THE ASPLE+ COMPILER

DESCRIPTION OF
THE ASPLE+ COMPILER

By

DARRELL CHARLES EADY, 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

September 1977

MASTER OF SCIENCE (1977) McMASTER UNIVERSITY

(Computation) Hamilton, Ontario

TITLE: Description of the ASPLE+ Compiler

AUTHOR: Darrell Charles Eady, B.Sc. (McMaster University)

SUPERVISOR: Dr. N. Solntseff

NUMBER OF PAGES: ix, 177

Abstract

The emphasis of my essay are on the description of the language syntax, listing the functions of the major routines, with emphasis put on the use of the current operator - next operator (CO - NO) tables, and a discussion on the code generated by each statement.

Acknowledgements

I wish to thank my supervisor Dr. N. Solntseff for his assistance during the work on this project. I would especially like to thank Dr. N. Solntseff and Mr. Chris Bryce for their invaluable assistance and time during the writing of this project.

Finally, I would like to thank my sister, Miss Dawn Eady for her excellent typing.

Table of Contents

CHAPTER 1

INTRODUCTION

Since virtually all compilers have much in common, it should be possible to study any good compiler for a language such as Fortran, Pascal or Cobol, and from a detailed analysis of that one compiler on a single machine, to obtain knowledge which could be readily transferred to or from the others. While possible, such a method is uneconomic in terms of effort, primarily because such compilers, in addition to their fundamentals or basic components, must or necessity contain a far larger proportion of code which is concerned only with the details of their particular environment and implementation. With that approach, the trees obscured the forest.

At the other extreme, one could distill, from the population of all compilers, those principles which they exhibit in common and study them in a completely abstract way. While this extreme holds the advantage over the other, it suffers from a paucity of detail which leaves a person with a compartmentalized knowledge which is often too

fragmentary to provide a firm base from which the person could design and implement a processor independently. The person may recognize a forest, but miss a tree.

That is why, I have chose the approach laid down by Halstead in A Laboratory Manual for Compiler and Operating System Implementation, to develope the compiler for the language ASPLE+.

Since a language was needed to write an Operating System, the language ASPLE+ has been developed.

1.1 The Design of ASPLE+

The language designed was based upon the language PASCAL.

There are three main differences between PASCAL and ASPLE+. These are

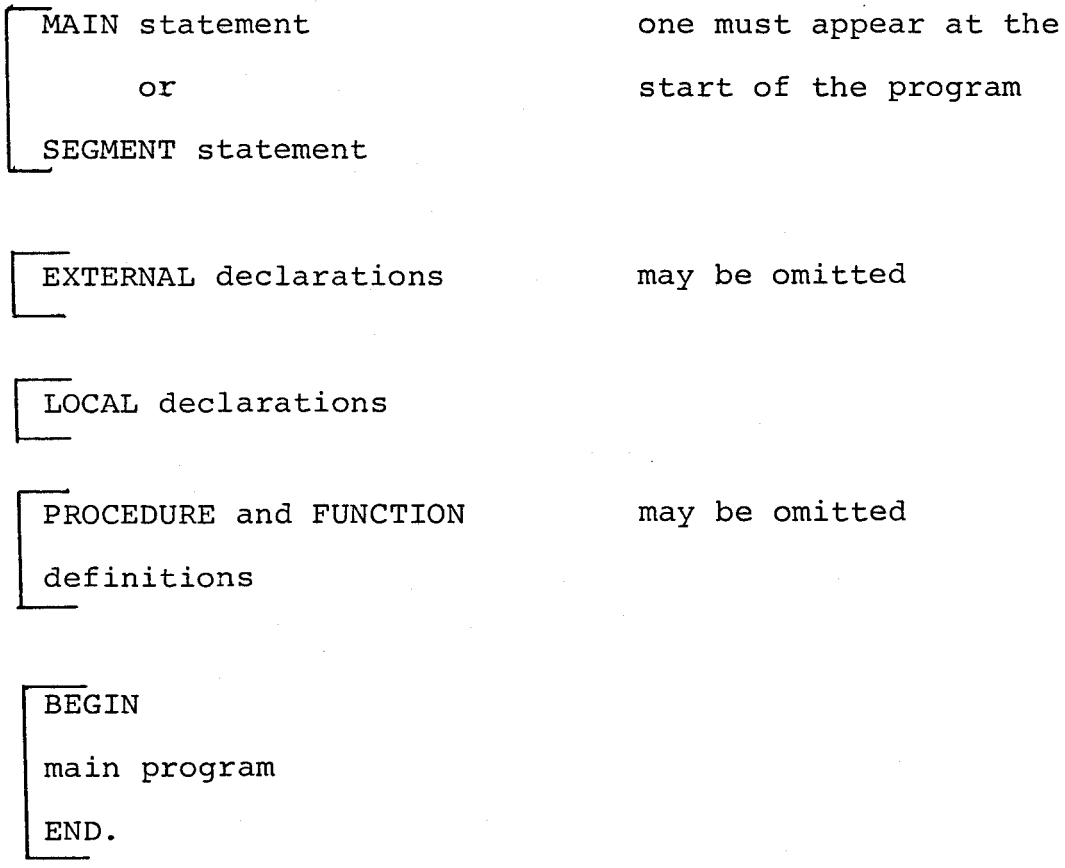
(i) ASPLE+ does not have any constant or type declarations.

(ii) ASPLE+ does not allow variables to be declared inside procedure or function definitions.

and (iii) ASPLE+ has the added feature of explicit segmentation.

Explicit segmentation was needed to simulate the Operating System feature of allowing concurrent processes to occur at the same moment in time.

Figure 1.1 shows the structural design of ASPLE+.



Structural Design of ASPLE+

Figure 1.1

1.2 Design Philosophy

The design philosophy has been based on the use of the current operator - next operator tables. These tables offer the advantages in speed of compilation, ease of extension and a staight forward error diagnostics. Both the statements and the declarations of ASPLE+, use CO - NO tables to check their syntax.

1.3 Generated Code

The code being generated by the declarations sets up the required storage space needed for each variable. It will also generate the appropriate code for variables that are entry points or declared externally.

The statements generate code in terms of marco calls. This has been done for the sake of portability, thus allowing the execution of the generated code, to be handled by any computer that has an assembly language, which is capable of calling macros.

1.4 Other Information

Before reading about the actual implementation of the compiler illustrated in chapters 2 to 11, it would be advisable to read Appendix A, which describes all global constants, types and variables used throughout the compiler.

CHAPTER 2

STRUCTURAL DESIGN OF THE

ASPLE+ COMPILER

The ASPLE+ Compiler consists of the following major subroutines.

A. NEXTCH

B. LXSCAN

C. READNAME

D. READSTRCHA

E. TESTENTRY

F. ENTERVARIABLE

G. PRINTTAB

H. ADVANCE

I. CHECKID

J. DECLARATION

K. GENCODE

L. STATEMENT

M. PROCFN

N. TESTVAR

O. BUILD

P. EMITOPCODE

Q. EMITNAME

R. EMITCHAR

S. EMITLABEL

T. EMITEOL

U. EMITNUM

V. ERROR

W. ERRORMSG

X. FATAL

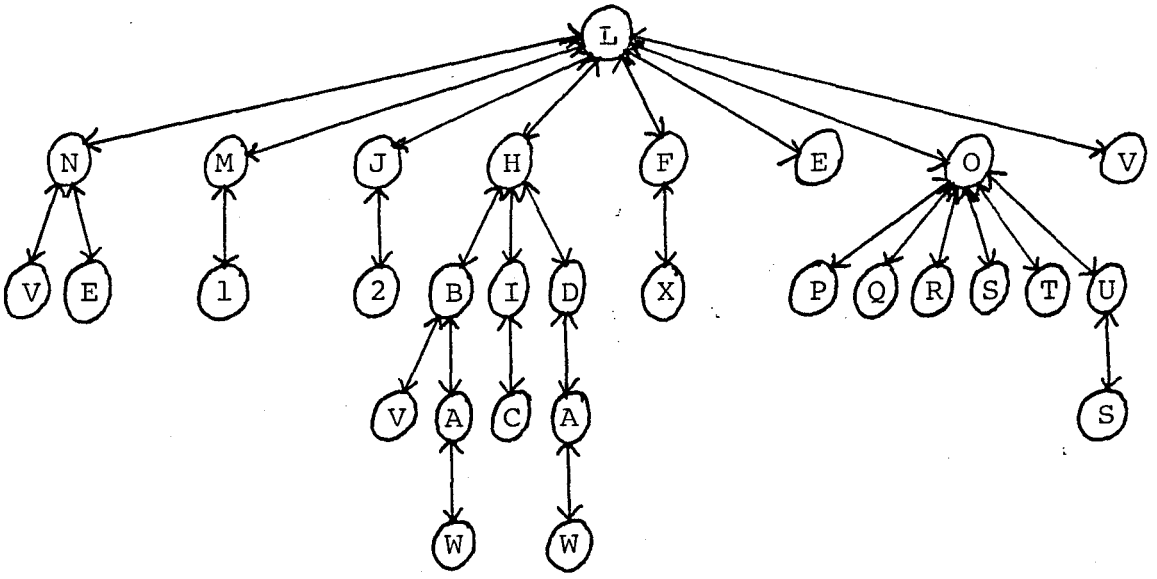
Y. INIT

The capital letters A to Y will also refer to the Operand Tables for these subroutines.

The heart of the compiler consists of the six routines NEXTCH, LXSCAN, ADVANCE, DECLARATION, STATEMENT and BUILD. The other nineteen routines are needed since they illustrate general features used in virtually all compilers. The following chapters are discussed in the order in which the subroutines were written and tested.

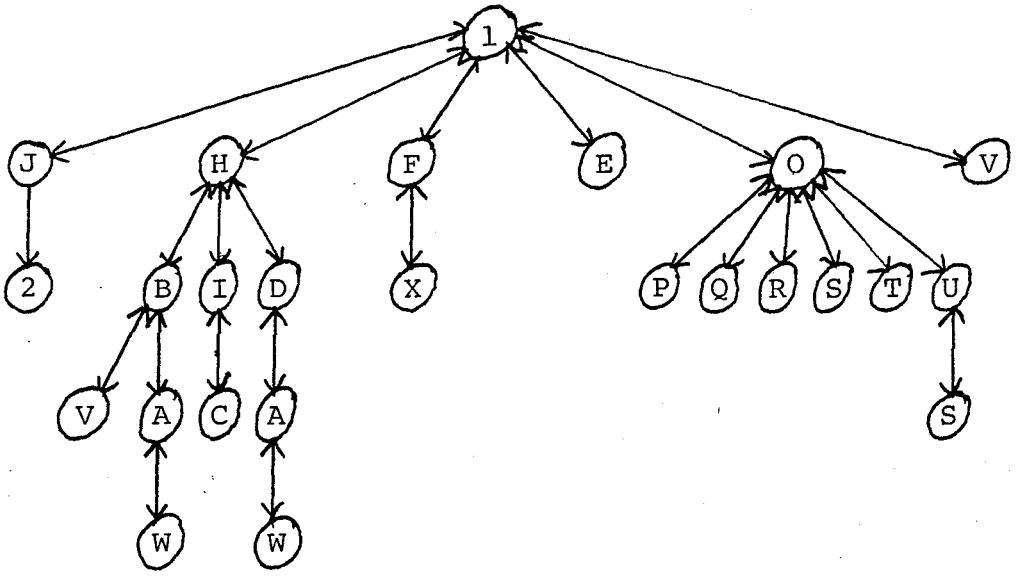
A subroutine map of the compiler is given in figure 2.1. This map is distinctly different from a flow chart but is much more useful. Since each of the modules of the

compliler is written as a closed subroutine it must return to the subroutine which called it. The map shows all possible paths among the twenty-five subroutines.



Path Map

Figure 2.1



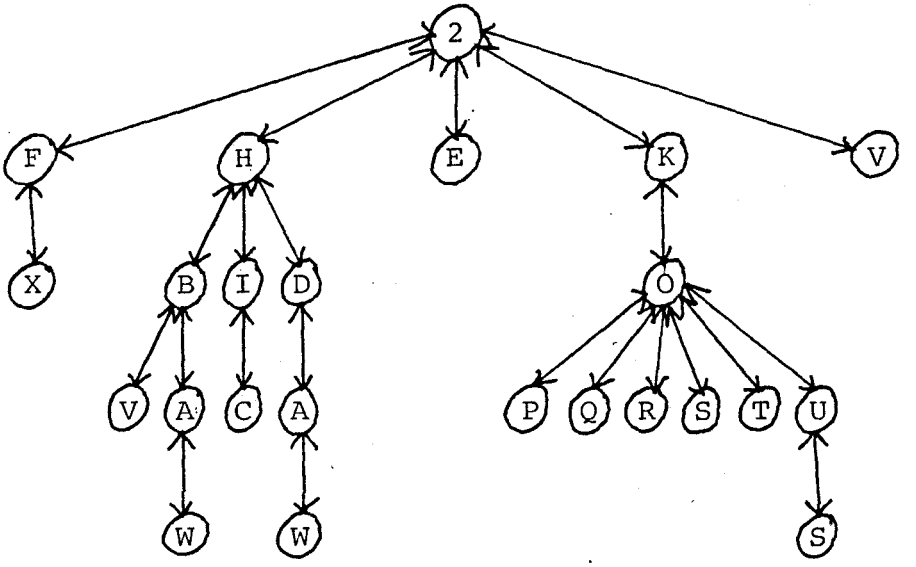


Figure 2.1 continued

CHAPTER 3

DESCRIBING THE LANGUAGE ASPLE+

This chapter describes the principal features of ASPLE+.

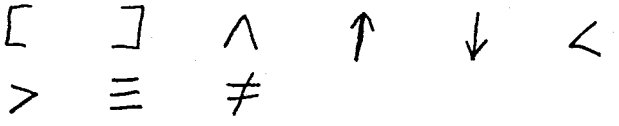
3.1 Character Set

First I would like to introduce the character set. The character set is made up of

(i) the special characters

( ) * + - /

V : ; . , =



(ii) the letters

A B C D E F G H I

J K L M N O P Q R

S T U V W X Y Z △

where △ is a blank

and (iii) the digits

0 1 2 3 4 5 6 7 8 9.

3.2 Reserved Words

ASPLE+ has forty-one reserved words, which are listed in Table 3 of Appendix B. Table 5 of Appendix B shows the special symbols used by ASPLE+.

3.3 Control Statements

Now let me introduce the six control statements. They are

(i) the Conditional statement

IF .. THEN .. ELSE .. ENDIF

(ii) the While statement

WHILE .. DO .. ENDWHILE

(iii) the Repeat statement

REPEAT .. UNTIL ..;

(iv) the GOTO statement

GOTO ..

(v) the For statement

FOR .. IN .. TO .. DO .. ENDFOR

or

FOR .. IN .. DOWNTO .. DO .. ENDFOR

and (vi) the Case statement

CASE .. ENDCASE

3.4 Assignment Statements

The assignment statements are evaluated from left to right. The assignment operator is a minus sign followed by a greater than sign (—>). The numerical operators used by the assignment statement are listed in Table 8 of Appendix B. The assignment statement deals only with simple operands. The operands can be

(i) an integer constant
(ii) an identifier
(iii) a string
(iv) a character
(v) an array
or (vi) a function call

3.5 Procedures and Functions

Procedures and functions can also be defined in ASPLE+. A procedure computes a series of values which are passed back through its parameters. Whereas, a function computes one value which is passed back in the function name. The times sign in a procedure or function definition declares an external procedure or function.

3.6 Explicit Segmentation

ASPLE+, also has the feature of explicit segmentation because it defines two types of programs that can be compiled separately. The two programs are the MAIN and SEGMENT programs. The main program calls the segment program which in turn can call another segment program, thus providing the feature of segmentation.

3.7 Declarations

All variables in ASPLE+ must be declared in either the external declarations or the local declarations. The variables are declared quite similar to Pascal.

The variables are declared as either

(i) an integer
(ii) a string
(iii) a character
(iv) a label
or (v) an array.

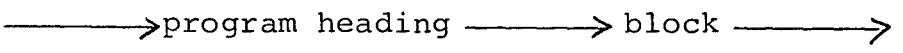
3.8 Syntax Graphs

The language syntax is defined by means of syntax graphs. A syntax graph defines the name and syntax of a language construct. The basic symbols are represented by capital letters and special characters.

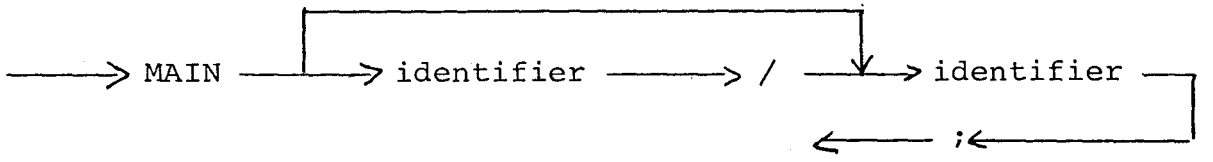
Constructs defined by other syntax graphs are represented by their names written in small letters. The correct sequence of basic symbols and constructs are formed by following the arrows.

