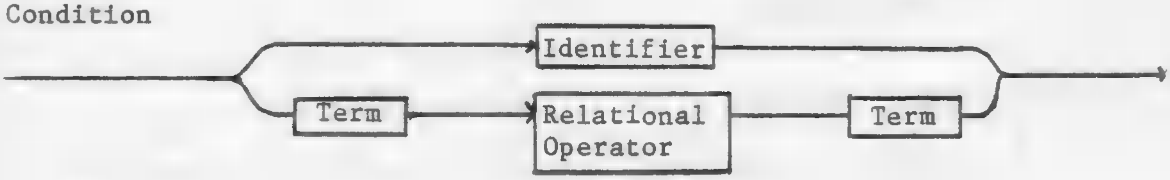
Constant



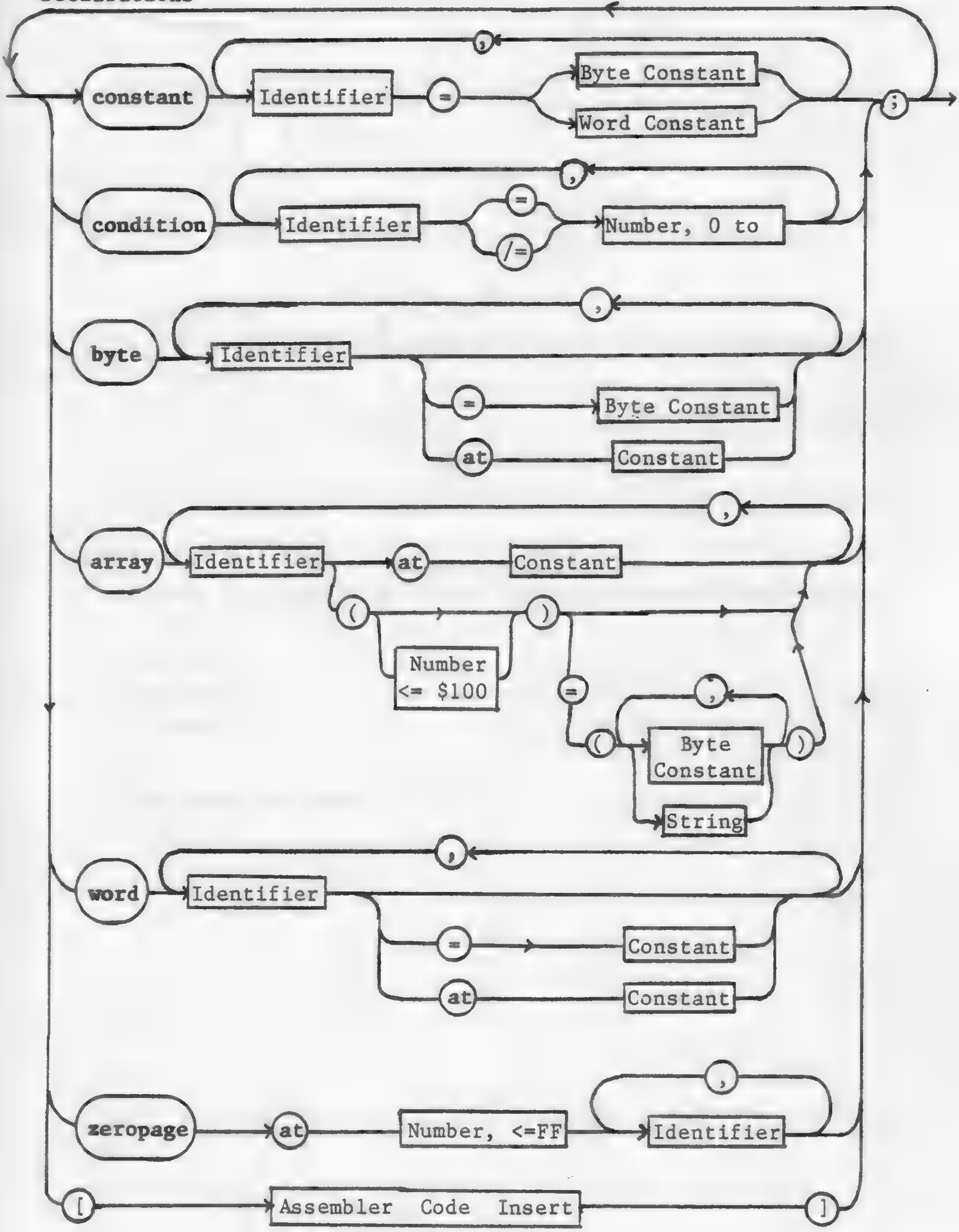
Destination



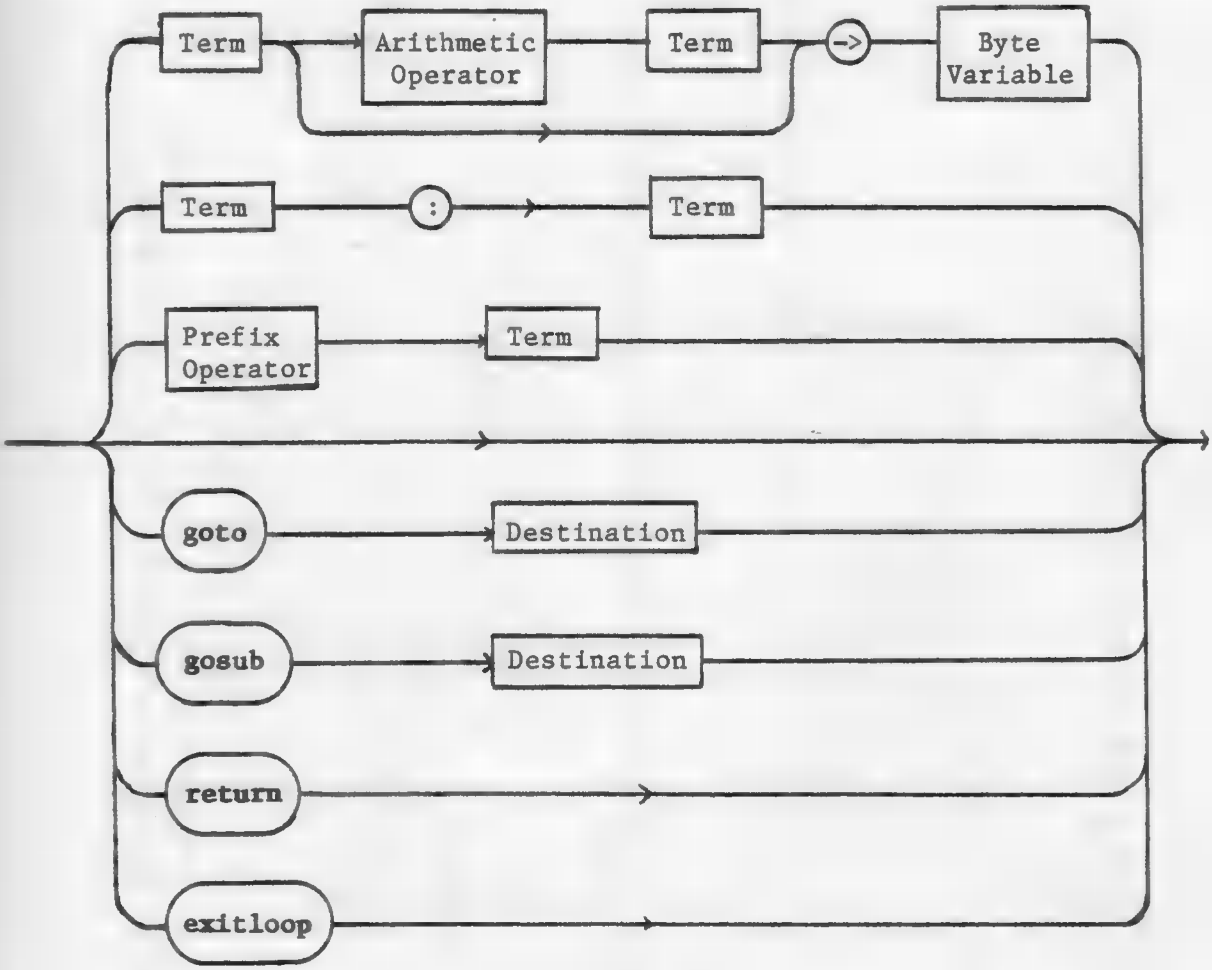
Condition

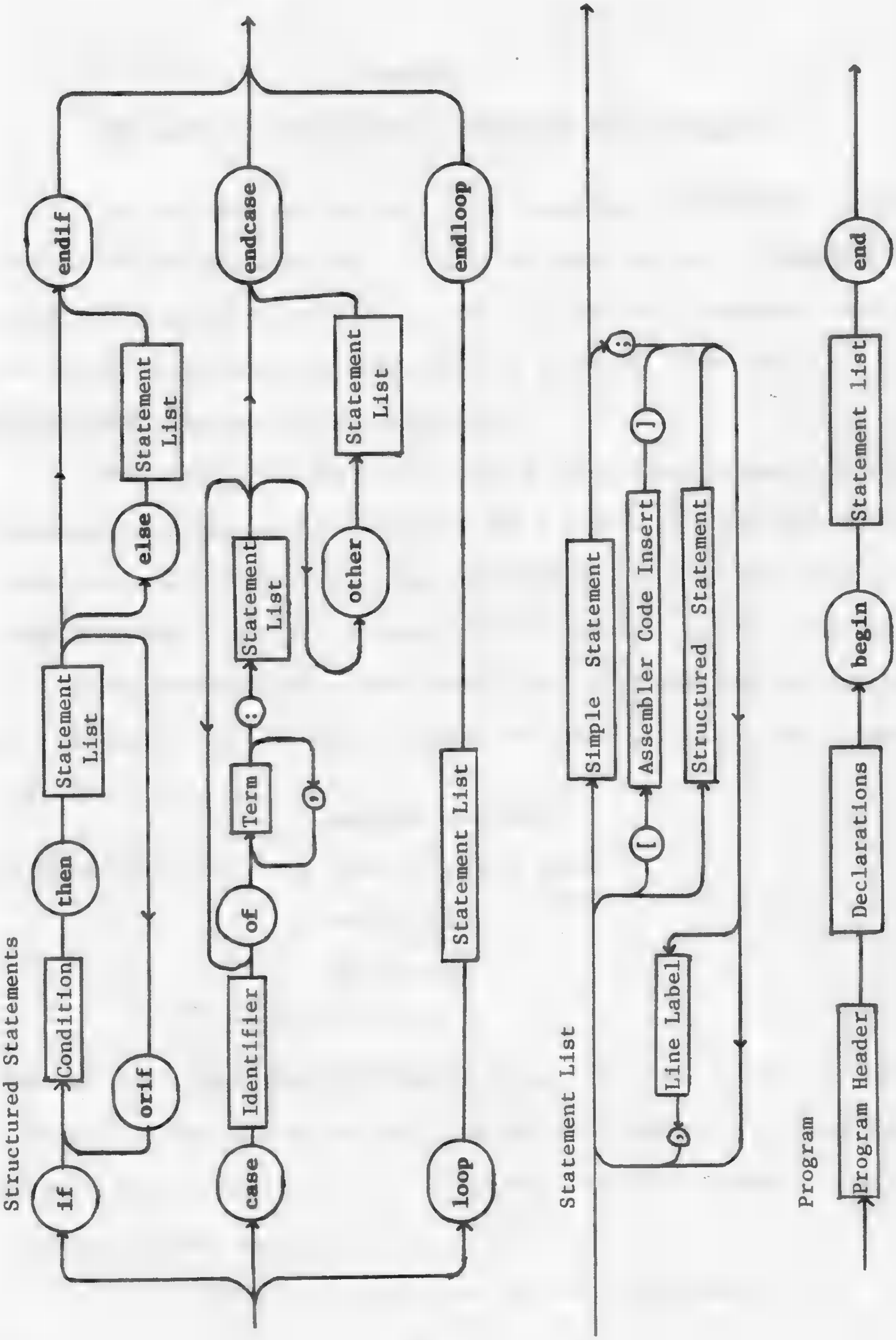


Declarations



Simple Statement




Structured Statements
if
Condition
then
Statement List
orif
else
Statement List
endif
case
Identifier
of
Term
,
:
Statement List
other
Statement List
endcase
loop
Statement List
endloop
Statement List
Line Label
:
Simple Statement
[
Assembler Code Insert
]
Structured Statement
;
Program
Program Header
Declarations
begin
Statement list
end

CHAPTER 3

THE LANGUAGE GRASSHOPPER: SEMANTICS AND TRANSLATION

In this chapter, the result of compiling GRASSHOPPER source code is completely described. Thus, a description of GRASSHOPPER semantics is given in terms of the MOS 6502 assembly language which is the object language of the compiler. In addition, the task of the GRASSHOPPER compiler is also described.

The object file must begin with a five byte header and be formatted in a way which is suitable for a source file for the OS-65D assembler. The formats of the GRASSHOPPER source code and the generated object code will be more fully discussed in Chapter 5.

The GRASSHOPPER program header statement contains the address in core where the executable code will be placed. The header statement:

program address;

is translated into three lines of object code:

```
       *= address
       JMP XS0000
XM0000 .BYTE $00
```

The line labelled XS0000 will be the first line of executable code generated in the object program. The variable XM0000 is a system math variable used in some mathematical expressions, but is never directly referenced by the user. [Section 3.3].

The object code generated by the translation of the declarations and the verb list of a GRASSHOPPER program will be discussed in Sections 3.1 to 3.5. There are two lines of assembler code added to the end of the object file:

```
JMP $2A51
.END
```

The jump instruction returns control to the Disk Operating System (DOS), and " .END " is an assembler directive, indicating the end of the assembler source code.

3.1 <u>Translation Of The Declarations</u>

With the exception of Condition identifiers, object code will be generated for every identifier declared; allocating storage to each variable and assigning a literal value to each constant identifier. The results of compiling the Constant, Byte, Word, Zeropage and Word declarations are summarized in Table 3.1.

All of the information required regarding a Condition variable is stored in its type flag. The value of the type flag is calculated using the number of the bit in the status register it represents, and whether it is declared to be equal or not equal to that bit, [Section 2.1.5]. Thus, for this declaration:

condition NAME = bit#;

the type flag is calculated:

$C8 **or** bit# -> type flag;

and for

condition NAME /= bit#;

the type flag is calculated:

$C0 **or** bit# -> type flag;

TABLE 3.1: Summary Of The Translation Of Identifier Declarations

Declaration Type	Example Of Source Code Input	Example Of Object Code Output	Data Type	Type Flag
constant	NAME = $FF, NAME = $A000,	NAME = $FF NAME = $A000	Byte Constant Word Constant	CONST1 CONST2
byte	NAME, NAME = $FF, NAME **at** $A000,	NAME .BYTE $00 NAME .BYTE $FF NAME = $A000	Byte Variable	VAR8
word	NAME, NAME = $FFFF, NAME **at** $A000,	NAME .WORD $00 NAME .WORD $FFFF NAME = $A000	Word Variable	VAR16
zeropage	**at** $50, NAME1, NAME2	NAME1 = $50 NAME2 = NAME1+2	Zero Page Variable	ZEROPG
array	NAME($10),	NAME =* *= *+1+$10	Array Variable	ARRAY
	NAME($2) = ($0,$1,$2),	NAME =* .BYTE $0 .BYTE $1 .BYTE $2		
	NAME($4) = ("HELLO"),	NAME =* .BYTE 'HELLO'		
	NAME($7) = ("HELLO"),	NAME =* .BYTE 'HELLO' *= *+1+$3		
	NAME **at** $A000,	NAME = $A000		

3.2 Translation Of The Operands

3.2.1 Byte And Word Operands

The large variety of operands possible has been outlined in Section 2.2.1. In Table 3.2, the translation of GRASSHOPPER operands not used as destinations in sequence control statements, is summarized. The following declarations are assumed:

constant CONST1 = $50, CONST2 = $9A40;

byte VAR8; **word** VAR16; **array** ARRAY($5);

zeropage at $50, ZEROPG;

TABLE 3.2: Summary Of The Translation Of Operands.

Source Code Operand	Object Code Operand	Addressing Mode
$50	#$50	Immediate
"A"	#'A	Immediate
$9A40	#$9A40	Immediate
CONST1	#CONST1	Immediate
CONST2 **lo**	#CONST2*$100/$100	Immediate
CONST2 **hi**	#CONST2/$100	Immediate
CONST2	#CONST2	Immediate
loc CONST1	CONST1	Absolute
loc CONST2	CONST2	Absolute
loc $9A40	$9A40	Absolute
VAR8	VAR8	Absolute
VAR16 **lo**	VAR16	Absolute
VAR16 **hi**	VAR16+1	Absolute
ZEROPG **lo**	ZEROPG	Absolute
ZEROPG **hi**	ZEROPG+1	Absolute
ARRAY,X	ARRAY,X	Absolute Indexed
ARRAY,Y	ARRAY,Y	Absolute Indexed
ind ZEROPG,X	(ZEROPG,X)	Indexed Indirect
ind ZEROPG,Y	(ZEROPG),Y	Indirect Indexed
ZEROPG,X	ZEROPG,X	Zero Page Indexed
ZEROPG,Y	ZEROPG,Y	Zero Page Indexed

Where the operand is the destination of a GOTO or GOSUB statement, the addressing mode is always absolute, thus:

Source Code	Object Code
gosub CONST2;	JSR CONST2
gosub LABEL;	JSR LABEL
goto $9A40;	JMP $9A40

3.2.2 Registers As Operands

The use of the accumulator and the X and Y index registers has been briefly discussed in Sections 2.1.6 and 2.2. The accumulator will be involved in most data-manipulations even if it is not specifically referenced in the GRASSHOPPER code, and the index registers are used when any form of indexed addressing is required. This section deals with the case where a register has been used as an explicit operand in a GRASSHOPPER statement. In this case, the assembler code instruction must be choosen according to which register is to be operated on, and on what operation is to be performed.

