

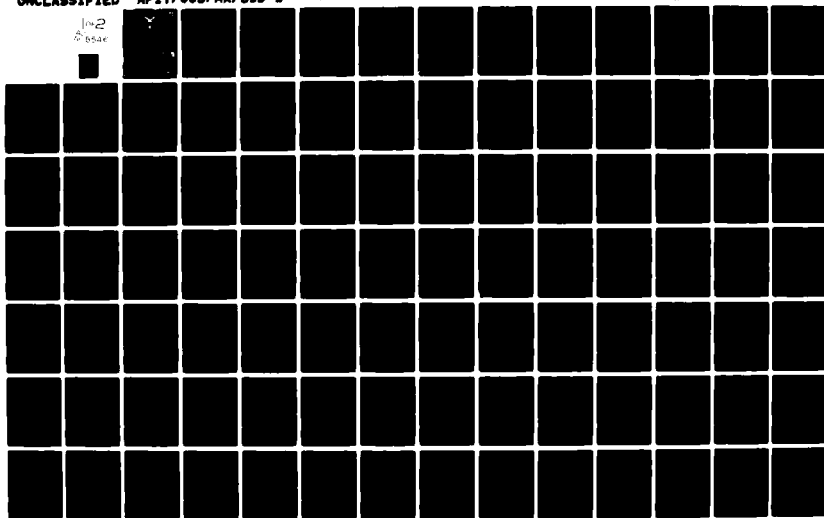
AD-A115 546

AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/6 9/2
AN INTERMEDIATE LANGUAGE AND INTERPRETER FOR THE ASSOL GRAPHICS--ETC(U)
DEC 81 K P ALBERT
AFIT/OC5/MA/81D-1

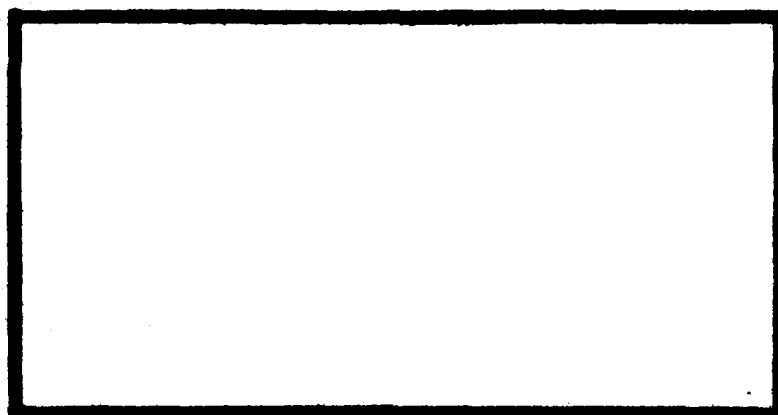
UNCLASSIFIED

NL

inv2
65ae



- AD A115546



DTIC
EXCERPTED
JUN 15 1982
H

UNITED STATES AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

DTIC FILE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

82 06 14 166

12

An Intermediate Language and
Interpreter for the
ASGOL Graphics Language

AFIT/GCS/MA/81D-1 Kevin P. Albert
Capt USAF

DTIC
ELECTE
JUN 15 1982

DISSEMINATION STATEMENT A
Approved for public release;
Distribution Unlimited

AFIT/GCS/MA/81D-1

An Intermediate Language and
Interpreter for the
ASGOL Graphics Language

Thesis

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

Kevin P. Albert, B.A.

Capt. USAF

Graduate Computer Systems

December 1981

Approved for public release; distribution unlimited.

DISTRIBUTION STATEMENT A

Approved for public release:
Distribution Unlimited

Preface

The construction of languages and compilers has always been an area of fascination for me. It is not surprising, then, that when the opportunity to do this thesis presented itself I did not hesitate.

I would like to take this opportunity to thank Major Michael C. Wirth for his support as my advisor on this thesis project. I would also like to thank Professor Charles Richard and Captain Roie Black for their ideas and comments. In addition I would like to thank Mike Luthman of AFWAL/ACD for his comments during the preparation of the users manual.

Finally, I would especially like to thank my wife Beth for her understanding and support during the course of the project. Without her encouragement the project would never have been completed on time.



Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

Contents

Preface	11
List of Figures	.v
Abstract	vi
I. Introduction	1
Problem and Scope	.2
Assumptions	.3
Approach	.3
Organization	.4
II. Requirements and Specifications	.6
User Requirements	.6
DISSPLA Requirements	.7
System Configuration	10
III. ASGOL Language Design	13
Format	13
Block Structure	16
Loop and Control Structures	19

IV. Parser	21
Scanner	21
Parser	24
LR(1)	24
Automatic Parser Generator	25
Construction	27
Semantic Routines	27
Symbol Table	29
Implementation of Labels	33
V. Intermediate Language and Interpreter	36
AIL	36
Interpreter	38
VI. Applications	42
VII. Recomendations	50
Segmentation	50
Error Recovery	51
Optimization of AIL	52
Functions and Procedures	52
Use of Additional DISSPLA Features	53
Bibliography	54
Appendix A: BNF of ASGOL	56
Appendix B: AIL Instructions	83
Appendix C: Conditional and Looping Productions	89
Appendix D: Compiler Routines	93
Appendix E: Interpreter Routines	98
Appendix F: Users Manual	103

List of Figures

<u>Figure</u>	<u>Page</u>
1. DISSPLA Processing	11
2. Functional System Chart	12
3. Program layout with Comments	14
4. ASGOL Program Levels	18
5. Scanner Organization	23
6. LR Parsing Steps	25
7. Parser organization	28
8. Hash Function Example	30
9. Hash Chaining	32
10. Lexical Level Termination	34
11. Interpreter Machine Structure	39
12. Use of Display Areas	41
13. Line and Arrow Instructions	43
14. Line and Arrow Plot	44
15. Bar and Linear Text	45
16. Bar and Linear Graph	46
17. Pie Graph Example	47
18. Text Instructions	48
19. Text Graph	49

Abstract

In an effort to make ASGOL (ALGOL-Structured Graphics Oriented Language) more powerful, conditional and looping instructions were added to the language. To do this the existing interpreter system was converted to a compiler/interpreter system. An intermediate language was designed as the compiler output and thus the source language for the interpreter.

An Intermediate Language and Interpreter for the ASGOL Graphics Language

I. Introduction

Anyone who has had a large amount of data to analyze is well aware that computer graphics is a very powerful tool. Not only is much time saved by having the computer do most of the processing, but by using a graphic format the results are presented in a clear, concise form. This is particularly important for any manager or commander who needs those results to make critical decisions.

Thus, there is a need for graphics that can be produced quickly and accurately with a minimum of effort. However, the people who possess the detailed knowledge and training for this job are not always available, leaving the responsibility for the creation of meaningful products to people with little knowledge of computers or graphics. Therefore, various graphic packages have evolved to simplify this process while still giving the desired results.

The Air Force Wright Aeronautical Laboratory (AFWAL/ACD) recognized the need for a management graphics system to interface with the Display Integrated Software System and Plotting Language (DISSPLA) graphics package (Ref 1) that is now being used. DISSPLA is a very complex system and requires a FORTRAN program to make it work. Unfortunately, the office in AFWAL that uses the system has no full time programmers. Therefore, it was essential that an easier way of using the system be found.

To solve this problem the lab sponsored a thesis by Lieutenant James D. Hart (Ref 2). The goal of Hart's effort was to develop a system that would be easily readable and easily used by anyone without a detailed understanding of FORTRAN or the DISSPLA system. In that thesis Lt. Hart designed a language (ASGOL) and set up an interpreter system to execute it.

Problem and Scope

There are two problems with Lt. Hart's system. The first and most important deals with its structure. Each instruction is executed and discarded as soon as it is decoded, leaving no way for conditional and looping type structures to be implemented. Since these structures are useful, this thesis effort involved the redesign of ASGOL and the interpreter system to include them. With the

exception of some simplification of ASGOL, only those changes necessary to implement the desired instructions were done.

The second problem involves the size of the system. Due to the large size requirements of DISSPLA, this system cannot be run interactively. As this was one of the sponsor's requirements, this problem was investigated.

Assumptions

The two major parts of the system, the LR(1) parser and the DISSPLA graphics system, were kept with few changes. It was accepted that a table driven LR(1) parser and the DISSPLA graphics package were the best tools available. Therefore, no attempt was made to identify or implement a replacement for either one.

Approach

The first step in this thesis effort was to identify the most useful conditional and looping structures. Then, in order to implement them within ASGOL, an intermediate language (AIL) was designed. This allowed the execution of the program to be delayed until after all the instructions had been decoded.

Two major changes were then made to the system to incorporate AIL. First the parser had to be modified to produce the appropriate AIL commands as the program instructions were decoded. After this was done it was necessary to design and implement a new interpreter to execute AIL and produce the plots. These modifications effectively changed the ASGOL system from a direct interpreter to a compiler/interpreter combination.

Organization

Included in the remainder of this thesis is a discussion of the requirements and specifications of the system (Chapter II) and a discussion of ASGOL (Chapter III).

Chapter IV describes the scanner and the parser while Chapter V covers AIL and its interpreter. Chapter VI is included to provide an analysis of the applications of the system. Finally, Chapter VII gives conclusions and recommendations for enhancements.

Several appendices are also included to provide more detailed information. Appendix A is a listing of the formal definition of ASGOL and Appendix B lists the AIL statements. Appendix C then details the steps involved in each production of the looping and control statements. Appendices D and E describe the software modules associated with the scanner, the parser, and the interpreter. With

Appendices A, B and C they make up a maintenance manual for the system. Finally a users manual is included as Appendix F.

II. Requirements and Specifications

User Requirements

Once a month AFWAL/ACD, the thesis sponsors, generate a Commander's report (a series of graphs and tables) that currently is created in part using the DISSPLA system. The tables, however, are done by hand. The ASGOL system was designed to generate the appropriate DISSPLA commands to allow the entire report to be created using the CDC CYBER computer. In performing this function it was requested that the ASGOL system have the following characteristics:

1. easily used,
2. capable of using data from user specified files,
3. capable of running selected parts of the report,
4. capable of graphics blow-up.

The ease of use requirement was met by designing ASGOL as a high-level language. This allowed it to be made machine independent and thus requires no specific knowledge of any computer system by the user. In addition, ASGOL was made free format to allow a user complete freedom to organize the program in a clear, meaningful manner. Perhaps more important to this requirement, though, was the incorporation in ASGOL of as many English language instructions as possible. A user needs only to state what

is wanted and does not have to worry about the translation of terms into DISSPLA commands. For example, the specification of plot type is done simply by naming the type desired; LINEAR, BAR, PIE, etc. ASGOL keeps track of the relationship of these terms to their DISSPLA routines. See Chapter 3 for more detail on the structure of ASGOL.

The capability to use data from user specified files was incorporated into ASGOL through the use of the INPUT instruction. This instruction allows the source tape number to be specified when the program is written making any data file available to the system.

The last two capabilities, running selected parts and graphic blow-up, are available in the DISSPLA Post Processor System (Ref 3). Therefore no special action was taken in the ASGOL system to duplicate any of these functions. The following description of the DISSPLA system will discuss these functions in more detail.

DISSPLA Requirements

The DISSPLA graphic system consists of a number of integrated subroutines that create a device independant plot file. On the CYBER these routines are written in FORTRAN IV and thus they require any program that calls them to be written in the same language. Any other language (PASCAL,

FORTTRAN V, FORTRAN 77, etc.) will not work because the I/O done by the DISSPLA routines will have unpredictable results (Ref 4). It was this fact that placed probably the most severe restriction on the ASGOL system. By forcing the system to be written in FORTRAN IV all of the convenience and power of the other languages was unavailable, resulting in a needlessly complex system.

To create a plot DISSPLA requires that information be specified in four areas: page size, axis lengths, origin, and step sizes. These areas must be specified in the order listed since each depends on the information in the areas preceeding it (Ref 2: Part A 3-1). It is ASGOL's responsibility to ensure that all needed information is specified in the correct order as the user is not required to have an understanding of the DISSPLA structure. When the information for a plot is being accumulated the DISSPLA system goes through a series of four levels. After each successive level is reached it is not possible to return to a lower level until the plot is finished. Therefore, since all information has a specific level or levels that it can be defined on, ASGOL must also ensure that a new level is reached only after all required information has been specified. Table 1 lists these levels and their meaning.

TABLE I
DISSPLA Levels

Level	Status
0	Initial Status
1	After Initialization Routine
2	Page Dimensions, Axis lengths and Physical Origin Have Been Defined
3	Step Size Defined. Plot is Fully Determined.