Here are the syntax graphs for ASPLE+.

main program

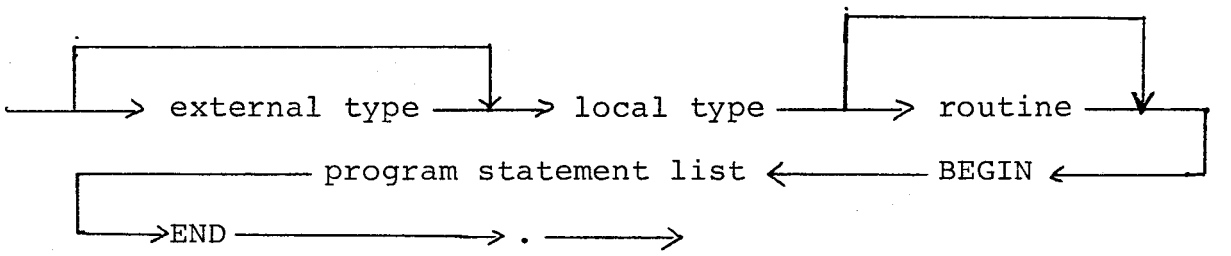


program heading

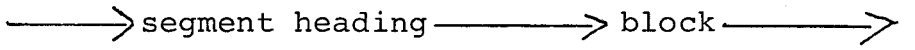


The slash indicates that a call to segment, named by the first identifier, will occur at the end of the program.

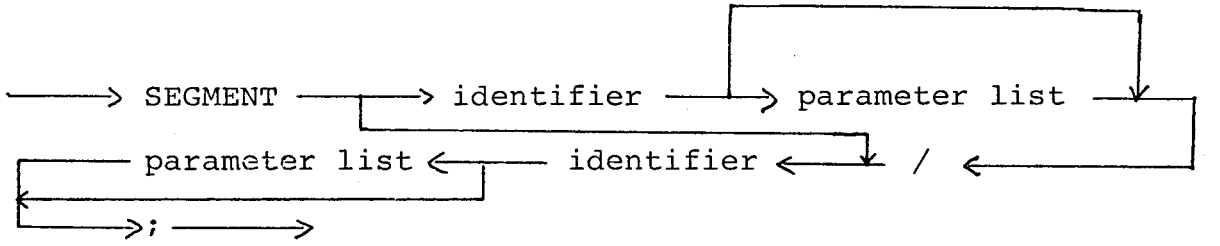
block



segment

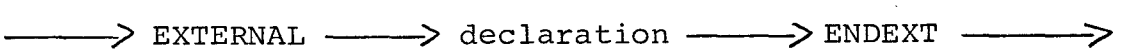


segment heading

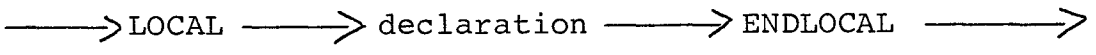


Again the slash indicates that this segment will call another segment which is identified by the first identifier.

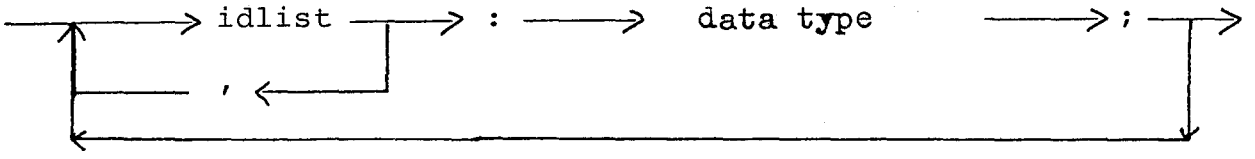
external type



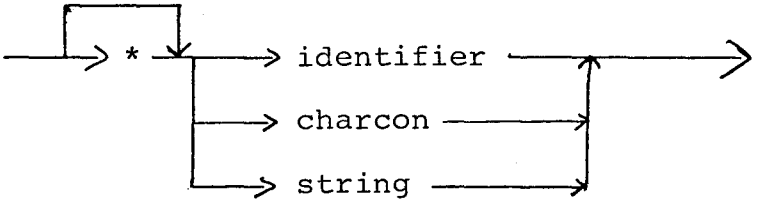
local type



declaration

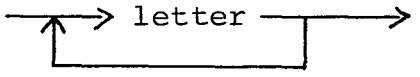


idlist

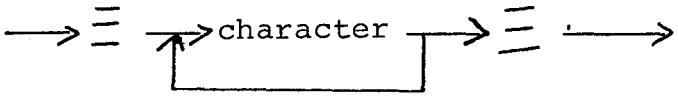


The times sign indicates the variable as an entry point.

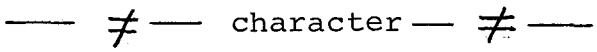
identifier



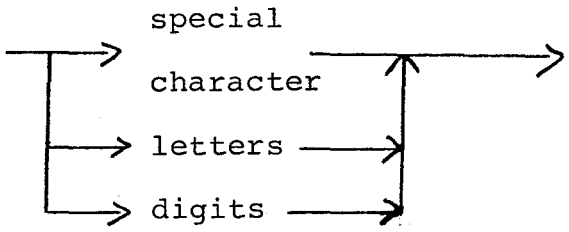
stringcon



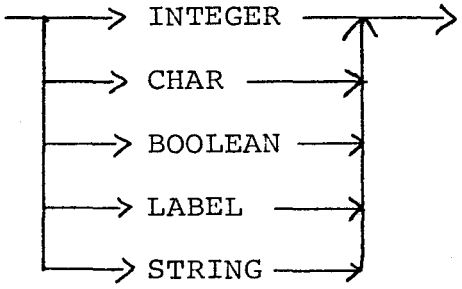
charcon



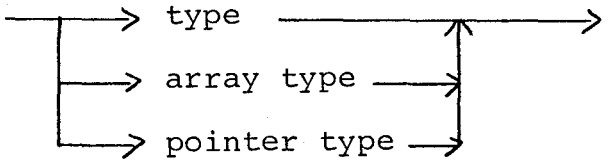
character



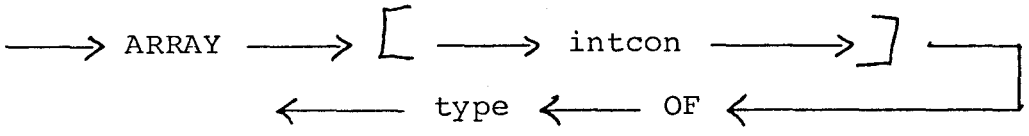
type



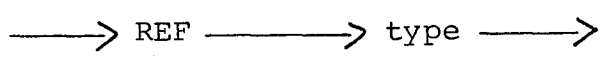
data type



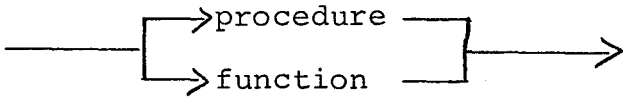
array type



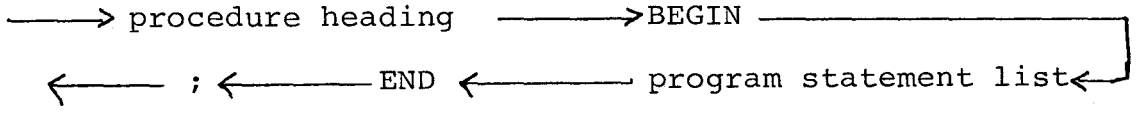
pointer type



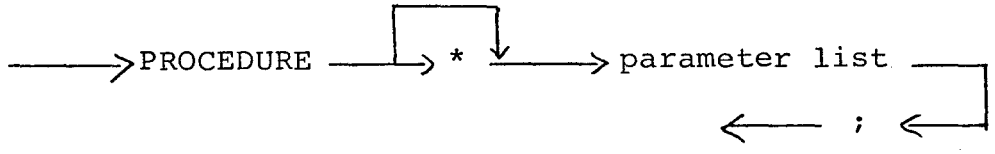
routine



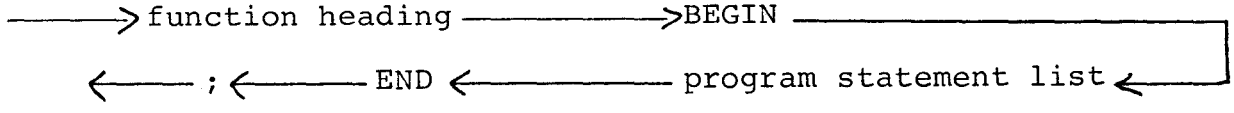
procedure



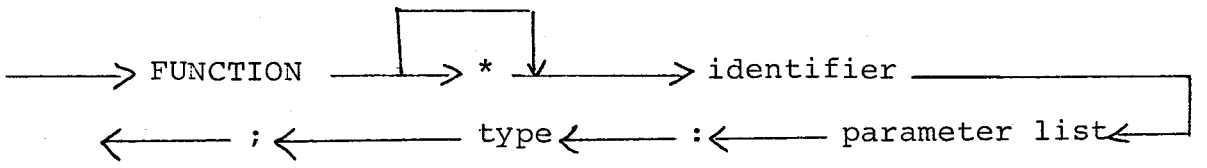
procedure heading



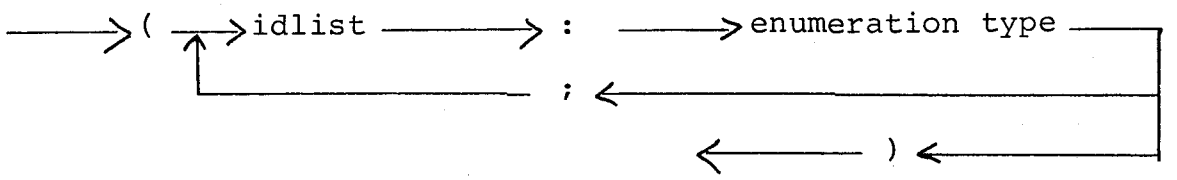
function



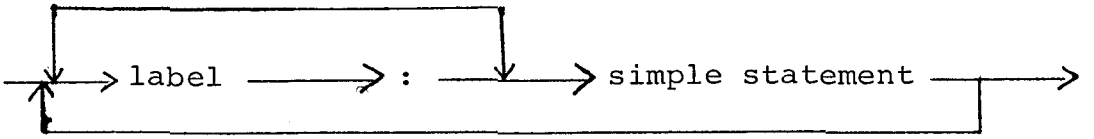
function heading



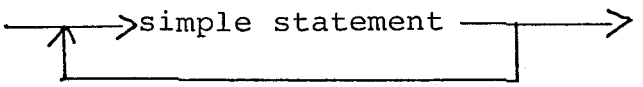
parameter list



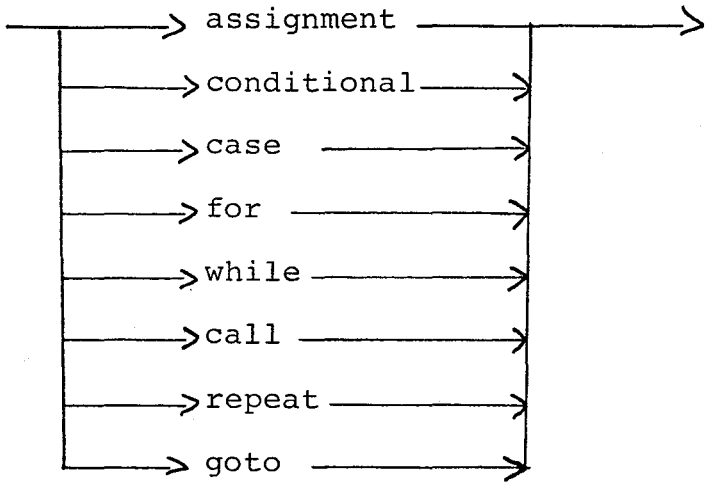
program statement list



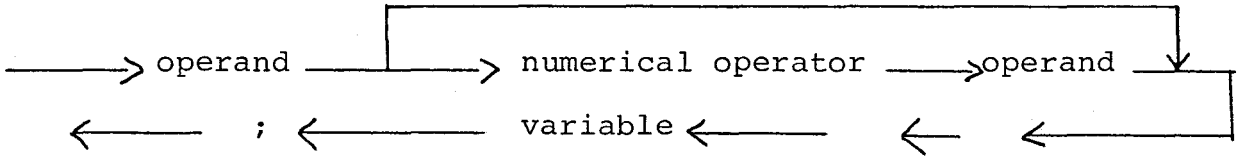
simple statement list



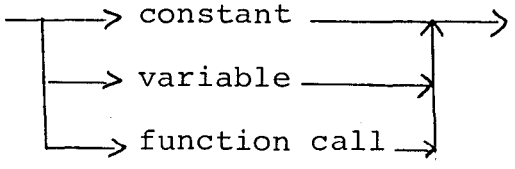
simple statement



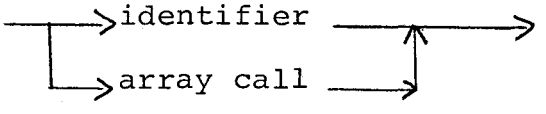
assignment



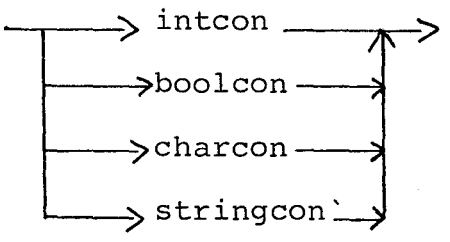
operand



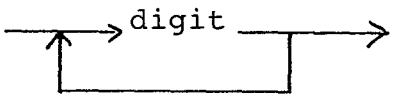
variable



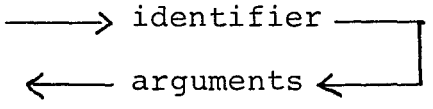
constant



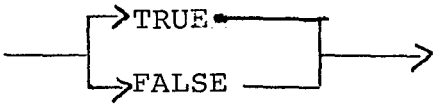
intcon



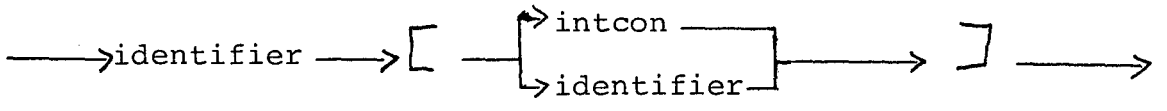
function call



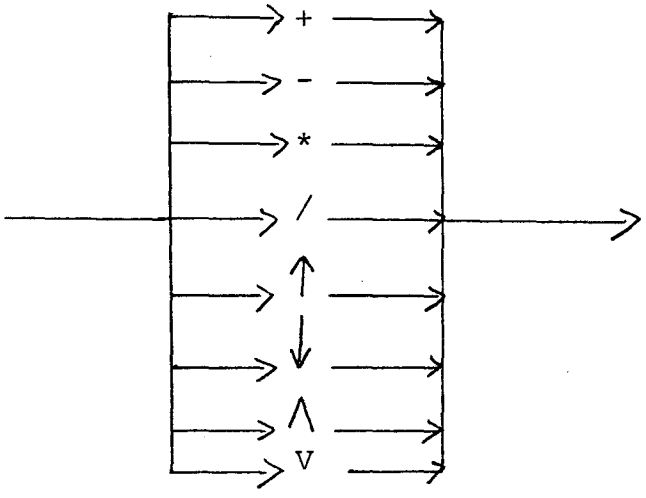
boolcon



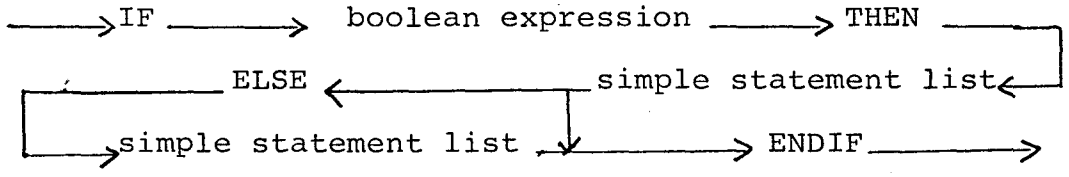
array call



numerical operator

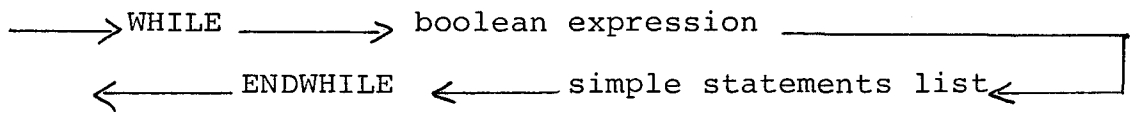


conditional



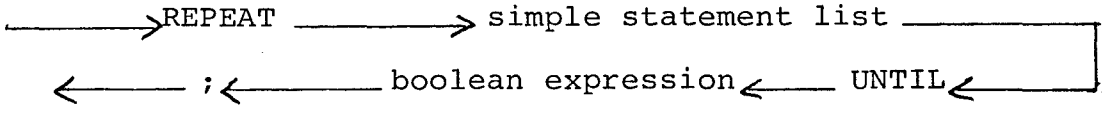
The conditional statement defines a boolean expression and two series of statements. If the boolean expression is true then the first series of statements are executed, else the second series of statements are executed. The second series of statements may be omitted.