The MCS6500 assembler language has many register specific instructions. Nine of these instructions specify an action and a register and require an operand field. For example:

```
LDA #$55
LDX #$55
LDY #$55
```

instruct that the accumulator, the X and the Y register, respectively, be loaded with the value in the operand field, $55. These instructions may be divided up into three groups:

1/ The comparison instructions: CMP; CPX; CPY.

2/ The load register instructions: LDA; LDX; LDY.

3/ The store register instructions: STA; STX; STY.

There are four additional instructions which require no operand field since the operand is implied in the instruction. These instructions are used to transfer between the accumulator and one of the index registers:

1/ a transfer from the accumulator to an index register:
TAX; TAY.

2/ a transfer from an index register to the accumulator:
TXA; TYA.

The choice of instruction will be more thoroughly discussed in Sections 3.3 and 4.2.2.

3.3 The Data Manipulation Statements

The translation of the three data-manipulation statements: the prefix operator statement; the assignment statement; and the comparison statement; will be described in this section. The relational expression is only used as the condition portion of the IF construct, so discussion of the translation of this expression will be reserved for Section 3.5.1.

Translation of these statements is complicated by two factors. The first is the use of registers as explicit operands. As was mentioned in Section 3.2.2, there are thirteen register specific instructions which must be used in these cases.

The second problem is that there are many cases where an operand's addressing mode is illegal for a desired assembler code instruction. Table 3.3 summarizes the illegal addressing modes which had to be dealt with when compiling the data-manipulation statements. Each column is labelled with the mnemonic for an assembler code instruction, and each row with a type of operand. An "X" in the table represents an illegal addressing mode, where alternate code must be generated.

TABLE 3.3: Illegal Addressing Modes For Assembler Code Instructions.

Operand	INC, DEC	CPX, CPY	LDX	LDY	STX	STY
ARRAY,X		X	X		X	X
ARRAY,Y	X	X		X	X	X
ZEROPG,X		X	X		X	
ZEROPG,Y	X	X		X		X
ind ZEROPG,X	X	X	X	X	X	X
ind ZEROPG,Y	X	X	X	X	X	X

In the tables that follow, the identifier GENERAL will represent all non-register operands whose addressing modes are legal for the assembler code instruction desired. The identifier SPECIAL will represent all non-register operands whose addressing modes require that alternative object code be generated.

There are several cases, in the translation of data-manipulation statements, where a temporary holding variable is needed. At the beginning of this chapter it was mentioned that space is allocated to a system math variable, XM0000. This variable will never be referenced by the user when writing a GRASSHOPPER source program, but will be used in the object code under certain circumstances. Examples of its use may be seen in Tables 3.7 and 3.8.

3.3.1 The Prefix Operator Statements

Statements in which prefix operators are used are of the following form:

inc Byte Variable;

dec Byte Variable;

The results of compiling these statements depend on the nature of the operands and are summarized in Table 3.4.

TABLE 3.4: Translation Of The Prefix Operations

Operand	Translation of **inc** Operand	Translation of **dec** Operand
A	CLC ADC #1	SEC SBC #1
X	INX	DEX
Y	INY	DEY
GENERAL	INC GENERAL	DEC GENERAL
SPECIAL	LDA SPECIAL CLC ADC #1 STA SPECIAL	LDA SPECIAL SEC SBC #1 STA SPECIAL

3.3.2 The Assignment Statements

The results of compiling simple assignment statements, where there is no calculation performed, have been summarized in Table 3.5. In this table, each row is labelled with the operand being assigned, and each column with the operand being assigned to. By examination of this table, and of Table 3.3, it can be seen that the following translation will occur:

Source Code	Object Code
A -> X;	TAX
ARRAY1,X -> Y;	LDY ARRAY1,X
ARRAY2,X -> X;	LDA ARRAY2,X TAX

TABLE 3.5: The Translation Of The Assignment Statement:
TERM -> RESULT;
Where One Or Both Operands May Be Registers.

Term	-> A	-> X	-> Y	-> GENERAL2	-> SPECIAL
A		TAX	TAY	STA GENERAL2	
X	TXA		TXA TAY	STX GENERAL2	TXA STA SPECIAL
Y	TYA	TYA TAX		STY GENERAL2	TYA STA SPECIAL
GENERAL1	LDA GENERAL1	LDX GENERAL1	LDY GENERAL1	LDA GENERAL1 STA GENERAL2	
SPECIAL		LDA SPECIAL TAX	LDA SPECIAL TAY		

The results of compiling assignment statements which include calculations:

TERM OPERATOR TERM -> RESULT;

have been summarized in Tables 3.6 and 3.7. Table 3.6 summarizes the simplest case where none of the operands are registers.

TABLE 3.6: The Translation Of The Assignment Statement:
TERM OPERATOR TERM -> RESULT
Where None Of The Operands Is A Register.

Assignment Statement					Object Code
OPERAND1	+	OPERAND2	->	RESULT	CLC LDA OPERAND1 ADC OPERAND2 STA RESULT
OPERAND1	-	OPERAND2	->	RESULT	SEC LDA OPERAND1 SBC OPERAND2 STA RESULT
OPERAND1	**and**	OPERAND2	->	RESULT	LDA OPERAD1 AND OPERAND2 STA RESULT
OPERAND1	**or**	OPERAND2	->	RESULT	LDA OPERAND1 ORA OPERAND2 STA RESULT
OPERAND1	**exor**	OPERAND2	->	RESULT	LDA OPERAND1 EOR OPERAND2 STA RESULT

Table 3.7 summarizes the translation of the calculation part of the assignment statement when one of the terms is a register. It is not legal in this implementation of GRASSHOPPER to have registers for both terms. The Boolean and the addition operations are all commutative, so that the same code may be produced whether the

register is the first or the second term. Only the addition operation is illustrated in Table 3.7, but the Boolean operations involving registers are compiled similarly. Since the assembler code instruction must operate on the accumulator the value in the register operand is first transferred into the accumulator. The non-register operand is is then added to the accumulator.

The subtraction operation is not commutative, so that the first term in the statement must be loaded into the accumulator then the second term must be subtracted from it. When the second term is a register it is first stored in the temporary variable XM0000 so that this subtraction may take place. In Table 3.7 subtraction with the accumulator and with the X register have been illustrated. Subtraction with the Y register is compiled similarly.

The translation of the actual assignment part of the assignment statement corresponds to the first row in Table 3.5, since the result of any of the boolean or arithmetic operations will be stored in the accumulator. Thus, by examining Tables 3.5, 3.6 and 3.7, it can be seen that the following translation will occur:

Source Code	Object Code
X **and** $F0 -> Y;	TXA AND #$F0 TAY

TABLE 3.7: The Translation Of The Calculation Part Of The Assignment Statement:
TERM OPERATOR TERM -> RESULT;
Where The Terms Include Registers.

Assignment Statements					Object Code
A	+	NAME	->	RESULT	CLC
NAME	+	A	->	RESULT	ADC NAME
X	+	NAME	->	RESULT	CLC
NAME	+	X	->	RESULT	TXA
					ADC NAME
Y	+	NAME	->	RESULT	CLC
NAME	+	Y	->	RESULT	TYA
					ADC NAME
A	-	NAME	->	RESULT	SEC
					SBC NAME
NAME	-	A	->	RESULT	SEC
					STA XM0000
					LDA NAME
					SBC XM0000
X	-	NAME	->	RESULT	TXA
					SEC
					SBC NAME
NAME	-	X	->	RESULT	SEC
					STX XM0000
					LDA NAME
					SBC XM0000

3.3.3 The Comparison Statement

The comparison statement is of the form:

TERM : TERM;

It compares two terms by subtracting the second term from the first term without storing the result. The purpose of this statement is to set the Carry, Zero and Negative bits of the processor status register which were described in Section 2.1.5. The translation of this statement is summarized in Table 3.8, note that the order of the terms must be preserved in the translation.

TABLE 3.8: Object Code Emitted For The Comparison Statement
TERM1 : TERM2;

TERM1	:	TERM2	Instructions
GENERAL1		GENERAL2	LDA GENERAL1 CMP GENERAL2
A		GENERAL	CMP GENERAL
GENERAL		A	STA XM0000 LDA GENERAL CMP XM0000
X		GENERAL	CPX GENERAL
X		SPECIAL	TXA CMP SPECIAL
GENERAL		X	STX XM0000 LDA GENERAL CMP XM0000
Y		GENERAL	CPY GENERAL
Y		SPECIAL	TYA CMP SPECIAL
GENERAL		Y	STY XM0000 LDA GENERAL CMP XM0000

3.4 Line Labels

There are two kinds of line labels which may appear in the object code. The statement line label which originates in the GRASSHOPPER source code is described in Section 2.5. When it is encountered it is entered into the label field of a line of object code. It may appear in an output line containing an assembler code instruction, for example:

```
LABEL,  A : NAME;
```

will be compiled to:

```
LABEL   CMP NAME
```

Or it may appear as an assembler code directive, for example:

```
LABEL;
```

is a label on an empty statement and will be compiled to:

```
LABEL   =*
```

The second kind of label is the system line label which is generated by the compiler, usually in the translation of the sequence control constructs. System line labels consist of: the letter "X"; a letter which indicates what construct generated the label; and a four place hexidecimal number which gives the sequence in which the labels were generated. The generation and use of these labels will be more thoroughly discussed in Section 7.2. The second character identifies the kind of label according to the following classifications:

TABLE 3.9: Summary Of System Line Labels.

Identifying Letter	Class Name	Class Description
S	XSTART	Program Start
M	XMATH	System Math Variable
L	XLOOP	Beginning Of A Loop
E	XLOPEX	End of a Loop
F	XIF	Steps in an If Construct
G	XENDIF	End of an If Construct
C	XCASE	Steps in a Case Construct
D	XENDCS	End of a Case Construct

In this report, system line labels which appear in examples will be represented by their class name. Sequence numbers will only be used when more then one label of a class appears in the same example.

3.5 Sequence Control Statements and Constructs

The four simple sequence control statements and the three sequence control constructs have been described in Section 2.3. The former are very simply compiled and have been summarized in Table 3.10. The EXITLOOP statement will also be illustrated in the discussion of the LOOP construct.

TABLE 3.10: The Translation Of The Simple Sequence Control Statements.

Statement	Object Code Generated
goto LABEL;	JMP LABEL
gosub LABEL;	JSR LABEL
return;	RTS
exitloop;	JMP XLOPEX

TABLE 3.11: The Translation Of IF Conditions Using Relational Expressions and CONDITION Variables, Assuming Preceeded By: AA:BB;

Relational Expression	Object Code	**condition** variable	Object Code
if AA = BB **then**	LDA AA CMP BB BNE XIF	**if** EQUAL **then**	BNE XIF
if AA /= BB **then**	LDA AA CMP BB BEQ XIF	**if** NEQUAL **then**	BEQ XIF
if AA < BB **then**	LDA AA CMP BB BCS XIF	**if** LESSTHEN **then**	BCS XIF
if AA >= BB **then**	LDA AA CMP BB BCC XIF	**if** NLESSTHEN **then**	BCC XIF
if AA > BB **then**	LDA AA CMP BB BEQ XIF BCC XIF	**if** NEQUAL **then** **if** NLESSTHEN **then**	BEQ XIF BCCXIF
if AA <= BB **then**	LDA AA CMP BB BEQ XIF1 BCS XIF2 XIF1 =*	-----	

3.5.1 Translation Of The IF Construct

The format of this construct has been described in Section 2.3.1. The result of compiling the initial phrase of the construct:

if condition **then**

will depend on:

1/ whether the condition is given as a Condition variable, or as a relational expression. In the latter case comparison code will be generated;

2/ what the condition is. In all cases a branching instruction will be generated.

Table 3.11 compares the object code code generated when relational expressions and Condition variables are used as the condition. Note that there is no equivalent using Condition variables to the relational expression AA <= BB.

In the relational expression, the occurrence of a register as one of the terms is handled differently than in the comparison statement. The two terms are compared in the same way as was summarized in Table 3.8, except that they are always compared as if the register was the first term. The order in which the terms actually occurred will be reflected in the branch instruction which is generated. For example, the following two phrases:

if A < BB **then...** **if** BB < A **then...**

will be compiled to the following segments of code:

```
CMP GENERAL          CMP GENERAL
BCS XIF              BCC XIF
```

Thus, the second phrase is compiled as if it had been stated:

if A >= BB **then**...

The translation of the IF construct, without the comparison and branch instructions, has been summarized in Table 3.12.

TABLE 3.12: Summary Of The Translation Of The IF Construct

```
Source Code Phrase               Object Code Generated
-----------------------------------------------------------
if condition then                    Comparison Code
                                     Branch to XIF

orif condition then                  JMP XENDIF
                                XIF =*
                                     Comparison Code
                                     Branch to XIF

else                                 JMP XENDIF
                                XIF =*

endif                        XENDIF =*
-----------------------------------------------------------
```

This table can be clarified with two examples, one of the simplest form of the IF construct:

```
      Source Code                      Object Code

 if  X = $FF then                         CPX #$FF
                                          BNE XIF
        inc loc $2A67                     INC $2A67
 endif                              XIF    =*
                                    XENDIF =*
```

and a second example with all the possible elements of the IF construct:

```
      Source Code                      Object Code

 if  X < $3F then                         CPX #$3F
                                          BCS XIF1
        gosub FIRSTQUARTER                JSR FIRSTQ
 orif X < $7F then                        JMP XENDIF
```

```
                                        XIF1    =*
                                                CPX #$7F
                                                BCS XIF2
        gosub SECONDQUARTER                     JSR SECOND
  orif X <= $FF then                            JMP XENDIF
                                        XIF2    =*
                                                CPX #$FF
                                                BEQ XIF3
                                                BCX XIF4
                                        XIF3    =*
        gosub LASTHALF                          JSR LASTHA
  else                                          JMP XENDIF
                                        XIF4    =*
        goto CRASH                              JMP CRASH
  endif                                 XENDIF  =*
```

3.5.2 Translation Of The CASE Construct

The format of this construct has been discussed in Section 2.3.2. The selector is first loaded into the accumulator, then each guard encountered is compared to the accumulator. The translation of the remainder of the construct is best described using an example:

```
          Source Code                          Object Code

  case DEVICE                                   LDA DEVICE
    of  CRT :                                   CMP #CRT
                                                BNE XCASE1
          gosub CATHODE                         JSR CATHOD
    of  PRINT1, PRINT2:                         JMP XENDCS
                                        XCASE1  =*
                                                CMP #PRINT1
                                                BEQ XCASE2
                                                CMP #PRINT2
                                                BNE XCASE3
                                        XCASE2  =*
          gosub PRINTERS                        JSR PRINTE
    other                                       JMP XENDCS
                                        XCASE3  =*
          goto BADOUT                           JMP BADOUT
  endcase                               XENDCS  =*
```

In this example, if there had been no OTHER portion, then the XCASE3 =* label would have appeared immediately before the XENDCS =* label.

3.5.3 Translation Of The Loop Construct

The format of this construct has been described in Section 2.3.3. Its translation is far simpler then that of the constructs previously described, and may be illustrated by showing the translation of the example given in Section 2.3.3:

Source Code	Object Code
loop	XLOOP =*
if CURRENT = LENGTH **then**	LDA CURRNT
	CMP LENGTH
	BNE XIF
exitloop	JMP XLOPEX
endif	XIF =*
	XENDIF =*
gosub TRANSFER;	JSR TRANSF
endloop	JMP XLOOP
	XLOPEX =*

In the case of nested constructs, the **exitloop** will refer to the innermost loop.

3.6 Summary

Figure 2.1, given at the end of Chapter 2 is an example of a GRASSHOPPER program which has been compiled, assembled and successfully run. The compiled version is shown in Figure 3.1, illustrating the assembler source code which is actually generated. In this figure, comments have been inserted which describe most of the original GRASSHOPPER code.