The DISSPLA system also consists of a number of Post Processors which prepare output for various display devices. Currently available on the CYBER are the following: CALCOMP, Tektronix 4010, Tektronix 4014, ZETA plotter, and CDC274 (Ref 3:2). These processors take the file created by the plotting routines and allow the plots to be seen on a number of devices (Figure 1). In addition the plots can be modified according to the following features:

1. selective plotting,
2. windowing of a plot (blow-up),
3. relocation of plots relative to each other,
4. scaling,

5. superimposing of plots (Ref 3:5).

These features are well defined in the DISSPLA manual and required only minor actions from the ASGOL system to be available.

System Configuration

The ASGOL system consists of three main parts; the Parser, the Semantic routines, and the Interpreter. The parser performs the syntax checking of ASGOL and when applicable calls the appropriate semantic routine. See Chapter 4 for details of the parser construction. The semantic routines in turn check the semantics of the language and generate the appropriate AIL instructions to perform the required plot functions. These routines are part of the parser and are also discussed in more detail in Chapter 4. Finally, the interpreter is responsible for decoding and executing the AIL instructions and generating the necessary DISSPLA commands. See Chapter 5 for a complete discussion of the structure and function of the interpreter.

By using a file to pass the AIL instructions from the parser to the interpreter the ASGOL system was separated into a two step process. This allows the parser to be run to check syntax without running the interpreter. In the

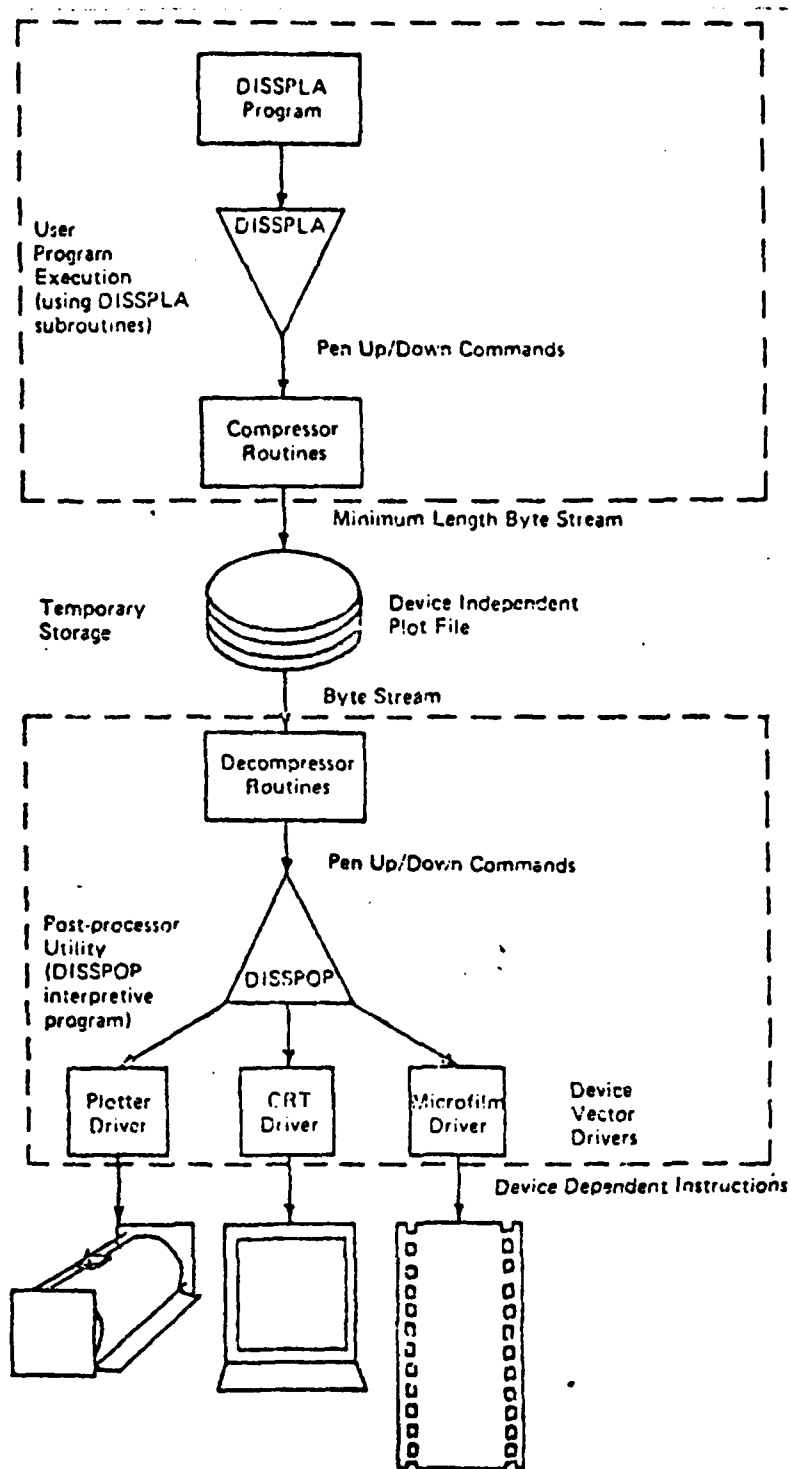


Figure 1. DISSPLA Processing

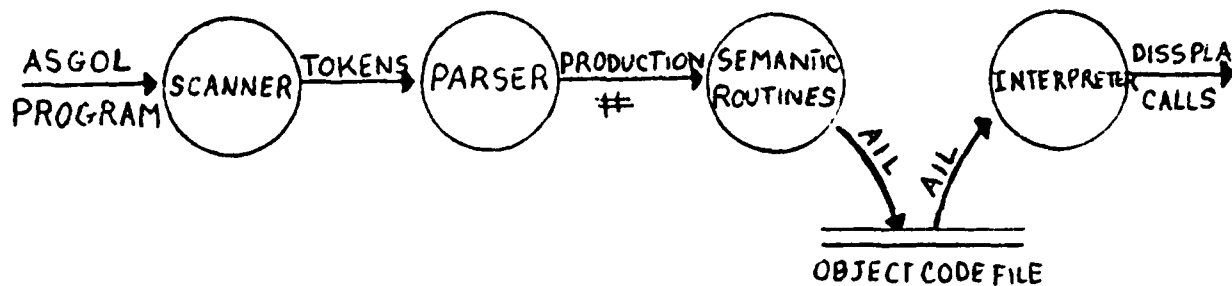


Figure 2. Functional System Chart

same way the interpreter can be run as often as desired without having to rerun the parser portion of the system. Figure 2 shows the relationship of the parts of the ASGOL system.

III. ASGOL Language Design

Format

The intention of ASGOL, as designed by Lt. Hart (Ref 1:7-17), was to be as independent of DISSPLA and as machine independent as possible. The first objective was met by assuring that a one-to-one correspondance of ASGOL statements to DISSPLA commands was not forced upon the user. ASGOL instructions were designed to be meaningful English language statements and, so, one statement could generate a sequence of DISSPLA commands or no commands at all. (See Appendix A for a listing of the ASGOL productions.) Although this makes the code generation in the parser more difficult, the advantages to the user are well worth the extra effort as this effectively frees the user from the need to know any details of the DISSPLA system.

By basing the language on ALGOL, a high-level language, the second objective was achieved. A high-level language can be largely machine independent, therefore, it is not necessary to even know what computer the system is to be run on to write the ASGOL program. However, this does not imply that the current implementation can be run on any machine since it contains several CDC dependent routines. It simply means that an ASGOL language contains no machine dependant

features.

For further ease in using the language, ASGOL was also made free format. This means that there is no column dependency (such as in FORTRAN) for the instructions. The user is totally free to space the instructions over one or several lines as necessary for program clarity. There is also no restriction on the number of lines or cards that can be used for this purpose.

PROGRAM example

DECLARE

CONSTANT PI = 3.14159 /* comment on same line */

VARIABLE

A, /* instruction over many lines */

B, /* with comments within the */

C, /* instruction. */

D : INTEGER

END DECLARE

END PROGRAM example.

Figure 3. Program Layout With Comments

To complement this freedom of spacing, the ability to place comments anywhere within a program, was provided. Comments begin with the characters /* and end with */. Anything in between is ignored by the compiler so comments can appear on the same line as an instruction or even within an instruction. Figure 3 shows examples of line spacing and comments. Note that the spacing can be used for indentation. With these features, the ASGOL program should be easily readable and understandable.

Unlike some other languages that have been designed to interface with DISSPLA (Ref 5), ASGOL was not structured to be open-ended. In an open-ended language it is necessary to specify only the beginning parameters in a list if the remaining are to be used with their default values. This approach works well until it becomes necessary to change only a parameter at the end of the list. In this case the default values, if known, have to be entered or a series of commas must be used to designate the parameters' positions. It is obvious that this can lead to difficulties if a default value is forgotten or if the number of commas is miscounted. To avoid these difficulties, any parameter that is optional in ASGOL will assume a default value if it is not specified. Just removing the parameter from the list is sufficient to accomplish this. For example, in the definition of a page the margin parameter is optional. If included the instruction becomes:

PAGE example (VERTICAL,LEFT RESET,1.0,TOP).

If, however, the default value is all that is needed the instruction simplifies to the following:

PAGE example (VERTICAL,1.0,TOP).

Not only is this easier to deal with, it eliminates the potential for errors present in the other approaches.

Block Structure

When large programs are written it is commonly agreed that they need to be divided into smaller modules whenever possible. This is true for the following reasons:

1. improved readability;
2. improved testability;
3. enhanced organization (Ref 6:281).

To accomplish this aim, subprograms have been incorporated into most languages. Initially the two types, procedure and function, were allowed to communicate with the main program only through parameters passed by the calling mechanism. This approach was sufficient as long as the desired amount of data to be shared was small. If not, the parameter list could be quite long, thus, creating two bad situations: "first, no one wants to type lists with 15 or 20 members; second, a long list is error prone in that a programmer could easily permute the order of arguments or leave one

out" (Ref 6:298). In an effort to eliminate this problem, FORTRAN allows COMMON blocks to be defined and shared among subprograms. Although the initial problem is solved by this approach others are created. In making the module interfaces more complex, the reliability of the program is decreased and therefore debugging time is increased.

From a software engineering point of view this is not really an acceptable situation. Therefore, a different type of program structure was developed. This is known as block structure and it uses the physical organization of the modules to determine data access rights. Only modules that are physically nested within another module can use the data within that module. This allows the user to set up a program in such a way that any module has access only to the data that it needs (Ref 6:298-299).

As an added advantage, a block structure approach allows some space to be saved at execution time. Since the data for a particular block is only needed while that block is executing, dynamic storage allocation can be used. When a block is activated, storage for its variables is allocated and when execution for that block is complete the storage is returned to be used by the next block. Depending on the program, this could result in a major savings of storage space (Ref 6:300).

```
PROGRAM one
  SECTION one
    PAGE one
      SEGMENT one
        END SEGMENT one
      END PAGE one
    END SECTION one
  END PROGRAM one.
```

Figure 4. ASGOL Program Levels

In the original design of ASGOL it was desired that it be as efficient and easy to use as possible. In addition the amount of storage used by the system was a concern. Therefore, it was decided to use a block structure. Not only was the design of ASGOL programs made simpler through the use of modularity but a savings in storage was also realized. Within an ASGOL program there are four possible levels that can be used. They are PROGRAM, SECTION, PAGE, and SEGMENT and they are the only procedure blocks available. Figure 4 shows the relationship of these levels. See Appendix F for a detailed discussion of their meaning and use.

Loop and Control Structures

Looping and control statements are an essential part of any language. Because few applications allow a strictly sequential approach, a language is severely restricted without these statements. Therefore, ASGOL was modified to include five looping and control statements.

IF THEN ELSE. This statement can take one of the following two forms:

IF expression THEN commands END IF

IF expression THEN commands ELSE commands END IF

FOR. The for statement also has two forms and is used to repeat a sequence of instructions a specific number of times.

FOR variable = expression TO expression BY expression

DO commands END FOR

FOR variable = expression DOWN TO expression BY

expression DO commands END FOR

WHILE DO. This statement is used to execute a sequence of instructions as long as a given expression is true.

WHILE expression DO commands END WHILE

REPEAT UNTIL. This statement is the same as the WHILE DO with the exception that the instructions are executed until the specified expression becomes true.

REPEAT commands UNTIL expression END REPEAT

CASE. This statement provides a multi-path branching capability for ASGOL. Although the same function could be accomplished using nested IF THEN ELSE statements, the CASE statement was provided for clarity.

```
CASE variable OF
    list : commands
    list : commands
    OTHERS : commands
END CASE
```

See Chapter 4 for a discussion of how these statements are handled by the semantic routines and Appendix F for rules concerning their use.

IV. Parser

Scanner

The first step in any compiler or interpreter system is to reduce the input language string to a form that is more easily used by the computer. It is both wasteful and unnecessary to deal with a character string such as "DECLARE" when an integer value would work just as well. The integer could convey the same meaning and also save space while simplifying processing. This is just one of the functions of the scanner.