while



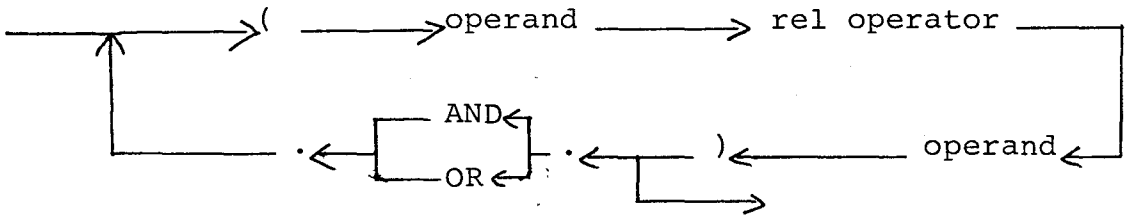
A while statement defines a boolean expression and a series of statements. If the boolean expression is false then the series of statements are not executed; otherwise, they are executed repeatedly until the boolean expression becomes false.

repeat

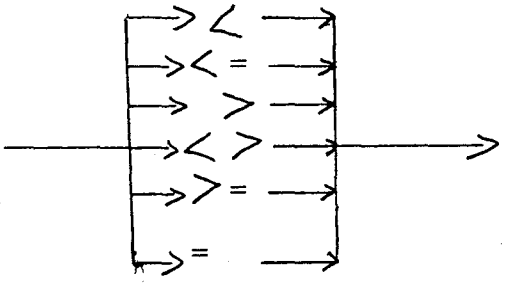


A repeat statement defines a series of statements and a boolean expression. The statements are executed at least once. If the boolean expression is false, they are executed repeated until the boolean expression becomes true.

boolean expression



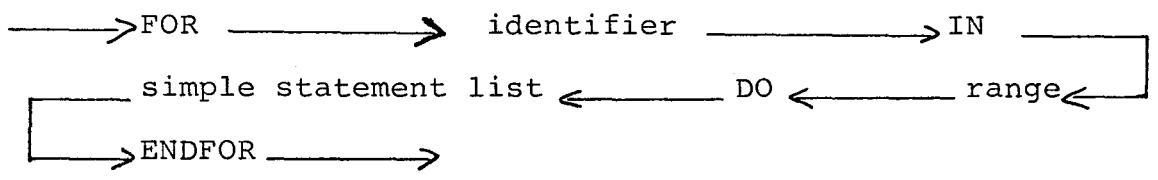
rel operator



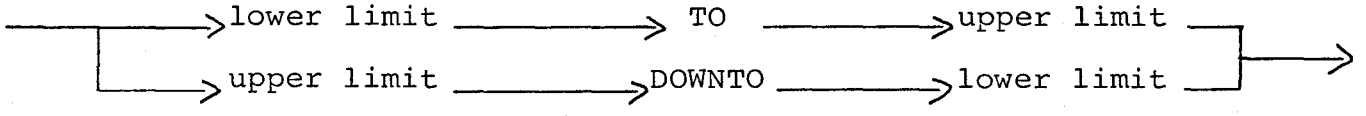
The meaning of the rel operators are given in Table 9 of Appendix B.

The boolean expression will always give the result of true or false.

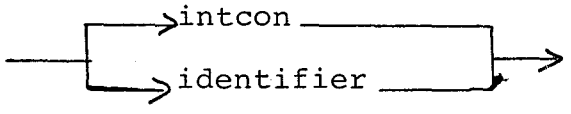
for



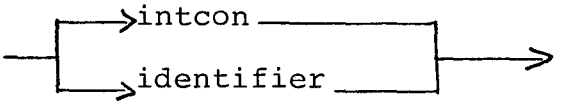
range



lower limit

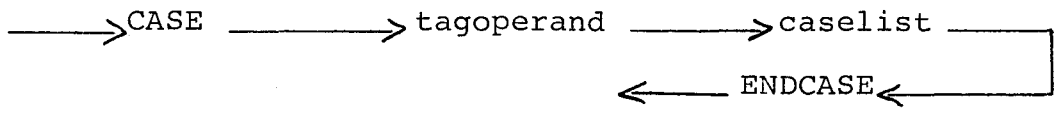


upper limit

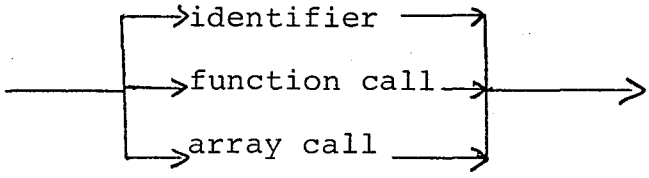


The identifier is incremented from its minimum value to its maximum value or decremented from its maximum value downto its minimum value. For each value of the identifier, the series of statements are executed. If the minimum value is greater than the maximum value, the series of statements will not be executed.

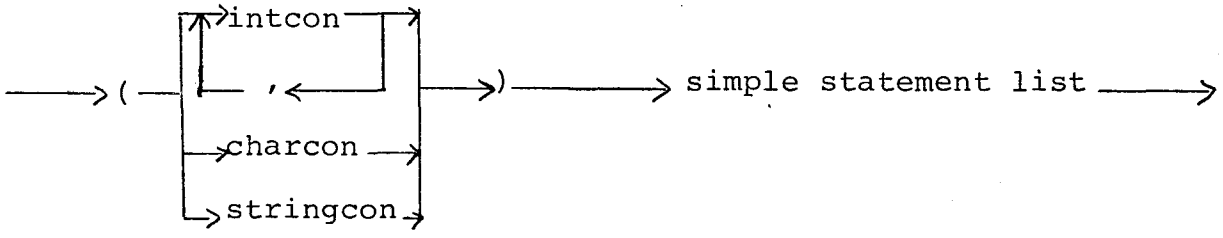
case



tagoperand

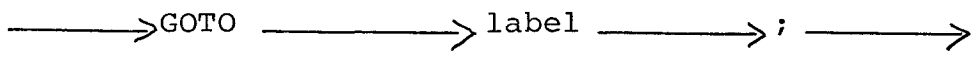


caselist



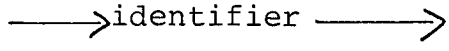
A case statement defines an enumeration expression and a statement block. The case statement executes the statement block belonging to that (tag) label.

goto

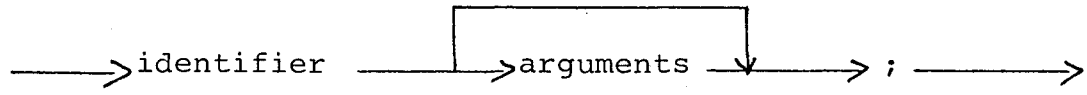


The goto statement gives the ASPLE+ language the feature of jumping to a labelled statement for the next instruction to be executed.

label

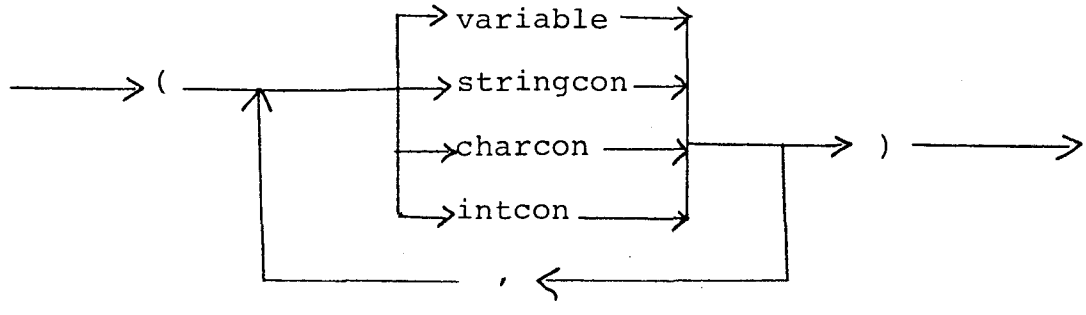


call



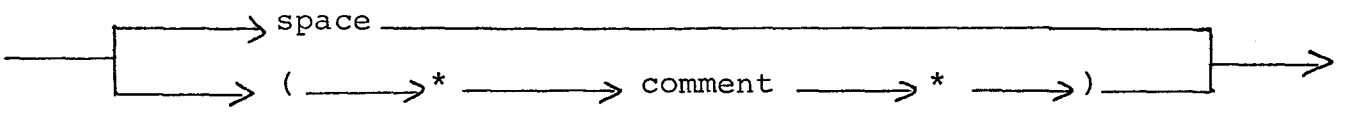
The call statements specifies the execution of a procedure or a segment.

arguments



The argument list defines the arguments used in a procedure, function or segment call.

separator



A comment is any sequence of characters not containing the combination (* or *).

CHAPTER 4

THE LEXICAL SCAN

The basic purpose of the lexical scan, embodied in the subroutines LXSCAN and NEXTCH, is

(i) To read a character into the array LINE

(ii) To unpack the array LINE

(iii) To remove both spaces and comments

(iv) To associate similar tokens with the same lexical class by giving a type value to each token that designates its class.

The lexical scan from ASPLE+ uses the Biggest Bite Principle. This principle finds and releases the largest token that can be found from a given point. For example ... < > ... will always yield the , < > , operator NEQ, not equal to, rather than a less than sign followed by a greater than sign.

4.1 Subroutine NEXTCH

Subroutine NEXTCH performs the lexical functions of

(i) reading an 80 column card into the array, LINE

and (ii) to unpack the array, LINE, character by character.

Subroutine NEXTCH uses the three global variables

(i) LL which is used as a counter for the length of the current line that has been read

(ii) CC which is used as a pointer to the current character that has been unpacked from the array, LINE

and (iii) CH which will contain the current character that has been unpacked from the array, LINE

Operand Table A, as well as the other operand tables, are provided as an aid to the implementor, so that the use of the global variables and constants can be quickly identified. Of even more value is the information that shows which global variables are busy-on-entry to this subroutine, indicating that there is a path through the subroutine along which the value of the variable will be fetched before it is calculated within the subroutine.

Similarly, a global variable which is busy-on-exit from the subroutine contains a value which will be used by the calling subroutine or perhaps by a subroutine which calls the calling subroutine.

Operand Table A lists the variables used by NEXTCH.

Operand Table A

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
CC	global variable	A,V,Y	YES	NO
CH	global variable	A,B,D	NO	YES
ERRPOS	global variable	A,V,Y	NO	YES
LINE	global variable	A	YES	NO
LL	global variable	A,U,Y	YES	NO

Subroutine NEXTCH is called by the subroutines LXSCAN and READSTRCHA and calls only the subroutine ERRORMSG, if the program is incomplete.

4.2 Subroutine LXSCAN

The lexical function of

(i) removing both spaces and comments

and (ii) associates a type value to each token that designates its class are performed by subroutine LXSCAN.

The second function is accomplished through the global variable SY. Table 10 of Appendix B lists the type values that a token can have. On return from subroutine LXSCAN the global variable SY will be set to one of these type values, indicating the designated token class.

Operand Table B lists the variables used by LXSCAN.

Operand Table B

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
ALNG	global constant	B	YES	NO
CH	global variable	A,B,D	YES	YES
ID	global variable	B,C,I	NO	YES
INUM	global variable	B,J,L	NO	YES
K	local variable	B,W	-	-
KMAX	global constant	B	YES	NO
NMAX	global constant	B	YES	NO
NMIN	global constant	B	YES	NO
SPS	global variable	B,Y	YES	NO
SY	global variable	B,H,I	NO	YES

Appendix A, list the meaning of the global constants and global variables. The global constants are used to check that

(i) the identifiers are not longer than ALNG

and (ii) the integer constants are not more than KMAX digits long and that the integer value INUM lies in the range NMAX to NMIN.

Since the biggest bite principle is used the global variable ID is needed to hold the identifier name and the global variable INUM is needed to contain the integer value.

Subroutine LXSCAN is called by subroutine ADVANCE and it calls the subroutines NEXTCH and ERROR.

CHAPTER 5

THE SYMBOL TABLE

The ASPLE+ compiler uses a linear Symbol Table. That is, the information is stored in the next available location. But an index array is keep so that the identifier names can be written out in alphabetical order.

The subroutines READNAME, READSTRCHA, TESTENTRY, ENTERVARIABLE and PRINTTAB are used in the development and printing out of the Symbol Table.

5.1 Subroutine READNAME

Subroutine READNAME has only one duty. That is to pack the first five characters (MAX) of the identifier, contained in array ID, into the array NAM and to ignore any following characters.

Operand Table C lists the variables used in READNAME.

Operand Table C

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
ID	global variable	B,C,I	YES	NO
KK	local variable	C,D,F,U	-	-
MAX	global constant	C,D,F,P,Q	YES	NO
NAM	global variable	C,D,L,M,N	NO	YES

Subroutine READNAME is called by subroutine CHECKID and does not itself call any other subroutines.

5.2 Subroutine READSTRCHA

Subroutine READSTRCHA has the following two duties.

(i) to read a character into the array NAM

and (ii) to read a string of any length and pack the first five characters (MAX) into the array NAM.

Operand Table D lists the variables used by READSTRCHA.

Operand Table D

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
CH	global variable	A,B,D	YES	NO
J	local variable	D,E,F,G,I,J,		
		L,O,P,Q,U,Y	-	-
KK	local variable	C,D,F,U	-	-
MAX	global constant	C,D,F,P,Q	YES	NO
NAM	global variable	C,D,L,M,N	NO	YES

Subroutine READSTRCHA is called by subroutine ADVANCE and calls subroutine NEXTCH to get the next character (CH) in the string or character constant.

5.3 Subroutine Testentry

Subroutine TESTENTRY has three major duties. They are

(i) to determine whether the name,NM, has previously been entered into symbol table TAB.

(ii) to set the boolean variable BL to false if the name,NM, has already been placed into symbol table TAB

and (iii) to set N to the storage location in symbol table TAB if the name,NM, is found.

These duties are performed by checking through the names in the symbol table linearly until either the name is found or the end of the symbol table is reached.

Operand Table E lists the variables used by TESTENTRY.

Operand Table E

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
BL	local variable	E,J,L,M,N	-	-
III	local variable	E,F,U	-	-
J	local variable	D,E,F,G,I,J,L, O,P,Q,U,Y	_	_
N	local variable	E,J,L,M,N,V,W	_	_
NM	local variable	E	-	-
TAB	global variable	E,F,G,J,L,M,N,Y	YES	NO
TPTR	global variable	E,F,G,J,M,Y	YES	NO

Subroutine TESTENTRY is called by subroutines TESTVAR, DECLARATION, STATEMENT and PROCFN and does not itself call any other subroutines.

5.4 Subroutine ENTERVARIABLE

Subroutine ENTERVARIABLE is the main symbol table routine. It has the following two duties

(i) To enter the variable name and all information about the variable name into the next available location, TPTR, in the symbol table.

and (ii) To rearrange the array INDEX so that the variable names can be written out in alphabetical order.

First of all, subroutine ENTERVARIABLE enters the information about the identifier. There are six pieces of information. These are

(i) The name, PT, of the identifier.

(ii) The object type, OB, of the identifier. The object type can be a variable, a procedure, a function or a segment name.

(iii) The amount of storage, SZ, required by the identifier if it is a variable or the number of parameters, SZ, a function, segment or procedure has.

(iv) The identifier type, TY, of the identifier. The identifier type is either integer, boolean, character, string or no type.

(v) Whether the identifier is a local or external declaration. This is done using variable STY.

and (vi) Whether the identifier is an entry point, pointer variable or neither. This is done through variable ENY.

Then the INDEX array is reorganized so that the ordering of the identifier names remain in alphabetical order. The INDEX array is only reorganized if the procedure and function parameter variable counter TPROC is zero or if I am dealing with a segment because the parameters for the procedures or functions are not to be shown by the symbol table when it is written out.

Operand Table F lists the variables used by ENTERVARIABLE.

Operand Table F

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
ENTRY	global variable	F,G	NO	NO
ENY	local variable	F,K,L,M	-	-
III	local variable	E,F,U	-	-
INDEX	global variable	F	YES	YES
J	local variable	D,E,F,G,I,J,L, O,P,Q,U,Y	-	-
KK	local variable	C,D,F,U	-	-
KL	local variable	F	-	-
MAX	global constant	C,D,F,P,Q	YES	NO
NAMES	global variable	F,G	NO	NO
OA	global variable	F,L,M	YES	NO
OB	local variable	F,J,K,L,M,N	-	-
OBJ	global variable	F,G,	NO	NO
PA	local variable	F,O	-	-
PT	local variable	F,K,O,Q	-	-
SIZE	global variable	F,G	NO	NO
STY	local variable	F,J,K	-	-
STYP	global variable	F,G,	NO	NO
SZ	local variable	F,J,K,L,M,O,U	-	-
TAB	global variable	E,F,G,J,L,M,N,Y	NO	NO
TEM	local variable	F	-	-
TEMP	local variable	F	-	-

Operand Table F

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
TPROC	global variable	F,J,L,M,N,Y	YES	NO
TPTR	global variable	E,F,G,J,M,Y	YES	YES
TY	local variable	F,H,I,J,K,L,M	-	-
TYP	global variable	F,G	NO	NO

The global variables NAMES, OBJ, SIZE, STYP, TYP and ENTRY contain the information when put into the symbol table.

Subroutine ENTERVARIABLE is called from subroutines DECLARATION, PROCFN, and STATEMENT and does not itself call any other subroutines.

5.5 Subroutin PRINTTAB

The last subroutine of this chapter, PRINTTAB, has one duty. That is to write out the information contained in the symbol table in a alphabetical order. PRINTTAB writes out

(i) the identifier name, NAMES

(ii) the object type, OBJ, of identifier

(iii) the identifier type, TYP

(iv) The variable STYP, indicates whether the identifier is a local or external declaration

(v) The variable ENTRY indicates whether the identifier is an entry point, pointer variable or neither.