FIGURE 3.1: Translation Of Figure 2.1

```
 10         *= $4000
 20         JMP XS0000
 30;            'PROGRAM' GRSHOP $4000:
 40 XM0000 .BYTE 00
 50;            'CONSTANT'    DOS    = $2A51,    INWEKO = $2340,
 60;                          OUTSTR = $2D73,    SEEKA  = $26BC,
 70;                          LDREAD = $2B1A,    SAVE   = $2C3A,
 80;                        CR = $0D,   LF = $0A,   TOTAL = $02;
 90 DOS    = $2A51
100 INWEKO = $2340
110 OUTSTR = $2D73
120 SEEKA  = $26BC
130 LDREAD = $2B1A
140 SAVE   = $2C3A
150 CR     = $0D
160 LF     = $0A
170 TOTAL  = $02
180;            'BYTE'    DSRNO 'AT' $265E,  DSRLEN 'AT' $265F,
190;                      SAVX;
200 DSRNO  = $265E
210 DSRLEN = $265F
220 SAVX   .BYTE 00
230;            'ARRAY'      ADDRESS() = ($91, $9D, $A9),
240;                         TRACK  () = ($16, $18, $20);
250 ADDRES =*
260        .BYTE $91
270        .BYTE $9D
280        .BYTE $A9
290 TRACK  =*
300        .BYTE $16
310        .BYTE $18
320        .BYTE $20
330;            'WORD'       ZADDRESS 'AT' $FF;
340 ZADDRE = $FF
350;            'BEGIN' $00 -> SAVX;
360 XS0000 =*
370        LDA #$00
380        STA SAVX
390;                 ASK,   'GOSUB' OUTSTR;
400 ASK    JSR OUTSTR
410       .BYTE CR,LF,'1/LOAD      2/UPDATE ?',$00
420;                        'GOSUB' INWEKO;
430        JSR INWEKO
440;                        'CASE' A
450;                           'OF' $1: 'LOOP'      ! retrieve !
```

```
460;                                          'GOSUB' NEXT;
470;                                          'GOSUB' LDREAD;
480;                                      'ENDLOOP'
490          CMP #$1
500          BNE XC0002
510 XL0004 =*
520          JSR NEXT
530          JSR LDREAD
540          JMP XL0004
550 XE0003 =*
560;                          'OF'  $2: 'LOOP'      ! save !
570;                                          'GOSUB' NEXT;
580;                                          $0C -> DSRLEN;
590;                                          'GOSUB' SAVE;
600;                                      'ENDLOOP'
610          JMP XD0001
620 XC0002 =*
630          CMP #$2
640          BNE XC0005
650 XL0007 =*
660          JSR NEXT
670          LDA #$0C
680          STA DSRLEN
690          JSR SAVE
700          JMP XL0007
710 XE0006 =*
720;                          'OTHER'   'GOTO' ASK;
730;                        'ENDCASE'
740          JMP XD0001
750 XC0005 =*
760          JMP ASK
770 XD0001 =*
780;  NEXT,  SAVX -> X;
790 NEXT     LDX SAVX
800;                        'IF' X = TOTAL 'THEN' 'GOTO' DOS
810;                        'ELSE'
820;                          ADDRESS,X -> ZADDRESS 'HI';
830;                          $00  ->  ZADDRESS 'LO';
840;                          $01  ->  DSRNO;
850;                          X + $1 -> SAVX;
860;                          TRACK,X  ->  A;
870;                          'GOSUB' SEEKA;
880;                        'ENDIF'
890          CPX #TOTAL
900          BNE XF0009
910          JMP DOS
920          JMP XG0008
930 XF0009 =*
940          LDA ADDRES,X
```

```
 950        STA ZADDRE+1
 960        LDA #$00
 970        STA ZADDRE
 980        LDA #$01
 990        STA DSRNO
1000        TXA
1010        CLC
1020        ADC #$1
1030        STA SAVX
1040        LDA TRACK,X
1050        JSR SEEKA
1060 XG0008 =*
1070;                     'RETURN';
1080;            'END'.
1090        RTS
1100        JMP $2A51
1110        .END
```

CHAPTER 4

A GRASSHOPPER COMPILER

The GRASSHOPPER compiler is written mostly in the assembly language of the MOS 6502 microcomputer, with two portions: SRCMGR [Section 5.3] and FATAL [Section 4.5] written in GRASSHOPPER.

The lexical analysis of a GRASSHOPPER source program is supervised by the routine, ADVANC which will be described in Chapter 6, and which is based on the basic scan used in Halstead's Pilot compiler, [Halstead 1974, p. 36]. The source code is treated as a series of operands and symbols which can be examined as groupings of symbol-operand-symbol triplets. The purpose of ADVANC is to obtain from the source code the next symbol-operand-symbol triplet and place representative tokens in the three Byte variables: CURSYM, CURITM and NXTSYM.

Symbols which may be returned in CURSYM and NXTSYM may be grouped into three categories:

1/ single characters from the ASCII character set. The only such characters to be returned in CURSYM and NXTSYM are those in the general category in Table 6.2;

2/ arithmetic, relational or comparison operators as described in Table 6.5;

3/ or the tokens associated with the key words used in Grasshopper. These tokens are listed in Table 4.1.

TABLE 4.1: The Key Words Used In GRASSHOPPER, And Their Tokens

Key Word	Token Identifier	Token
and	LOGAND	$8D
array	ARRAY	$B8
at	AT	$A8
begin	BEGIN	$A2
byte	VAR8	$B6
case	CASE	$D3
condition	CONDI	$C0
constant	CONST	$B2
dec	KEYDEC	$91
else	ELSE	$E2
end	END	$FF
endcase	ENDCA	$F3
endif	ENDIF	$F1
endloop	ENDLOP	$F2
exitloop	EXITLP	$D5
exor	EXOR	$8F
gosub	GOSUB	$D6
goto	GOTO	$D1
hi	HI	$A6
if	IF	$D2
inc	KEYINC	$90
ind	IND	$A9
lo	LO	$A5
loc	LOC	$A7
loop	LOOP	$D4
of	OF	$E1
or	OR	$8E
orif	ORIF	$E4
other	OTHER	$E3
program	PROGRM	$A0
return	RETURN	$D7
then	THEN	$D0
word	VAR16	$BA
zeropage	ZEROPG	$B4

The token placed in CURITM identifies the next operand type, in almost all cases additional information is stored in other variables to describe and identify the operand. This is summarized in Table 6.1, if there is no operand, CURITM is zero.

Syntactic and Semantic analysis are done in the translating routines supervised by: HEADER; DCLARE and VRBLST, which will be described in Sections 4.1, 4.3 and 4.4. The source code is scanned by repeatedly calling ADVANC and analyzing the current symbol-operand-symbol triplet. There is some error detection, covering syntax errors, nesting errors and operand type errors, this is the subject of Section 4.5.

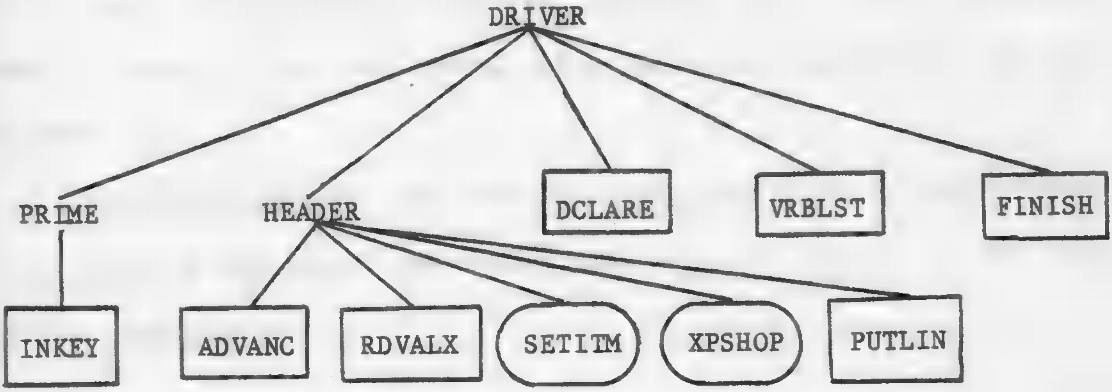
As this scan proceeds, the translating routines generate object code using the format and routines discussed in Chapter 7. The operand field is generated by the routines outlined in Section 4.2. The operator field of the object code, which was briefly discussed in Section 7.1.2, will always contain either an opcode mnemonic, or an assembler directive. All possible operators are contained in the array OPLIST. The translating routines to be discussed in Sections 4.3 and 4.4 must set the operator field by placing the displacement of the desired operator in OPLIST into the byte variable OPDISP.

While efficiency of object code was an objective when writing the compiler, separate optimization of object code has not been attempted in this implementation.

4.1 Overview Of The GRASSHOPPER Compiler

The whole process of compilation is supervised by the routine DRIVER, which may be considered to be the ultimate root of all the subroutine maps shown in this report.

Figure 4.1: Subroutine Map Of The Translator



DRIVER activates PRIME at the beginning of a translation to: initialize the disk I/O buffers [section 5.1]; reserve the first five bytes of the object file for the header described in Table 5.2; to set all variables in the data space used by the compiler to zero; and to enter the key words into the symbol tables.

The array, KEYLST, contains all the key words used, preceded by their tokens. The contents of this array are first read into STBUF1 then entered into TABLES using the routine INKEY discussed in section 6.2.2. The keys are not transferred directly from KEYLST to TABLES because INKEY uses the same table building routines as INNAME and these routines expect to find each new item in the statement buffer.

HEADER simply reads the program header statement and generates the three lines of object code described at the beginning of Chapter 3. If subroutine capabilities are extended in GRASSHOPPER the routine HEADER will become more extensive.

DCLARE compiles the variable and constant declarations and will be more thoroughly discussed in section 4.3. All identifier names are entered into the symbol table under the supervision of this routine.

After DCLARE has been executed DRIVER sets the variable DCLFLG to 1 so that subsequently the symbol tables may be referenced but not altered. [Section 6.2]

VRBLST compiles the executable body of the program, hereafter refered to as the verb list. This will be further discussed in section 4.4.

After the program has been compiled, DRIVER calls the routine FINISH to complete the object program and to finish transferring it to the disk file.

A jump to the Disk Operating System (DOS) is placed at the end of the object code, followed by the assembler directive " .END ". FINISH then completes the transfer of the object disk I/O buffer, [section 5.1], and inserts the header described in Table 5.2 at the beginning of the object file, [section 5.4].

After FINISH has been executed the file in OBJECT will be in a form suitable for processing as source by the assembler/editor. DRIVER then returns control to the operating system.

4.2 Translating The Operand

The task of translating an operand has, with two exceptions, been delegated to the six routines which will be discussed in Sections 4.2.1 and 4.2.2. Which of these routines will be called will depend on the kind of operand which is legal for the context.

The first exception is a character string used to initialize an array in the declarations, and in this case, the routine INSTRG is called directly by the routine which translates array declarations. The second exception is the case of a conditional identifier being used as the condition in an IF statement. This operand is translated in the routine STIF, since there is no other context, outside of the declarations, where a conditional identifier is legal.

Examination of Table 6.1 will show that for several possible values of CURITM which may be returned by ADVANC, there will be information on the operand stored in other variables. The most important of these is the Byte variable NAMFLG. When an identifier is read by RDNAME, its type flag will be put into NAMFLG, the different possible values of which are summarized in Table 4.2. Also, when the operand is the accumulator, or the X or Y register, then the ASCII code value for "A", "X" or "Y" respectively will have been put into NAMFLG by SRCHNM, [Section 6.2.1].

TABLE 4.2: Summary Of The Possible Values Of NAMFLG.

NAMFLG	Actual Value Of NAMFLG	Corresponding CURITM	Type of Operand
"A"	$41	ACC	Accumulator
"X","Y"	$58,$59	REG	Index Register
CONST1	$B2	NAME	Byte Constant
CONST2	$B3	NAME	Word Constant
ZEROPG	$B4	NAME	Zero Page Variable
VAR8	$B6	NAME	Byte Variable
ARRAY	$B8	NAME	Array Variable
VAR16	$BA	NAME	Word Variable
	$C0 to $CF	NAME	Condition Variables

4.2.1 Byte and Word Operands

There are five routines which translate the non-register operands discussed in Section 3.2.1. Each of these routines will return $FF in the accumulator if an appropriate operand was found and has been translated, and returns $00 otherwise. The source code operands which can be translated, and the required results of translation have already been summarized in Table 3.2.

OPB1LT is called when a Byte constant is expected, in every case the addressing mode will be immediate. OPB1VR is called when a Byte variable is expected, and will translate legal operands in all the addressing modes, other then immediate, which are shown in Table 3.2. OPBYT1 is called when a Byte variable or constant is expected and corresponds to the Term syntax graph given in Section 2.6.

OPB2LT is called when the expected operand is a Word constant identifier, or a literal. OPJMP is called when the destination of a GOTO or GOSUB statement is expected, and will except the same kinds of operands as OPB2LT, as well as a statement label.

4.2.2 Registers As Operands

The routine OPREG is called when the operand may be the accumulator, or the X or Y register. The MCS6500 assembler language, which is the language of the object code, has many register specific instructions, thirteen of these have been outlined in Section 3.2.2. The mnemonics for related instructions have been arranged together in OPLIST so that they always occur with the accumulator specific instruction first and the Y register specific instruction last.

The task of OPREG is to determine whether the operand is one of the three registers, and if it is, to store a value in the Byte variable, REGFLG, which can be used to choose the appropriate assembly code instruction from OPLIST. The accumulator is set to zero if a register is found, and to the value of CURITM if not.

```
procedure OPREG;
begin  case CURITM
           of REG : if  NAMFLG = X then  $4 -> REGFLG
                        else  $8 -> REGFLG  endif  $0 -> A;
           of ACC : $0 -> REGFLG;  $0 -> A;
           other CURITM -> A;
       endcase;
end
```

The thirteen register specific instructions are broadly divided into two groups: those which require an operand field; and those which do not.

The first group consists of: the comparison instructions; the load register instructions; and, the store register instructions. A specific instruction within one of these groups can be chosen using the register mapping routine, REGMP1, which will expect to find the OPLIST displacement for the first of one of these series in the accumulator. REGMP1 will store a value in OPDISP using the following function:

A + REGFLG -> OPDISP;

Thus, if before REGMP1 was called the accumulator contained the displacement for " LDA ", and REGFLG contained the value $4, then OPDISP will be set to the displacement for " LDX " in OPLIST. [Section 3.2.2]

The second group consists of transfers: from the accumulator to an index register; and from an index register to the accumulator. When the corresponding register mapping routine, REGMP2, is called, it will expect to find the displacement minus four of either " TAX " or " TXA " in the accumulator. If REGFLG = $0, the operand is the accumulator and there is no action. Otherwise the following code is executed:

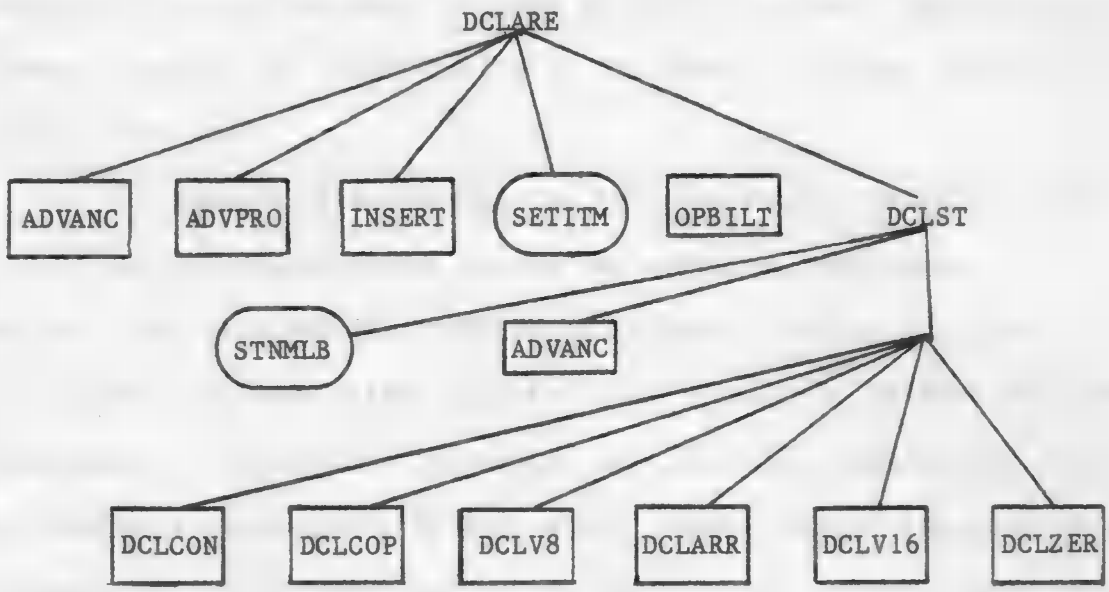
A + REGFLG -> OPDISP; **gosub** PUTLIN;

Thus REGMP2 will emit a line of object code, except in the trivial case of a transfer from the accumulator to to the accumulator.

4.3 Compilation Of The Declarations

The routines DCLARE and DCLST together supervise the process of compiling the declarations. DCLARE supervises the compilation of each declaration statement, DCLST supervises the compilation of each new identifier in a declaration statement. The subroutine map for this process is shown in Figure 4.2.