Since the input STREAM is treated as one continuous character string, the scanner must perform all the following functions:

1. Identify reserved words, PROGRAM, DECLARE, FRAME, etc;
2. Identify special symbols, =, <, >, etc;
3. Identify and convert numeric constants, 10, 27.4, 131, etc;
4. Identify and collect character constants, "ABC", "X123", etc;
5. Identify and collect identifiers.

ASGOL's scanner performs all of these functions and conveys the results to the rest of the system through its calling parameter, ITOK, and several associated variables, RVAL, IVAL, SYMSTR, STRING, and SLNGTH. For reserved words

and special symbols ITOK's value is an index into the vocabulary array. No other information is required for processing of this type of information since it is usually only necessary to know that the reserved word or special symbol is present. The remaining three functions, however, require more information to be meaningful.

When a numeric constant is encountered, ITOK, is set to a predefined value and the constant is converted to machine format. It is then placed into one of two variables; IVAL, if it is an integer, or RVAL, if it is real. These variables are in the COMMON block LEXCOM so they are thus available to the rest of the system. Character constants are handled in a similar manner with a predefined value being assigned to ITOK. No conversion is performed though. Instead the characters are placed one character at a time, as they are read, into the array STRING. When the end of the string is detected, STRING contains the entire character string with one character stored per word. The string's length is then put into the variable SLNGTH. STRING and SLNGTH can also be found in the COMMON block LEXCOM. If the scanner cannot identify a character string as either a reserved word or a special symbol, it assumes the string is an identifier. In this case the string is put into the array SYMSTR, also found in LEXCOM, and ITOK is set to the predefined identifier value.

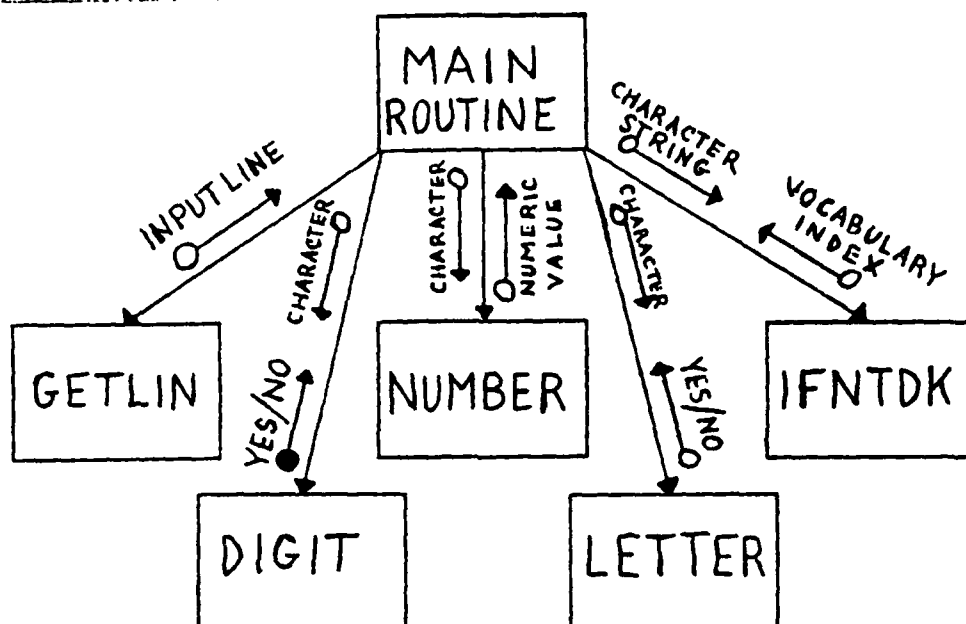


Figure 5. Scanner Organization

Construction. Figure 5 shows the relationships between the various scanner routines. They consist of the main routine, the input routine GETLIN, the conversion routine NUMBER, and the identification routines IFNDTK, DIGIT, and LETTER. See Appendix E for more information on the specific functions of each of these routines.

Parser

LR(1). During the design of ASGOL, the decision was made to use an LR parsing approach. Although the major motivation was probably the ability to utilize the Automatic Parser Generator obtained from Lawrence Livermore Laboratory (Ref 7), there are many other advantages to this technique. When compared to an LL parsing approach it has been shown that LR parsing is not only more powerful (any LL(K) grammar is also LR(K)) but there are available commercially proven tools to construct LR parsers from a given grammar (Ref 8:157). In addition LL parsing is a top down recursive technique and since the ASGOL system was forced to be written in FORTRAN IV, which does not easily support recursion, LL parsing was not practical.

The LR parsing technique is a left-to-right, bottom-up approach. It starts with an input string and then reduces it to a goal symbol (Ref 8:158). This is accomplished by performing the steps in Figure 6. When neither a reduction nor a transition is possible a syntatic error has been detected, meaning that the rules of the grammar have not been followed correctly. At this point parsing cannot continue unless some kind of error recovery is performed. Unfortunately error recovery for a table driven parser is a difficult process. Therefore little was done within the scope of this thesis to find a solution to this problem.

```
STEP1:  GET TOKEN FROM SCANNER
STEP2:  IF REDUCTION POSSIBLE
        PERFORM REDUCTION
        GO TO STEP1
STEP3:  IF TRANSITION POSSIBLE
        PERFORM TRANSITION
        GO TO STEP1
```

Figure 6. LR Parsing Steps

See Chapter 7 for a further discussion of this problem.

Automatic Parser Generator. The basis of the parser used in the ASGOL system is a series of tables produced by an automatic parser generator obtained from the Lawrence Livermore Laboratory (Ref 7). This program takes a context free grammar expressed in a modified BNF format and generates tables that describe an LR(1) parsing automaton. These tables are then incorporated into the parser and are all that is required to verify the syntax of the language. This approach is particularly useful if any changes are made to the language since they can be included simply by generating new tables (Ref 7:1).

This, however, does not mean that a language can be expanded indefinitely. As a language grows the size of the table output also grows because the table output is roughly linearly proportional to the number of states in the automaton. These states are generated at a rate of about one for every right hand side symbol in the BNF. Since the average production length of most languages seems to be about two symbols, the parser will have roughly twice as many states as productions (Ref 7:7). When space is a concern, as it was with the ASGOL system, this can have a limiting effect on the size of the language.

In addition to this concern there is a problem with this table approach that is not discussed in the Automatic Generator documentation but needs to be mentioned. As long as syntax checking is all that is required this system works well. However, ASGOL, as well as other languages, also needs to have semantic checking and code generation done. The parser is set up to do this by passing the production number of the reductions that are being performed to a set of semantic routines. These routines can then do any required validation and code generation. As long as no changes are made to the language there is no problem, but when a change is made and new tables generated the production numbers may change. Since the production number is the key to the semantic routines a way of correlating the numbers to their appropriate routines is necessary. This

can be done two ways. First, if the language is small enough the routines can be shifted as the numbers change. However, as the number of productions increases this approach can result in a great deal of work and a potential for mistakes when the shifting is done. The second way to solve this problem is to create a cross reference table that keeps track of the routines' location. When new tables are then generated this table is all that needs to be changed. The ASGOL parser uses this approach through the table XREF. Appendix A shows the relationship of the productions and their respective semantic routines for the BNF of ASGOL.

Construction. The parser consists of several routines that determine reductions, transitions, and semantics. Figure 7 shows the relationship of these routines. See Appendix 4 for a detailed description of each.

Semantic Routines

It is not enough just to know that a program is syntactically correct. When variables are used in a program it is usually necessary to know more than just their names. Other information, such as type, dimension, etc., is also needed if the data referred to by that variable is to be processed correctly. It is one of the functions of the

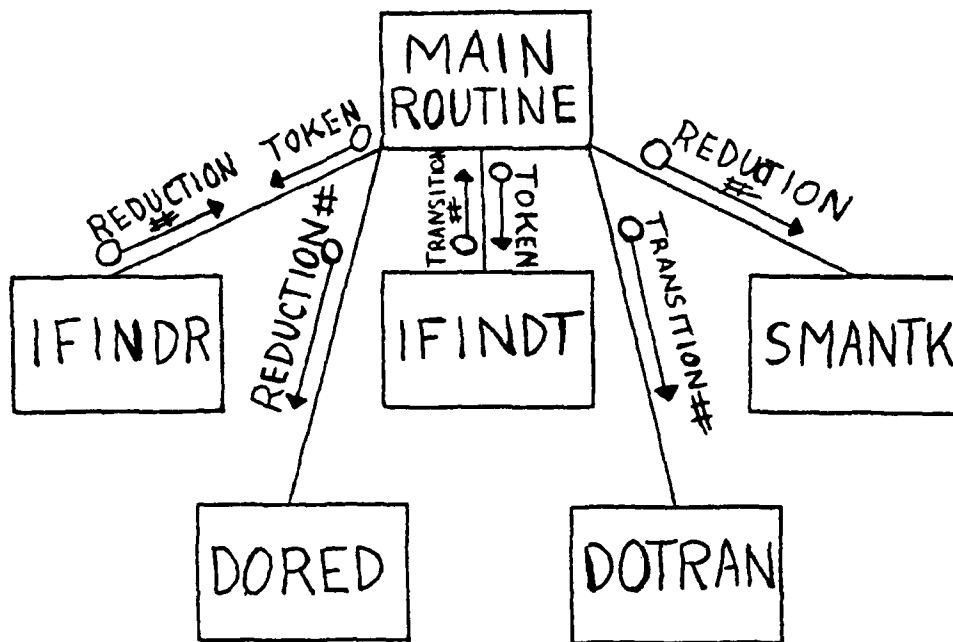


Figure 7. Parser Organization

semantic routines to keep track of this information and to insure that all references to variables are consistent with it.

The other main function of the ASGOL semantic routines is to generate AIL instructions. These are instructions to the interpreter and will be discussed in detail in Chapter 5.

Symbol Table. The symbol table is made up of 9 arrays, BSCTYPE, LEXLEV, POINT, ARRAY, DEFINED, MODE, ENTRYPT, NEXTENT, and NAME. By storing in these arrays information about each variable used in the program, the semantic routines are able to verify that a variable is being used correctly. However, since most programming languages, ASGOL included, have an extremely large set of possible identifier names, (FORTRAN contains approximately $1.62 \times 10^{**9}$), it is not possible to have one entry in these arrays for each possible identifier (Ref 9:111). Instead only variables that have been used in the program are put into the symbol table. This solution does decrease the amount of storage space required but it also leaves the problem of searching the table for desired identifiers. Of the many different organizational and searching techniques (Ref 9, Ref 10), a hashing approach was chosen for the ASGOL system.

"A hashing function partitions the set of all source codes into equivalence classes such that two source codes are in the same equivalence class if and only if they have the same bit pattern" (Ref 10:114). This is a strict definition of a hashing function and simply means that the function operates on an identifier to obtain a value. The value is then used as an index into the symbol table. The array ENTRYPT is used for this purpose as shown in Figure 8. The value obtained from evaluating in sequence the following three hash functions is used as a pointer into the array