Operand Table G lists the variables used by PRINTTAB.

Operand Table G

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
ENTRY	global variable	F,G,	YES	NO
J	local variable	D,E,F,G,I,J,L, O,P,Q,U,Y	-	-
KKK	local variable	G	-	-
NAMES	global variable	F,G	YES	NO
OBJ	global variable	F,G	YES	NO
SIZE	global variable	F,G	YES	NO
STYP	global variable	E,G	YES	NO
TAB	global variable	E,F,G,J,L,M,N,Y	YES	NO
TPTR	global variable	E,F,G,J,M,Y	YES	NO
TYP	global variable	F,G	YES	NO

The global variable TPTR contains the value of the number of identifier names in the symbol table to be written out.

Subroutine PRINTTAB is called from the main program and does not itself call any subroutines.

CHAPTER 6

THE BASIC SCAN

Having developed the subroutines NEXTCH, READNAME, LXSCAN and READSTRCHA in the proceeding sections, it is now a simple task to scan the source string. The objective is to obtain the next operator-operand-operator combination that is used in analyzing the source string.

This basic operation is done by subroutines CHECKID and ADVANCE.

6.1 Subroutine CHECKID

Subroutine CHECKID has the following duties

(i) to check the reserved word table IDTAB for the identifier name ID.

(ii) to set the boolean variable BOL

and (iii) to set the global variable SY or the local variable TY depending on the value of J.

Subroutine CHECKID firsts checks the reserved word list, given in Table 3 of Appendix B, for the identifier ID. If the reserved words INTEGER, CHAR, BOOLEAN, LABEL, OR, AND XOR or STRING are found then the boolean variable BOL is set to false and the local variable TY is to be set to the appropriate value. If any other reserved word is found the global variable SY is set to the appropriate reserved work operator. If no reserved word is found subroutine READNAME is called.

Operand Table I lists the variables used by CHECKID.

Operand Table I

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
BOL	local variable	H,I,L	-	-
ID	global variable	B,C,I	YES	NO
IDTAB	global variable	I,Y	YES	NO
J	local variable	D,E,F,G,I,J,L,O,P, Q,U,Y	-	-
SY	global variable	B,H,I,	NO	YES
TY	local variable	F,H,I,J,K,L,M	-	-

Subroutine CHECKID is called from ADVANCE and calls subroutine READNAME.

6.2 Subroutine ADVANCE

The main subroutine in the basic scan is subroutine ADVANCE.

Subroutine ADVANCE has two basic purposes which are

(i) to obtain the next operator-operand-operator combination

and (ii) to obtain the information about the operand.

ADVANCE performs these two duties by first setting the next operator (NO) to the current operator (CO) and then calling LXSCAN for the next token. A number of simple tests are performed on the token, to see if it is an identifier, in which CHECKID is called, an integer, a string or character in which READSTRCHA is called, or another operator. In the latter case the operand member of the triple is the null operand.

At the label marked FILL NEXT OP, if the boolean variable BOL is true then we have found the other operator and set it to the next operator, NO. If the boolean variable, BOL, is false then we have found the operand and must call LXSCAN for the next operator.

The global variable YSY contains the information about the operand. It can have the values of

(i) SNAM specifying the operand to be an identifier

(ii) NUM specifying the operand as an integer

(iii) STR specifying the operand as a string

(iv) CHA specifying the operand as a character

or (v) BLNK specifying the empty operand.

Operand Table H list the variables used by ADVANCE.

Operand Table H

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
BOL	local variable	H,I,L	-	-
CO	local variable	H,J,L,M,N	-	-
NO	local variable	H,J,L,N	-	-
SY	global variable	B,H,I	NO	NO
TY	local variable	F,H,I,J,K,L,M	-	-
YSY	global variable	H,L,N,	NO	YES

Subroutine ADVANCE is called from DECLARATION, STATEMENT and PROCFN and calls the subroutines READSTRCHA, CHECKID and LXSCAN.

CHAPTER 7

COMPILING DECLARATIVE STATEMENTS

WITH A CO-NO TABLE

The use of the current operator-next operator (CO-NO) table, often called a transition matrix, is both old and rather space consuming, but it still appears to offer a basic advantage in speed of compilation, as well as ease of extension and straightforward error diagnostics. In addition to these tangible advantages, inherent in the CO-NO concept are several seldom-exploited capabilities which make them worthy of interest. Among the latter are their facility in allowing the overlapping and mixing of statement types, and their facility in allowing the potential N * N unique operations from a character set of N operators.

In designing the Co-No table (if one would do the task manually) the basic approach consists of setting up a matrix with operators across the top, corresponding to the current operator position, and along the side representing the next operator. Intersections of illegal CO-NO combinations are marked by x's while legal combinations are left blank.

The approach is illustrated below, in which only a few of the set of operators are shown

Current Operator

		CO		
NEXT	,	:	;	⌋
OPERATOR ,		x	x	x
NO :		x		x
;	x		x	x
⌋	x	x	x	x

7.1 Subroutine DECLARATION

Table 1 of Appendix D lists and gives the meaning of each operator used in the CO-NO table for the declarations. Table 2 of Appendix D shows the CO-NO table used in subroutine DECLARATION. The declarations are compiled by subroutine DECLARATION.

The major functions of subroutine DECLARATION are

(i) to enter each identifier name declared, by the external or local declarations, into the symbol table, TAB

(ii) to allocate storage space, SZ, needed for each identifier

(iii) to mark the identifier as either an entry point, pointer variable, or neither.

and (iv) if the identifier is a procedure or function parameter to stack it in array stack.

Operand Table J lists the variables used by DECLARATION.

Operand Table J

Operand	Class	Used By	Buys-on-Entry	Buys-on-Exit
BL	local variable	E,J,L,M,N	-	-
BOOL	local variable	J,L,N	-	-
CO	local variable	H,J,L,M,N	-	-
ENT	local variable	J,K	-	-
I	local variable	J,L	-	-
INUM	global variable	B,J,L	NO	NO
J	local variable	D,E,F,G,I,J,L,O,P, Q,U,Y	-	-
LI	local variable	J	-	-
LNUM	local variable	J,L,M	-	-
LOCTNS	global variable	J,L,M,N	NO	YES
N	local variable	E,J,L,M,N,V,W	-	-

Operand Table J

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
NK	global variable	J	NO	YES
NO	local variable	H,J,L,M	-	-
OB	local variable	F,J,K,L,M,N	-	-
STACK	global variable	J,L,M	NO	YES
STY	local variable	F,J,K,L,M	-	-
SZ	local variable	F,J,K,L,M,O,U	-	-
TAB	global variable	E,F,G,J,L,M,N,Y	NO	NO
TPTR	global variable	E,F,G,J,M,Y	NO	NO
TY	local variable	F,I,H,J,K,L,M	-	-

Appendix C describes the local variables used by DECLARATION and GENCODE.

DECLARATION is called from PROCFN and STATEMENT and calls ENTERVARIABLE, TESTENTRY an GENCODE.

7.2 Subroutine GENCODE

Subroutine Gencode has one duty and that is to generate the code for the external and local declarations.

Operand Table K lists the variables used by GENCODE.

Operand Table K

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
ASTPLUSONE	local variable	K	-	-
BSS	local variable	K	-	-
DEF	local variable	K	-	-
ENT	local variable	J,K	-	-
ENY	local variable	J,K,L,M	-	-
EXTS	local variable	K	-	-
NUL	global variable	K,L,O,M	NO	YES
OB	local variable	F,J,K,L,M,N	-	-
PT	local variable	F,K,O,Q	-	-
STY	local variable	F,J,K,L,M	-	-
SZ	local variable	F,J,K,L,M,O,U	-	-
TY	local variable	F,I,H,J,K,L,M	-	-
TYS	local variable	K,L,O	-	-

I will now show some examples of declarations and the code generated.

EXAMPLE 1

Any variable that is declared in the external declaration appears as

```
EXT  name
```

where name is the variable name.

EXAMPLE 2

an array

```
A:  ARRAY[10]  OF INTEGER;
A   DEF   *   +   1
    BSS    10
```

EXAMPLE 3

an array as an entry point

```
*  ABC  :  ARRAY[10] OF INTEGER  ;
ABC   DEF   *   +   1
      ENT   ABC
      BSS   10
```

EXAMPLE 4

This example illustrates a variable that is an entry point.

```
*  ABCDE  :  INTEGER  ;
ABCDE  BSS  1
ENT  ABCDE
```

Subroutine GENCODE is called from DECLARATION and calls BUILD.

CHAPTER 8

COMPILING THE STATEMENTS

Like the declarations, the statements are also handled by a CO-NO table. There are eight statements defined by the language ASPLE+. These are

(i) the assignment
(ii) the conditional
(iii) the case
(iv) the for
(v) the while
(vi) the repeat
(vii) the goto
and (viii) the routine calls

As stated in the last chapter, the CO-NO table works on the current operator (CO) - next operator (NO) principle.

8.1 Subroutine STATEMENT

A CO - NO table is used by subroutine STATEMENT to check the syntax of each statement used in the source program.

Table 3 of Appendix H lists and defines the operators used in the CO - NO table inconjunction with STATEMENT. Since this CO - NO table is so large, instead of listing all the current operators across the top and all the next operators down the side, Table 4 of Appendix D gives the name of the next operator and beside that it lists all the current operators that it can follow.

The main function of subroutine STATEMENT is to call subroutine ADVANCE for the next current operator (CO) - next operator (NO) pair. Then based on the value of the next operator (NO), STATEMENT calls one of the subroutines that it defines inside itself. This subroutine then checks the syntax of the source line, by checking to see that the next operator (NO) follows the correct current operator (CO) as specified in the CO - NO table.

The subroutines defined inside STATEMENT correspond to the names in capital letters in the CO - NO table and are defined in Appendix E.

Subroutine STATEMENT has a number of minor functions. These are

(i) to call subroutine DECLARATION if the key words LOCAL or EXTERNAL are found.

(ii) to call subroutine TESTENTRY to see if the identifier name has been declared

(iii) to call subroutine TESTVAR to test the compatibility of variables in the statements.

(iv) to call subroutine ERROR when errors occur

(v) to call subroutine PROCFN to declare the procedure, function and segment headers.

and (vi) to call subroutine BUILD to generate code.

Operand Table L list the variables used by STATEMENT.

Operand Table L

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
BL	local variable	E,J,L,M,N	-	-
BOL	local variable	H,I,L	-	-
BOOL	local variable	J,L,N	-	-
CAS	global variable	L,N	YES	NO
CO	local variable	H,J,L,M,N	-	-
COUNT	local variable	L,Y	-	-
ENY	local variable	F,K,L,M	-	-
ENYY	local variable	L	-	-
I	local variable	J,L	-	-
II	local variable	L,N	-	-
INUM	global variable	B,J,L	NO	NO

Operand Table L

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
J	local variable	D,E,F,G,I,J,L,O,P, Q,U,Y	-	-
JK	local variable	L	-	-
JKC	local variable	L	-	-
JKK	local variable	L	-	-
JLK	local variable	L	-	-
JPK	local variable	L	-	-
L	local variable	J,L,O,S	-	-
LJK	local variable	L	-	-
LLNUM	local variable	L	-	-
LNUM	local variable	J,L	-	-
LOCTNS	global variable	J,L,M	YES	NO
LOUT	global variable	L,O	NO	NO
LP	global variable	L,O,Y	NO	NO
LUCK	local variable	L	-	-
N	local variable	E,J,L,M,N,V,W	-	-
NAM	global variable	C,D,L,M,N	YES	NO
NO	local variable	H,J,L,M	-	-
NUL	global variable	K,L,M,O	YES	NO
OA	global variable	F,L,M	NO	NO
OB	local variable	F,J,K,L,M,N	-	-

Operand Table L

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
OPERT	local variable	L,M	-	-
SK	global variable	L,O,Y	NO	NO
SKK	global variable	L,)	NO	NO
SPNAM	global variable	L,M	NO	NO
SPNM	global variable	L,M	NO	NO
STACK	global variable	J,L,M	YES	NO
STCK	local variable	L	-	-
STK	global variable	L,O	NO	NO
STY	local variable	F,J,K,L,M,O,U	-	-
SWK	local variable	L	-	-
SZ	local variable	F,J,K,L,M,O,U	-	-
TAB	global variable	E,F,G,J,L,M,N,Y	YES	YES
TEST	global variable	L,N	NO	NO
TK	global variable	L,O	NO	NO
TP	global variable	L,O	NO	NO
TPROC	global variable	F,J,L,M,N,Y	NO	NO
TST	local variable	L,N	-	-
TY	local variable	F,I,H,J,K,L,M	-	-
TYS	local variable	K,L,O	-	-
YSY	global variable	H,L,N	NO	NO

All local variables are described in Appendix F. All global variables are described in Appendix A.

Subroutine STATEMENT is called from the main program and calls subroutines BUILD, DECLARATION, TESTENTRY, TESTVAR, ERROR, PROCFN and ADVANCE.

8.2 Subroutine PROCFN

Subroutine PROCFN also uses a CO - NO table. This table is given in Table 5 of Appendix D.

The functions of PROCFN are

(i) to enter the procedure, function or segment name into the symbol table.

(ii) to enter the parameters of a procedure, function or segment into the symbol table.

(iii) to enter the parameter names type into the array VARS, in the location, LOCTNS, of the symbol table. This location is the procedure of function name. This is done by DECLARATION.

(iv) to enter the function type into the symbol table, in location, LOCTNS

(v) to enter the segment parameter into the symbol table and to stack the segment name into array SPNAM.

and (vi) calls subroutine BUILD to generate the code necessary to define a procedure or function definition.

Operand Table M lists the variables used by PROCFN.

Operand Table M

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
BL	local variable	E,J,L,M,N	_	_
CO	local variable	H,J,L,M,N	-	-
ENY	local variable	F,K,L,M	-	-
LNUM	local variable	J,L,M	-	-
LOCTNS	global variable	J,L,M,N	NO	YES
N	local variable	E,J,L,M,N,V,W	-	-
NAM	global variable	C,D,L,M,N	NO	NO
NO	local variable	H,J,L,M	-	-
NUL	global variable	L,M,O	YES	NO
OA	global variable	F,L,M	YES	NO
OB	local variable	F,J,K,L,M,N	-	-
OPERT	local variable	L,M	-	-
SPNAM	global variable	L,M	NO	YES
SPNM	global variable	L,M	YES	YES
STACK	global variable	J,L,M	NO	YES

Operand Table M

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
STY	local variable	F,J,K,L,M	-	-
SZ	local variable	F,J,K,L,M,O,U	-	-
TAB	global variable	E,F,G,J,L,M,N,Y	YES	YES
TPROC	global variable	F,J,L,M,N,Y	YES	YES
TPTR	global variable	E,F,G,J,M,Y	YES	YES
TY	local variable	F,I,H,J,K,L,M,	-	-

All local variables have the same meaning as in STATEMENT except that variable ENY indicates an external procedure or function.

Subroutine PROCFN is called from STATEMENT and calls DECLARATION, ERROR, ADVANCE and BUILD.

8.3 Subroutine TESTVAR

The last subroutine to be mentioned in this chapter is TESTVAR. Subroutine TESTVAR has five basic functions. They are

(i) to test the variable name to see if, it has been declared.

(ii) to set the boolean variable, BOOL, to false if the variable name is a procedure or function name and to set the parameter counter TPROC to zero.

(iii) to set the variable TEST, to the appropriate type value, if I am starting to parse a new statement.

(iv) to enter the tagoperand type of the case statement into array CAS if I am dealing with the case statement

and (v) to check the variable type, TST, compatibility

(i) against TEST if BOOL is true.

(ii) against the array CAS element II if I am dealing with the CASE statement

or (iii) against the parameter definition of a FUNCTION or Procedure if BOOL is false.

Operand Table N lists the variables used by TESTVAR.

Operand Table N

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
BL	local varaible	E,J,L,M,N	-	-
BOOL	local variable	J,L,N	-	-
CAS	global variable	L,N	YES	YES
CO	local variable	H,J,L,M,N	-	-
II	local variable	L,N	-	-
LOCTNS	local variable	J,L,M,N	-	-
N	local variable	E,J,L,M,N,V,W	-	-
NAM	global variable	C,D,L,M,N	YES	NO
OB	local variable	F,J,K,L,M,N	-	-
TAB	global variable	E,F,G,J,L,M,N,Y	YES	YES
TEST	global variable	L,N	YES	YES
TPROC	global variable	F,J,L,M,N,Y	YES	YES
TST	local variable	L,N.	-	-
YSY	global variable	H,L,N	YES	NO

All local variables have the same meanings as described in Appendix F.

Subroutine TESTVAR is called from STATEMENT and does not itself call any other subroutines.

This ends the discussion about the statements.

CHAPTER 9

GENERATING CODE

This chapter discusses the subroutines used to build a command that has been generated from the ASPLE+ statement. There are seven subroutines used to generate these commands. They are BUILD, EMITOPCODE, EMITNAME, EMITCHAR, EMITLABEL, EMITEOL and EMITNUM. All commands generated will be written to the file CODE.

9.1 Subroutine BUILD

The major subroutine is BUILD. Subroutine BUILD will build a command or part of a command depending on the value of L. The variable L takes on one of the following integer values.

(i) The value five indicates that I am building the command for a procedure or function call or a procedure or function definition and that I am adding to that command the parameter

PA is the argument name of a procedure or function call or the name of a parameter in the definition of a procedure or function.