Figure 4.2: Subroutine Map For The Translation Of The Declarations



The syntax graph for the declarations, given in Section 2.6, shows an initial seven way choice between the six kinds of declarations statements and the assembler code insert. The six routines which may be indirectly called from DCLST correspond to the first six of these choices and will do the actual compilation. The assembler code insert is detected by DCLARE, resulting in a call to INSERT, which transfers the insert directly to the object program.

Throughout this phase of the compilation there is a short routine, ADVPRO, used as a stepping stone to ADVANC:

```
procedure ADVPRO;
begin
   $01 -> DCLFLG;    gosub ADVANC;    $00 -> DCLFLG;
end
```

ADVPRO is used when the expected operand is a numeric literal or a pre-declared constant, used, for example, to assign a value to a new constant. This is necessary because if DCLFLG is zero, ADVANC will attempt to enter any identifier into the symbol tables instead of reading from them.

The token associated with the key word which identifies each type of declaration statement serves two additional purposes. It is stored in the byte variable TYPFLG, to be used either as the type flag stored with each identifier, [Section 6.2], or as a factor in the calculation of this value. Secondly, the four least significant bits are used as a displacement in the array DCLBF1 which contains the addresses of the routines required to specifically translate each type of declaration statement, and in the array DCLBF2 which contains the displacements in OPLIST of the assembler directives used for part or all of the compilation of each declaration statement type. For example, the value of the token for ARRAY is $B8, the address of DCLARR which translates array declarations is located in DCLBF1 at displacements 8 and 9. DCLBF2(8) contains the displacement in OPLIST of " =* ".

DCLARE examines the beginning of each statement to determine whether it is:

1/ an assembler code insert, in which case INSERT is called;

2/ the beginning of the verb list, in which case control is returned to DRIVER;

3/ a GRASSHOPPER declaration statement. The address of the specific translating routine required is read from DCLBF1 into the word variable REST, to be used for an indirect jump to that routine by DCLST. The variable OPDISP is set from the array DCLBF2. TYPFLG will be set equal to the token value of the identifying key word, for example $B8 for the translation of an Array declaration statement. The routine DCLST will then be called.

4/ an illegal statement for the declaration section, in which case an error stop is issued.

```
procedure DCLARE;
condition  SUCCESS /= $1;
begin
  loop
    if NXTSYM = BEGIN  then  return  ! end of the declarations !
    orif NXTSYM = "["  then             ! assembler code insert !
        gosub INSERT;   ";" -> NXTSYM;     gosub ADVANC
    else                              ! new declaration statement !
        NXTSYM  ->  TYPFLG;         NXTSYM and $0F -> Y;
        DCLBF2,Y  ->  OPDISP;
        DCLBF1,Y  ->  REST lo;   inc Y;   DCLBF1,Y  ->  REST hi;
```

```
        case NXTSYM
            of CONDI :  ! Condition Variables !
            of ZEROPG :  ! Zeropage Variables !
                gosub ADVANC;     gosub ADVPRO;
                if (CURSYM = AT) then
                    gosub OPB1LT;
                    if  SUCCESS  and (CURITM /= STRING)
                    then  gosub SETITM
                    else  goto FATAL($08)  endif
                else  goto FATAL($13)  endif
           other         ! Array, Constant, Byte or Word !
                if (NXTSYM and $F0) = $B0
                then  gosub SETITM
                else  goto FATAL($17)  endif
        endcase
        gosub DCLST
    endif
  endloop
end
```

DCLST sets each newly declared identifier as a line label then calls the routine which is specific for the kind of declaration statement in which it occurred. The address of this routine is stored in the word variable REST. When the end of the declaration statement is reached, control is returned to DCLARE.

```
procedure DCLST;
begin
   "," -> NXTSYM;
                          ! loop for each identifier declared !
   loop
      $00 -> ITMBFY;    gosub ADVANC;
      if CURSYM = ";" then  return    ! end declaration statement !
      orif  (CURSYM = ",") and (CURITM = NAME)  then
            ! set the new identifier as a line label, then
             call the routine specific to that data type !
             gosub STNMLB;     gosub  loc REST
      else  ! syntax error in declaration statement !
             gosub FATAL($11)
      endif
   endloop
end
```

4.4 Compilation Of The Statement List

The brief routine, VRBLST initiates the compilation of the entire verb list of a GRASSHOPPER program. Using the routine, XPLLAB, described in Section 7.2, VRBLST emits the line:

XS0000 =*

to the object file, thereby labeling the beginning of the executable code.

```
procedure VRBLST;
begin
    XSTART  ->  A;         gosub XPLLAB;
    ";"  ->  NXTSYM;       gosub ADVANC;     goto STMLST;
end;
```

There are three routines which supervise the rest of the compilation process, these are: STMLST, STMNXT and STDELM. Control passes between these according to what area of the Statement List syntax graph a compilation is in, [Section 2.6]. Thus, STMLST is called when entering the statement list syntax graph; STMNXT is called when the next statement in a statement list is required; and STDELM is called when an intermediate or terminating statement delimiter is expected.

STMLST supervises the compilation of: the simple statements; the assembler code inserts; and the first phase of the structured statements.

```
procedure STMLST;
begin
  case statement
     of empty statement :     goto STDELM;
     of simple statement :   translate;     goto STDELM;
     of assembler code insert :  transfer;   goto STMNXT;
     of structured statement :  translate up to where a
          new statement list is expected;    goto STMLST;
     other error, unidentified statement,    goto FATAL($18)
  endcase
end;
```

STMNXT corresponds to a continuation of the cycle of the Statement List syntax graph. When STMNXT is called, CURSYM is a statement terminating delimiter, and may be followed by a statement label.

```
procedure STMNXT;
begin
    if  CURITM = LINLAB then  place label in line label field of
          current line of object code;
          if of form " LABEL; "  then
               generate object code   " LABEL =* ";    goto STDELM
          orif of the form " LABEL,... "  then
               continue scan... goto STMLST
          else  syntax error...goto FATAL($19)  endif
    else  goto STMLST  endif
end;
```

STDELM is called when an intermediate or terminating statement delimiter is expected to follow.

```
procedure STDELM;
begin
   case CURSYM
      of END : if system line label stack is empty
               then return to DRIVER;
               else goto FATAL($31) endif
      of terminating statement delimiter : translate;   goto STMNXT;
      of intermediate statement delimiter : translate;   goto STMLST;
      other missing statement delimiter, goto FATAL($20)
   endcase
end
```

Two points should be clarified before these routines are more completely described. At the beginning of the compilation of any statement, when NXTSYM is the first symbol in the new statement, CURITM will be equal to zero on all but two occasions: when the new statement is an arithmetic, comparison or prefix operator statement, to be compiled by the routine STEXPR; and when the new statement is labelled.

The second point is that the token values for intermediate and terminating statement delimiters are in the ranges $E0 to $EF and $F0 to $FF respectively.

For the purposes of this description, the routines SIMPLE and STRUCT may be considered to call the routines required to translate the simple and the structured statements, respectively.

```
procedure STMLST;
begin
  loop
    if  CURITM /= 0  then
          gosub STEXPR;   gosub ADVANC;   goto  STDELM;
    else  gosub ADVANC;
          if CURSYM  =  ";"  then   goto STMNXT
          orif  CURSYM and $E0 = $E0  then   goto STDELM
          else
             case CURSYM
                of  GOTO, GOSUB, EXITLP, RETURN :
                     gosub SIMPLE;    gosub ADVANC;   goto STDELM;
                of  IF, CASE, LOOP :     gosub STRUCT;
                of  "[" :  gosub INSERT;      ";"  ->  NXTSYM;
                           gosub ADVANC;      goto STMNXT;
                other  goto FATAL($18)
             endcase
          endif
    endif
  endloop
end;
```

```
procedure STMNXT;
begin
  loop
      if  CURITM = LINLAB  then    gosub SETLAB;
            if NXTSYM = ";" then
                   XEQUST -> OPDISP;     gosub PUTLIN;     gosub ADVANC
            orif  NXTSYM = ","  then  gosub ADVANC;  goto STMLST
            else  goto FATAL($19)   endif
      else  goto STMLST  endif
  endloop
end
```

For the purposes of this description, the routines DINTER and DMTERM may be considered to translate the intermediate and terminating statement delimiters, respectively.

```
procedure STDELM;
begin
  case CURSYM
     of END : if  XPOINT /= 0  then  goto FATAL($31)
                 else  return  endif
     of ";", ENDLOP, ENDIF, ENDCA :
                 gosub DMTERM;    goto STMNXT;
     of ELSE, ORIF, OF, OTHER :
                 gosub DINTER;    goto STMLST
     other   goto FATAL($20)
  endcase
end.
```

4.5 Error Detection And Diagnostics

In the current implementation of GRASSHOPPER, the handling of errors in the source code is rather primitive. A wide range of errors are detectable, but, with one exception, there is no error recovery attempted, so that the compilation ceases on the first error detected. The single exception is the case of the same identifier being declared more then once. In this case a warning is issued by the routine WARN, called from INNAME, and all but the first declaration are ignored.

When a fatal error is detected, an error identifier is put into the accumulator, and the routine FATAL is called. In the algorithm descriptions given in this report, this is represented as:

goto FATAL(identification);

but in the compiler the transfer to fatal is always a subroutine jump. FATAL prints out the error identifier, then pulls the return address saved in the stack used by the MOS 6502 for subroutine jumps and prints this out. The line of source code currently being scanned is printed followed by a selected core dump of global variables and arrays. The contents of ITMBUF and the statement label buffer, LABELS, are also printed in their ASCII characters.

This information will just fit on the CRT terminal screen. If a printer is activated for the run, then a hardcopy of this dump can be obtained. Since NEXTLN always displays each line it enters into STBUF1, the source program, up to the error, will also be printed. Figure 4.3 gives an example of a compilation run, where the source code is the example given in Figure 2.1, with SAVX in the declarations istyped resulting in a second declaration of SAVE and then a fatal error in the verb list.

The most important output to a user are the contents of the statement buffer, and the error identification. This identifier may be either a letter or a decimal number. These are summarized in Tables 4.3 and 4.4, respectively.

With the exception of "B", a letter will indicate a bug in the compiler. The code to detect these errors was left in the compiler assuming that there will be further developement.

TABLE 4.3: Compilation Error Summary: Letters.

Error	Detecting Routine	Cause Of Error Stop
A	NEXTLN	STBUF1 has been overflowed.
B	NEWREC	Symbol table overflow, Need to increase Space alloted
C	OPSAVE	Save Buffer overflow
D	OPREST	Save buffer underflow
E	RDNUMX, RDVALX	Expecting number symbol, $, @, %.
G	RDNMIN	Overflow of STBUF1, or ITMBUF

There are thirty-one possible error numbers covering a wide range of scanning, syntactic and semantic errors. Table 4.4 largely summarizes the causes of these errors, only two of them require further comment.

Error number 1 will only result if there is a character detected that has no legal context in the source code, except perhaps in a comment, string or insert. Completely illegal characters include "&" and most of the characters with ASCII values less then $20.

The absence of a right delimiter on keys, comments, inserts and strings can result in a wide variety of error messages, only one of which is number 3. If it happens with the 'KEY' format, the error message number 2 will result, since the first following character is taken as an illegal character for this format. The other three cases result in scanning errors. The following code will be read as part of the comment, insert or string until one of two things happen. A right delimiter may be encountered, for example:

! comment **gosub** ANYTHING; ! another comment !

In this example, the subroutine call will be read as comment and there will be an attempt to read the second comment as code. The error numbers which may result include, but are not limited to: 2, 4, 10, 11, 17, 18, 19, 20 and 23. This will not happen with the insert because the left and right delimiters are not the same. In this case, unless there is a nesting error, or a declaration missed, the error may not be detected by the compiler. It will be during assembly because GRASSHOPPER code will have been inserted into the object file.

If no right delimiter is detected before the end of file is read, then error number 3 will result, since SETSTR, INSERT and NEXTLN all check for the escape character inserted as an end of file flag by SRCMGR, [Section 3.2].

TABLE 4.4: Compilation Error Summary: Numbers.

Error	Cause of Error Stop	Detecting Routine
1	Illegal character found during scan of source code.	CHRTYP
2	Unrecognizable key found.	RDKEY
3	Missing right delimiter for: String(")	SETSTR
	Insert(])	INSERT
	Comment(!)	NEXTLN
4	Decimal numbers not implemented.	RDNUMX
5	Empty string not allowed.	INSTRG
6	Incomplete file, missing "**end**".	ADVANC
7	Word literal is required for program header statement.	HEADER
8	Bad or missing initialization in a declaration statement. May be a word value where byte value needed.	DCLCON, DCLV8, DCLARR, DCLV16, DCLCOP
9	Bad array length declared.	DCLARR
10	A, X, Y, S, P are reserved as variable names by assembler, "X" as first letter reserved by compiler	SRCHNM
11	Error in declaration statement, may be missing "," between identifiers, or ";" at end of statement.	DCLST
12	Error in Constant declaration, expect "=".	DCLCON
13	Error in Zeropage declaration, expect: **at** Byte Value.	DCLARE
14	Error in Byte or Word declaration expect **at**, "=", "," or ";".	DCLV8, DCLV16
15	Error in Array declaration, expect "(" or **at**.	DCLARR
16	Error in Condition declaration, expect "=", or "/=".	DCLARR

TABLE 4.4: continued.

Error	Cause of Error Stop	Detecting Routine
17	Unrecognizable declaration statement, May be a missing **begin**.	DCLARE
18	Unrecognizable statement in the Verb list.	STMLST
19	May be undeclared identifier, or bad syntax in statement label.	STMNXT
20	Expect intermediate or terminating statement delimiter, probably ";".	STDELM
21	Bad or missing operator.	STEXPR, STCOND, STIF
22	Bad or missing destination in GOTO or GOSUB statement.	STGOSB, STGOTO
23	Bad or missing operand. May be wrong data type or undeclared identifier.	STEXPR, STCOND, STIF, STCASE, STOF
24	Missing or bad index on indexed data type, Arrays and Zeropage.	OPINDX
25	Error in IF or ORIF statement, expect **then**	STIF, STORIF
26	Error in CASE statement, expect **of**	STCASE
27	Error in OF phrase of CASE statement, expect "," or ":".	STOF
28	System line label stack overflow, the limit of nesting has been passed.	XPUSH
29	Nesting error, probably overlap of stuctured statements.	XPULL, XFNDJP STDELM
30	Nesting error, attempt to pull from empty sytem line label stack, may be too many structure termination delimiters, i.e. an extra **endif**.	XPULL, XFNDJP
31	End of file before system line label stack is empty, structured statement incomplete, i.e. missing **endif**.	STDELM

FIGURE 4.3: Example Of An Error Dump

```
 'PROGRAM' GRSHOP $4000;
 !          This is a simplified version of the program
      I used to store and retrieve the assembled version
      of GRASSHOPPER.                                    !

 'CONSTANT'       DOS    = $2A51,      INWEKO = $2340,
                  OUTSTR = $2D73,      SEEKA  = $26BC,
                  LDREAD = $2B1A,      SAVE   = $2C3A,

                  CR = $0D,   LF = $0A,   TOTAL = $02;

 'BYTE'       DSRNO 'AT' $265E,  DSRLEN 'AT' $265F,  SAVE;
****** RE-DECLARATION OF: SAVE

 'ARRAY'      ADDRESS() = ($91, $9D, $A9),
              TRACK  () = ($16, $18, $20);

 'WORD'       ZADDRESS 'AT' $FF;

 'BEGIN' $00 -> SAVX;

******ERROR #23 AT $AD9F,  FOUND IN:
 'BEGIN' $00 -> SAVX;

        VARIABLES
0F 00 BA 53  89 3B 07 00  74 02 0D 00  80 01 01 09
0D 10 08 71  B0 15 00 00  00 00 53 00  00 00 01 01
01 00 00 00  52 95 CE 9F

       ITEM BUFFER
24 30 30 35  46 00 00 00  00 00 00 00  00 00 00 00
00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00 00
$005F
       LABELS
00 00 00 00  00 00 53 41  56 58 00 00
SAVX

       VECTORS
71 5F 71 32  71 7A 71 B0  71 68 71 9E  71 A7 71 95
00 00 71 05  70 99 71 29  70 90 71 17  70 EA 71 0E
DONE, T= 01
A*
```

CHAPTER 5

INPUT/OUTPUT AND FILE MANAGEMENT

5.1 Disk Input/Output Buffers

The compiler must read from a source file written in Grasshopper and write to an object file in assembler code. Since the memory available could be quickly exhausted if these files were kept in core during translation, a system has been chosen which requires only part of each of these files in core at a time. This is done using the OS-65D supported disk input/output buffers [Ohio Scientific 1978,pp. 57-59].

For each disk I/O buffer used there is a area on the disk which contains the complete file being accessed. The buffer itself is long enough to contain the amount of information to be stored on each track of this file, usually $C pages. When the input or output flag has been set to indicate this buffer a call to a system I/O routine will input from or output to that buffer. When the end of the buffer has been reached, a track boundary has been crossed and transfer between core and disk will be initiated and performed by the system. Thus only one track of the file is in core at a time.