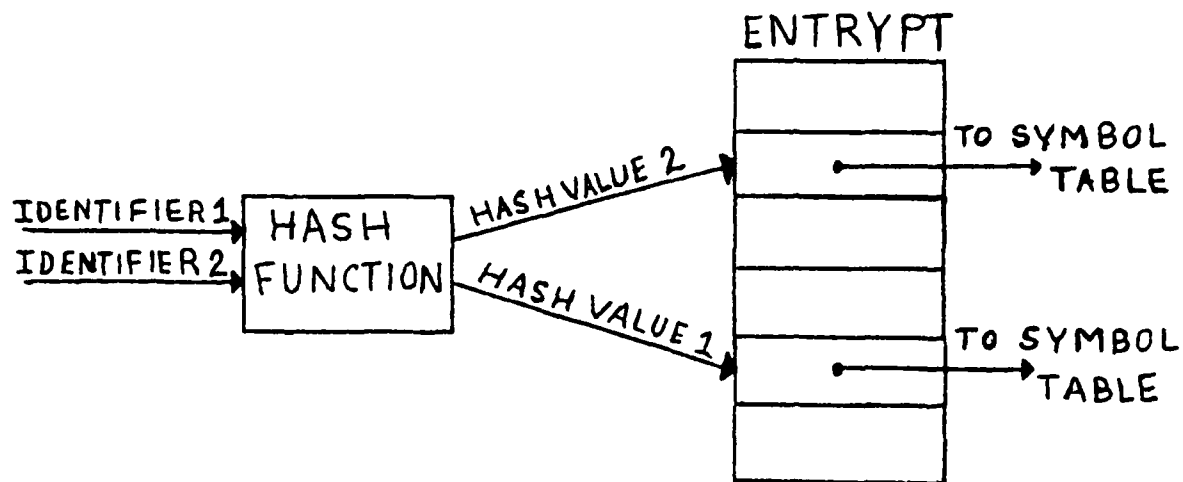


Figure 8. Hash Function Example

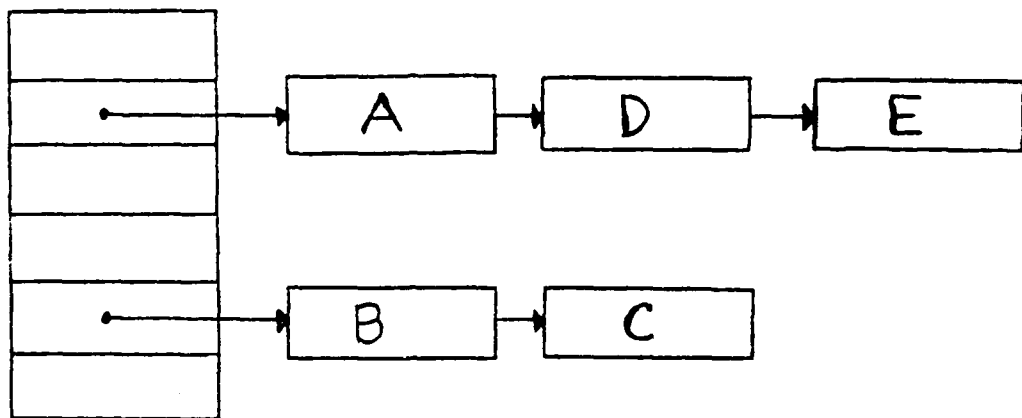
ENTRYPT which then contains a pointer into the other symbol table arrays.

```

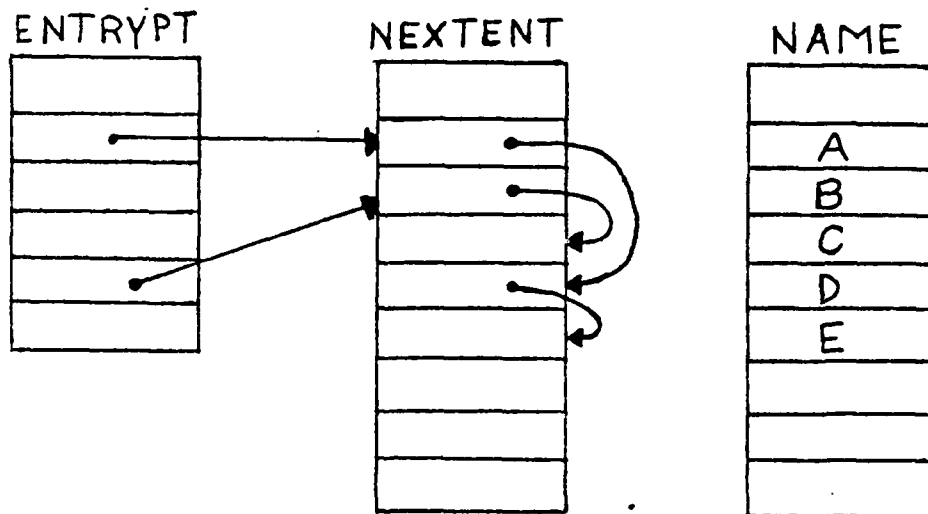
value = sym(1) + 3*sym(2) + 5*sym(3) + 7*sym(4) +
        11*sym(5) + 13*sym(6) + 17*sym(7) +
        19*sym(8)
value = int(value * 160795.0/262144.0 * 50 + 0.5)
value = mod(value,50) + 1
  
```

With the large number of identifiers possible and the limited number of entries in ENTRYPT, 50, it is clear that not all of the values can be unique. In other words two identifiers may yield the same value. This is called a collision and can be handled in several ways. The ASGOL system uses the array NEXTENT to help resolve these conflicts. With this array all identifiers that result in the same value are put into a chain as shown in Figure 9. Since all the arrays, except ENTRYPT, of the symbol table are parallel the information for any variable in the chain can be obtained simply by using the current index of the NEXTENT array.

Because of space limitations it was necessary to restrict the symbol table to only 100 identifiers. Although this does not seem to be enough for a large program, it is sufficient if the block structure of ASGOL is utilized. By only keeping identifiers that are part of active blocks the same space in the table can be used again and again. This, however, presents the problem of identification of the variables to be removed from the table when a block ends. The array LEXLEV is used for this purpose. As each new block is entered the lexical level of the system is increased and this value is placed into the symbol table for every variable defined in that block. In addition as each variable is added to the table it is put at the head of a hash chain. Since identical variable names can be used in



(A). Hash Chains



(B). Hash Chain Implementation

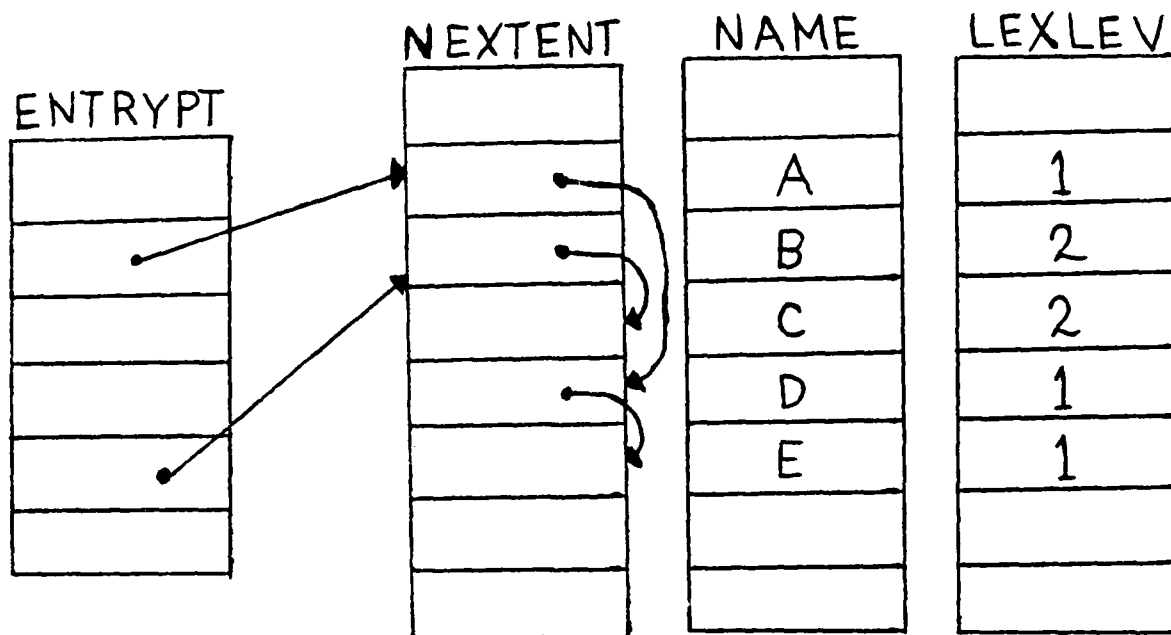
Figure 9. Hash Chaining

different this also assures the latest definition will always be the one used. This also makes deallocation of the variables easier. When a block is exited all the variables in the table that have the same lexical number as the block are removed. To do this each chain is searched to the end or until a lower lexical number is found. Clearly not all of the table has to be searched resulting in faster execution. Figure 10 shows how this has been implemented.

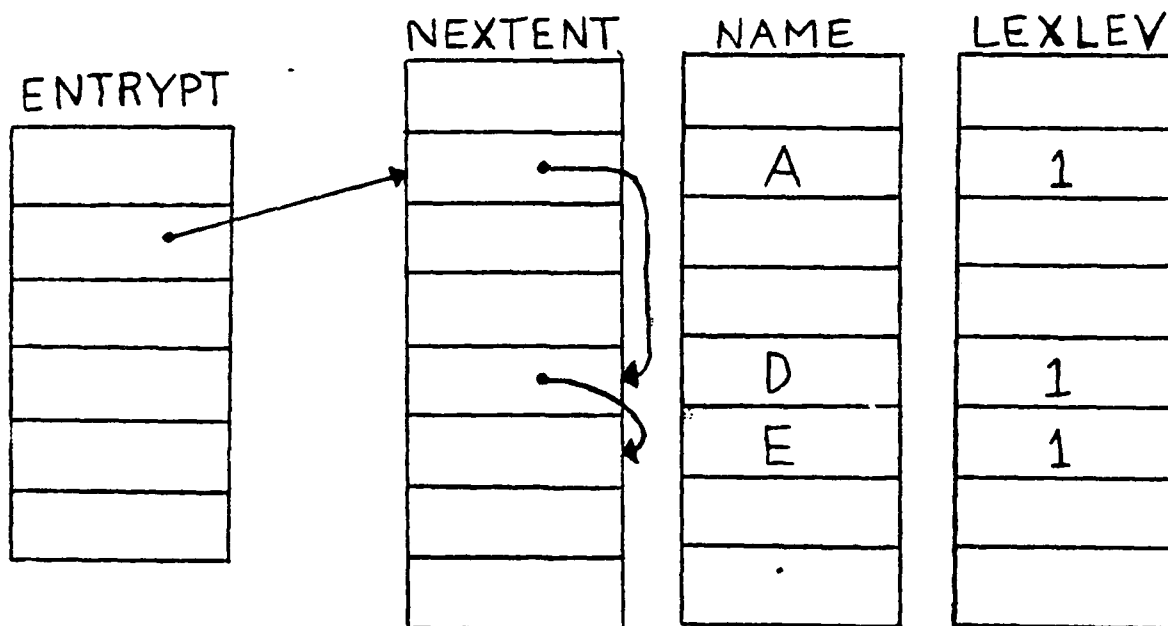
Implementation of Labels. When looking at the productions for the conditional and looping statements, Appendix C, the primary confusion is likely to be the handling of labels in the intermediate language. Since some labels must be referenced before they are defined it is not always readily apparent what is happening during code generation.

To resolve labels the ASGOL system uses a stack and two tables, BRSTACK, and LABSTK. Whenever a label is needed a special routine, GETLAB, is called to generate a unique label number. This number is then used for all subsequent references to that label.

If the label is being referenced by a jump instruction then the label number and the address in the instruction stack of the AIL instruction are put into the BRSTACK table. This information is thus held until the label is defined. At this point two things happen. First the location of the



(A). During Execution of Level 2



(A). After Termination of Level 2

Figure 10. Lexical Level Termination

;

label in the instruction stack and the label number are saved in the table LABSTK. Then the BRSTACK table is searched and any references to that label are resolved by the following formula:

$$\text{ADDRESS} = \text{ADDRESS OF LABEL} - \text{ADDRESS OF REFERENCE}$$

It is clear that this formula generates an address relative to the point where the label is referenced. The advantage here is that the instructions are not restricted to any particular location in the interpreter. Not only are modifications made easier this way but larger programs can be run a segment at a time.

In addition to the tables mentioned, labels may be stored temporarily on a stack if an instruction is using more than one label at any particular time. This is strictly temporary to allow the reductions to keep track of the labels being used.

Finally, when a label is no longer needed it is removed from the LABSTK table and all remaining references in the BRSTACK table are also removed. Since all labels are generated and used within the system, transparent to the user, there is no difficulty in knowing when a label is no longer needed. Therefore the possibility of unresolved or unreferenced labels does not exist.

V. Intermediate Language and Interpreter

AIL

Because execution was delayed to allow for the looping and control structures, a way of saving the program instructions was required. The ASGOL Intermediate Language(AIL), see Appendix B, was designed for this purpose. These instructions are created by the parser (semantic routines) and are subsequently executed by the interpreter to generate DISSPLA calls. Although an intermediate language approach can result in decreased running speed of the program, it can also have advantages that offset any losses if it is designed correctly (Ref 11:10). In addition, in the ASGOL system, it allows for the breakup of the processing into two parts. AIL can be stored on a file by the parser and executed by the interpreter as often as desired.

An intermediate language(IL) usually consists of a few simple operations that are semantically equivalent to the original program (Ref 12:466). ILs are not designed to be used by people, but instead are entirely internal to the system and are usually unknown to the user (Ref 11:10). This is true with AIL as the user will probably never see these instructions. ILs can take several forms including:

1. Quadruples,
2. Triples,
3. Postfix, and
4. Infix (Ref 12:466).

A quadruple is a 4-tuple consisting of an operation code, two operands, and a result destination. For example, the expression $A = B * C$ would be represented by the sequence:

$*, B, C, A.$

A triple is essentially the same as a quadruple, except that the result destination is implied by the location of the instruction (Ref 12:467). For example, $A + B * C$ would be represented by

(1) $*, B, C$

(2) $+, A, (1).$

The (1) in expression (2) refers to the result generated by expression (1).

Both the infix and postfix notations, also known as Polish and Reverse Polish respectively after the Polish logician J. Lukasiewicz who first employed them (Ref 13:162), are continuous strings where each operation refers to either the two preceeding or two following operands. This type of representation is convenient for arithmetic expressions because parenthesis are not needed. However, it is very cumbersome when optimization and other functions are desired (Ref 12:470).