(ii) The value four indicates that I am generating a JUMP command. This JUMP command is generated by the ENDWHILE or ELSE statements.

(iii) The value three means that I am building the IF command and I am adding to it, the logical operator to test for and the label to jump to if the test on the logical operator is true.

(iv) The value two indicates the completion of a command that has been started. The last item to be done depends on the value of TYS.

(v) The value one indicates that a complete command will be emitted.

or (vi) The value zero indicates that part of a command will be emitted.

For the values zero and one a label will be emitted, first, depending on the value of NUL. Then the opcode is emitted, followed by the rest of the command which depends on the value of TYS.

Operand Table O lists the variables used by BUILD.

Operand Table O

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
COUT	global variable	O,Q,U	YES	NO
J	local variable	D,E,F,G,I,J,L,O,P,Q,U,Y	-	-
L	local variable	J,L,O,S	-	-
LP	global variable	L,O,Y	YES	YES
LOUT	global variable	L,O	YES	YES
NUL	global variable	K,L,M,O	YES	YES
O	local variable	O,P	-	-
PA	local variable	F,O	-	-
PT	local variable	F,K,O,Q	-	-
SK	global variable	L,O,Y	YES	YES
SKK	global variable	L,O	YES	YES
STK	global variable	L,O	YES	YES
SZ	local variable	F,J,K,L,M,O,U	-	-
TK	global variable	L,O	YES	YES
TP	global variable	L,O	YES	YES

Subroutine BUILD is called by DECLARATION PROCFN, GENCODE and STATEMENT and calls the subroutines EMITNAME, EMITNUM, EMITOPCODE, EMITLABEL and EMITCHAR.

9.2 Subroutine EMITOPCODE

Subroutine EMITOPCODE writes the opcode value O to the file CODE. Table four of Appendix B lists the opcode values.

Operand Table P lists the variables used by EMITOPCODE.

Operand Table P

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
CODE	global variable	P,Q,R,T,U	YES	YES
J	local variable	D,E,F,G,I,J,L,O,P, Q,U,Y	-	-
MAX	global constant	C,D,F,P,Q	YES	YES
O	local variable	O,P	-	-

Subroutine EMITOPCODE is called from BUILD and does not itself call any other subroutines.

9.3 Subroutine EMITNAME

PT, a variable name used in the program, is written to the file CODE by subroutine EMITNAME.

Operand Table Q lists the variables used by EMITNAME.

Operand Table Q

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
CODE	global variable	P,Q,R,T,U	YES	YES
COUT	global variable	O,Q,U	YES	YES
J	local variable	D,E,F,G,I,J,L,O,P, Q,U,Y	-	-
MAX	global constant	C,D,F,P,Q	YES	NO
PT	local variable	F,K,O,Q	-	-

Subroutine EMITNAME is called from BUILD and does not itself call any other subroutines.

9.4 Subroutine EMITCHAR

Subroutine EMITCHAR will write one character to the file CODE.

Operand Table R lists the variables used by EMITCHAR.

Operand Table R

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
C	local variable	R	-	-
CODE	global variable	P,Q,R,T,U	YES	YES

Subroutine EMITCHAR is called from BUILD and EMITLABEL and does not itself call any other subroutines.

9.5 Subroutine EMITLABEL

The labels generated by outputted code are written to the file CODE, by subroutine EMITLABEL. These labels will always start with the two characters F $.

Operand Table S lists the variables used by EMITLABEL.

Operand Table S

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
L	local variable	J,L,O,S	-	-

The variable L is the label number to be emitted.

Subroutine EMITLABEL is called from BUILD and calls the subroutines EMITCHAR and EMITNUM.

9.6 Subroutine EMITEOL

Subroutine EMITEOL will make sure that a new line will be started when a new command is generated.

Operand Table T lists the variable used by EMITEOL.

Operand Table T

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
CODE	global variable	P,Q,R,T,U	YES	YES

Subroutine EMITEOL is called from BUILD and does not itself call any other subroutines.

9.7 Subroutine EMITNUM

The last subroutine to be talked about in this chapter is EMITNUM. Subroutine EMITNUM will write the integer number, SZ, to the file CODE.

This is accomplish by breaking the number, SZ, down into single digits which are stored in the array NUMBER. Then the array NUMBER is written to CODE one element at a time.

Operand Table U list the variables used by EMITNUM.

Operand Table U

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
CODE	global variable	P,Q,R,T,U	YES	YES
COUT	global variable	O,Q,U	NO	YES
III	local variable	E,F,U	-	-
J	local variable	D,E,F,G,I,J,L,O, P,Q,U,Y	_	_
KK	local variable	C,D,F,U	-	-
LL	local variable	A,U,Y	-	-
NUMBER	local variable	Y	-	-
SZ	local variable	F,J,K,L,M,O,U	-	-

Subroutine EMITNUM is called from BUILD and EMITLABEL and does not itself call any other subroutines.

All macros generated by the code and built through subroutine BUILD are described in Appendix G.

CHAPTER 10

THE ERROR ROUTINES

ASPLE+ has three routines to handle the errors in input programs. These routines are ERROR, ERRORMSG and FATAL.

10.1 Subroutine ERROR

Subroutine ERROR has two duties which are

(i) to indicate where and in what source line the error occurred

and (ii) to specify the error number that occurred.

These duties are performed by placing, under the line in error, asterisks in columns two to five and the indicating where the error occurred by placing an up arrow and the error number, N.

Operand Table V lists the variables used by ERROR.

Operand Table V

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
CC	global variable	A,V,Y	YES	YES
ERRPOS	global variable	A,V,Y	YES	YES
ERRS	global variable	V,Y	YES	YES
N	local variable	E,J,V	-	-

Subroutine ERROR is called from STATEMENT, LXSCAN, PROCFN, TESTVAR, and DECLARATION but does not itself call any other subroutines.

10.2 Subroutine ERRORMSG

Subroutine ERRORMSG has one duty and that is to write out the error messages that have occurred in the source program.

Operand Table W lists the variables used by ERRORMSG.

Operand Table W

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
ERRS	global variable	V,W	YES	NO
K	local variable	B,W	-	-

ERRORMSG is called from the main program but does not itself call any other subroutines.

10.3 Subroutine FATAL

The last subroutine, FATAL, has the duty of writing out the fatal error meesage.

Operand Table X lists the variables used by FATAL.

Operand Table X

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
MSG	local variable	X	-	-
N	local variable	E,J,L,M,N,V,W	-	-

Subroutine FATAL is called from ENTERVARIABLE but does not itself call any other subroutines.

Appendix H lists the error messages.

CHAPTER 11

INITIALIZATION

This chapter describes the duties of the initialization subroutine INIT. This subroutine

(i) initializes the special symbol array SPS

(ii) initializes the reserved word table IDTAB

and (iii) initializes the global variables.

All global constants, types and variables are defined in Appendix A.

Table two of Appendix B lists the special symbols that are initialized by the array SPS.

Table three of Appendix B lists the reserved words that are initialized by IDTAB.

Operand Table Y lists the variables used by INIT.

Operand Table Y

Operand	Class	Used By	Busy-on-Entry	Busy-on-Exit
CC	global variable	A,V,Y	NO	YES
CACC	global variable	Y	NO	YES
CNAM	global variable	Y	NO	YES
CNUM	global variable	Y	NO	YES
COUNT	global variable	L,Y	NO	YES
ERRPOS	global variable	A,V,Y	NO	YES
ERRS	global variable	V,Y	NO	YES
IDTAB	global variable	I,Y	NO	YES
J	local variable	D,E,F,G,I,J,L,O,P, Q,U,Y	-	-
LL	global variable	A,U,Y	NO	YES
LP	global variable	L,O,Y	NO	YES
SK	global variable	L,O,Y	NO	YES
SPS	global variable	B,Y	NO	YES
TAB	global variable	E,F,G,J,L,M,N,Y	NO	YES
TP	global variable	L,O,Y	NO	YES
TPROC	global variable	F,J,L,M,N,Y	NO	YES
TPTR	global variable	E,F,G,J,M,Y	NO	YES

This subroutine is called from the main program and does not call any other subroutines.

CHAPTER 12

WHAT THE TEST PROGRAM

ILLUSTRATES

The test program illustrates the syntax of the ASPLE+ language and the code generated by each statement. The test program has been set up in such a way that the outputted code generated by ASPLE+ can be seen directly after the input line of the source program.

12.1 The Program Statement

The first statement in the source program must be one of the described statements in figure 1.1 of chapter 1. This statement will indicate the name of the source program and the segment to be called, if there is one.

The code generated from this statement indicates the name of the program and reserves storage space for the names * + 1, * + 2, * + 3, * + 4 and * + 5 which are used in generating code in certain situations.

12.2 External Declarations

The external declarations of the test program demonstrate the declarations of

(i) variable names which are less than five characters in length. These names illustrate the six declaration types of integer, boolean, character, string, label and pointer variables.

(ii) variable names which are greater than five characters in length. These names show how the variable name is cut down to five characters.

and (iii) strings and characters. The string names are also cut down to a length of five characters.

The code generated by each statement is also shown under the source line being looked at.

12.3 Local Declarations

The local declarations declared in the test program illustrate

(i) the declaration of an array as an integer, a boolean, a character and a string

(ii) the declaration of a variable, a string, a character, and an array as entry points.

and (iii) how a number of variables, strings and characters can be declared in the same line.

The code generated by each statement is also shown under the source line being looked at.

12.4 Procedures and Functions

The test program also demonstrates how to declare external procedures and functions as well as the procedures and functions which are local to the program.

The local procedure and functions are just stubs since I am just illustrating the code generated by the procedure or function definitions.

12.5 Assignment Statements

The first part of the main program illustrates

(i) the initialization of constants by the assignment statement

(ii) the eight numerical operators used in an assignment statement

(iii) the assignment of a boolean variable

(iv) how the accumulator can be used as an operand in the assignment statement

(v) the use of an array in an assignment statement

and (vi) how a character or string can be assigned.

The output code generated to handle an assignment statement is shown directly after the assignment statement being scanned.

12.6 Control Statements

The control statements (WHILE, CASE, REPEAT, IF, FOR and GOTO) generate the code illustrated in the test program.

The WHILE, IF and illustrate boolean expressions that can be used by this language.

12.7 Call Statements

The last features of the test program illustrates how functions, procedures and segments are called.

A function must occur as an operand in an assignment statement. There are three assignment statements which demonstrate the code generated when a function occurs as an operand.

The next two source program statements show the code generated by procedure calls.

When the END statement of the main program is encountered, it will generate the code to call a segment if the program card specifies the segment to be called. Since the test program indicates a segment to be called the appropriate code is generated. Otherwise, the code indicating the end of the program is generated.

CHAPTER 13

CONCLUSION

13.1 Project Comments

Work on the project proceededvery smoothly. I was able to test each part of the compiler as it was added and therefore ensuring that the new addition had no effect on what was already working.

I have tested the compiler with seven test programs up to this point. But because the outputted code generates macro calls, I am unable to check for any logic errors in the reentrant code, since the macros have not been written.

13.2 Program Assessment

The program code is fairly readable, fairly well commented and presents a logical flow of control.

The subroutines invoke a modular design contributing to a reasonably efficient program with minimized redundancy.

The program is easily modified because of the current operator-next operator feature in the two major subroutines DECLARATION and STATEMENT.

The error dump provides the user with valuable information not contained in the error number.

13.3 Program Extensions

There are a number of changes that can be made to the compiler to make it much more effective.

The first change can be made inconjunction with the error messages. The error messages can be made explicit when talking about CO - NO table errors. Instead of just indicating a CO - NO table error, the error message should explain what the compiler actually expected. For example, the error message could read

; EXPECTED,

: EXPECTED or

OF EXPECTED.

Another change can be made in connection with boolean expressions and array subscripting. As currently implemented both of these features allow simple operands but should be adjusted to accept simple expressions, such as

(i) A [I + 2] or A [I / 3]

and (ii) IF (I / 3 = 0) THEN.

A third change can be made in the program statement MAIN. As implemented now, the segment which it calls cannot have any arguments. Therefore, the change to be made, is to set up the program statement MAIN, to allow the segment to be called, to have arguments.

The last change I would like to suggest is the implementation of the boolean operator . NOT . before a variable name. This operator will be recognized but has not been implemented.

All of the above changes would make this compiler much more effective.

13.4 Wrap-Up

The project qualifies as a success since the basic aim has been fulfilled. I wish I had more time to tidy up the program since there are a number of instances, where specific code gets repeated a number of times. Given this time the project would have been polished up and further testing could have been accomplished.

Considerable knowledge and experience has been gained, from this project, through the use of the real time interactive system provided by SCOPE. I have also gained a better understanding about compilers in particular the cross compiler.

From this project I have also learned the importance of backup files since I accidentally purged one of my main files which I had to reenter.

I have also learned the importance of documentation in a program. Because it took me almost one day to figure out what was happening in a program that I was looking over.

APPENDICES

APPENDIX A

This appendix defines all global constants, types and variables used throughout the ASPLE+ Compiler.

Global Constants

ALNG — is set to ten. This is the maximum number of characters to be read into the array ≠ ID ≠.

ERRMAX — is set to twelve. This is the number of error messages that ASPLE+ has.

KMAX — is set to five. This is the maximum number of significant digits that an integer can have.

LEN — is set to forty-one. This is the number of key words that the language ASPLE+ has.

LLNG — is set to eighty. This is the maximum number of characters to be read by ASPLE+.

MAX — is set to five. This is the maximum length of a variable identifier.

NMAX — is set to 32767. This is the size of the largest positive integer that ASPLE+ can handle.

NMIN — is set to -37768. This is the size of the largest negative number that ASPLE+ can handle.

TMAX — is set to 128. This is the length of the identifier table ≠ TAB ≠.

Global Types

ALFA — refers to a packed character array, of length ≠ ALNG ≠.

ARRY — refers to an integer array which has five elements.

NAME — refers to a packed character array of length ≠ MAX ≠.

OBJECT — takes on one of the following values when a identifier is being declared.

(i) VARIABLE means the identifier name has been declared as a variable name.

(ii) PROZEDURE means the identifier name has been declared as a procedure name.

(iii) FUNKTION means the identifier name has been declared as a function name.

(iv) SEKMENT means the identifier name has been declared as the segment name.

(v) ARRAYS means the identifier names has been declared as an array.

(vi) STNG means the identifier name has been declared as a string.

(vii) CHRCH menas the identifier names has been declared as a character.

OPCODE — refers to a packed character array of length ≠ MAX ≠.

STYPE — will indicate whether a variable being declared is a local variable (LOC) or an external variable (EXT).

SYMBOL — defines a global type that takes on the values illustrated in TABLE 1 of Appendix B.

TYPES — takes on one of the following values when an identifier is being declared or when a key word has been found.

(i) NOTYP which means the variable being declared has no type.

(ii) INTS means the identifier has been declared as an integer.

(iii) BOOLS means the identifier has been declared as a boolean variable.

(iv) CHARS means the identifier has been declared as a character.

(v) STG means the identifier has been declared as a string.

(vi) LABL menas the identifier has been declared as a label.

(vii) ANDE means that the key word ≠ AND ≠ has been found between two boolean expressions.

(viii) ORE means the key word ≠ OR ≠ has been found between two boolean expressions.

TYPESY defines what type of operand I am working with. It takes on the values specified below.

(i) SNAM indicates that the operand is an identifier name.

(ii) NUM indicates that the operand is a number.

(iii) STR indicates that the operand is a string.

(iv) CHA indicates that the operand is a character.

(v) BLNK indicates that there is a blank operand.

Global Variables

CACC — defines the ≠ OPCODE ≠ value for the accumulator ≡ ACC≡ .

CAS — defines an array which is used to check the validity of the tag operands of a ≠ CASE ≠ statement.

CC — is a pointer to the current character being scanned by procedured ≠ LXSCAN ≠.

CH — has the value of the last character read from the source program.

CNAM — defines the ≠ OPCODE ≠ value for a name ≡ NAM≡ .

CNUM — defines the ≠ OPCODE ≠ value for a number ≡ NUM≡ .

CODE — is a file of text where the generating output code is written too.

COUNT — is used as the BEGIN-END counter.

COUT — defines an integer value used in the aid of generating code

ERRPOS — defines an integer which aids in the writing out the error meesage numbers.

ERRS — defines a set of integers from zero to ≠ ERMAX ≠. This aids in the printing of the error messages.

ID — will contain the identifier names that are read by procedure LXSCAN. This names can be up to ≠ ALNG ≠ in length.

IDTAB — is a table, defining the reserved (key) words used by the language ASPLE+. They are listed in TABLE 3 of Appendix B.

INDEX — defines an integer array which is ≠ TMAX ≠ in length. It is used in conjunction with the identifier table ≠ TAB ≠. It will hold the locations of the variable names in the

symbol table, such that, the symbol table can be written in alphabetical order.

INUM — will contain the value of the integer formed by subroutine ≠ LXSCAN ≠.

LINE — defines a character array which is ≠ LLNG ≠ in length. This array will hold one line of the source program that is read in by procedure ≠ NEXTCH ≠.

LL — defines the length of the source line read in by procedure ≠ NEXTCH ≠.

LOCTNS — will define the location of the procedure or function name in the symbol table when a call on this procedure or function occurs. This aids in parameter checking for procedures and functions.

LOUT — defines an integer for generating labels when generating output code.