Before the translation begins, the disk buffer parameters, which are described in Table 5.1, are initialized by PRIME to the values shown. The first disk I/O buffer is used for reading from the source file while the second is used to write the object file. After translation is complete, the second disk I/O buffer may contain information which has not yet been transferred to disk so FINISH outputs zeros to the object file buffer until the last track has been transferred.

TABLE 5.1: Parameters Required For The Disk I/O buffers

Parameter		Location = Initialized Value Disk Buffer 1	Disk Buffer 2
Buffer Start	Low	$2326 = $00	$232E = $00
	Hi	$2327 = $42	$232F = $4E
Buffer End	Low	$2328 = $00	$2330 = $00
	Hi	$2329 = $4E	$2331 = $5A
First track, (BCD)		$232A = $65	$2332 = $68
Last track, (BCD)		$232B = $67	$2333 = $76
Current track, (BCD)		$232C = $64	$2334 = $67

The input and output flags previously mentioned are used to specify the I/O devices to be used by a system input/output routine. Each bit of an I/O flag refers to a different device, the disk I/O buffers 1 and 2 are specified by bits 5 and 6 respectively. The values of these flags before translation are saved in the variables SINFLG and SOTFLG by PRIME. When either of the disk I/O buffers is used the appropriate flag is set to that buffer. Immediatly after use the flag is reset from SINFLG or SOTFLG.

5.2 Origins Of The Source File

Using the OS-65D assembler/editor, a source file is created which includes line numbers, a carriage return (CR) at the end of each line, and in which all repeated character strings have been packed into a two character code. The source file is positioned in core, usually starting at $317E, with a five byte header beginning at $3179 which gives information regarding the length and position of the file, [Ohio Scientific 1976, p.4]. This header is outlined in Table 5.2. SRCMGR pre-processes the source file to remove all line numbers, unpack repeated character strings and add the ASCII escape character ($1B) as an end of file flag. As the file is processed it is placed on the disk using Disk I/O buffer two.

TABLE 5.2: Source File Header, [Ohio Scientific 1978, app. p. 4]

Byte #	Memory Address	Parameter
0	$3179	Source Start, Hi
1	$317A	Source Start, Low
2	$317B	Source End, Hi
3	$317C	Source End, Low
4	$317D	Number Of Tracks
5	$317E	Usual starting address of the source file

5.3 Input From Source

The compiler expects to find the source file on disk as pre-processed and placed by SRCMGR, and accesses it using Disk I/O buffer one. Reading of the source file is restricted to three routines: NEXTLN, SETSTR and INSERT.

NEXTLN performs the first step in the lexical scan which is supervised by the routine ADVANC [Chapter 6]. It reads one line of code, terminated by a carriage return, from the source file into the buffer STBUF1. All comments are deleted and only one space of any set of consecutive spaces is included. When a character string or assembler code insert is encountered the leading delimiter (" or [respectively) is stored, followed by a carriage return to end line input. This is done because the character string and insert are special cases which are read by SETSTR and INSERT respectively.

SETSTR is called from INSTRG which is called when a character string is to be read from the source file where it is delimited by double quotes, into the object file where single quotes are to be used. SETSTR reads the string from the source file into the buffer ITMBUF in portions of no more then 40 characters. On returning from SETSTR the accumulator is set to indicate: 00/ that the end of the string was reached with no new characters transferred; 01/ that the end of the string was found after reading one or more characters into ITMBUF; or 02/ that the end of the string was not encountered.

INSERT transfers an assembler code insert from the source where it is enclosed in square brackets, [text], to the object. The removal of brackets is the only change made to such inserts and it is the user′s responsibilty to insure that the insert is reasonable.

If the right delimiter of a comment, character string or assembler code insert is missing then the left and right delimiters will not match up and eventually the escape character [Section 5.2] will be encountered. In this case an error stop is issued.

5.4 Output To Object

The first five bytes of the object file must be reserved for the header described in Section 5.2, this is done in PRIME. After the object file has been written FINISH retrieves the first track, calculates and places the values of the header parameters then returns this back to disk. This manipulation of the first track is done using the OS-65D routines for reading and writing a single track.

During the translation process itself the writing of a character into the object file is restricted to PUTACC which is called from: PUTLIN, INSERT, PUTXLB, and FINISH. PUTACC contains PUTOUT which writes each character using system routine OUTCH and increments the variable COUNT. COUNT is a two byte variable which always contains the current length of the object file, it is used by FINISH to calculate the end address for this file when it is loaded as the source for the assembler. PUTACC calls PUTOUT for three different purposes;

1/ to output the character received in the accumulator;

2/ to output two null characters ($00) after each carriage return (CR), as blank line numbers;

3/ to output the repeat count when packing repeated character strings into the two character code used by the assembler/editor.

PUTACC will now be more precisely described:

```
procedure PUTACC(ACC);
      procedure PUTOUT(ACC);
      condition  ZERO = $1;
      begin   OUTCH(A);    inc COUNT lo;
              if  ZERO  then   inc COUNT hi   endif
      end
begin
     A  ->  SAVA;
     case  LASTCH
         of CR:                              ! blank line label !
                gosub PUTOUT($00);    gosub PUTOUT($00);
                SAVA  ->  LASTCH;     gosub PUTOUT(SAVA);
         of SAVA:   dec REPEAT;          !repeated character !
         other  if  REPEAT/=$00  then   ! last character was the end
                        of a repeated character string !
                        gosub PUTOUT(REPEAT);    $00 -> REPEAT;
                endif
                                   ! output the current character !
                SAVA  ->  LASTCH;     gosub PUTOUT(SAVA)
     endcase
end
```

CHAPTER 6

DESCRIPTION OF THE LEXICAL SCAN

The routines discussed in this chapter are used for the lexical analysis of the source code. These routines operate on code placed in the buffer STBUF1 by NEXTLN and the X register is reserved for the indexed addressing of this buffer. The routine ADVANC supervises the lexical scan and is based on the basic scan used in Halstead's Pilot compiler, [Halstead 1974, p. 36], the purpose of which is to obtain the next symbol-operand-symbol triplet. Each item is inspected and the appropriate routine is called for reading an item of its type. When a carriage return is encountered the routine NEXTLN is called to extract the next line from source and place it in STBUF1. The following algorithm gives a crude outline of what ADVANC does:

```
begin
    NEXTSYM  ->  CURSYM;
    if  there is an operand  then
        read it and put its kind into CURITM; will
        call one of:  RDNAME, INNAME or RDNUMX
    endif
    Read next symbol into NXTSYM; may call one of:
    RDKEY1,  RDKEY2  or  GETSYM
end
```

This algorithm will be discussed in more detail in Section 6.4. After execution of ADVANC, CURSYM will contain the previous value of NXTSYM, NXTSYM will contain the next operator and CURITM will identify the next operand type. Table 6.1 summerizes the types of operands which may be found. An operator may be the token associated with a key word or may be a function of the CHRFLG value for a single ASCII character as discussed in the next section. Character strings and assembler code inserts are not read by ADVANC, instead the leading delimiter is stored as NXTSYM, and in the case of a string, CURITM is set to STRING.

TABLE 6.1: Summary of Operand Types

CURITM	Operand Type	Additional Information Stored
NAME	Identifier	Stored in TABLES Data Type -> NAMFLG Record Location -> ZNAMFG
STRING	Character string	
NUMBER	Number, larger then BYTE2	Stored in ITMBUF Length -> ITMLEN
BYTE1	One Byte Number	Same as for NUMBER
BYTE2	Two Byte Number	Same as for NUMBER
LINLAB	Statement Label as line label	Stored in LABLIN
JMPLAB	Statement Label as operand	Stored in LABJMP
REG	X or Y Register	Stored in NAMFLG
ADDRES	Constant used as Address	Operand not yet read, CURITM set when **ind** or **loc** is encountered
ACC	Accumulator	
PREOP	Operand preceded by Prefix Operator	Operand not yet read, CURITM set when a Prefix operator is encountered

In this chapter, descriptions of several routines will be given using pseudo-GRASSHOPPER. In these descriptions, constant declarations corresponding to the constant identifiers described in Tables 4.1, 6.1 and 6.6 may be assumed, as well as the following declarations:

```
byte     CURITM,  CURSYM,  NXTSYM,   CHRFLG,  FOUND,
         SAVX,    SAVHI,   TYPFLG;

word     LEGAL;

array    LABJMP at $9060,    LABLIN at $905A,
         NAMVCT at $9066,    KEYVCT at $9076,
         STBUF1 at $5A00;

zeropage at $50,  ZRECRD,  ZNAMFG,  ZNEXT;
```

6.1 Character Recognition

Recognition of the type and range of each item requires recognition of character types and specification of what character types are legal for what items.

Recognition of a character's type is accomplished by the routine CHRTYP. The ASCII value of the character is used to find a number which has been encoded to give information on that character's type and use. This is done by subtracting $20 and using the result as a displacement in the array CHRBUF, the value found at this displacement is then placed in the byte variable CHRFLG. If the CHRFLG value found is equal to zero an illegal character has been read and an error stop occurs.

The value placed in CHRFLG has been encoded to give information as shown in Tables 6.2 and 6.3.

TABLE 6.2: CHRFLG Values For Character Types

Character Type	CHRFLG	Characters
Illegal	$00	
Operator	$00 < CHRFLG < $0F	/ < > = - + * :
General	$0F	( ) " ; . , [ ] carriage return, escape and space.
Number Type Symbol	$10	$ % @
Letters and Digits	$60 < CHRFLG < $E0	A...Z, a...z, 0 1 2 3 4 5 6 7 8 9

TABLE 6.3: CHRFLG Values For Letters and Digits

BIT #	CONTENTS					
7	1	letter			0	not a letter
6	1	a -> f	0	g ->z	1	digit, 0 to 9
5	1	UPPER CASE	0	lower case	1	
0 -> 4	0					

The information encoded into CHRFLG is used mostly to detect the type and range of an item being scanned, and, in the case of operators, to give information to be used during the translation of expressions. The four actual uses of CHRFLG will now be described.

I/ ADVANC, which determines how each item in source is to be read in, will first test for special symbols then use CHRFLG to differentiate between identifiers, lower case keys, decimal numbers and other numbers. The algorithm for making this distinction is as follows;

```
with  CHRFLG  do
   if BIT7 = $1 then            ! letter !
          if  BIT5 = $1  then  read an identifier
          else   read key in lower case  endif
   orif  BIT6 = $1  then  read decimal number
   orif  BIT4 = $1  then read a number with base 2, 8 or 16
   else  store symbol in  NXTSYM
   endif
endwith
```

The complete algoritm for ADVANC is listed in Section 6.4.

II/ CLEGAL is called when scanning an identifier or a key word to determine whether the current character is part of that item. CLEGAL first calls CHRTYP to set CHRFLG, then makes an indirect jump to the routine whose address is stored in LEGAL. This routine then tests the specific bits of CHRFLG significant to the item being scanned. The subroutine map for CLEGAL is given in Figure 6.1. Table 6.4 describes specifically what is being tested in each case.

FIGURE 6.1: Subroutine Map For CLEGAL

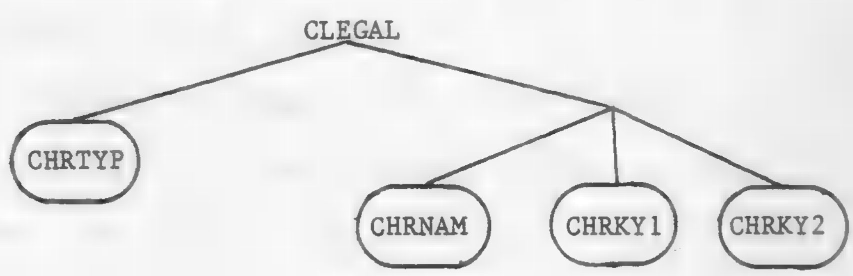


TABLE 6.4: Summary of CHRNAM, CHRKY1 and CHRKY2

Routine	Item	Legal	Character Type
CHRNAM	Identifier	%XX1X0000	numeral, or upper case letter
CHRKY1	'KEY'	%1XXX0000	letter, upper or lower case
CHRKY2	key	%1X0X0000	lower case letter.

After execution of any of these routines, the second, or "Z" bit of the processor status register will be set to 1 if the character is legal or re-set to 0 if illegal.

III/ GETSYM uses CHRFLG to calculate a token value for each operator and stores this value in NXTSYM. In Table 6.2 the CHRFLG value of an operator is listed as between \$00 and \$0F. Table 6.5 shows the actual CHRFLG values for the operators, as well as the values which must be entered into NXTSYM. NXTSYM for all the single character operators is found by adding \$80 to CHRFLG, the value for two character operators is found by doing this calculation for the first character and adding one. In the algorithm given below, note that STBUF1,X is the character immediately following that for which CHRFLG was found. The full algorithm for GETSYM is listed in Section 6.4.

```
if   CHRFLG  /=  $0F  then
     if ((CHRFLG < $06) and (STBUF,X = "=")) or
        ((CHRFLG = $08) and (STBUF,X = ">"))
     then   inc CHRFLG;   inc X;   endif
     CHRFLG  or  $80  ->  NXTSYM;
endif
```

Table 6.5 only lists the operators represented by symbols, as opposed to key words, which are included in Table 4.1. The operators for multiplication (*) and division (/) are included, but are not available in this implementation of GRASSHOPPER.

TABLE 6.5: CHRFLG and NXTSYM Values For The Operators

Operator	CHRFLG	NXTSYM	Operator	CHRFLG	NXTSYM
/	$01	$81	=	$07	$87
/=		$82	-	$08	$88
<	$03	$83	->		$89
<=		$84	+	$0A	$8A
>	$05	$85	*	$0B	$8B
>=	$06	$86	:	$0C	$8C

IV/ <u>RDNUMH</u>, which transfers a hexadecimal number from STBUF1 to ITMBUF, recognizes the end of the number when an illegal character is read. A character is legal if it is a digit, 0 to 9, or an upper case letter, A to F, so CHRFLG must be %X110 0000, thus:

if (CHRFLG **and** %01110000) = %01100000

then legal for hexadecimal number

else not legal, end of number **endif**

6.2 The Symbol Tables

The symbol tables are maintained in two very simple hash tables using a method similar to that found in [Lewis 1976,p.79]. The first two letters of an item are added together and the lowest three bits are extracted from the sum. The resulting number is multiplied by two to give a displacement in a list pointer vector. If the item only has one letter then the same calculation is performed without the initial addition. These calculations are performed by the routine MAP.

If the value found at this displacement is zero there is no corresponding list and the item has not been tabulated. Otherwise the value found is the address of the first record in a linked list which is searched until there is a match or the end of the list is reached. In the latter case the item has not been tabulated.

The buffer TABLES contains all language key words, followed by all user identifier names, no line labels are tabulated. There are separate list pointer vectors maintained for keys (KEYVCT), which are entered by PRIME, and for identifiers (NAMVCT), which are entered during translation of the declarations. The individual records are formated as shown in Figure 6.2. Constants are used for key positions in a record so that the record format may be easily changed. These constants are described in Table 6.6.

The routines which search and build these tables are designed so whenever possible the same code can be used on both tables; for this reason the record format is the same for keys as for identifiers.

The addressing method allows access to all of core, but by adjusting the values of TABLES and OVER the table space can be placed and its size limited. The lowest three bytes are used in the mapping function so that the result will be the same whether upper or lower case letters are used.

I felt that a more elaborate method was not required for this implementation but tried to program so that the method could be easily refined or altered without side effects.