AIL does not strictly follow any of these notations. Instead it employs a variable format that ranges from just an operation code, CONT, to a 5-tuple, GRAPH,T,F,S,I. It is based on AOC (ALGOL Object Code) as presented by Barrett and Couch (Ref 12:376-463). AOC essentially forms the basis of AIL with instructions added to AOC to provide the graphic capabilities needed for ASGOL. AOC was chosen for this purpose because it can support essentially all of the features of ALGOL 60 (Ref 12:377) upon which ASGOL was based. In addition it provides the variable instruction format. By employing a stack as either an operand or a result destination the amount of information required in any one instruction was reduced. The main advantage to this approach is that less space is required for the instructions than if a straight 5-tuple was used. More processing time is required but the wasted space of the 5-tuple approach is avoided.

Interpreter

A compiler that generates target machine code directly, instead of an intermediate language, is usually more efficient and produces more efficient code (Ref 12:473). To take advantage of this situation, AIL was designed around a machine that was based on the AOC machine shown in Barrett

and Couch (Ref 12:377-463). This machine contains two stacks, a display for each stack, two stack pointers, a display pointer, an instruction area, and an instruction pointer as shown in Figure 11.

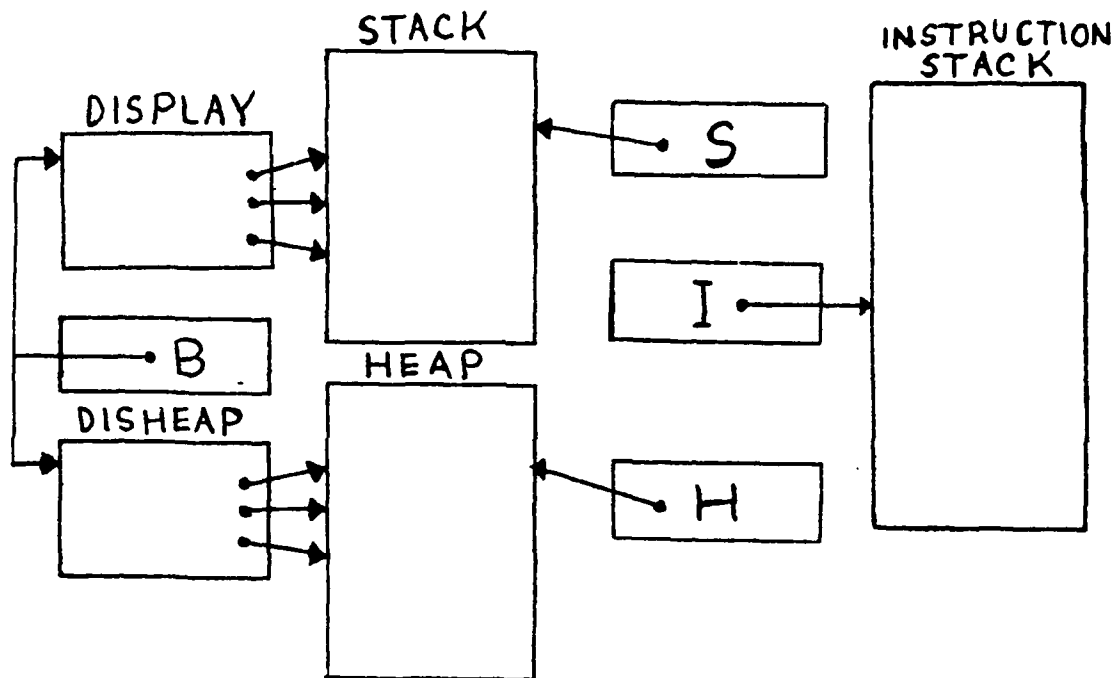


Figure 11. Interpreter Machine Structure

The area **STACK** is used to store data and intermediate results. Virtually all of the implied operands and result locations are in this area. The only exceptions to this

statement occur when character strings or array data are being used.

ASGOL allows dynamic dimensioning of arrays and strings. This means that the compiler does not know what the dimensions are going to be. Therefore at compile time it can only set up pointers and cannot reserve any space. The needed space is allocated at execution time and is always found in the stack HEAP. This is the sole function and purpose for this area.

To keep track of the block levels in each stack a display area is used for each. They are DISPLAY and DISHEAP and are used as shown in Figure 12.

Each area is an array of pointers to the beginning of each block. They serve two major functions. First they provide relatively quick access to the data defined at each level and second, they allow for quick, easy deallocation of blocks.

The other major area of the machine is the instruction stack. The AIL instructions are read into this area initially. The interpreter then starts a sequential execution of the instructions. As each is decoded the interpreter performs any manipulations required and generates DISSPLA calls when appropriate. This process continues until all of the instructions have been processed and the job is complete.

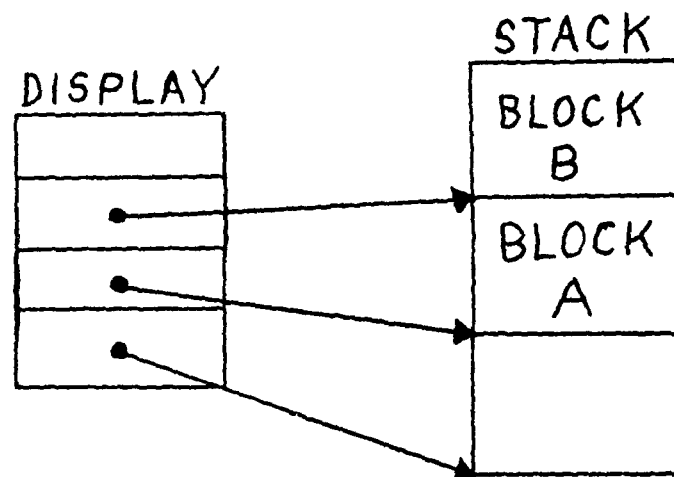


Figure 12. Use of Display Areas

VI. Applications

It is the purpose of this chapter to show examples of ASGOL code and the resulting plots for many of the applications of this system. They are taken from the examples presented by Lt. Hart (Ref 1). Only the major applications will be demonstrated, however, and no attempt is made to explain them here. For detailed explanations refer to the user's manual found in Appendix F.

```

.
.
.
PAGE example_4(HORIZONTAL,CENTER, . . .
MARGIN(1.25 INCHES,0 INCH)
DRAW LINE(0 INCH,3.25 INCH,3.25 INCH,0 INCH)
DRAW LINE(3.25 INCH,0 INCH,6.5 INCH,3.25 INCH)
DRAW ARROW .01(3.25 INCH,3.25 INCH,
               0 INCH,6.5 INCH)
DRAW ARROW 32(0 INCH, 6.5 INCH,
               6.5 INCH,6.5 INCH)
DRAW LINE(6.5 INCH,3.25 INCH,3.25 INCH,6.5 INCH)
DRAW ARROW 03(0 INCH,0 INCH,3.25 INCH,6.5 INCH)
DRAW LINE(3.25 INCH,6.5 INCH,0 INCH,3.25 INCH)
END PAGE example_4
.
.
.

```

Figure 13. Line and Arrow Instructions

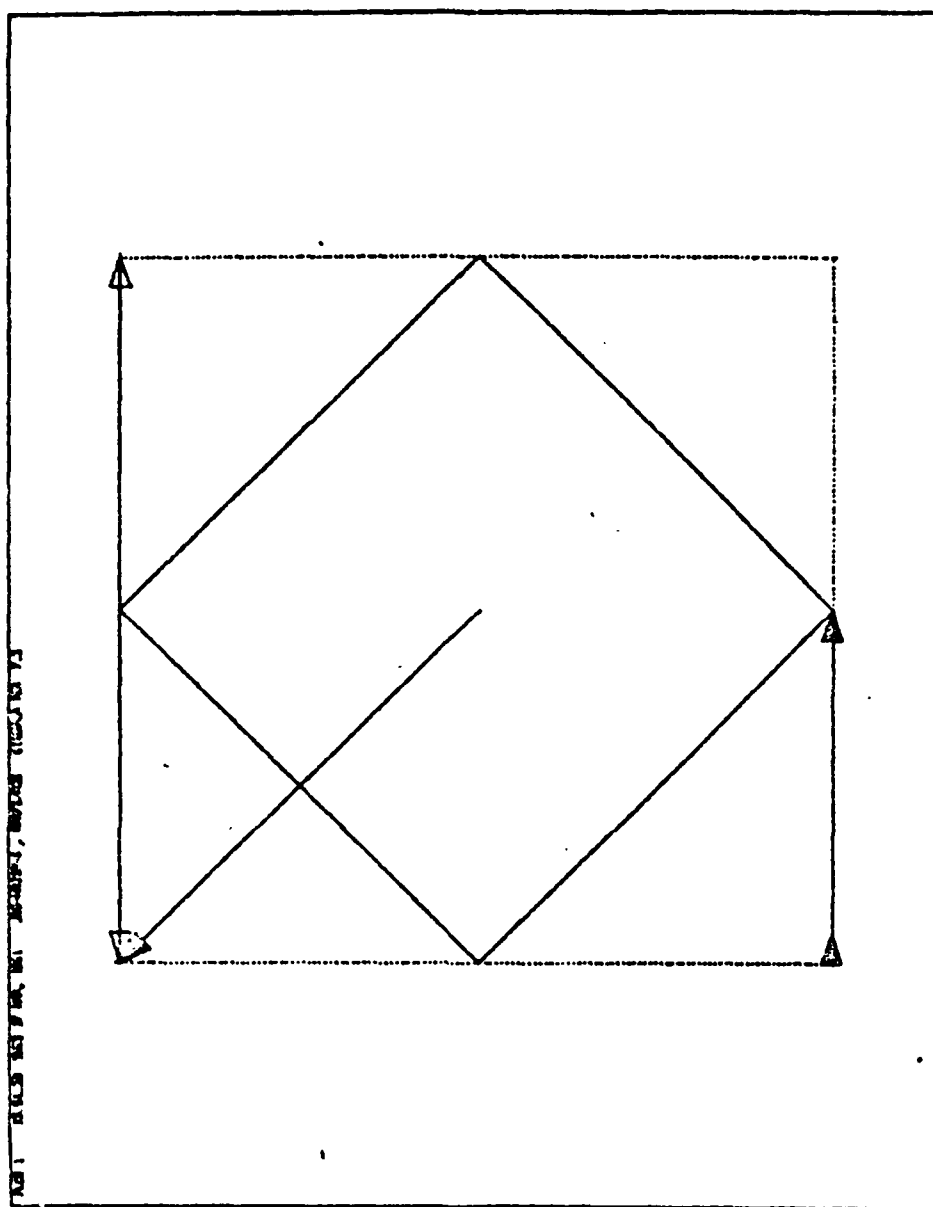


Figure 14. Line and Arrow Plot

```

PROGRAM exampleb_1
PAGE drawb_1(HORIZONTAL,LEFT RESET,0,TOP)
  SEGMENT draw_bar(0 INCH, 4.25 INCH, /* x dimen */
                  0 INCH, 6.5 INCH) /* y dimen */
  FRAME /* enclose plot */
  /* scaling of axes within the subplot area
     will occur automatically */
  X AXIS =("hours", /* hours of the day */
          0,24, /* min and max */
          5, /* # of plots */
          0) /* no ticks */
  Y AXIS =("decibels", /* noise level */
          0,120, /* 0-120 Db */
          7, /* every 20 Db */
          0) /* no ticks */
  LEGEND "freeway" /* title legend */
        (LEFT TOP, /* location */
         "weekdays",
         "weekends",
         "average") /* note order */
  GRAPH(BAR,3,
        DATA/10,70,56,87,10, /* weekdays */
          6,37,43,40,9, /* weekends */
          36.8:5/) /* average */
  END SEGMENT draw_bar
  SEGMENT draw_lin
    (4.25 INCH,8.5 INCH, /* x dimen */
     0 INCH,6.5 INCH) /* y dimen */
  X AXIS =("hours",0,24,5,5)
  Y AXIS =("decibels",0,120,7,1)
  GRAPH "noise level"(LINEAR,3,
                    DATA /10,70,56,87,10,
                      6,37,43,40,9,
                      36.8:5 /)

  END SEGMENT draw_lin
END PAGE drawb_1
END PROGRAM exampleb_1.