LP	is the counter variable for the array ≠ STK ≠.
NAM	contains the identifier names declared in the ASPLE+ program.
NK	defines an integer which counts the parameter names, of a function or procedure definition, that have already been used in the main program.
NUL	defines a blank name of length ≠ MAX ≠.
OA	defines one of four ≠ OBJECT ≠ type values which are SEKMENT, PROZEDURE, FUNKTION or VARIABLE.
SK	is the counter variable for the array ≠ SKK ≠.
SKK	defines an array which stacks the labels generated by the ≠ FOR ≠ and ≠ REPEAT ≠ statements.

SPNAM — defines an array which stores the
(i) name of the segment to be called at the end of the program
(ii) name of the main program or the main segment.

SPNM — is the index to the array ≠ SPNAM ≠.

SPS — defines a character array of special symbols. These special symbols are listed in TABLE 2 of Appendix B.

STACK — defines a stack for storing identifier names.

STK — defines an integer array that stacks the labels being generated by the boolean expressions of the ≠ IF ≠ statement.

SY — contains the value of the last symbol type read by procedure ≠ LXSCAN ≠.

TAB — defines the symbol table which holds the information below

(i) NAMES is the identifier name

(ii) STYP declares the identifier name as a local or external declaration

(iii) OBJ defines the ≠ OBJECT ≠ type

(iv) TYP defines the ≠ TYPES ≠ type

(v) SIZE has two meanings

(a) defines the number of storage locations if the identifier name is a variable, array, string or character

and (b) defines the number of parameters a procedure, function or segment has.

(vi) VARS is an array which defines the procedure,function or segment parameter ≠ TYPES ≠ types.

(vii) ENTRY indicates whether an identifier name is an entry point, pointer variable, or neither.

TEST takes on one value of ≠ TYPES ≠. It is used in testing the validity of variable checking in all statements.

TK	defines an integer array which stacks the labels generated by the ≠ WHILE ≠ and ≠ CASE ≠ statements.
TP	is the counter variable for the array ≠ TK ≠.
TPROC	is used to count the number of parameters a procedure or function definition has.
TPTR	is the counter variable for the symbol table.
YSY	will contain one value of ≠ TYPESY ≠ when exitting from the procedure ≠ ADVANCE ≠.

APPENDIX B

TABLE 1

Symbol Types

Values	Meaning
INTCON	integer value
COLON	: (colon)
BECOMES	(assignment sign)
NEG	not equals
GTR	greater than
GEQ	greater than or equal to
LSS	less than
LEQ	less than or equal to
EQUL	equal to
LPARENT	left parenthesis
CHARCON	character
STRING	string
IDENT	identifier name
PLUS	plus sign
MINUS	minus sign

Values	Meaning
TIMES	times sign
IDIV	divide sign
SHIFT	shift sign
ROTATE	rotate sign
ANDS	and sign
ORS	or sign
RPARENT	right parenthesis
COMMA	comma
LBRACK	left bracket
RBRACK	right bracker
SEMICOLON	semicolon (;)
PERIOD	period
OFSY	the word ≠ OF ≠
REFSY	the word ≠ REF ≠
ENDEXTSY	the word ≠ ENDEXT ≠
EXTSY	the word ≠ EXTERNAL ≠
LOCSY	the word ≠ LOCAL ≠
ENDLOCSY	the word ≠ ENDLOCAL ≠
THENSY	the word ≠ THEN ≠
ELSESY	the word ≠ ELSE ≠
ENDIFSY	the word ≠ ENDIF ≠
ENDCASESY	the word ≠ ENDCASE ≠
ENDWHSY	the word ≠ ENDWHILE ≠

Values	Meaning
ENDFORSY	the word ≠ ENDFOR ≠
UNTILSY	the word ≠ UNTIL ≠
IFSY	the word ≠ IF ≠
CASESY	the word ≠ CASE ≠
WHILESY	the word ≠ WHILE ≠
DOSY	the word ≠ DO ≠
FORSY	the word ≠ FOR ≠
INSY	the word ≠ IN ≠
TOSY	the word ≠ TO ≠
DOWNTOSY	the word ≠ DOWNTO ≠
REPEATSY	the word ≠ REPEAT ≠
GOTOSY	the word ≠ GOTO ≠
BEGINSY	the word ≠ BEGIN ≠
ENDSY	the word ≠ END ≠
FUNCTSY	the word ≠ FUNCTION ≠
PROCSY	the word ≠ PROCEDURE ≠
SEGSY	the word ≠ SEGMENT ≠
MAINSY	the word ≠ MAIN ≠
NOTSY	the word ≠ NOT ≠
FALSESY	the word ≠ FALSE ≠
TRUESY	the word ≠ TRUE ≠
ARRAYSY	the word ≠ ATTAY ≠

TABLE 2

SPS Symbols

Symbol	Meaning
+	plus sign
-	minus sign
*	times sign
/	divide sign
)	right parenthesis
(	left parenthesis
=	equal sign
,	comma
[	left bracket
]	right bracket
;	semicolon
V	the and operator
∧	the or operator
↑	the shift operator
↓	the rotate operator

TABLE 3

List of Reserved Words

AND	ARRAY	BEGIN
BOOLEAN	CASE	CHAR
DO	DOWNTO	ELSE
END	ENDCASE	ENDEXT
ENDFOR	ENDIF	ENDLOCAL
ENDWHILE	EXTERNAL	FALSE
FOR	FUNCTION	GOTO
IF	IN	INTEGER
LABEL	LOCAL	MAIN
NOT	OF	OR
PROCEDURE	REF	REPEAT
SEGMENT	STRING	THEN
TO	TRUE	UNTIL
WHILE	XOR	

TABLE 4

List of Opcode Values

ADACC	ADINX	ADNAM
ADNUM	ANACC	ANINX
ANNAM	ANNUM	BEGIN
BSS	CALL F	CALLP
CALLS	DEF	DEFNF
DEFNP	DVACC	DVINX
DVNAM	DVNUM	ENDDF
ENT	EQ	EXT
GE	GT	IFBOL
IFCHA	IFEXP	IFSTR
JUMP	LDBOL	LDCHA
LDINX	LDNAM	LDNUM
LDSTR	LE	LT
NE	NOP	ORACC
ORINX	ORNAM	ORNUM
RTACC	RTINX	RTNAM
RTNUM	SBACC	SBINX
SBNAM	SBNUM	SHACC
SHINX	SHNAM	SHNUM
TMACC	TMINX	TMNAM
TMNUM		

TABLE 5

Special Symbols

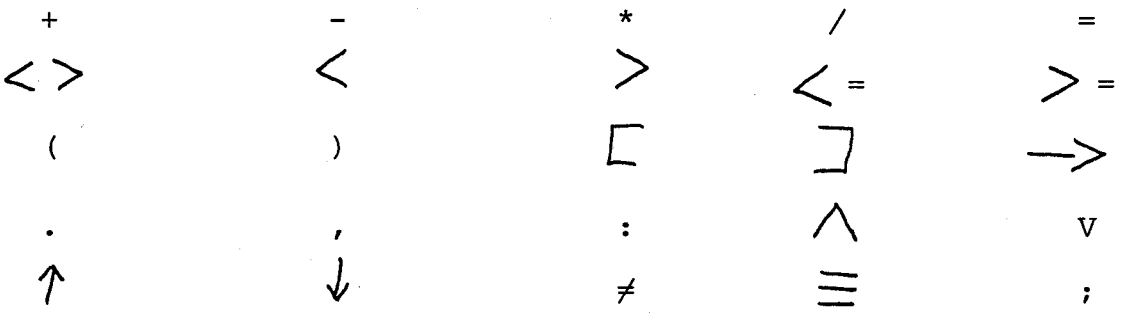


TABLE 6

Enumeration Operators

Symbol	Meaning
->	assignment
<	less than
=	equal to
>	greater than
< =	less than or equal to
> =	greater than or equal to
< >	not equal to

TABLE 7

Boolean Operators

. AND .

. NOT .

. OR .

TABLE 8

Numerical Operators

Symbol	Meaning
+	plus sign
-	minus sign
*	times sign
/	divide sign
$\wedge$	and sign
V	or sign
$\uparrow$	shift sign
$\downarrow$	rotate sign

TABLE 9

Rel Operators

Symbol	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
< >	not equal to

TABLE 10

Token Classes

Class	Meaning
ANDS	and sign
BECOMES	assignment operator
CHARCON	character
COLON	colon
COMMA	comma
EQUL	equal operator
GEQ	greater than or equal to operator
GTR	greater than operator
IDENT	identifier
IDIV	slask
INTCON	integer constant
LBRACK	left bracket
LEQ	less than or equal to operator
LPARENT	left parenthesis
LSS	less than operator
MINUS	minus sign
NEQ	not equal to operator
ORS	or sign

Class	Meaning
PERIOD	period
PLUS	plus sign
RBRACK	right bracket
ROTATE	rotate sign
RPARENT	right parenthesis
SEMICOLON	semicolon
SHIFT	shift sign
TIMES	times sign

APPENDIX C

This defines the local variables used by subroutines DECLARATION and GENCODE.

BL	is a boolean variable that indicates whether a variable name has been entered into the symbol table.
BOOL	is a boolean variable that indicates whether or not I am dealing with procedure or function parameter declaration.
BSS	indicates the opcode value ≡BSS≡
CO	is the current operator to CO - No table
DEF	indicates the opcode value ≡DEF≡

ENT	indicates (i) in DECLATION, which variable names are entry points or pointers and (ii) in GENCODE, the opcode value . ≡ENT≡ .
ENY	indicates to GENCODE, which variable names are entry points or pointers.
EXTS	indicates the opcode value ≡EXT≡ .
I	is the counter variable for array STACK.
J	is a control variable for a FOR statement.
LI	is also used as a counter for the array STACK

LNUM	indicates whether a procedure or function is declared as external.
N	is a parameter to TESTENTRY.
NO	is the next operator of CO - No table.
OB	indicates whether the identifier name is an array or a variable.
PT	indicates the identifier name in GENCODE
STY	indicates whether an identifier is declared externally or locally.
SZ	indicates the storage space needed for an identifier.
TY	indicates the variable type.

TYS	indicates to subroutine BUILD what kind of operand it is working with.

APPENDIX D

This appendix has the CO - NO tables used by PROCFN, DECLARATION and STATEMENT defined in it.

TABLE 1

Operators for Declaration

Operators	Meaning
LOCSY	the word LOCAL
RPARENT	), a right parenthesis
SEMICOLON	;, a semicolon
REFSY	the word REF
COMMA	, a comma
COLON	:, a colon
ARRAYSY	the word ARRAY
LBRACK	[, a left bracket
RBRACK	], a right bracket
OFSY	the word OF
ENDEXTSY	the word ENDEXT
CHARCON	≠, character variable
STRING	≡, string variable
ENDLOCSY	the word ENDLOCAL
TIMES	*, asterisk

TABLE 2

CO - NO Table for DECLARATION

Current Operator / CO — Next Operator / NO	LOCSY	RPARENT	SEMICOLON	REFSY	COMMA	ARRAYSY	LBRACK	RBRACK	OFSY	ENDEXTSY	CHARCON	STRING	ENDLOCSY	TIMES	EXTST	COLON
LOCSY	X	X		X	X	X	X	X	X		X	X	X	X	X	
RPARENT	X	X	X		X	X	X	X		X	X	X	X	X	X	
SEMICOLON	X	X	X		X	X	X	X		X	X	X	X	X	X	
REFSY	X	X	X		X	X	X	X	X	X	X	X	X	X	X	
COMMA		X		X		X	X	X	X	X			X			X
ARRAYSY	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
LBRACK	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X
RBRACK	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X
OFSY	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X
ENDEXTSY	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
CHARCON		X		X		X	X	X	X	X		X	X			X
STRING		X		X		X	X	X	X	X	X		X			X
ENDLOCSY	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
TIMES		X		X		X	X	X	X	X	X	X	X	X		X
EXTSY	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
COLON		X		X		X	X	X	X	X			X			X

TABLE 3

Operators For Statement

Operator	Meaning
UNTILSY	the word UNTIL
EXTSY	the word EXTERNAL
LOCSY	the word LOCAL
ENDIFSY	the word ENDIF
ENDCASESY	the word ENDCASE
ENDFORSY	the word ENDFOR
ENDWHSY	the word ENDWHILE
THENSY	the word THEN
ELSESY	the word ELSE
CASESY	the word CASE
IFSY	the word IF
WHILESY	the word WHILE
FORSY	the word FOR
REPEATSY	the word REPEAT
DOSY	the word DO
INSY	the word IN
DOWNTOSY	the word DOWNTO
TOSY	the word TO
GOTOSY	the word GOTO

Operator	Meaning
BEGINSY	the word BEGIN
ENDSY	the word END
PROCSY	the word PROCEDURE
FUNCTSY	the word FUNCTION
SEGSY	the word SEGMENT
MAINSY	the word MAIN
NOTSY	the word NOT
BECOMES	assignment operator,
LSS	<
LEQ	<=
GTR	>
GEQ	>=
EQUL	=
NEQ	<>
LPARENT	(
RPARENT	)
FALSESY	the word FALSE
TRUESY	the word TRUE
SEMICOLON	;
RBRACK	]
LBRACK	[
PERIOD	.
COLON	:

Operator	Meaning
COMMA	,
STRING	≡
CHARCON	≠
PLUS	+
MINUS	-
TIMES	*
SHIFT	↑
ROTATE	↓
ANDS	∧
ORS	V
IDIV	/

TABLE 4

Next Operator	Current Operators that next Operator can follow
UNTILSY	SEMICOLON, ENDIFSY, ENDFORSY, ENDWHSY, ENDCASESY
EXSTY	SEMICOLON
LOCSY	SEMICOLON
ELSESY ENDIFSY ENDCASESY ENDFORSY ENDWHSY	ENDIFSY, SEMICOLON, ENDCASESY, ENDFORSY, ENDWHSY
THENSY	RPARENT
CASESY IFSY WHILESY FORSY REPEATSY	THENSY, ENDIFSY, ENDCASESY, ENDWHSY, COLON ELSESY, DOSY, ENDFORSY, REPEATSY, RPARENT SEMICOLON, BEGINSY
DOSY	TOSY, DOWNTOSY
INSY	FORSY
DOWNTOSY TOSY	INSY

GOTOSY	ENDFORSY, ENDIFSY, THENSY, ELSESY, ENDCASESY, ENDWHSY, RPARENT, SEMICOLON
BEGINSY	SEMICOLON, ENDLOCSY
ENDSY	SEMICOLON, ENDIFSY, ENDWHSY, BEGINSY, ENDFORSY, ENDCASESY
PROCSY FUNCTSY	SEMICOLON, ENDLOCSY
MAINSY SEGSY	SEMICOLON
NOTSY	PERIOD
BECOMES	PLUS, MINUS, TIMES, SHIFT, ROTATE, ORS, ANDS, IDIV, ENDFORSY, ENDWHSY, ENDCASESY, ENDIFSY, THENSY, ELSESY, DOSY, REPEATSY, COLON. BEGINSY, SEMICOLON, RBRACK, STRING, CHARCON, TRUESY, FALSESY, RPARENT
LSS,LEG GTR,GEQ NEQ,EQUL	LPARENT, RBRACK
LPARENT	ENDCASESY, ENDIFSY, ENDFORSY, ENDWHSY, SEMICOLON, REPEATSY, THENSY, ELSESY, BEGINSY, DOSY, CASESY, IFSY, WHILESY, UNTILSY, PERIOD, RBRACK, STRING, CHARCON, RPARENT, PLUS, MINUS, TIMES, IDIV, ROTATE, SHIFT, ANDS, ORS

FALSESY TRUESY	LSS, LEQ, GTR, GEQ, NEQ, EQUL, BEGINSY ENDWHSY, REPEATSY, ENDCASESY, THENSY, ELSESY, ENDFORSY, SEMICOLON, RPARENT, DOSY
SEMICOLON	GOTOSY, ENDSY, BEGINSY, SEMICOLON, MAINSY, IDIV, BECOMES, FALSESY, TRUESY, RBRACK, RPARENT, ENDIFSY, ENDFORSY, ENDCASESY, THENSY, REPEATSY, ELSESY, DOSY
RBRACK	LBRACK
LBRACK	PLUS, MINUS, TIMES, IDIV, ROTATE, SHIFT, COLON, ANDS, ORS, ENDIFSY, THENSY, ELSESY, ENCASESY, ENDWHSY, RPARENT, BECOMES, LPARENT, BEGINSY, REPEATSY, ENDFORSY, DOSY, CASESY, SEMICOLON, LEQ, LSS, GTR, GEQ, NEQ, EQUL
PERIOD	ENDSY, LPARENT, RPARENT, PERIOD, NOTSY
COLON	SEMICOLON, ENDCASESY, ENDFORSY, ENDFISY, ENDWHSY, DOSY
COMMA	COMMA, LPARENT
STRING CHARCON	DOSY, THENSY, ELSESY, ENDWHSY, ENDCASESY ENDFORSY, REPEATSY, ENDIFSY, SEMICOLON, BEGINSY, LSS, LEQ, NEQ, EQUL, GTR, GEQ, LPARENT, COMMA

PLUS MINUS TIMES IDIV SHIFT ROTATE ANDS ORS	THENSY, ENDIFSY, ENDFORSY, ENDWHSY, ELSESY, ENDCASESY, REPEATSY, BEGINSY, COLON, SEMICOLON, DOSY, RBRACK, RPARENT
RPARENT	LSS, LEQ, NEQ, EQUL, GTR, GEQ, PERIOD, STRING, CHARCON, FALSESY, TRUESY, RBRACK, COMMA, LPARENT

TABLE 5

CO - NO Table for PROCFN

Current Operator / CO / Next Operator / NO	FUNCTION	RPARENT	TIMES	SEMICOLON	COLON	PROCEDURE	SEGMENT	IDIV	LPARENT
FUNCTION	X	X	X	X	X	X	X	X	X
RPARENT)	X	X	X	X	X	X	X	X	X
TIMES *		X	X	X	X		X	X	X
SEMICOLON ;	X			X					X
COLON :				X	X	X	X	X	X
PROCEDURE	X	X	X	X	X	X	X	X	
SEGMENT	X	X	X	X	X	X	X	X	X
IDIV /	X		X	X	X	X		X	X
LPARENT (		X	X	X	X				X

APPENDIX E

This Appendix describes the procedures declared inside procedure ≠ STATEMENT ≠ and some of the smaller procedures called by ≠ STATEMENT ≠.