FIGURE 6.2: Record Format For The Symbol Tables

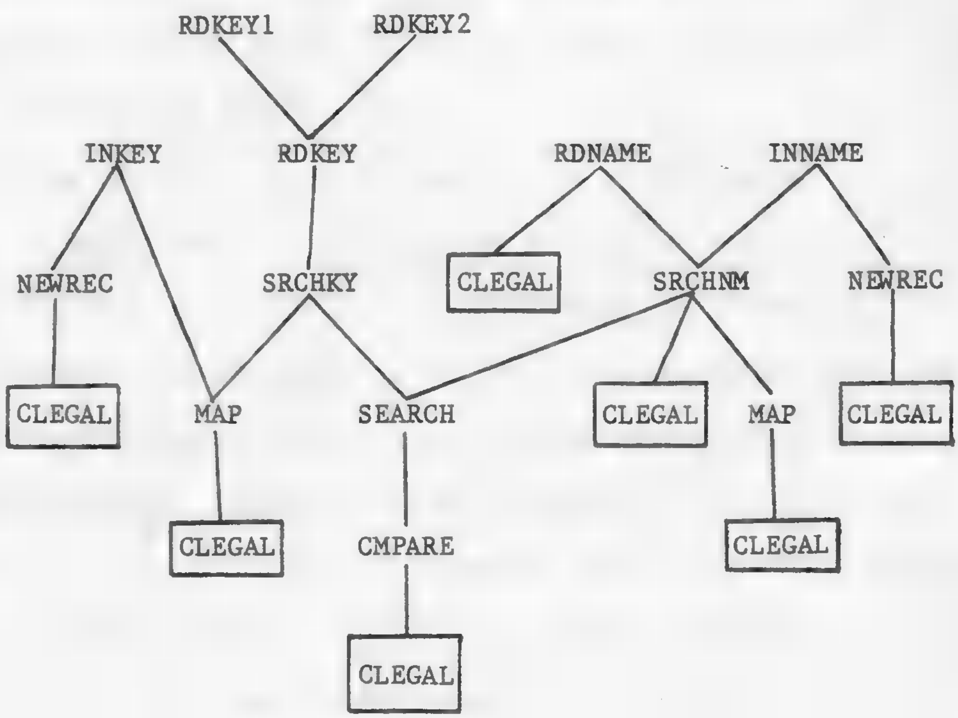
0	
1	
2	Up to six characters of the
3	key word or identifier name.
4	
5	
6	High address of next record.
7	Low address
8	Type flag (name), token (key)

TABLE 6.6: Constants Describing Format of Symbol Table Records

Identifier	Value	Meaning
TABLES	$7000	Address for Start of Tables
TABLHI	$70	High Part of Start Address
OVER	$7B	High Part of Overflow Address
MAX	$05	Maximum of six characters stored
LINK	$06	Address for next record stored in Positions seven and eight
FLAG	$08	Key Word Token, or Identifier Type is Stored in the ninth position
SIZE	$09	Size of Record

Figure 6.3 is a subroutine map for the routines which access TABLES. INKEY is called by PRIME, the other root routines of this map are called from ADVANC.

FIGURE 6.3: Subroutine Map For The Symbol Tables



These routines can be subdivided into three groups; 1/ Table Searching Routines: SRCHKY, SRCHNM, SEARCH and CMPARE; 2/ Table Building Routines: NEWREC, INKEY and INNAME; 3/ and Table Reading Routines: RDKEY1, RDKEY2, RDKEY and RDNAME.

6.2.1 Table Searching Routines

Three Zeropage variables are used for indirect indexed addressing of the contents of the symbol tables; ZRECRD points to the record currently being examined; ZNAMFG points to the last identifier record found by the list searching routine, SEARCH; and ZNEXT indicates the next empty record to be used in building the tables.

The variable FOUND is a flag which is set to indicate the result of a search for a key word or identifier in the symbol tables. There are two possible values:

00/ The item was found, ZRECRD has been set to point to its record in the tables.

01/ The item has not been found in the symbol tables.

SRCHKY, which supervises the search for a key word, and SRCHNM which supervises the search for an identifer, both use the routine MAP to identify a linked list. If that list is not empty, they will set ZRECRD to the address of its first record, and call the list searching routine, SEARCH. SRCHNM differs from SRCHKY in that it must first determine if the variable is a reserved or illegal name, [discussed at the beginning of Chapter 2]. A fatal error is issued if: an attempt is made to declare one of the reserved identifiers: A, X, Y, S or P; if an attempt is made to use S or P in the statement list; or if an identifier beginning with "X" is encountered. When A, X or Y is encountered in the statement list, CURITM is set to REG for the X and Y registers, and to ACC for the accumulator.

The process of searching a single linked list for a key or identifier is supervised by the routine SEARCH. ZRECRD initially points to the head of the list, SEARCH will reset ZRECRD from the link field of the record it points to until the comparison routine, CMPARE, returns FOUND = 0 or the list has been completely checked.

```
procedure  SEARCH;
begin
   loop
      gosub CMPARE;
      if  FOUND = 0  then  exitloop
      else  LINK -> Y;                    ! try next record !
            if  ind ZRECRD,Y /= 0  then
                ind ZRECRD,Y -> SAVHI;   inc Y;
                ind ZRECRD,Y -> ZRECRD lo;   SAVHI -> ZRECRD hi;
            else  exitloop   endif        ! end linked list !
      endif
   endloop
   return
end
```

The actual comparison of an item in STBUF1 with the contents of a table record is done by CMPARE. Before execution ZRECRD will point to the record, X and SAVX will indicate the item's position in STBUF1. If the item does not match, X will be restored to this value, so that the item can be compared to the next record. The variable FOUND will be set to indicate whether or not a match was made.

6.2.2 Table Building Routines

The key words are entered into TABLES by INKEY which is called from PRIME, [Section 4.1], during the initialization of the compiler. MAP is used to find the required linked list then the routine NEWREC discussed below is called to enter the key word into that list.

Identifiers are entered by INNAME during the translation of the declarations. SRCHNM is called to determine whether the identifier has already been entered, if it has a warning is issued and the initial declaration will stand. Otherwise NEWREC is called to enter the new record.

Every key and identifier in TABLES is entered by the routine NEWREC into the next available record. The records used have been outlined in Figure 6.2 and consist of: a six byte name field; a two byte link field and a single byte flag field.

The key word or identifier will be read from the STBUF1 into the name field. If the name is shorter the rest of this field is filled with zeros, if it is longer the rest of the name is ignored.

The new record is entered at the beginning of a linked list by putting its address into the pointer vector NAMVCT or KEYVCT and by loading its link field with the address of its succeeding record in that linked list. NEWREC also puts the address of the new record into ZNAMFG.

Lastly, the value in TYPFLG is put into the FLAG field of the new record. TYPFLG will have been set to a key word's token by PRIME or an identifiers type flag by DCLARE.

6.2.3 Table Reading Routines

The three routines which access the completed tables are: RDKEY1 which searches for a key word which has occurred in the source code delimited by single quotes; RDKEY2 which searches for a key word which has been given in lower case letters; and RDNAME which searches for an identifier.

RDKEY1 and RDKEY2 simply set LEGAL to the address of CHRKY1 or CHRKY2 respectively, then call RDKEY.

```
procedure  RDKEY;
begin  gosub SRCHKY;
       if  FOUND = $00  then  FLAG -> Y;
           ind ZRECRD,Y  ->  NXTSYM;           ! key token !
       else  goto FATAL($02)  endif
       return
end
```

The function of RDNAME is complicated by the fact that statement labels are not declared, and are therefore not stored in the symbol tables. RDNAME assumes any undeclared identifer, which is not a reserved name, [Section 6.2.1], is a statement label. Thus:

```
if  the identifier is found in the tables  then
      put its type into NAMFLG;
      put the address of its record in the tables into ZNAMFG.
orif  CURSYM is a terminating statement delimiter, [Section 2.4]  then
      put the name in the buffer LABLIN to be used as line label;
      LINLAB -> CURITM;
else  put the name in the buffer LABJMP to be used as an operand;
      JMPLAB -> CURITM;
endif
```

6.3 Reading Numeric Literals

Each numeric literal encountered is copied from the buffer STBUF1 to the buffer ITMBUF under the supervision of RDNUMX which is called from ADVANC. When RDNUMX is called the variable ITMBFY will contain the displacement of the next available position in ITMBUF. This value is put in the Y register which is then used for indexed addressing of ITMBUF. After RDNUMX has been executed, the variable CURITM will be set to BYTE1, BYTE2 or NUMBER, [Table 6.1].

6.4 Scanning The Source Code

A crude algorithm for ADVANC was given at the beginning of this chapter. The use of CHRFLG for recognition of the type of each item to be read was developed in Section 6.1, and the routines for reading each item type have been discussed in Sections 6.1 to 6.3. There is an additional routine within ADVANC, called ADVKEY which is called after both RDKEY1 and RDKEY2 to detect: the prefix operators, **dec** and **inc**; and the absolute or indirect addressing operators, **loc** and **ind**. In the case of prefix operators the operand will not be read during the current call of ADVANC, but CURITM will be set to PREOP.

```
procedure  ADVKEY;
begin
   if  CURITM = $00  then
       if (NXTSYM = LOC) or (NXTSYM = IND)  then  ADDRESS -> CURITM
       orif  (NXTSYM and $F0) = $90  then  PREOP -> CURITM  endif
   endif
end
```

NXTSYM will be given a new value every time ADVANC is called. This value may be: a key word token found by RDKEY1 or RDKEY2; an ACSII character such as ";" read by GETSYM; or an operator such as "+" or "->" for which the token must be calculated by GETSYM. This last case has already been discussed in Section 6.1. GETSYM must also set CURITM to STRING when the string flag (") is encountered.

```
procedure GETSYM;
    ! finds NXTSYM when the next symbol is not a key word !
constant   STRFLG = $22;   ! ASCII code for (") !
begin
    STBUF1,X  -> NXTSYM;       inc X;
    if  NXTSYM = STRFLG  then  ! literal character string !
       if CURITM = $0 then   STRING -> CURITM  endif
    orif  CHRFLG  /=  $0F  then
       if  ((CHRFLG < $06)  and  (STBUF1,X = "="))  or
           ((CHRFLG = $08)  and  (STBUF1,X = ">"))
       then   inc CHRFLG;   inc X;    endif
       CHRFLG  or  $80  ->  NXTSYM;
    endif
end
```

The algorithm for ADVANC will now be completely described. The basic function of this routine is to obtain the next symbol-operand-symbol triplet, and it will be used by virtually every translating routine discussed in Chapter four.

```
procedure ADVANC;

begin
   $0 -> CURITM;    NXTSYM -> CURSYM;

   if (CURSYM /= "[") and (CURSYM /= END) then

     loop

       case STBUF1,X

         of CR:     gosub NEXTLN;

         of ESC:    goto FATAL($06);

         of KEY:    inc X;   gosub RDKEY1;   ! 'KEY' format found !

                    gosub ADVKEY;   exitloop;

         of SP:     inc X;

         other  gosub CHRTYP(STBUF1,X);

                with  CHRFLG  do

                   if BIT7 = 1 then                        ! letter !

                     if  BIT5 = 1  then  NAME -> CURITM;

                       if  DCLFLG = 1  then  gosub RDNAME

                       else   gosub INNAME   endif

                     else  gosub RDKEY2;   gosub ADVKEY;   exitloop

                     endif

                   orif  BIT6 = 1  then   goto FATAL($04)

                   orif  BIT4 = 1  then   gosub RDNUMX

                   else  gosub GETSYM;   exitloop   endif

                endwith
       endcase

     endloop

   endif

 end
```

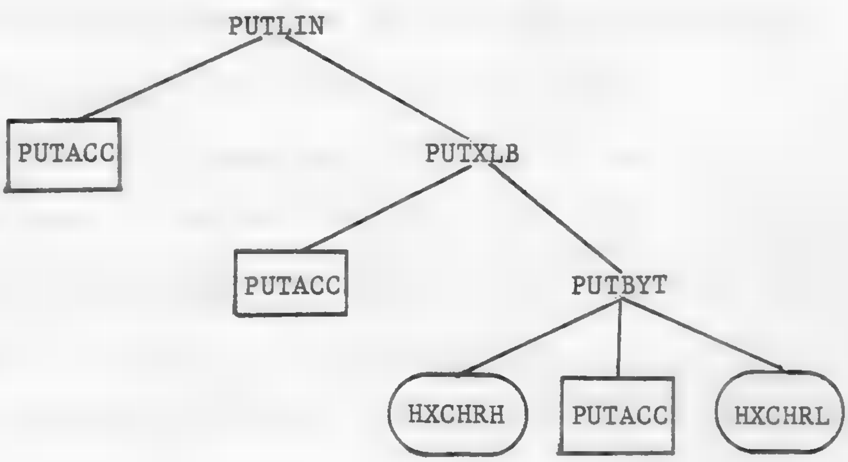
CHAPTER 7

GENERATION OF OBJECT CODE

The final step in the translation of any portion of a GRASSHOPPER program is the output of code to the object program. The output of each line of object code is supervised by the routine PUTLIN except where an assembler code insert has been used, in which case INSERT is called. A major part of the task of the translating routines discussed in chapter four is the preparation of information to be used by PUTLIN.

The format of object code produced by the translator, and its output using PUTLIN is discussed in this chapter. Figure 7.1 shows a subroutine map for the routines used by PUTLIN. In addition to these there are two groups of utility routines discussed in sections 7.2 and 7.3 which are used by the translating routines to to prepare a line of object code.

FIGURE 7.1: Subroutine Map For PUTLIN



7.1 The Object Code

Each line of code output by PUTLIN contains the following fields: a blank line number consisting of two bytes of value zero; a line label field which may simply be eight spaces, or a space then a line label of up to six letters followed by enough spaces to fill the eight byte field; an operator field of variable length followed by a single space; an operand field of variable length; and finally a carriage return to indicate end-of-line.

Before PUTLIN is called by a translating routine, information regarding the label, operator and operand fields must be placed in the variables ZLABEL, ZITEM, ITMLEN and OPDISP, and often in one or more of the buffers: ITMBUF, LABLIN or LABJMP.

7.1.1 The Label Field

A label on a line of object code may fall into one of three categories: identifiers as line labels; user statement labels; and system line labels. The first type only occurs during translation of the declarations when an identifier found in TABLE may be used as a line label to reserve its location in the data space. A user statement label originates in the GRASSHOPPER source code and will have been placed in the buffer LABLIN by RDNAME.

System line labels are generated by the translator and are more thoroughly discussed in section 7.2. When encountered, PUTLIN calls the routine PUTXLB which outputs the letter "X", followed by the contents of the variables XKIND and XCRRNT.

The two-byte variable ZLABEL, located on page zero, gives information on the existence and location of a line label in the following way:

case ZLABEL **hi** ! PUTLIN !

of $0: no line label, output eight spaces;

of $FF: system line label, call PUTXLB;

other

there is either a user statement label, or an identifier used as a line label. Its address in LABLIN or TABLE has been stored in ZLABEL. PUTLIN will access the label to be output by using ZLABEL for indirect indexed addressing;

endcase

7.1.2 The Operator Field

The Operator field will always be used and will contain an item from the buffer OPLIST found at the displacement given in the variable OPDISP. The operator will be either an opcode mnemonic or an assembler directive.

7.1.3 The Operand Field

The contents of the operand field may be:

1/ non-existent;

2/ a system line label used as a branch operand;

3/ a user statement label to be read from the buffer LABJMP;

4/ the name of an identifier which is stored in TABLE;

5/ the contents of ITMBUF which may be; a literal number; an identifier with additional characters for indexed addressing or for indirect indexed addressing; or an identifier as part of a value calculation.

In cases 3, 4 and 5 the address of the item to be used as operand will be stored in ZITEM which will be used for indirect indexed addressing of that item. In general, the variable ITMLEN characterizes the operand to PUTLIN in the following way:

case ITMLEN

of $FF: there is no operand, no action;

of $FE: system label is used as operand, call PUTXLB;

other

ITMLEN gives the length of the operand which is stored at the address given in ZITEM;

endcase

7.2 System Line Labels

System line labels are labels which have been created by the translator to be inserted into the object code. The format of these labels must insure that the labels generated be unique, never conflict with user identifiers or labels, and that a very large number of labels be possible. A six character label format was chosen which consists of: first the letter "X"; second a letter which indicates what construct generated the label; followed by a four place hexidecimal number which gives the sequence in which the labels are generated.

These labels are pushed into a F.I.L.O stack, SYSTLB, when created, and pulled when needed. SYSTLB is a 2^8 byte buffer which can contain up to 85 entries, overflow of this stack will result in an error stop. The variables required to use and maintain the stack are: XKIND which is assigned the classifying letter; XNEXT, a two byte variable which is incremented each time a new label is created; XCRRNT, a two byte variable which is given the value of the numerical part of a label being pulled from the stack; and XPOINT which gives the current displacement in the stack.

The following nine routines are available for accessing and manipulating the stack, but only two of them operate directly on SYSTLB: XPUSH and XPULL.

XPUSH pushes XKIND and XNEXT into the stack, increments XPOINT by three to point to the next available position, and increments XNEXT by one. If the stack will overflow then XPUSH causes an error stop.

XPULL pulls the top label from the stack, placing it in XKIND and XCRRNT. XPOINT is decremented by three to point to the next label in the stack. An attempt to pull from an empty stack results in an error stop.

XPSHLB and XPSHOP both set XCRRNT from XNEXT then call XPUSH, note that XNEXT must be set before these routines are called. In both cases the new label is going to be used in the next line of object code, as line label in the first case and as operand in the second. Thus, XPSHLB sets ZLABEL hi to $FF and XPSHOP sets ITMLEN to $FE.

XPLJMP and XPLLAB are called with an expected value for XKIND in the accumulator. This value is saved then XPULL is called to pull the most recent label from the stack. If the type of the pulled label agrees with the saved expected value then one line of object code is emitted, otherwise a nesting error has occured and an error stop results. The object code emitted is:

JMP LABEL or

LABEL =* respectively.

XPLBNC performs the same functions as XPLLAB except that there is no check made of label kind.

XFNDJP is called to search for the most recent occurence of a label of a particular type, the type specification is received in the accumulator. XFNDJP saves the current value of XPOINT and calls XPULL repetitively until a label of the required type is found or the stack is empty. The latter case is a nesting error and results in an error stop, otherwise a line of object code is emitted:

JMP XLABEL

then the original value of XPOINT is restored so that in effect no label is pulled from the stack.