```

Figure 15. Bar and Linear Text

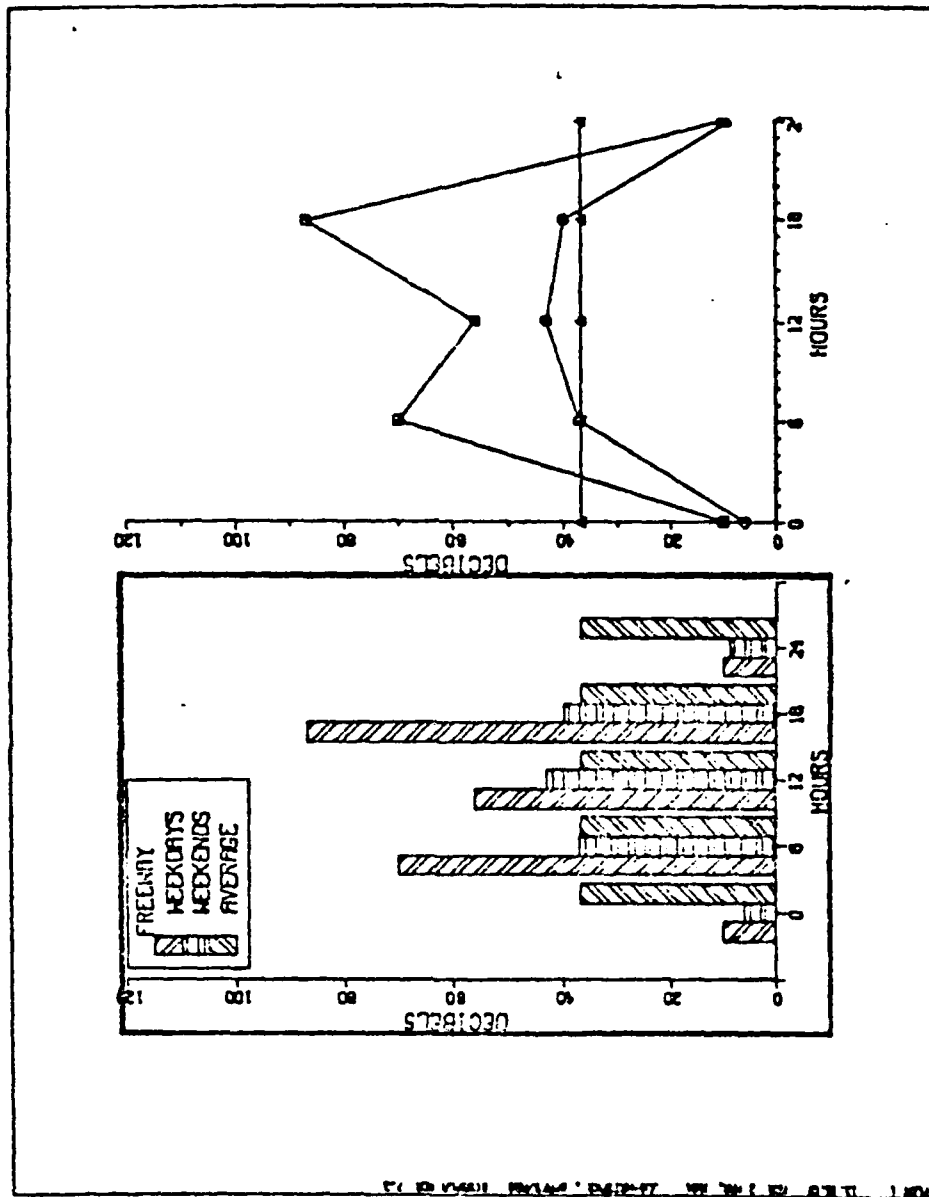


Figure 16. Bar and Linear Graph

```

PROGRAM examplepie
PAGE drawpie(VERTICAL,CENTER,5,RIGHT TOP)
MARGIN(1 INCH,1 INCH)
FRAME
LEGEND ("cars",      /* automobiles */
        "trucks",    /* 2 and 4 wheel */
        "vans")      /* customized */
GRAPH "automobile sales"
      (PIE,          /* define pie graph */
       3,            /* # of sections */
       DATA / 47.6, /* car sales */
              32.4,  /* truck sales */
              15/)   /* van sales */

END PAGE drawpie
END PROGRAM examplepie.

```

FIGURE 31 Pie Graph Example

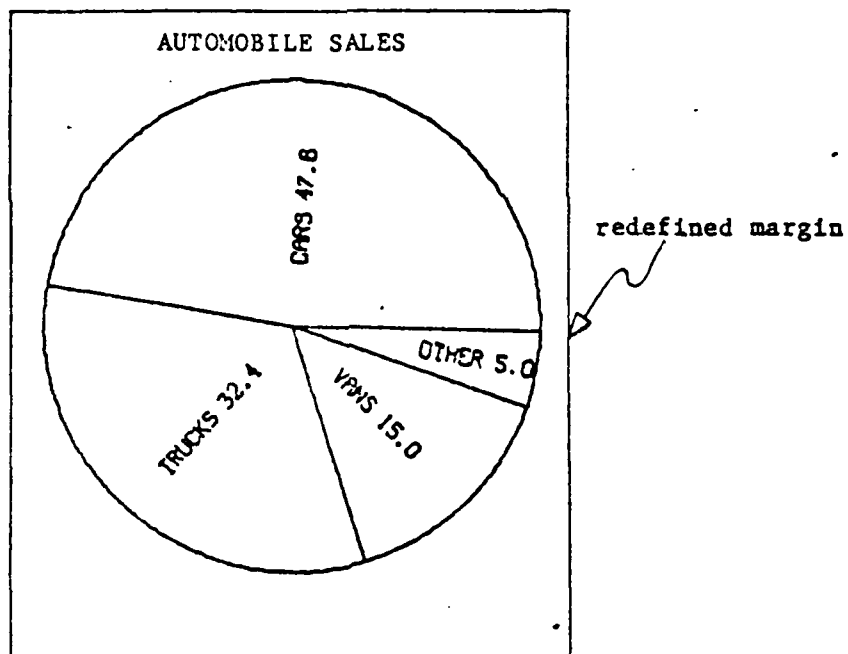


Figure 17. Pie Graph Example

```

.
.
.
PAGE text_examp(VERTICAL,center,0,top)
  SEGMENT l_just(0.00 INCH,4.50 INCH,
                3.25 INCH,9.00 INCH)
    FRAME
    TEXT(LEFT JUSTIFIED SIMPLE,
         "this is#nlan example#nl"&
         "of justified#nltext",
         1,CONTINUE,.50 INCHES)
  END SEGMENT l_just
  SEGMENT r_just(3.25 INCH,4.5 INCH,
                6.50 INCH,9.00 INCH)
    FRAME
    TEXT(RIGHT JUSTIFIED SIMPLE,
         "this is#nlan example#nl"&
         "of justified#nl text",
         1,CONTINUE,.50 INCHES)
  END SEGMENT r_just
  SEGMENT t_cent(0.00 INCH,0.00 INCH,
                3.25 INCH,4.5 INCH)
    FRAME
    TEXT(TOP CENTERED SIMPLE,
         "top#npcentered",
         1,100,.25 INCHES)
  END SEGMENT t_cent
  SEGMENT cent(3.25 INCH,0.00 INCH,
                6.50 INCH,4.50 INCH)
    FRAME
    TEXT(CENTERED SIMPLE,
         "this part not printed."&
         "centered#nltext",
         23,17,.25 INCHES)
  END SEGMENT cent
END PAGE text_examp
.
.
.

```

Figure 18. Text Instructions

THIS IS AN EXAMPLE OF JUSTIFIED TEXT	THIS IS AN EXAMPLE OF JUSTIFIED TEXT
TOP CENTERED.	CENTERED TEXT

Figure 19. Text Graph

VII. Recommendations

There are five major areas that are not part of the ASGOL system that could be useful. Therefore they deserve to be explored in more detail. However, this exploration was beyond the scope of this thesis and alone could provide sufficient material for a thesis. These areas are:

- (1). Segmentation,
- (2). Error Recovery,
- (3). Optimization of AIL,
- (4). Procedures and Functions, and
- (5). Utilization of DISSPLA.

It is important to note that these areas are not listed in order of importance.

Segmentation

Of the five areas this is probably the most critical at this point. One of the original requirements of the system was that it be designed to run under INTERCOM on the CYBER in a memory space of approximately 65K words(octal). Presently neither part of the system, the compiler or the interpreter with the DISSPLA routines, will run in less than 100K.

There are two approaches to the solution of this problem, overlays and segmentation. Segmentation is recommended because none of the segmentation directives are included in the text of the program. Rather, they are input to the LOADER. This provides a great deal more flexibility to the system. The problem of segmenting the DISSPLA package is currently being worked on so this should be available to anyone who would be expanding ASGOL. Although the compiler has not been segmented, a modular design, with as little interaction between the modules as possible, was used. This approach should allow segmentation with little difficulty.

Error Recovery

For a compiler to be useful it should not terminate after finding only one error. It would be most beneficial if it would find as many errors as possible during one run. This not only saves programmer time but it also saves computer time making debugging easier and quicker. This is generally a difficult problem. It is particularly difficult with table driver parsers like the one used in the ASGOL system. With the large amount of research that has been done in this area, it is easy to envision this problem solution as a follow-on thesis effort.

Optimization of AIL

When space is a problem, as in the ASGOL system, the time required to optimize the AIL code could be well spent. By removing redundant code, redundant labels, and code from within loops for example, not only space but execution time can be saved. Like error recovery there is much theoretical material available in this area.

Functions and Procedures

Functions and procedures are very useful structures in a programming language. They allow code to be executed a number of times, using different data each time, without the need for multiple copies of the instructions. Most of the features of the machine that are required to support these structures already exist in the ASGOL system. Therefore their inclusion in ASGOL is primarily a matter of determining the scope of the routines (a PAGE procedure could not be called from a SEGMENT level plot for example) and changing the BNF to include them. In addition this change would require the modification and addition of semantic routines to generate the appropriate AIL code.

Use of Additional DISSPLA Features

Many of the features of DISSPLA have been included in ASGOL, but by no means all. As use of ASGOL increases it is reasonable to expect that many of the other features will be desired. It is not difficult to imagine, then, that a follow-on to this thesis could invariably involve the addition of many of these DISSPLA features to the ASGOL system.

Bibliography

1. Hart, James D. ASGOL - An ALGOL-Structured Graphics Oriented Language. Ms Thesis. Wright-Patterson AFB, Ohio: Air Force Institute of Technology, March 1981.
2. ISSCO. DISSPLA User's Manual, Version 8.2. San Diego: Integrated Software Systems Corporation, 1978.
3. Preliminary Instructions. Use of the DISSPLA Plotting Library for the DCD6000 - CYBER74. Wright-Patterson AFB, January 1980.
4. Control Data Corporation. FORTTRAN Extended Version 4 Reference Manual. Sunnyvale, California, CDC Publications and Graphics Division, 1979.
5. Dahl, David A. MAPPER. A Graphics Oriented Language. Los Alamos, California: Los Alamos Scientific Laboratory, June 1972.
6. Pyster, Arthur B. Compiler Design and Construction. New York: Van Nostrand Reinhold Company, 1980.
7. Wetherell, Charles and Alfred Shannon. LR Automatic Parser Generator and LR(1) Parser. Livermore, California: Lawrence Livermore Laboratory, June 1979.
8. Backhouse, Roland C. Syntax of Programming Languages Theory and Practice. London: Prentice-Hall International, Inc., 1979.
9. Aho, Alfred V., et al. The Design and Analysis of Computer Algorithms (Third Printing). Reading, Massachusetts: Addison-Wesley Publishing Company, 1976.

10. Wegner, Peter. Programming Languages, Information Structures, and Machine Organization. New York and St. Louis: McGraw-Hill Book Company, 1968.
11. Randell, B. and L. J. Russell. ALGOL-60 Implementation. London and New York: Academic Press, 1964.
12. Barrett, William A. and John D. Couch. Compiler Construction: Theory and Practice. Science Research Associates, Inc., 1979.
13. Lee, John A. N. The Anatomy of a Compiler. New York, Amsterdam, and London: Reinhold Publishing Corporation, 1967.

Appendix A
BNF of ASGOL

This appendix is designed to serve two functions. First it is a formal definition of the ASGOL language. It also functions, however, as a cross reference between the ASGOL productions and the semantic routines. For each production two numbers are included. The first is the production number and the second is the address for the semantic routine in the semantic program.

1
<SYSTEM GOAL SYMBOL> ::= END <PROGRAM> END

2 100
<PROGRAM> ::= <PROGRAM HEAD> <PROGRAM BODY> <PROGRAM END> .