CONTROL — calls procedure ≠ BUILD ≠ to define the predefined variables used in generating the output code.

FUNPRO — calls procedure ≠ BUILD ≠ when a call to a procedure or function occurs.

INITOP — initializes the opcode ≠ OPERT ≠ for the numerical operators given in TABLE 8 of Appendix B.

INTNAM — calls procedure ≠ BUILD ≠ to produce the proper output code if the procedure or function argument is a number. Otherwise the procedure or function argument name is stacked.

REST — completes the last three letters of the opcode ≠ OPERT ≠ depending on whether I am dealing with a number, array or the accumulator.

SIGN - SIGNS — initializes the opcode ≠ OPERT ≠ to the appropriate boolean operator of the boolean expression.

All the remaining procedures have one main function, that is to check that the next operator (NO) follows the appropriate current operator (CO).

BECOME — parses part of the assignment statement and calls ≠ BUILD ≠ to generate the proper reentrant code.

BEGINS — means I am parsing the beginning of the main program or the beginning of a function, procedure or segment.

CHARCN — means I am dealing with a character.

CMMA — means I am dealing with a comma that separates the arguments in a procedure or function call, or separates the tag operands of a ≠ CASE ≠ statement.

COLONN deals with a labelled statement.

DIVIDE deals with the slash in the segment or main statement or the divide sign.

DOS deals with the word ≠ DO ≠ which belongs to either a ≠ WHILE ≠ or ≠ FOR ≠ statement. If it is a ≠ FOR ≠ statement I must check the validity of the variable or number before the word ≠ DO ≠.

DOWNTOTO means I am parsing part of the ≠ FOR ≠ statement.

ELSEE and THEN means I am parsing the ≠ IF ≠ statement. The ELSE part of the ≠ IF ≠ statement is optional.

ENDCASE checks that the top element on the stack ≠ STCK ≠ is the word ≠ CASE ≠.

ENDFOR checks that the top element on the stack ≠ STCK ≠ is the word ≠ FOR ≠.

ENDIF	checks that the top element on the stack ≠ STCK ≠ is the word ≠ IF ≠.
ENDS	means that I am dealing with the end of the main program, function, procedure or segment.
ENDWHILE	checks that the top element of the stack ≠ STCK ≠ is the word ≠ WHILE ≠.
EXTLOC	means that I am beginning to declare the external or local declarations.
FUNCT, PROC, SEG	are dealing with the function, procedure, or segment definitions. This is done by calling procedure ≠ PROCFN ≠.
GOTOS	deals with the parsing of the ≠ GOTO ≠ statement.
INS	parses part of the ≠ FOR ≠ statement and checks the validity of the variable before the word ≠ IN ≠.

LEFTBRK — parses part of an array and must check to see if the variable infront of the left bracket is declared as an array.

LEFTPAR — marks the start of parsing of either

(i) a boolean expression

(ii) a tag operand of the ≠ CASE ≠ statement

(iii) a function or procedure call.

LLEGGN — parse the boolean operator of the boolean expression of the statements ≠ IF ≠, ≠ WHILE ≠ and ≠ UNTIL ≠.

MAINS — parse the main program statement.

NOTS — parses part of the boolean expression.

OPTER — parses the operators ORS, ROTATE, PLUS, TIMES, SHIFT, ANDS, MINUS of the assignment statement.

PERIODS — deals with the parsing of the

(i) period at the end of the main program

or (ii) period of a boolean expression.

RIGHTBRK deals with the parsing of the array subscript.

RIGHTPAR marks the end of parsing either the

(i) boolean expression

(ii) procedure or function call

(iii) tag operand of a ≠ CASE ≠ statement

SEMI marks the end of parsing either the

(i) assignment statement

(ii) goto statement

(iii) until statement

STATE means that I am starting to parse either the

(i) ≠ FOR ≠ statement

(ii) ≠ WHILE ≠ statement

(iii) ≠ REPEAT ≠ statement

(iv) ≠ IF ≠ statement

or (v) ≠ CASE ≠ statement.

This procedure stacks one of the words ≠ FOR ≠, ≠ WHILE ≠, ≠ REPEAT ≠, ≠ IF ≠ or ≠ CASE ≠ in the top element of the stack ≠ STCK ≠.

STRINGS	parses a string.
TRUEFALSE	deals with the parsing of the boolean variables ≠ TRUE ≠ and ≠ FALSE ≠.
UNTILS	checks that the top element on the stack ≠ STCK ≠ is the word ≠ REPEAT ≠.

APPENDIX F

This appendix defines the local variables used in the procedure ≠ STATEMENT ≠. The variable names may also be used as local variables in other procedures but there definitions are the same.

BL — is a boolean variable that indicates on return from ≠ TESTENTRY ≠ whether a variable has been declared or is undefined.

BOL — is a boolean variable that indicates that the ≠ ELSE ≠ statement entered the reentrant code to jump to a labelled statement.

BOOL — is a boolean variable that has a value of ≠ FALSE ≠ if the identifier name is a procedure or function call.

CO — is the current operator of the CO - NO table.

ENNY — is used in conjunction with the ≠ FOR ≠ statement to let me know if the ≠ FOR ≠ statement used the word ≠ TO ≠ or ≠ DOWNTO ≠.

ENY — is used to indicate whether a variable is an entry point, pointer variable or neither.

I — is the counter variable for the array ≠ STCK ≠

II — is the counter variable for the array ≠ CAS ≠.

J — is used as the control variable in the ≠ FOR ≠ statement.

JK — takes on the values

(i) 2 if the statement is an ≠ UNTIL ≠

(ii) 1 if the statement is a ≠ GOTO ≠ or if there is and ≠ ELSE ≠ statement which has no ≠ GOTO ≠ statement before it.

or (iii) 0 otherwise

JKC — is a counter variable for the array ≠ STACK ≠ when used for procedure or function calls.

JKK — indicates whether an ≠ IF ≠ statement has an ≠ ELSE ≠ (JKK = 1) statement.

JLK — indicates where, in array, ≠ STACK ≠, the control variable for the ≠ FOR ≠ statement is located.

JPK — indicates what boolean operator was found in the boolean expression of the ≠ IF ≠, ≠ WHILE ≠ and ≠ UNTIL ≠ statements.

LJK — is the counter variable for the array ≠ SWK ≠.

LLNUM — indicates that two boolean expressions are combined by the words ≠ AND ≠ or ≠ OR ≠ (LLNUM = 1000)

LNUM — takes on certain values to help generate code

LUCK — indicates that the words ≠ BEGIN ≠ and ≠ END ≠ belong to procedure or function definitions (LUCK = 20).

N — will contain the storage location of an identifier name on return from ≠ TESTENTRY ≠.

NO — is the next operator of the CO - NO table.

OB — is used to indicate whether an identifier name is an array or not.

OPERT — contains one opcode values in TABLE 4 of Appendix B that is used in generating the output code.

STCK — is an array that stacks the words ≠ FOR ≠, ≠ WHILE ≠, ≠ REPEAT ≠, ≠ IF ≠ and ≠ CASE ≠.

STY — tells me whether a variable is a local or external declaration.

SWK — is an array that contains the name of the ≠ CASE ≠ statement variable.

SZ	contains an integer value to be used by procedures ≠ ENTERVARIABLE ≠, ≠ BUILD ≠ and ≠ REST ≠.
TST	is used to aid in testing the validity of each statement.
TY	indicates whether a variable is an integer, character, string, boolean or a label.
TYS	indicates to procedure ≠ BUILD ≠ what type of identifier it is working with. It can be a variable name, integer or a blank.

APPENDIX G

This appendix defines the macro calls.

BEGIN Macro

This macro indicates the start of a procedure, function, segment or main program. This macro has one argument which is the name of the procedure, function, segment or main program.

Example

BEGIN ACT

CALL Macros

There are three CALL macros which are

(i) CALLF which defines a call to a function
(ii) CALLP which defines a call to a procedure
(iii) CALLS which defines a call to a segment.

These CALL macros can have one to ten arguments. The first argument is the procedure, function or segment. name. The other arguments are the arguments to the procedure, function or segment.

Upon calling the CALL macros, the accumulator will contain an integer value specifying the number of agruments a procedure, function or segment has.

Example

```
LDNUM   2
CALL    ACT, J, K
is a call on procedure
ACT which has two arguments J and K
```

DEFN Macros

There are two DEFN Macros which are

(i) DEFNF which will define a function

(ii) DEFNP which will define a procedure.

These two macros can have one to ten arguments. This first argument defines the function or procedure name.

The rest of the arguments will define the parameters for the procedure or function.

Before any procedure or function definition, the accumulator will be loaded with an integer to indicate how many parameters the procedure or function has.

Examples

```
(1)  LDNUM    0
     DEFNP    ACT
```

This defines the procedure ACT which has no parameters.

```
(2)  LDNUM    4
     DEFNF    ACT, I, J, K, L
```

This defines the function ACT which has the four parameters I, J, K, and L.

ENDDF Macro

This macro indicates the end of a procedure, function, segment or main program. It has one argument, which names the procedure, function, segment or main program.

Example

```
ENDDF    ACT
```

I have also assumed that on return from a function call, the value of the function will be left in the accumulator.

IF Macros

The first two letters of the macro name are IF. The last three letters specify what type of macro I am working with. These letters can be

(i) BOL to represent a boolean IF

(ii) CHA to represent a character IF

(iii) STR to represent a string IF

(iv) EXP to represent an expression IF.

The IF macros have three parameters which are

(i) the address of the variable to be tested against the accumulator

(ii) the operator being tested for. These are

LE	less than or equal to
LT	less than
GE	greater than or equal to
GT	greater than
NE	not equal
EQ	equal to

(iii) the address to jump to if the test is true.

Examples

```
(1)  LDNUM    10
     IFEXP     Y, GT, F$1
```

If the accumulator contents (10) is greater than the value stored at location Y then jump to address F$1.

```
(2)  LDBOL    TRUE
     IFBOL    BOOL, NE, F$2
```

If the accumulator contents (TRUE) is not equal to the boolean variable BOOL then jump to address F$2.

```
(3)  LDCHA    ;
     IFCHA    * + 1, EQ, F$3
```

If the accumulator contents (;) is equal to the contents of the storage location * + 1 then jump to address F$3.

```
(4)  LDSTR    P.SYS
     LFSTR    * + 3, NE, F$4
```

If the accumulator contents (P.SYS) is not equal to the contents of the storage location * + 3 then jump to the address F$4.

JUMP Macro

This macro defines a jump to a specified address when the next instruction is to be performed.

Example

```
JUMP     ONE
```

Load Macros

LDBOL Will load the value TRUE or FALSE into the accumulator.

Example

```
LDBOL     TRUE
```

LDCHA will load a character into the accumulator

Example

```
LDCHA     $
LDCHA     .
```

LDINX will load the value stored in an array element into the accumulator. This macro has two parameters. The first parameter is the address of the start of the array. The second parameter gives the address of the value of the array subscript wanted.

Example

```
LDINX     A, I
```

LDNAM will load the value stored at the address of the specified name into the accumulator.

Example

```
LDNAM    IJK
LDNAM    CHUCK
```

LDNUM will load an integer number into the accumulator.

Example

```
LDNUM    100
```

LDSTR will load a string into the accumulator.

Example

```
LDSTR    ABC
LDSTR    P.SYS
```

NOP Macro

This macro means that there is no operation to be performed.

Numerical Operator Macros

The first two letters of the macro name tells me what operators I am working with. For example

ADD	AD
SUBTRACT	SB
TIMES	TM
DIVIDE	DV
SHIFT	SH
ROTATE	RT
AND	AN
OR	OR

The last three letters of the macro call, state what I am working with. They can be

(i) NUM to represent a number

(ii) NAM to represent a name

(iii) INX to represent an array name and the subscript wanted.

All these macros work in conjunction with the accumulator.

Here are some examples

(1) ADNUM 10 will add the integer number ten to the accumulator contents.

(2) SBNAM I will subtract the contents stored at location I from the accumulator contents.

(3) TMNUM 3 will multiply the accumulator contents by three.

(4) DVINX A,I will divide the accumulator contents by the value stored at address A[I].

(5) SHNAM N will shift the accumulator contents by the value stored at location N.

(6) RTNUM 1 will rotate the accumulator contents by the integer number one.

(7) ANINX B,J will an the accumulator contents with the value stored at the address B[J].

(8) ORNAM T will or the accumulator contents with the value stored at location T.

All resulting values will be left in the accumulator.

APPENDIX H

This appendix defines the error messages.

Fatal Error

1. The compiler table for the Identifier table is two small.

Diagnostic Errors

0. Begin - Ends do not match up
1. Undefined character
2. Variable Type Conflict
3. CO - NO Table error
4. Multi - defined variable
5. Declaration Type error
6. UNTIL, ENDIF, ENDCASE, ENDWHILE or ENDFOR is missing
7. Undefined variable
8. Procedure or Function Parameter conflict
9. Undefined label
10. The number of parameters does not agree with the procedure or function definition.