In addition to the above, there is a routine XNOPSH which sets XCRRNT from XNEXT, sets ITMLEN to $FE then calls PUTLIN to emit a line of object code with the new label as operand. The operator must be set before XNOPSH is called. Note that the new label is not pushed into the stack.

7.3 To Output A Line Of Object Code

PUTXLB is called from PUTLIN when a system line label is encountered either as an operand or line label. It will output to object; the letter "X"; the letter found in XKIND; then the four hexidecimal characters representing the value found in XCRRNT. HXCHRH and <u>HXCHRL</u> are used to output the ASCII characters for the high and low parts respectively, of a hexidecimal number.

```
procedure   PUTXLB;

     procedure  PUTBYT(A);

     begin   gosub HXCHRH(A);    gosub PUTACC(A);

             gosub HXCHRL(A);    gosub PUTACC(A);

     end

begin    gosub PUTACC("X") ;     gosub PUTACC(XKIND);

         gosub PUTBYT(XCRRNT hi);     gosub PUTBYT(XCRRNT lo);

end
```

<u>PUTLIN</u>, which supervises the output of each line of object code, will now be more precisely described:

```
procedure PUTLIN;

begin

   $40  ->  OTFLAG;    gosub PUTACC(SP);

   case ZLABEL hi                                  ! put the LABEL !

      of $0:  gosub PUTACC($F9);                    ! put 8 spaces !

      of $FF:  gosub PUTXLB;                    ! system line label !

               $0  ->  ZLABEL hi;   gosub PUTACC(SP);

      other                                        ! user line label !

         $0 -> Y;

         loop

            if  ind ZLABEL,Y = $0 then

               gosub PUTACC(SP);  gosub PUTACC(Y + $FA);  exitloop
```

```
          else  gosub PUTACC(ind ZLABEL,Y);
              if  Y = MAX then   gosub PUTACC(SP);   exitloop
              else  inc Y  endif
          endif
       endloop
       $0 -> ZLABEL hi
   endcase
                                                 ! put the OPERATOR !
   OPDISP  ->  Y;
   loop  gosub PUTACC(OPLIST,Y);
         if OPLIST,Y = SP  then  exitloop   else  inc Y  endif
   endloop
                                          !  put the OPERAND, if any !
   if   ITMLEN  =  $FE then  gosub PUTXLB
   orif ITMLEN /=  $FF then
        if  IMMFLG = $0  then                     ! immediate operand !
            gosub PUTACC("#");   $01 -> IMMFLG  endif
        $00  ->  Y;
        loop  if  ind ZITEM,Y = $00 or Y > ITMLEN  then  exitloop
              else  gosub PUTACC(ind ZITEM,Y);   inc Y   endif
        endloop
   endif
   $FF  ->  ITMLEN;   gosub PUTACC(CR);   SOTFLG  ->  OTFLAG;
end
```

Note that the last character output is the carriage return which signals end-of-line, and that the flags ZLABEL and ITMLEN are both reset to indicate empty fields.

There are nine routines used by the translating routines when preparing a line of object code. These routines apply to frequently occurring types of operands and line labels and need only brief explanation here.

SETITM puts the address of the buffer ITMBUF into ZITEM, so that the contents of ITMBUF will be the operand.

SETLAB puts the address of the buffer LABLIN into ZLABHI, so that the line is labelled with a user line label.

SETZOP stores "00" in ITMBUF as operand and sets ITMLEN to $01.

STLBOP puts the address of the buffer LABJMP into ZITEM, and sets ITMLEN to 5 so that a user line label is the operand.

STNMIT puts the identifier found in the TABLE record found at the address in ZNAMFG into the buffer ITMBUF, the current displacement in ITMBUF is put into ITMBFY, and the length of the identifier (ITMBFY - 1) is put into ITMLEN. SETITM is then called. Thus an identifier name has been put in ITMBUF and set as the operand, this is done when the identifier name is only part of the operand to be output in the object code.

STNMLB puts the address found in ZNAMFG into ZLABEL so that an identifier found in TABLE will be the line label. this is only used during the translation of the declarations.

STNMOP puts the address found in ZNAMFG into ZITEM and sets ITMLEN to 5 so that an identifier found in TABLE will be the operand.

STV16H is called when the high part of a WORD variable is to be operand. This is done by using STNMIT to put the identifier into ITMBUF, then using LITITM to add the characters "+1" so that the second byte of the variable is referenced.

LITITM is used to add strings of characters to the contents of ITMBUF. The character strings which may by used are found in the buffer LITBUF, each separated by the null character (00). The displacement of the string in LITBUF is to be put into the accumulator before LITITM is called.

CHAPTER 8

DISCUSSION

8.1 Testing The Compiler

The first part of the compiler to be written was the character recognition routine, [Section 6.1], followed by the routines required to search, build and read the symbol tables. These were tested with short, specific test routines, usually followed by a core dump of the symbol tables and of the zero page variables. Preliminary versions of ADVANC, DRIVER and FATAL were written and from this point on all new code could be tested in the environment of the current state of the GRASSHOPPER compiler.

As each new section was added to the developing compiler, it was tested using source code written in the current state of the GRASSHOPPER language, designed to cause all paths of the new code to be executed. Appendix B contains two examples of these test programs, TESTXY and TESTIF. TESTXY is devoted to the special addressing problems involving indexed operands outlined in Section 3.3 and in Table 3.3. The object code required for both the general and special cases was outlined in Tables 3.4, 3.4, 3.6 and 3.8. In TESTXY, the relational expression, simple assignment statement and the comparison statement are tested for all cases where one operand is a register and the other is an indexed operand. TESTXY is successfully compiled to correct object code.

TESTIF tests the compilation of:

if condition **then**

where the condition may be a Condition variable or a relational expression, [Section 3.5.1]. The generation of comparison and branching code is tested for the eight kinds of Condition variables, as well as for all possible cases of:

TERM RELATIONAL OPERATOR TERM;

where both of the terms are variables, and where one or the other is a register. TESTIF is successfully compiled to correct object code, and an assembled listing of its object code is included in Appendix B.

Error detection code was tested by attempting to compile bad code. After major revisions and additions were made, tested and debugged, the old test programs were brought up to date and re-compiled and their object codes quickly checked. At the time this chapter was being written, they were compiled again, and the object code checked in great detail: no errors in the compiler were found by this check.

As the compiler became more advanced, working GRASSHOPPER programs were written, compiled, their object code examined and tested. The purpose of these programs was to test the usability of the language itself, and to test the compiler with real programs. These programs were generally re-written as the compiler became more powerful. The most important of these was the GRASSHOPPER version of SRCMGR, [Section 5.3], which eventually replaced the SRCMGR program originally written in assembler.

When I decided to re-write FATAL, [Section 4.5], to give a more readable error dump, I wrote it in GRASSHOPPER. This was done mainly because there was no reason to continue writing in assembler when GRASSHOPPER had become a usable programming language, but also to show that programs originally written in assembler code and programs compiled from GRASSHOPPER could be successfully linked. Thus, the compiler has been tested successfully with two programs in practical use.

The GRASSHOPPER program, GRSHOP, shown in Figure 2.1 is compiled to produce the assembler code object program shown in Figure 3.1. Examination of the object code, and comparison to the same program written in assembler will show an increase of 13 bytes in the version compiled from GRASSHOPPER. This increase was caused by four unneccessary jumps generated in the object code and the inefficient compilation of one of the assignment statements.

The first of the unneccessary jumps occurs in the following section of source code:

```
          Source Code                  Object Code
      if  X = TOTAL  then               CPX #TOTAL
                                        BNE XF0009
        goto DOS                        JMP DOS
      else...                           JMP XG0008
                                 XF0009 =*
```

Obviously the second jump generated is redundant, as is the jump to the operating system inserted at the end of the program by FINISH, [Section 4.1], in this program. Both of these could be easily found and eliminated by a second optimizing pass, saving six bytes in the object code.

The other two unneccessary jumps are not immediatlely obvious, and would not be easily found by an optimizing pass. Each of the two statement lists in the Case construct in Figure 2.1 when compiled, are terminated by a jump to the end of the Case construct, [Section 3.5.2]. In this program these jumps are never executed because the statement lists are enclosed in Loop constructs which contain no Exitloop statements. NEXT, called in each loop, will transfer control to the operating system when the program is complete.

The remaining extra byte results from the way the following piece of code is compiled.

Source Code	Actual Object Code	Better Object Code
X + $1 -> SAVX;	TXA	STX SAVX
	CLC	INC SAVX
	ADC #$1	
	STA SAVX	

It is possible to write the compiler to detect such cases and generate code accordingly, this was beyond the scope of this project but could be attempted as an extension to the compiler. The actual cost is small, but if a programmer is really cramped for space, the addition could be written:

```
X -> SAVX;    inc SAVX;
```

which would be compiled to the object code listed as "Better Object Code", above.

8.2 Use Of GRASSHOPPER

The purpose of developing GRASSHOPPER was to provide a language which could be used for systems programming on the MOS 6502 microcomputer. It is a readable language, with the structured constructs and strict data-typing of many high-level languages. On the other hand it retains the flexibility and much of the efficiency of assembler code, which is required in a language to be used for systems programming. As was mentioned in the previous section, GRASSHOPPER was used successfully in the re-writing and expanding of two compiler programs.

GRASSHOPPER, in its current state, is not suitable as a teaching language. I have assumed that the user has a good understanding of assembler code programming, and a new programmer could too easily get into trouble using GRASSHOPPER. There are changes which would make it more suitable for the student programmer. HEADER could be altered to restrict the starting address to over $3179, so that the operating system is not over-written.

The indirect indexed and indexed indirect operands are currently written:

ind ZEROPG,Y;

ind ZEROPG,X;

respectively, [Section 2.1.4]. This could be made more explicit by changing the syntax to:

ZEROPG,**ind**,Y;

ZEROPG,X,**ind**;

While this would result in more fatal syntax errors, it may reinforce the difference in the addressing modes.

The direct access to the accumulator will be a dangerous source of error if the user does not fully understand assembler code programming. If GRASSHOPPER should ever be used for teaching purposes, it sould be altered so that the accumulator cannot be explicitly accessed by the user.

With these changes, GRASSHOPPER could be useful as a teaching language, where the student has had experience programming in a high level language and is starting to learn the techniques of microcomputer programming. It is a simple, readable language, which can be easily learned. Its data types, data addressing modes and data manipulation operations are those which are available to the assembler code programmer on the MOS 6502. Thus a student could be introduced to working in the environment of a microprocessor without having to learn the assembly language itself.

8.3 How The Language Be Further Developed

The greatest deficiency in the current implementation of GRASSHOPPER is the primitive state of subroutine calls. Parameter passing and the developement of function subprograms would both be useful.

The data manipulation statements could be expanded in several ways, including the implementation: of Word operand arithmetic; of multiplication and division operations; and of complex assignment statements. In the relational expression and the comparison and assignment statements:

REGISTER OPERATOR REGISTER

is currently not allowed. It may be useful to extend the compiler to permit two registers in such expressions. Another useful extension would be the implementation of a multiple assignment, so that:

Expression -> RESULT1 -> RESULT2 -> RESULT3;

would be a legal assignment format. Also, GRASSHOPPER could be easily extended to allow numbers of base ten.

The WITH construct described in the introduction to this report, and used in algorithm descriptions, could be a useful addition.

A second pass to the compiler should be written, both for the purposes of error checking and optimization. Error checking could then be done on line labels and for branch out of range, the latter of which could be corrected by the second pass. Optimization would include searching for consecutive jumps with no path to any but the first, and storing from a register then reloading the same operand into the same register. Checking for branch out of range and the separate optimization of object code would have been made difficult in the first pass by the assembler code inserts. A second pass which would read all of the assembler code object program, both compiled and inserted, would be easier to write.

Error recovery should be attempted, so that a parsing scan of the source code can continue after a fatal error has been detected.

APPENDIX A

INDEX OF ROUTINES

Name	Described	Important References
ADVANC	6.4	CH.4, 5.3, CH.6, CH.5, 8.1
ADVKEY	6.4	
ADVPRO	4.3	
CHRKY1	6.1	6.2.3
CHRKY2	6.1	6.2.3
CHRNAM	6.1	
CHRTYP	6.1	
CLEGAL	6.1	
CMPARE	6.2.1	
DCLARE	4.3	4.0, 4.1, 6.2.2
DCLARR	4.3	
DCLCON	4.3	
DCLCOP	4.3	
DCLST	4.3	
DCLV8	4.3	
DCLV16	4.3	
DCLZER	4.3	
DRIVER	4.1	4.3, 4.4, 8.1
FATAL	4.5	numerous
FINISH	4.1	5.1, 5.4
GETSYM	6.1, 6.4	6.0
HEADER	4.1	4.0, 8.2
INKEY	6.2.2	4.1
INNAME	6.2.2	4.1, 4.5, 6.0, 6.4
INSERT	5.3	4.3, 4.4, 4.5
INSTRG	5.3	4.2
LITITM	7.3	
MAP	6.2	
NEWREC	6.2.2	
NEXTLN	5.3	4.5, 6.0, 6.4
OPB1LT	4.2.1	4.3
OPB1VR	4.2.1	
OPB2LT	4.2.1	
OPBYT1	4.2.1	
OPJMP	4.2.1	
OPREG	4.2.2	
PRIME	4.1	5.1, 5.4, 6.2
PUTACC	5.4	7.3

Index Of Routines, Continued

APPENDIX B

EXAMPLES OF TEST PROGRAMS

FIGURE B.1: Test Program, For Special Addressing Of Indexed Operands

```
'PROGRAM' $4500;
[;                                                      TESTXY ]

'ARRAY'    ARRAY ($02) = ($01, $02);

'ZEROPAGE' 'AT' $50, ZEROPG;

'BEGIN'
[;                                      Relational Expressions ]
'IF' A = ARRAY,X 'THEN'[;                 A = ARRAY,X] 'ENDIF'
'IF' ARRAY,X = A 'THEN'[;                 ARRAY,X = A] 'ENDIF'
'IF' X = ARRAY,X 'THEN'[;                 X = ARRAY,X] 'ENDIF'
'IF' ARRAY,X = X 'THEN'[;                 ARRAY,X = X] 'ENDIF'

[;                                Simple Assignment Statements ]
[;                                        ARRAY,REG -> REG ]
  ARRAY,X -> A;      ARRAY,Y -> A;
  ARRAY,X -> X;      ARRAY,Y -> X;
  ARRAY,X -> Y;      ARRAY,Y -> Y;

[;                                       ZEROPG,REG -> REG ]
  ZEROPG,X -> A;     ZEROPG,Y -> A;
  ZEROPG,X -> X;     ZEROPG,Y -> X;
  ZEROPG,X -> Y;     ZEROPG,Y -> Y;

[;                                 'IND' ZEROPG,REG -> REG ]
  'IND' ZEROPG,X -> A;    'IND' ZEROPG,Y -> A;
  'IND' ZEROPG,X -> X;    'IND' ZEROPG,Y -> X;
  'IND' ZEROPG,X -> Y;    'IND' ZEROPG,Y -> Y;

[;                                        REG -> ARRAY,REG ]
  A -> ARRAY,X;      A -> ARRAY,Y;
  X -> ARRAY,X;      X -> ARRAY,Y;
  Y -> ARRAY,X;      Y -> ARRAY,Y;

[;                                       REG -> ZEROPG,REG ]
  A -> ZEROPG,X;      A -> ZEROPG,Y;
  X -> ZEROPG,X;      X -> ZEROPG,Y;
  Y -> ZEROPG,X;      Y -> ZEROPG,Y;
```

```

[;                                REG -> 'IND' ZEROPG,REG ]
   A -> 'IND' ZEROPG,X;   A -> 'IND' ZEROPG,Y;
   X -> 'IND' ZEROPG,X;   X -> 'IND' ZEROPG,Y;
   Y -> 'IND' ZEROPG,X;   Y -> 'IND' ZEROPG,Y;
[;
;                                  Conditional Statements ]
[;                                      REG : ARRAY,REG ]
   A : ARRAY,X;       A : ARRAY,Y;
   X : ARRAY,X;       X : ARRAY,Y;
   Y : ARRAY,X;       Y : ARRAY,Y;

[;                                     REG : ZEROPG,REG ]
   A : ZEROPG,X;      A : ZEROPG,Y;
   X : ZEROPG,X;      X : ZEROPG,Y;
   Y : ZEROPG,X;      Y : ZEROPG,Y;

[;                                REG : 'IND' ZEROPG,REG ]
   A : 'IND' ZEROPG,X;    A : 'IND' ZEROPG,Y;
   X : 'IND' ZEROPG,X;    X : 'IND' ZEROPG,Y;
   Y : 'IND' ZEROPG,X;    Y : 'IND' ZEROPG,Y;

[;                                      ARRAY,REG : REG ]
   ARRAY,X : A;       ARRAY,Y : A;
   ARRAY,X : X;       ARRAY,Y : X;
   ARRAY,X : Y;       ARRAY,Y : Y;

[;                                     ZEROPG,REG : REG ]
   ZEROPG,X : A;      ZEROPG,Y : A;
   ZEROPG,X : X;      ZEROPG,Y : X;
   ZEROPG,X : Y;      ZEROPG,Y : Y;

[;                                'IND' ZEROPG,REG : REG ]
   'IND' ZEROPG,X : A;   'IND' ZEROPG,Y : A;
   'IND' ZEROPG,X : X;   'IND' ZEROPG,Y : X;
   'IND' ZEROPG,X : Y;   'IND' ZEROPG,Y : Y;
[;
;                                Prefix Operator Statements ]
[;                                            DECREMENT ]
   'DEC' ARRAY,X;          'DEC' ARRAY,Y;
   'DEC' ZEROPG,X;         'DEC' ZEROPG,Y;
   'DEC' 'IND' ZEROPG,X;   'DEC' 'IND' ZEROPG,Y;

[;                                            INCREMENT ]
   'INC' ARRAY,X;          'INC' ARRAY,Y;
   'INC' ZEROPG,X;         'INC' ZEROPG,Y;
   'INC' 'IND' ZEROPG,X;   'INC' 'IND' ZEROPG,Y;
'END'
```