3 200
<PROGRAM HEAD> ::= PROGRAM <IDENTIFIER>

4 400
<PROGRAM BODY> ::= <DECLARATION LIST> <SECTION BLOCK LIST>

5 500
<PROGRAM END> ::= END PROGRAM <IDENTIFIER>

6 600
<DECLARATION LIST> ::= <NULL>

7 700 DECLARE <CONSTANT DECLARATION PART> <VARIABLE
DECLARATION PART> END DECLARE

8 800
<CONSTANT DECLARATION PART> ::= CONSTANT <CONSTANT
DECLARATION LIST>

9 900 <NULL>

10 1000
<CONSTANT DECLARATION LIST> ::= <CONSTANT DECLARATION>

11 1100 <CONSTANT DECLARATION LIST> <CONSTANT
DECLARATION>

12 1200
<CONSTANT DECLARATION> ::= <IDENTIFIER> = <CONSTANT
EXPRESSION>

13 33500
<CONSTANT EXPRESSION> ::= <INTEGER CONSTANT>

14 33600 <REAL CONSTANT>

15 33700 TRUE

16 33800 FALSE

17 33900 <CHARACTER STRING>

18 1300
<VARIABLE DECLARATION PART> ::= <NULL>

19	1400	VARIABLE <VARIABLE DECLARATION LIST>
20	1500	<VARIABLE DECLARATION LIST> ::= <VARIABLE DECLARATION>
21	1600	<VARIABLE DECLARATION LIST> <VARIABLE DECLARATION>
22	1700	<VARIABLE DECLARATION> ::= <IDENTIFIER LIST> : <TYPE>
23	1800	<IDENTIFIER LIST> ::= <IDENTIFIER>
24	1900	<IDENTIFIER LIST> , <IDENTIFIER>
25	2000	<TYPE> ::= <BASIC TYPE>
26	2100	<ARRAY TYPE>
27	2200	STRING (<EXPRESSION>)
28	2300	AXIS
29	2400	UNIT
30	2600	<ARRAY TYPE> ::= ARRAY <BOUNDS> OF <BASIC TYPE>

31 2700
 <BOUNDS> ::= (<ARRAY BOUNDS>)

32 2800
 <ARRAY BOUNDS> ::= <EXPRESSION>

33 2900 <ARRAY BOUNDS> , <EXPRESSION>

34 3000
 <BASIC TYPE> ::= INTEGER

35 3100 REAL

36 3200 BOOLEAN

37 3300 CHARACTER

38 3400
 <SECTION BLOCK LIST> ::= <SECTION LISTING>

39 3500 <PAGE LISTING>

40 3600
 <SECTION LISTING> ::= <SECTION>

41 3700 <SECTION LISTING> <SECTION>

42 3800
 <SECTION> ::= <SECTION HEAD> <SECTION BODY> <SECTION END>

43 3900
 <SECTION HEAD> ::= SECTION <IDENTIFIER>

44 4100
<SECTION BODY> ::= <DECLARATION LIST> <PAGE LISTING>

45 4200
<SECTION END> ::= END SECTION <IDENTIFIER>

46 4300
<PAGE LISTING> ::= <PAGE>

47 4400
 <PAGE LISTING> <PAGE>

48 4500
<PAGE> ::= <PAGE HEAD> <PAGE BODY> <PAGE END>

49 4600
<PAGE HEAD> ::= PAGE <IDENTIFIER> (<PAGE PARAMETER>)

50 4800
<PAGE BODY> ::= <DECLARATION LIST> <PAGE BLOCK LIST>

51 4900
<PAGE END> ::= END PAGE <IDENTIFIER>

52 5000
<PAGE PARAMETER> ::= <DIRECTION> , <MARGIN SET> <NUMBER> ,
<LOCATION>

53 5100
<LOCATION> ::= <LOCATION SIGNAL> TOP

54 5200
 <LOCATION SIGNAL> BOTTOM

55 5300
<LOCATION SIGNAL> ::= LEFT

56 5400 RIGHT

57 5500 INSIDE

58 5600 OUTSIDE

59 5700 <NULL>

60 5800
<NUMBER> ::= <EXPRESSION>

61 5900
<DIRECTION> ::= VERTICAL

62 6000 HORIZONTAL

63 6100
<MARGIN SET> ::= LEFT RESET ,

64 6200 RIGHT RESET ,

65 6300 CENTER ,

66 6400 <NULL>

67 6500
<PAGE BLOCK LIST> ::= <STRUCTURE COMMAND LIST>
<INSTRUCTIONS>

68	6600	<STRUCTURE COMMAND LIST> <SEGMENT LISTING>
69	6700	<STRUCTURE COMMAND LIST> ::= <NULL>
70	6800	<STRUCTURE COMMAND LIST> <STRUCTURE COMMAND>
71	6900	<STRUCTURE COMMAND> ::= <MARGIN INSTRUCTION>
72	7000	<FRAME INSTRUCTION>
73	7100	<ASSIGNMENT INSTRUCTION>
74	7200	<CHANGE INSTRUCTION>
75	7300	<IF INSTRUCTION>
76	7400	<WHILE INSTRUCTION>
77	7500	<FOR INSTRUCTION>
78	7600	<REPEAT INSTRUCTION>
79	7700	<CASE INSTRUCTION>
80	7800	<SEGMENT LISTING> ::= <SEGMENT>

81 7900 <SEGMENT LISTING> <SEGMENT>

82 8000
 <SEGMENT> ::= <SEGMENT HEAD> <SEGMENT BODY> <SEGMENT END>

83 8100
 <SEGMENT HEAD> ::= SEGMENT <IDENTIFIER> (<SEGMENT
 PARAMETER>)

84 8300
 <SEGMENT BODY> ::= <DECLARATION LIST> <SEGMENT BLOCK LIST>

85 8400
 <SEGMENT BLOCK LIST> ::= <STRUCTURE COMMAND LIST>
 <INSTRUCTIONS>

86 9600
 <SEGMENT END> ::= END SEGMENT <IDENTIFIER>

87 9700
 <SEGMENT PARAMETER> ::= ALL

88 99000 <UNIT VALUE> , <UNIT VALUE> , <UNIT VALUE> ,
 <UNIT VALUE>

89 10200
 <MARGIN INSTRUCTION> ::= MARGIN <MARGIN VALUE>

90 10300
 <MARGIN VALUE> ::= (<UNIT VALUE> , <UNIT VALUE>)

91 10400 <NULL>

92 10500
 <UNIT VALUE> ::= <EXPRESSION> <UNITS>

93 10600
<UNITS> ::= INCH

94 10700
 INCHES

95 10800
 <IDENTIFIER>

96 10900
<AXIS DEFINITION> ::= (<TITLE> , <TYPE AXIS> , <MIN> ,
<MAX> , <DELTA> , <TICKS>)

97 11000
<TITLE> ::= <CHARACTER STRING>

98 11100
 <IDENTIFIER>

99 11200
<TYPE AXIS> ::= LINEAR

100 11300
 LOG

101 11400
 LOGARITHMIC

102 11500
 MONTH

103 11600
<MIN> ::= <EXPRESSION>

104 11700
 <MONTH>

105 11800
<MAX> ::= <EXPRESSION>

106 11900
 <MONTH>

107 12000
<MONTH> ::= JAN

108 12100
 FEB

109 12200
 MAR

110 12300
 APR

111 12400
 MAY

112 12500
 JUN

113 12600
 JUL

114 12700
 AUG

115 12800
 SEP

116 12900
 OCT

117 13000
 NOV

118	13100	DEC
119	13200	<DELTA> ::= <EXPRESSION>
120	13300	<TICKS> ::= <EXPRESSION>
121	13400	<FRAME INSTRUCTION> ::= FRAME <FRAME THICKNESS>
122	13500	<FRAME THICKNESS> (<EXPRESSION>)
123	13600	<NULL>
124	13700	<ASSIGNMENT INSTRUCTION> ::= <SET INSTRUCTION>
125	13800	<INPUT INSTRUCTION>
126	13900	<OUTPUT INSTRUCTION>
127	14000	<SET INSTRUCTION> ::= SET <SET VARIABLE> = <EXPRESSION>
128	14100	SET STRING <IDENTIFIER> = <EXPRESSION>
129	14200	SET UNIT <IDENTIFIER> = <UNIT VALUE>
130	14300	SET AXIS <IDENTIFIER> <AXIS SPECIFICATION>

131 14400
<SET VARIABLE> ::= <VARIABLE>

132 14700
<AXIS SPECIFICATION> ::= = <AXIS DEFINITION>

133 14800
 . TITLE = <TITLE>

134 14900
 . TYPE = <TYPE AXIS>

135 15000
 . MIN = <MIN>

136 15100
 . MAX = <MAX>

137 15200
 . DELTA = <DELTA>

138 15300
 . TICKS = <TICKS>

139 15400
<IF INSTRUCTION> ::= <IF HEAD> <TRUE BRANCH> <FALSE BRANCH>
END IF

140 15800
<IF HEAD> ::= IF <EXPRESSION>

141 16200
<ELSE> ::= ELSE

142 15900
<TRUE BRANCH> ::= THEN <STRUCTURE COMMAND LIST>

143 16000
<FALSE BRANCH> ::= <ELSE> <STRUCTURE COMMAND LIST>

144 16000
 <NULL>

145 16300
<WHILE INSTRUCTION> ::= <WHILE HEAD> DO <STRUCTURE COMMAND
LIST> END WHILE

146 16500
<WHILE HEAD> ::= <WHILE> <EXPRESSION>

147 16400
<WHILE> ::= WHILE

148 16600
<FOR INSTRUCTION> ::= <FOR HEAD> DO <STRUCTURE COMMAND LIST>
END FOR

149 16700
<FOR START> ::= FOR <SET VARIABLE> = <EXPRESSION>

150 18000
<FOR HEAD> ::= <FOR START> <WAY> <BY CLAUSE>

151 16800
<WAY> ::= TO <EXPRESSION>

152 16900
 DOWN TO <EXPRESSION>

153 17000
<BY CLAUSE> ::= BY <EXPRESSION>

154 17100
 <NULL>

155 17200
 <REPEAT INSTRUCTION> ::= <REPEAT> <STRUCTURE COMMAND LIST>
 <REPEAT END>

156 17300
 <REPEAT> ::= REPEAT

157 17400
 <REPEAT END> ::= UNTIL <EXPRESSION> END REPEAT

158 17500
 <CASE INSTRUCTION> ::= <CASE HEAD> <CASE SEQUENCE> END CASE

159 17700
 <CASE HEAD> ::= CASE <VARIABLE> OF

160 17800
 <CASE SEQUENCE> ::= <CASE LIST> : <STRUCTURE COMMAND LIST>

161 17900
 <CASE SEQUENCE> <CASE LIST> : <STRUCTURE
 COMMAND LIST>

162 18200
 <CASE LIST> ::= <INTEGER CONSTANT LIST>

163 18300
 <CHARACTER STRING LIST>

164 18400
 OTHERS

165 18500
 <CHARACTER STRING LIST> ::= <CHARACTER STRING>

166 18600
 <CHARACTER STRING LIST> , <CHARACTER STRING>

167 18700
 <VARIABLE> ::= <IDENTIFIER> <ARRAY SPECIFICATIONS>

168 18800
 <ARRAY SPECIFICATIONS> ::= <BOUNDS>

169 18900
 <NULL>

170 19000
 <INPUT INSTRUCTION> ::= INPUT <IDENTIFIER> : <SOURCE>

171 19100
 <SOURCE> ::= TERMINAL

172 19200
 TAPE <INTEGER CONSTANT>

173 19300
 <DIRECT INPUT>

174 19400
 <DIRECT INPUT> ::= DATA / <CONSTANT SET> /

175 19500
 <CONSTANT SET> ::= <INTEGER CONSTANT SET>

176 19600
 <REAL CONSTANT SET>

177 19700
 <BOOLEAN CONSTANT SET>

178 19800
 <CHARACTER STRING>

179 19900
 <INTEGER CONSTANT SET> ::= <INTEGER CONSTANT> <LIST NUMBER>

180 20000 <INTEGER CONSTANT SET> , <INTEGER CONSTANT>
 <LIST NUMBER>

181 20100
 <REAL CONSTANT SET> ::= <REAL CONSTANT> <LIST NUMBER>

182 20200 <REAL CONSTANT SET> , <REAL CONSTANT> <LIST
 NUMBER>

183 20300
 <BOOLEAN CONSTANT SET> ::= <BOOLEAN VALUE> <LIST NUMBER>

184 20400 <BOOLEAN CONSTANT SET> , <BOOLEAN VALUE>
 <LIST NUMBER>

185 20500
 <LIST NUMBER> ::= : <INTEGER CONSTANT>

186 20600 <NULL>

187 20700
 <OUTPUT INSTRUCTION> ::= OUTPUT <EXPRESSION> : <PORT>

188 20800
 <PORT> ::= TERMINAL

189 20900 TAPE <INTEGER CONSTANT>

190 21000 OUTPUT

191 21100
 <CHANGE INSTRUCTION> ::= CHANGE <FROM SET> TO <TO SET>

192 21200
<FROM SET> ::= <CASE SET> ROMAN

193 21300
 <CASE SET> ITALIC

194 21400
 <CASE SET> SCRIPT

195 34000
<TO SET> ::= SPECIAL

196 21500
 MATH

197 21600
 <CASE SET> GREEK

198 21700
 <CASE SET> RUSSIAN

199 21800
 HEBREW

200 21900
<CASE SET> ::= UPPER

201 22000
 LOWER

202 22100
<EXPRESSION> ::= <CONDITION EXPRESSION>

203 22200
 <CONDITION EXPPESSION> <LOGIC OP> <CONDITION
EXPRESSION>

204	22300	$\langle \text{CONDITION EXPRESSION} \rangle ::= \langle \text{SIMPLE EXPRESSION} \rangle$
205	22400	$\langle \text{CONDITION EXPRESSION} \rangle \langle \text{CONDITION OP} \rangle \langle \text{SIMPLE EXPRESSION} \rangle$
206	22500	$\langle \text{SIMPLE EXPRESSION} \rangle ::= \langle \text{TERM} \rangle$
207	22600	$+ \langle \text{TERM} \rangle$
208	22700	$- \langle \text{TERM} \rangle$
209	22800	$\langle \text{SIMPLE EXPRESSION} \rangle + \langle \text{TERM} \rangle$
210	22900	$\langle \text{SIMPLE EXPRESSION} \rangle - \langle \text{TERM} \rangle$
211	23000	$\langle \text{TERM} \rangle ::= \langle \text{FACTOR} \rangle$
212	23100	$\langle \text{TERM} \rangle * \langle \text{FACTOR} \rangle$
213	23200	$\langle \text{TERM} \rangle / \langle \text{FACTOR} \rangle$
214	23300	$\langle \text{TERM} \rangle \text{ MOD } \langle \text{FACTOR} \rangle$
215	23400	$\langle \text{TERM} \rangle \text{ REM } \langle \text{FACTOR} \rangle$