11. Variable is not a function

12. Variable is not declared as an array.

APPENDIX I

This is a test program illustrating the different types of sentences used in the language ASPLE+.

```
(* THIS TEST PROGRAM ILLUSTRATES THE SYNTAX OF THE
≠ ASPLE+ ≠ LANGUAGE AND THE  OUTPUT   CODE GENERATED
BY EACH STATEMENT. *)

(* THE PROGRAM STATEMENT GIVES THE PROGRAM NAME AND THE
SEGMENT TO BE CALLED AT THE END OF THE PROGRAM. *)

MAIN SEGMT / TEST;

IDENT TEST

*+1      BSS 1

*+2      BSS 1

*+3      BSS 1

*+4      BSS 1

*+5      BSS 1

(* THEN EXTERNAL DECLARATIONS DEMONSTRATE THE SYNTAX
FOR DECLARING VARIABLE NAMES. *)

EXTERNAL

(* THE NEXT SIX STATEMENTS DECLARE VARIABLE NAMES WHICH
ARE LESS THAN FIVE CHARACTERS IN LENGTH AS

        (A)  AN INTEGER

        (B)  A BOOLEAN

        (C)  A CHARACTER

        (D)  A STRING
```

```
        (E)  A POINTER
    AND (F)  A LABEL          *)
A  :  INTEGER ;
EXT A
EOS  :  BOOLEAN ;
EXT EOS
XYZZ  :  CHAR ;
EXT XYZZ
BCD  :  STRING ;
EXT BCD
EE  :  REF INTEGER;
EXT EE
ONE  :  LABEL ;
EXT ONE
(* THE NEXT COUPLE OF LINES INDICATE HOW A VARIABLE
NAME IS CUT DOWN TO FIVE CHARACTERS WHEN BEING DECLARED *)
ABCDEFGHI  :  INTEGER ;
EXT ABCDE
WXYZZABUVW  :  BOOLEAN ;
EXT WXYZZ
(* A STRING CAN BE DECLARED IN ASPLE+ AS FOLLOWS *)
≡ABCD≡  :  INTEGER ;
EXT ABCD
```

```
(* THIS LINE SHOWS HOW A STRING NAME IS CUT DOWN TO
FIVE CHARACTERS *)

=P.SYSTEM= : INTEGER ;

EXT P.SYS

(* THIS LINE INDICATES HOW A CHARACTER CAN BE DECLARED *)

≠ $ ≠ : INTEGER ;

EXT $

(* THE NEXT LINE INDICATES THE END OF THE EXTERNAL
DECLARATIONS *)

ENDEXT

(* NOW LETS LOOK AT WHAT OTHER TYPES OF DECLARATIONS CAN
BE HANDLED. THIS IS DONE THROUGH THE USE OF THE LOCAL
DECLARATIONS *)

LOCAL

(* I CAN DECLARE AN ARRAY AS

        (A)  INTEGER

        (B)  BOOLEAN

        (C)  STRING

    AND (D)  CHARACTER

AS FOLLOWS *)

AB  :  ARRAY[10]OF INTEGER ;

AB      DEF *+1

        BSS 10

WCABD  :  ARRAY[5] OF BOOLEAN ;

WCABD   DEF *+1

        BSS 5
```

```
STRG   :  ARRAY [25] OF STRING ;
STRG      DEF *+1
          BSS 25
CHR    :  ARRAY [25] OF CHAR ;
CHR       DEF *+1
          BSS 25
(* VARIABLES, STRINGS AND CHARACTERS CAN BE DECLARED
AS ENTRY POINTS AS FOLLOWS *)
* WXYZ   :  INTEGER ;
WXYZ      BSS 1
ENT WXYZ
*≡P.SYTEM≡ :  INTEGER ;
P.SYT     BSS 1
ENT P.SYT
* ≠]≠  :  INTEGER ;
 ]        BSS 1
ENT ]
(* ARRAYS CAN DECLARED AS ENTRY POINTS AS FOLLOWS *)
*ABD   :  ARRAY [10]  OF INTEGER ;
ABD       DEF *+1
ENT ABD
BSS 10
* UVZZ  :  ARRAY [5] OF INTEGER ;
```

```
UVZZ    DEF *+1
ENT UVZZ
BSS 5
(* THE FOLLOWING LINES SHOW HOW TO DECLARE MORE THAN
ONE VARIABLE IN THE SAME LINE *)
BL,BOL,BOOL,  :  BOOLEAN ;
BL      BSS 1
BOL     BSS 1
BOOL    BSS 1
LABE, GABE, TABE, SABE  :  ARRAY [5] OF INTEGER ;
LABE    DEF *+1
        BSS 5
GABE    DEF *+1
        BSS 5
TABE    DEF *+1
        BSS 5
SABE    DEF *+1
        BSS 5
≠ . ≠, ≠ , ≠ , ≠< ≠  :  CHAR ;
.       BSS 1
,       BSS 1
<       BSS 1
≡AMN≡, ≡AUVYZZZ≡  :  STRING ;
AMN     BSS 1
```

```
AUVYZ    BSS 1
* LMK,≡AMNOPGQ≡, ≠>≠  :  INTEGER ;
LMK      BSS 1
ENT LMK
AMNOP    BSS 1
 >       BSS 1
TWO, THREE, FOUR  :  LABEL ;
I, J, *≡ZAZDZZZ≡, ZZZ, * ≠ + ≠  :  INTEGER ;
I        BSS 1
J        BSS 1
ZAZOZ    BSS 1
ENT ZAZOZ
ZZZ      BSS 1
+        BSS 1
ENT  +
(* THE NEXT LINE MARKS THE END OF THE LOCAL DECLARATIONS *)
ENDLOCAL
(* THIS IS HOW TO DECLARE AN EXTERNAL PROCEDURE *)
(* THE ≠ BEGIN ≠ OF A PROCEDURE OR FUNCTION INDICATES
THE START OF THE PROCEDURE OR FUNCTION.  THE ≠ END ≠
INDICATES
THE END OF A PROCEDURE OR FUNCTION *)
PROCEDURE * MOD (M,N  :  INTEGER) ;
EXTNUM 0
EXT MOD
```

```
BEGIN

BEGIN MOD

END ;

ENDDF MOD

(* THIS IS HOW TO DECLARE AN EXTERNAL FUNCTION *)

FUNCTION * SIN (L  :  INTEGER)  : INTEGER ;

LDNUM 0

EXT SIN

BEGIN

BEGIN SIN

END ;

ENDDF SIN

(* DECLARING A LOCAL PROCEDURE WITH PARAMETERS *)

PROCEDURE MOVE (U  :  INTEGER ; BOSS  :  BOOLEAN) ;

LDNUM 2

DEFNP MOVE, U, BOSS

BEGIN

BEGIN MOVE

(* THIS IS ONLY A STUB SINCE IT IS ONLY A TEST PROGRAM
SHOWING HOW THE STATEMENTS ARE SET UP AND THE CODE
GENERATED FROM THEM *)

END ;

ENDDF MOVE

(* DECLARING A LOCAL FUNCTION WITH PARAMETERS IS SHOWN
IN THE NEXT FEW LINES *)
```

```
FUNCTION ADD (V  : INTEGER ; W  :  INTEGER)  :  INTEGER ;

LDNUM 2

DEFNF ADD, V, W

BEGIN

BEGIN ADD

(* IS A STUB FOR DEMONSTRATION ONLY *)

END ;

ENDDF ADD

(* DECLARING A PROCEDURE WITH NO PARAMETERS IS CARRIED
OUT AS FOLLOWS *)

PROCEDURE SUB ;

LDNUM 0

DEFNP SUB

BEGIN

BEGIN SUB

END ;

ENDDF SUB

(* DECLARING A FUNCTION WITH NO PARAMETERS IS DONE AS
FOLLOWS *)

FUNCTION BOLN  :  BOOLEAN ;

LDNUM 0

DEFNF BOLN

BEGIN
```

```
BEGIN BOLN

END ;

(* START OF MAIN PROGRAM *)

BEGIN

BEGIN TEST

(* INITIALIZATION OF CONSTANTS *)

0 ---> I ;

LDNUM 0

STNAM I

10 ---> J ;

LDNUM 10

STNAM J

10 ---> A ;

LDNUM 10

STNAM A

(* THE STATEMENTS ILLUSTRATE THE ASSIGNMENTS STATEMENTS
USING (A) THE PLUS SIGN *)

A  +  A ---> ABCDEFGHI ;

LDNAM A

ADNAM A

STNAM ABCDE

(* (B) THE MINUS SIGN *)

A  -  A ---> ZZZ ;

LDNAM A
```

```
SBNAM A
STNAM ZZZ
(* (C) THE DIVIDE SIGN *)
ABCDEFGHI —> A ;
LDNAM ABCDE
STNAM A
(* (D) THE TIMES SIGN *)
ZZZ   *   A —> ZZZ ;
LDNAM ZZZ
TMNAM A
STNAM ZZZ
(* (E) THE AND SIGN *)
A ∧ ZZZ —> ZZZ ;
LDNAM A
ANNAM ZZZ
STNAM ZZZ
(* (F) THE OR SIGN *)
ABCDEFGHI ∨ ZZZ —> A ;
LDNAM ABCDE
ORNAM ZZZ
STNAM A
(* (G) THE SHIFT SIGN *)
A ↑ 10 —> ZZZ ;
```

```
LDNAM A

SHNUM 10

STNAM ZZZ

(* AND (H) THE ROTATE SIGN *)

ZZZ ↓ A →A ;

LDNAM ZZZ

RTNAM A

STNAM A

(* INITIALIZATION OF A BOOLEAN VARIABLE *)

TRUE → EOS ;

LDBOL TRUE

STNAM EOS

(* HOW THE ACCUMULATOR CAN BE USED IN THE ASSIGNMENT
STATEMENT.  IT REPRESENTS THE NUL OPERAND. *)

A  +  A→ ;

LDNAM A

ADNAM A

ZZZ  + →A ;

LDNAM ZZZ

ADACC

STNAM A

+  A → ZZZ ;

ADNAM A

STNAM ZZZ
```

```
(* THE ASSIGNMENT STATEMENT USING AN ARRAY AS AN
OPERAND *)
AB [10] + A -> AB [ZZZ] ;
LDNUM 10
STNAM *+4
LDINX AB, *+4
ADNAM A
STINX AB, ZZZ
A + AB [J] -> ZZZ ;
LDNAM A
ADINX AB, J
STNAM ZZZ
LABE [ZZZ] / SABE [A] -> GABE [5] ;
LDINX LABE, ZZZ
DVINX SABE, A
STNAM *+1
LDNUM 5
STNAM *+4
LDNAM *+1
STINX GABE, *+4
(* THE ASSIGNMENT OF A STRING *)
= ABCDEFGHI= -> STRG [J] ;
LDCTR ABCDE
```

```
STINX STRG, J
≡ ZWXUVVFGH≡ -> STRG [25] ;
LDSTR ZWXUV
STNAM *+1
LDNUM 25
STNAM *+4
LDNAM *+1
STINX STRG, *+4
(* ASSIGNING A CHARACTER *)
≠  :  ≠ -> CHR [ZZZ] ;
LDCHA
STINX CHR, ZZZ
≠  Z  ≠ -> CHR [23] ;
LDCHA Z
STNAM *+1
LDNUM 23
STNAM *+4
LDNAM *+1
STINX CHR, *+4
(* A LABELLED STATEMENT *)
CNE  :  A  +  ZZZ -> SABE [J] ;
ONE LDNAM A
ADNAM ZZZ
```

```
STINX SABE, J
(* THE WHILE STATEMENT *)
WHILE  (  A >=  10  )  DO
F $ 1  LDNAM A
STNAM *+1
LDNUM 10
IFEXP *+1, GT, F $ 2
A  *  ZZZ ->A ;
LDNAM A
TMNAM ZZZ
STNAM A
ZZZ  /  A ->AB [2] ;
LDNAM ZZZ
DVNAM A
STNAM *+1
LDNUM 2
STNAM *+4
LDNAM *+1
STINX AB, *+4
ENDWHILE
JUMP F $ 1
A -> ZZZ ;
F $ 2  LDNAM A
STNAM ZZZ
```

```
(* THE REPEAT STATEMENT *)
REPEAT
I  +  1 ->I ;
F $ 3  LDNAM I
       ADNUM 1
       STNAM I
≠  (  ≠ ->CHR [I] ;
LDCHA (
STINX CHR, I
=ABC= ->STRG [I] ;
LDSTR ABC
STINX STRG, I
UNTIL  (  I  =  25  ) ;
LDNAM I
STNAM *+1
LDNUM 25
IFEXP *+1, NE, F $ 3
(* THE FOR STATEMENT USING THE RESERVED WORD ≠ TO ≠ *)
FOR I IN 1 TO J DO
LDNUM 1
F $ 4  STNAM I
IFEXP J, GT, F $ 5
```

```
10 -> AB[I] ;
LDNUM 10
STINX AB, I
ENDFOR
LDNAM I
ADNUM 1
JUMP F $ 4
(* THE FOR STATEMENT USING THE RESERVED WORD ≠ DOWNTO ≠ *)
FOR I IN 5 DOWNTO 1 DO
F $ 5  LDNUM 5
F $ 6  STNAM I
LDNUM 1
STNAM *+1
LDNAM I
IFEXP *+1, LT, F $ 7
FALSE -> WCABD[I] ;
LDBOL FALSE
STINX WCABD, I
I -> TABE[I] ;
LDNAM I
STINX TABE, I
 -> GABE[I] ;
LDACC
```

```
STINX GABE, I
--> SABE[I] ;
LDACC
STINX SABE, I
ENDFOR
LDNAM I
SBNUM 1
JUMP F $ 6
(* THE IF STATEMENT WHICH USES THE GOTO STATEMENT *)
IF ( EOS = TRUE ) THEN
F $ 7  LDBOL TRUE
IFBOL EOS, EQ, F $ 8
GOTO TWO ;
JUMP TWO
ELSE
A  +  I --> I ;
F $ 3 LDNAM A
ADNAM I
STNAM I
ZZZ + -> ZZZ ;
LDNAM ZZZ
ADACC
STNAM ZZZ
```

```
ENDIF
(* THE IF STATEMENT WITHOUT THE ≠ ELSE ≠ *)
IF ( AB[ I] < LABE[ ZZZ] ) THEN
LDINX AB, I
STNAM *+1
LDINX LABE, ZZZ
IFEXP *+1, LE, F $ 9
GOTO ONE ;
JUMP ONE
ENDIF
I + 1->I ;
F $ 9  LDNAM I
ADNUM 1
STNAM I
(* THE CASE STATEMENT *)
TWO  :  CASE TABE  5
TWO  LDNUM 5
STNAM *+4
(0) FALSE -> EOS ;
LDINX TABE, *+4
STNAM *+3
LDNUM 0
IFEXP *+3, NE, F $ 11
```

```
LDBOL FALSE
STNAM EOS
(1)  A -> ZZZ ;
JUMP F $ 10
F $ 11  LDNUM 1
IFEXP *+3, NE, F $ 12
LDNAM A
STNAM ZZZ
J  +  I -> A ;
LDNAM J
ADNAM I
STNAM A
TRUE -> EOS ;
LDBOL TRUE
STNAM EOS
(2,4) FALSE -> EOS ;
JUMP F $ 10
F $ 12  LDNUM 2
IFEXP *+3, NE, F $ 13
JUMP F $ 14
F $ 13  LDNUM 4
IFEXP *+3, NE, F $ 15
```

```
F $ 14  LDBOL FALSE
STNAM EOS
(3,5) ZZZ -> A ;
JUMP F $ 10
F $ 15  LDNUM 3
IFEXP *+3, NE, F $ 16
JUMP F $ 17
F $ 16  LDNUM 5
IFEXP *+3, NE, F $ 18
F $ 17  LDNAM ZZZ
STNAM A
J + I -> ZZZ ;
LDNAM J
ADNAM I
STNAM ZZZ
TRUE -> EOS ;
LDBOL TRUE
STNAM EOS
ENDCASE
F $ 18  NOP
F $ 10  NOP
(* USING A FUNCTION IN AN ASSIGNMENT STATEMENT *)
```

```
I + ADD (J, ZZZ) ⟶ I ;
LDNAM I
STNAM * + 1
LDNUM 2
CALLF ADD, J, ZZZ
STNAM * + 2
LDNAM * + 1
ADNAM * + 2
STNAM I
SIN (10)  /  J ⟶ A ;
LDNUM 10
STNAM * + 1
LDNUM 1
CALLF SIN, * + 1
DVNAM J
STNAM A
BOLN ⟶ EOS ;
LDNUM 0
CALLF BOLN
STNAM EOS
(* DEMONSTRATION OF PROCEDURE CALLS *)
MOD (I, J) ;
LDNUM 2
CALLP MOD, I, J
```

```
SUB ;

LDNUM 0

CALLP SUB

(* THE ≠ END ≠ STATEMENT OF THE MAIN PROGRAM SHOWS
THE CODE GENERATED FOR THE SEGMENT CALL SPECIFIED BY
THE MAIN PROGRAM AS WELL AS INDICATING THE END OF THE
MAIN PROGRAM *)

END.

LDNUM 0

CALLS SEGMT

ENDOF TEST
```

IDENTIFIER TABLE

A	VARIABLE	INTEGER	EXT	1	
AB	ARRAY	INTEGER	LOC	10	
ABCD	VARIABLE	INTEGER	EXT	1	
ABCDE	VARIABLE	INTEGER	EXT	1	
ABD	ARRAY	INTEGER	LOC	10	ENTRY POINT
ADD	FUNCTION	INTEGER	LOC	2	
AMN	VARIABLE	STRING	LOC	1	
AMNOP	VARIABLE	INTEGER	LOC	1	
AUVYZ	VARIABLE	STRING	LOC	1	
BCD	VARIABLE	STRING	EXT	1	
BL	VARIABLE	BOOLEAN	LOC	1	
BOL	VARIABLE	BOOLEAN	LOC	1	
BOLN	FUNCTION	BOOLEAN	LOC	0	

BOOL	VARIABLE	BOOLEAN	LOC	1	
CHR	ARRAY	CHARACTER	LOC	25	
EE	VARIABLE	INTEGER	EXT	1	POINTER
EOS	VARIABLE	BOOLEAN	EXT	1	
FOUR	VARIABLE	LABEL	LOC	1	
GABE	ARRAY	INTEGER	LOC	5	
I	VARIABLE	INTEGER	LOC	1	
J	VARIABLE	INTEGER	LOC	1	
LABE	ARRAY	INTEGER	LOC	5	
LMK	VARIABLE	INTEGER	LOC	1	ENTRY POINT
MOD	PROCEDURE		LOC	2	ENTRY POINT
MOVE	PROCEDURE		LOC	2	
ONE	VARIABLE	LABEL	EXT	1	
P.SYS	VARIABLE	INTEGER	EXT	1	
P.SYT	VARIABLE	INTEGER	LOC	1	ENTRY POINT
SABE	ARRAY	INTEGER	LOC	5	
SEGMT	SEGMENT		LOC	1	
SIN	FUNCTION	INTEGER	LOC	1	ENTRY POINT
STRG	ARRAY	STRING	LOC	25	
SUB	PROCEDURE		LOC	0	
TABE	ARRAY	INTEGER	LOC	5	
TEST	VARIABLE		LOC	1	
THREE	VARIABLE	LABEL	LOC	1	

TWO	VARIABLE	LABEL	LOC	1	
UVZZ	ARRAY	INTEGER	LOC	5	ENTRY POINT
WCABD	ARRAY	BOOLEAN	LOC	5	
WXYZ	VARIABLE	INTEGER	LOC	1	ENTRY POINT
WXYZZ	VARIABLE	BOOLEAN	EXT	1	
XYZZ	VARIABLE	CHARACTER	EXT	1	
ZAZDZ	VARIABLE	INTEGER	LOC	1	ENTRY POINT
ZZZ	VARIABLE	INTEGER	LOC	1	
+	VARIABLE	INTEGER	LOC	1	ENTRY POINT
$	VARIABLE	INTEGER	EXT	1	
,	VARIABLE	CHARACTER	LOC	1	
.	VARIABLE	CHARACTER	LOC	1	
⌐	VARIABLE	INTEGER	LOC	1	ENTRY POINT
<	VARIABLE	CHARACTER	LOC	1	
>	VARIABLE	INTEGER	LOC	1	

BIBLIOGRAPHY

(1) Halstead, A Laboratory Manual for Compiler and Operating System Implementation. New York, American Elsevier Publishing Company, Inc, 1974

(2) A Pocket Guide to the 2100 Computer, California, Hewlett Packard Company, 1972

(3) Dr. N. Solntseff, A Pascal - 6000 Primer, Hamilton, McMaster University Press, 1975

(4) Grishman, Ralph, Assembly Language Programming, New York, Algorithmics Press, 1974.