FIGURE B.2: Test Program, For The Translation Of:
if condition **then**

```
 10 'PROGRAM' $4500;
 20[;                                                       TESTIF ]
 30 'BYTE'    AA = $10,  BB = $20,  CC;
 40 'CONDITION' CARRY = $0, NCARRY /= $0,
 50             ZERO  = $1, NZERO  /= $1,
 60             OVER  = $6, NOVER  /= $6,
 70             NEG   = $7, NOTNEG /= $7;
 80
 90 'BEGIN'
100[;
110;                                           CONDITION variables ]
120      AA : BB;
130     'IF'   CARRY  'THEN' [;                          CARRY  ]
140     'ORIF' NCARRY 'THEN' [;                          NCARRY ]
150     'ORIF' ZERO   'THEN' [;                          ZERO   ]
160     'ORIF' NZERO  'THEN' [;                          NZERO  ]
170     'ORIF' OVER   'THEN' [;                          OVER   ]
180     'ORIF' NOVER  'THEN' [;                          NOVER  ]
190     'ORIF' NEG    'THEN' [;                          NEG    ]
200     'ORIF' NOTNEG 'THEN' [;                          NOTNEG ] 'ENDIF'
210[;
220;                                        Relational Expressions
230;                                           None-Register Terms ]
240     'IF'   AA  = BB 'THEN' [;                      AA  = BB ]
250     'ORIF' AA /= BB 'THEN' [;                      AA /= BB ]
260     'ORIF' AA <  BB 'THEN' [;                      AA <  BB ]
270     'ORIF' AA <= BB 'THEN' [;                      AA <= BB ]
280     'ORIF' AA >  BB 'THEN' [;                      AA >  BB ]
290     'ORIF' AA >= BB 'THEN' [;                      AA >= BB ] 'ENDIF'
300[;
310;                                           With Register Terms
320;                                                   TERM = TERM ]
330     'IF'    A  = BB 'THEN' [;                        A = BB ]
340     'ORIF' BB  = A  'THEN' [;                      BB = A  ]
350     'ORIF'  X  = BB 'THEN' [;                        X = BB ]
360     'ORIF' BB  = X  'THEN' [;                      BB = X  ]
370     'ORIF'  Y  = BB 'THEN' [;                        Y = BB ]
380     'ORIF' BB  = Y  'THEN' [;                      BB = Y  ] 'ENDIF'
390[;
400;                                                  TERM /= TERM ]
410     'IF'    A /= BB 'THEN' [;                       A /= BB ]
420     'ORIF' BB /= A  'THEN' [;                     BB /= A  ]
430     'ORIF'  X /= BB 'THEN' [;                       X /= BB ]
440     'ORIF' BB /= X  'THEN' [;                     BB /= X  ]
450     'ORIF'  Y /= BB 'THEN' [;                       Y /= BB ]
460     'ORIF' BB /= Y  'THEN' [;                     BB /= Y  ] 'ENDIF'
```

```
[;
;                                             TERM < TERM ]
    'IF'    A <  BB 'THEN' [;                  A <  BB ]
    'ORIF' BB <  A  'THEN' [;                 BB <  A  ]
    'ORIF'  X <  BB 'THEN' [;                  X <  BB ]
    'ORIF' BB <  X  'THEN' [;                 BB <  X  ]
    'ORIF'  Y <  BB 'THEN' [;                  Y <  BB ]
    'ORIF' BB <  Y  'THEN' [;                 BB <  Y  ] 'ENDIF'
[;
;                                            TERM <= TERM ]
    'IF'    A <= BB 'THEN' [;                  A <= BB ]
    'ORIF' BB <= A  'THEN' [;                 BB <= A  ]
    'ORIF'  X <= BB 'THEN' [;                  X <= BB ]
    'ORIF' BB <= X  'THEN' [;                 BB <= X  ]
    'ORIF'  Y <= BB 'THEN' [;                  Y <= BB ]
    'ORIF' BB <= Y  'THEN' [;                 BB <= Y  ] 'ENDIF'
[;
;                                             TERM > TERM ]
    'IF'    A >  BB 'THEN' [;                  A >  BB ]
    'ORIF' BB >  A  'THEN' [;                 BB >  A  ]
    'ORIF'  X >  BB 'THEN' [;                  X >  BB ]
    'ORIF' BB >  X  'THEN' [;                 BB >  X  ]
    'ORIF'  Y >  BB 'THEN' [;                  Y >  BB ]
    'ORIF' BB >  Y  'THEN' [;                 BB >  Y  ] 'ENDIF'
[;
;                                            TERM >= TERM ]
    'IF'    A >= BB 'THEN' [;                  A >= BB ]
    'ORIF' BB >= A  'THEN' [;                 BB >= A  ]
    'ORIF'  X >= BB 'THEN' [;                  X >= BB ]
    'ORIF' BB >= X  'THEN' [;                 BB >= X  ]
    'ORIF'  Y >= BB 'THEN' [;                  Y >= BB ]
    'ORIF' BB >= Y  'THEN' [;                 BB >= Y  ] 'ENDIF'

 'END' ;
```

FIGURE B.3: Translation of Figure B.2

```
     4500                   *= $4500
     4500 4C0745            JMP XS0000
                  ;                                              TESTIF
     4503 00      XM0000 .BYTE 00
     4504 10      AA     .BYTE $10
     4505 20      BB     .BYTE $20
     4506 00      CC     .BYTE 00
     4507=        XS0000 =*
                  ;
                  ;                                   CONDITION variables
     4507 AD0445            LDA AA
     450A CD0545            CMP BB
     450D 9003              BCC XF0002
                  ;                   CARRY
     450F 4C3245            JMP XG0001
     4512=        XF0002 =*
     4512 B003              BCS XF0003
                  ;                   NCARRY
     4514 4C3245            JMP XG0001
     4517=        XF0003 =*
     4517 D003              BNE XF0004
                  ;                   ZERO
     4519 4C3245            JMP XG0001
     451C=        XF0004 =*
     451C F003              BEQ XF0005
                  ;                   NZERO
     451E 4C3245            JMP XG0001
     4521=        XF0005 =*
     4521 5003              BVC XF0006
                  ;                   OVER
     4523 4C3245            JMP XG0001
     4526=        XF0006 =*
     4526 7003              BVS XF0007
                  ;                   NOVER
     4528 4C3245            JMP XG0001
     452B=        XF0007 =*
     452B 1003              BPL XF0008
                  ;                   NEG
     452D 4C3245            JMP XG0001
     4530=        XF0008 =*
     4530 3000              BMI XF0009
                  ;                   NOTNEG
     4532=        XF0009 =*
     4532=        XG0001 =*
                  ;
                  ;                                 Relational Expressions
                  ;                                   None-Register Terms
     4532 AD0445            LDA AA
     4535 CD0545            CMP BB
```

```
4538 D003                 BNE XF000B
                ;              AA  = BB
453A 4C7545               JMP XG000A
453D=           XF000B =*
453D AD0445               LDA AA
4540 CD0545               CMP BB
4543 F003                 BEQ XF000C
                ;              AA /= BB
4545 4C7545               JMP XG000A
4548=           XF000C =*
4548 AD0445               LDA AA
454B CD0545               CMP BB
454E B003                 BCS XF000D
                ;              AA <  BB
4550 4C7545               JMP XG000A
4553=           XF000D =*
4553 AD0445               LDA AA
4556 CD0545               CMP BB
4559 F002                 BEQ XF000E
455B B003                 BCS XF000F
455D=           XF000E =*
                ;              AA <= BB
455D 4C7545               JMP XG000A
4560=           XF000F =*
4560 AD0445               LDA AA
4563 CD0545               CMP BB
4566 F005                 BEQ XF0010
4568 9003                 BCC XF0010
                ;              AA >  BB
456A 4C7545               JMP XG000A
456D=           XF0010 =*
456D AD0445               LDA AA
4570 CD0545               CMP BB
4573 9000                 BCC XF0011
                ;              AA >= BB
4575=           XF0011 =*
4575=           XG000A =*
                ;
                ;                             With Register Terms
                ;                                     TERM = TERM
4575 CD0545               CMP BB
4578 D003                 BNE XF0013
                ;               A = BB
457A 4CA245               JMP XG0012
457D=           XF0013 =*
457D CD0545               CMP BB
4580 D003                 BNE XF0014
                ;              BB = A
4582 4CA245               JMP XG0012
4585=           XF0014 =*
```

```
4585 EC0545            CPX BB
4588 D003              BNE XF0015
                ;              X = BB
458A 4CA245            JMP XG0012
458D=           XF0015 =*
458D EC0545            CPX BB
4590 D003              BNE XF0016
                ;             BB = X
4592 4CA245            JMP XG0012
4595=           XF0016 =*
4595 CC0545            CPY BB
4598 D003              BNE XF0017
                ;              Y = BB
459A 4CA245            JMP XG0012
459D=           XF0017 =*
459D CC0545            CPY BB
45A0 D000              BNE XF0018
                ;             BB = Y
45A2=           XF0018 =*
45A2=           XG0012 =*
                ;
                ;                                     TERM /= TERM
45A2 CD0545            CMP BB
45A5 F003              BEQ XF001A
                ;             A /= BB
45A7 4CCF45            JMP XG0019
45AA=           XF001A =*
45AA CD0545            CMP BB
45AD F003              BEQ XF001B
                ;            BB /= A
45AF 4CCF45            JMP XG0019
45B2=           XF001B =*
45B2 EC0545            CPX BB
45B5 F003              BEQ XF001C
                ;             X /= BB
45B7 4CCF45            JMP XG0019
45BA=           XF001C =*
45BA EC0545            CPX BB
45BD F003              BEQ XF001D
                ;            BB /= X
45BF 4CCF45            JMP XG0019
45C2=           XF001D =*
45C2 CC0545            CPY BB
45C5 F003              BEQ XF001E
                ;             Y /= BB
45C7 4CCF45            JMP XG0019
45CA=           XF001E =*
45CA CC0545            CPY BB
45CD F000              BEQ XF001F
                ;            BB /= Y
```

```
45CF=          XF001F =*
45CF=          XG0019 =*
               ;                                    TERM < TERM
               ;
45CF CD0545           CMP BB
45D2 B003             BCS XF0021
               ;           A <  BB
45D4 4C0246           JMP XG0020
45D7=          XF0021 =*
45D7 CD0545           CMP BB
45DA F005             BEQ XF0022
45DC 9003             BCC XF0022
               ;          BB <  A
45DE 4C0246           JMP XG0020
45E1=          XF0022 =*
45E1 EC0545           CPX BB
45E4 B003             BCS XF0023
               ;           X <  BB
45E6 4C0246           JMP XG0020
45E9=          XF0023 =*
45E9 EC0545           CPX BB
45EC F005             BEQ XF0024
45EE 9003             BCC XF0024
               ;          BB <  X
45F0 4C0246           JMP XG0020
45F3=          XF0024 =*
45F3 CC0545           CPY BB
45F6 B003             BCS XF0025
               ;           Y <  BB
45F8 4C0246           JMP XG0020
45FB=          XF0025 =*
45FB CC0545           CPY BB
45FE F002             BEQ XF0026
4600 9000             BCC XF0026
               ;          BB < Y
4602=          XF0026 =*
4602=          XG0020 =*
               ;
               ;                                    TERM <= TERM
4602 CD0545           CMP BB
4605 F002             BEQ XF0028
4607 B003             BCS XF0029
4609=          XF0028 =*
               ;           A <= BB
4609 4C3546           JMP XG0027
460C=          XF0029 =*
460C CD0545           CMP BB
460F 9003             BCC XF002A
               ;          BB <= A
4611 4C3546           JMP XG0027
```

```
4614=       XF002A =*
4614 EC0545        CPX BB
4617 F002          BEQ XF002B
4619 B003          BCS XF002C
461B=       XF002B =*
           ;            X <= BB
461B 4C3546        JMP XG0027
461E=       XF002C =*
461E EC0545        CPX BB
4621 9003          BCC XF002D
           ;           BB <= X
4623 4C3546        JMP XG0027
4626=       XF002D =*
4626 CC0545        CPY BB
4629 F002          BEQ XF002E
462B B003          BCS XF002F
462D=       XF002E =*
           ;            Y <= BB
462D 4C3546        JMP XG0027
4630=       XF002F =*
4630 CC0545        CPY BB
4633 9000          BCC XF0030
           ;           BB <= Y
4635=       XF0030 =*
4635=       XG0027 =*
           ;
           ;                                     TERM > TERM
4635 CD0545        CMP BB
4638 F005          BEQ XF0032
463A 9003          BCC XF0032
           ;            A > BB
463C 4C6846        JMP XG0031
463F=       XF0032 =*
463F CD0545        CMP BB
4642 B003          BCS XF0033
           ;           BB >  A
4644 4C6846        JMP XG0031
4647=       XF0033 =*
4647 EC0545        CPX BB
464A F005          BEQ XF0034
464C 9003          BCC XF0034
           ;            X >  BB
464E 4C6846        JMP XG0031
4651=       XF0034 =*
4651 EC0545        CPX BB
4654 B003          BCS XF0035
           ;           BB >  X
4656 4C6846        JMP XG0031
4659=       XF0035 =*
4659 CC0545        CPY BB
```

```
465C F005              BEQ XF0036
465E 9003              BCC XF0036
                 ;        Y > BB
4660 4C6846            JMP XG0031
4663=           XF0036 =*
4663 CC0545            CPY BB
4666 B000              BCS XF0037
                 ;        BB > Y
4668=           XF0037 =*
4668=           XG0031 =*
                 ;
                 ;                              TERM >= TERM
4668 CD0545            CMP BB
466B 9003              BCC XF0039
                 ;        A >= BB
466D 4C9B46            JMP XG0038
4670=           XF0039 =*
4670 CD0545            CMP BB
4673 F002              BEQ XF003A
4675 B003              BCS XF003B
4677=           XF003A =*
                 ;        BB >= A
4677 4C9B46            JMP XG0038
467A=           XF003B =*
467A EC0545            CPX BB
467D 9003              BCC XF003C
                 ;        X >= BB
467F 4C9B46            JMP XG0038
4682=           XF003C =*
4682 EC0545            CPX BB
4685 F002              BEQ XF003D
4687 B003              BCS XF003E
4689=           XF003D =*
                 ;        BB >= X
4689 4C9B46            JMP XG0038
468C=           XF003E =*
468C CC0545            CPY BB
468F 9003              BCC XF003F
                 ;        Y >= BB
4691 4C9B46            JMP XG0038
4694=           XF003F =*
4694 CC0545            CPY BB
4697 F002              BEQ XF0040
4699 B000              BCS XF0041
469B=           XF0040 =*
                 ;        BB >= Y
469B=           XF0041 =*
469B=           XG0038 =*
469B 4C512A            JMP $2A51
                       .END
```

REFERENCES

GRAHAM, R.M. [1975] Principles of Systems Programing. Toronto: John Wiley & Sons, Inc.

HALSTEAD, M.H. [1974], A Laboratory Manual for Compiler and Operating System Implementation. New York: American Eslevier Publishing Company.

LEWIS II, P.M., D.J.ROSENKRANTZ, and R.E.STEARNS [1976], Compiler Design Theory. Don Mills: Addison-Wesley.

MOS TECHNOLOGY, [1976], MCS6500 Microcomputer Family Programing Manual. Norristown, PA.: Mos Technoloty, Inc.

OHIO SCIENTIFIC, [1976], 65L-13 OSI 6502 Resident Assembler/Editor. Hiram, Ohio 44234: Ohio Scientific Instruments.

OHIO SCIENTIFIC, [1978], OS-65D C3.0 User's Manual,Preliminary Copy. Ohio Scientific, Inc.

PRATT, T.W. [1975], Programming Languages: Design and Implementation. Englewood Cliffs, N.J.: Prentice-Hall.

WIRTH, N. [1976], Algorithms + Data Structures = Programs. Englewood Cliffs, N.J.: Prentice-Hall.