216	23500	<FACTOR> ::= <PRIMARY>
217	23600	<FACTOR> ** <PRIMARY>
218	23700	<PRIMARY> ::= NOT <PRIMARY>
219	23800	(<EXPRESSION>)
220	23900	<VARIABLE>
221	24000	<CONSTANT>
222	24100	<CHARACTER STRING>
223	24200	FLOAT (<EXPRESSION>)
224	24300	INTEGER (<EXPRESSION>)
225	24400	FRACTION (<EXPRESSION>)
226	24500	SIN (<EXPRESSION>)
227	24600	COS (<EXPRESSION>)
228	24700	TAN (<EXPRESSION>)

229	24800	INV SIN (<EXPRESSION>)
230	24900	INV COS (<EXPRESSION>)
231	25000	INV TAN (<EXPRESSION>)
232	25100	ABSOLUTE (<EXPRESSION>)
233	25200	STRING POINT (<IDENTIFIER>)
234	25300	LENGTH (<IDENTIFIER>)
235	25400	<LOGIC OP> ::= <
236	25500	<=
237	25600	=
238	25700	/=
239	25800	>=
240	25900	>
241	26000	<CONDITION OP> ::= AND

242	26100	OR
243	26200	XOR
244	26300	<CONSTANT> ::= <INTEGER CONSTANT>
245	26400	<REAL CONSTANT>
246	26500	<BOOLEAN VALUE>
247	26600	<INTEGER CONSTANT LIST> ::= <INTEGER CONSTANT>
248	26700	<INTEGER CONSTANT LIST> , <INTEGER CONSTANT>
249	26800	<BOOLEAN VALUE> ::= TRUE
250	26900	FALSE
251	27000	<INSTRUCTIONS> ::= <DRAW INSTRUCTION LIST>
252	27100	<GRAPH INSTRUCTION>
253	27200	<TEXT INSTRUCTION>
254	27300	<NULL>

255 27400
 <DRAW INSTRUCTION LIST> ::= <DRAW INSTRUCTION>

256 27500
 <DRAW INSTRUCTION LIST> <DRAW INSTRUCTION>

257 27600
 <DRAW INSTRUCTION> ::= DRAW ARROW <ARROW STYLE> (<UNIT
 VALUE> , <UNIT VALUE> , <UNIT VALUE> , <UNIT VALUE>)

258 27700
 DRAW LINE (<UNIT VALUE> , <UNIT VALUE> ,
 <UNIT VALUE> , <UNIT VALUE>)

259 27800
 <ARROW STYLE> ::= <INTEGER CONSTANT>

260 27900
 <IDENTIFIER>

261 28000
 <GRAPH INSTRUCTION> ::= <GRAPH PREPARATION LIST> GRAPH
 <TITLE OPTION> (<STACK FORMATTING> <FRAME OPTION>
 <INTERPOLATION TYPE> , <NUMBER PLOTS> , <XARRAY> <YARRAY>)

262 28100
 <GRAPH PREPARATION LIST> ::= <NULL>

263 28200
 <GRAPH PREPARATION LIST> <GRAPH PREPARATION
 INSTRUCTIONS>

264 28300
 <GRAPH PREPARATION INSTRUCTION> ::= <GRID INSTRUCTION>

265 28400
 <LEGEND INSTRUCTION>

```
266      28500      <AXIS INSTRUCTION>
```

```

267      28600
<GRID INSTRUCTION> ::= GRID ( <EXPRESSION> , <EXPRESSION> )

```

```

268      28700
<LEGEND INSTRUCTION> ::= LEGEND <TITLE OPTION> ( <LOCATION>
, <TITLE LIST> )

```

```
269      28800
<TITLE LIST> ::= <TITLE>
```

```
270      28900      <TITLE LIST> , <TITLE>
```

```

271      29000
<AXIS INSTRUCTION> ::= <AXIS> = <AXIS DEFINITION>

```

```
272      29100      <AXIS> = <IDENTIFIER>
```

```
273      29200
<AXIS> ::= XAXIS
```

274 29300
YAXIS

```
275      29400
<TITLE OPTION> ::= <TITLE>
```

276	29500	<NULL>
-----	-------	--------

```
277      29600
<STACK FORMATTING> ::= STACK OF <INTEGER CONSTANT> ,
```

278 29700
 <NULL>

279 29800
<FRAME OPTION> ::= FRAMED ,

280 29900
 <NULL>

281 30000
<INTERPOLATION TYPE> ::= LINEAR

282 30100
 STEP

283 30200
 BAR

284 30300
 STACKED BAR

285 30400
 SPLINE

286 30500
 SMOOTH

287 30600
 PIE

288 30700
<NUMBER PLOTS> ::= <EXPRESSION>

289 30800
<TEXT INSTRUCTION> ::= TEXT (<JUSTIFICATION> , <TEXT STYLE>
 , <STRING NAME> , <START> , <LENGTH> , <ANGLE> <SIZE>)

290 30900
 <TEXT STYLE> ::= SIMPLE

291 31000
 CARTOG

292 31100
 COMPLX

293 31200
 DUPLX

294 31300
 GOTHIC

295 31400
 SCMLPX

296 31500
 SIMPLX

297 31600
 TRIPLX

298 31700
 <STRING NAME> ::= <IDENTIFIER>

299 31800
 <CHARACTER STRING>

300 31900
 <START> ::= <EXPRESSION>

301 32000
 NEXT

302 32100
 <LENGTH> ::= <EXPRESSION>

303 32200
 CONTINUE

304 32300
<SIZE> ::= , <UNIT VALUE>

305 32400
 <NULL>

306 32500
<ANGLE> ::= <EXPRESSION>

307 32700
<JUSTIFICATION> ::= <HORIZONTAL FORMAT> JUSTIFIED

308 32800
 <VERTICAL FORMAT> CENTERED

309 32900
<HORIZONTAL FORMAT> ::= LEFT

310 33000
 RIGHT

311 33100
 L-R

312 33200
<VERTICAL FORMAT> ::= TOP

313 33300
 BOTTOM

314 33400
 <NULL>

315 8500
<XARRAY> ::= <IDENTIFIER>

316 8600 <DIRECT INPUT>

317 8700
<YARRAY> ::= , <IDENTIFIER>

318 8800 , <DIRECT INPUT>

319 8900 <null>

Appendix B AIL Instructions

This appendix is intended to serve as a guide to the AIL instructions. Also included are statements denoting the effect of these instructions on the stack. The following notation is used:

S - STACK Pointer,
B - DISPLAY Pointer,
D - DISPLAY Stack,
I - Instruction Stack Pointer, and
C(S) - Contents of STACK at Location S.

LC C	Load Constant $S = S + 1$ $C(S) = C$
LA B, J	Load Address $S = S + 1$ $C(S) = D(B) + J$
LV B, J	Load Value $S = S + 1$ IF B=0 THEN $C(S) = C(S+J)$ ELSE $C(S) = C(D(B)+J)$
STD B, J, S	Store Direct $C(D(B)+J) = C(S)$ $S = S - 1$
ST	Store $C(C(S-1)) = C(S)$ $S = S - 2$
CONT	Load Contents $C(S) = C(C(S))$

INCS	Increment Stack Pointer $C(S+C(S)) = S$ $S = S + C(S)$
INCS N	Increment Stack Pointer $S = S + N$
DECS	Decrement Stack $S = S - C(S) - 1$
DECS N	Decrement Stack $S = S - N$
NEG	Negate $C(S) = -C(S)$
EQ C	Equal IF $C(S-1) = C(S)$ THEN $C(S-1) = 1.0$ ELSE $C(S-1) = 0.0$ $S = S - 1$
LS	Less Than IF $C(S-1) < C(S)$ THEN $C(S-1) = 1.0$ ELSE $C(S-1) = 0.0$ $S = S - 1$
GT	Greater Than IF $C(S-1) > C(S)$ THEN $C(S-1) = 1.0$ ELSE $C(S-1) = 0.0$ $S = S - 1$
NOT	 IF $C(S) = 1.0$ THEN $C(S) = 0.0$ ELSE $C(S) = 1.0$

AND

IF $C(S-1) = 1.0$ AND $C(S) = 1.0$
THEN $C(S-1) = 1.0$
ELSE $C(S-1) = 0.0$
 $S = S - 1$

OR

IF $C(S-1) = 1.0$ OR $C(S) = 1.0$
THEN $C(S-1) = 1.0$
ELSE $C(S-1) = 0.0$
 $S = S - 1$

JP L

Unconditional Jump
 $I = L$

JIF L

Jump If False
IF $C(S) = 0.0$ THEN $I = L$
 $S = S - 1$

BE N

Block Entry
 $C(S+1) = D(B)$
 $B = B + 1$
 $D(B) = S + 2$
 $S = S + N + 1$

EB

Exit Block
 $S = D(B) - 2$
 $B = B - 1$

XOR

Exclusive OR
IF $C(S-1) \neq C(S)$
THEN $C(S-1) = 1.0$
ELSE $C(S-1) = 0.0$
 $S = S - 1$

HALT

Stop the Machine

ADD

Addition
 $C(S-1) = C(S-1) + C(S)$
 $S = S - 1$

SUB	Subtraction $C(S-1) = C(S-1) - C(S)$ $S = S - 1$
MULT	Multiplication $C(S-1) = C(S-1) * C(S)$ $S = S - 1$
DIV	Real Division $C(S-1) = C(S-1) / C(S)$ $S = S - 1$
IDIV	Integer Division $C(S-1) = \text{INT}(C(S-1)) / \text{INT}(C(S))$ $S = S - 1$
MOD	$C(S-1) = \text{MOD}(C(S-1), C(S))$ $S = S - 1$
EXP	Exponentiation $C(S-1) = C(S-1) ** C(S)$ $S = S - 1$
INTEGER	Convert to Integer $C(S) = \text{INT}(C(S))$
FRACTION	Save Fraction $C(S) = C(S) - \text{INT}(C(S))$
SINE	$C(S) = \text{SIN}(C(S))$
COSINE	$C(S) = \text{COS}(C(S))$
TAN	Tangent $C(S) = \text{TAN}(C(S))$

INVSIN	Inverse Sine C(S) = INVSIN(C(S))
INVCOS	Inverse Cosine C(S) = INVCOS(C(S))
INVTAN	Inverse Tangent C(S) = INVTAN(C(S))
ABS	Absolute Value IF C(S) < 0 THEN C(S) = -C(S)
PAGE D,M,N,L	Page Set Up
MAS B,J,N	Make Array Space
AVA B,J,N	Array Variable Address
START	Start Plot Routines
LENGTH B,J	Return Length of String C(S) = LENGTH(C(S))
FRAME T	Draw Frame
CHANGE F,T	Change Character Set
ARROW T	Draw Arrow S = S - 5
LINE T	Draw Line S = S - 4
GRID	Draw Grid S = S - 2
LEGEND T,L,N	Define Legend

GRAPH T,F,S,I

Plot Graph

TEXT J,S,L

Plot Text

MSS

Make String Space

ENDPLT

End Plotting

READ T

Read from Tape

PRINT T

Write to Tape

MSL

Make String Literal

AD-A115 846

AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL--ETC F/6 9/2
AN INTERMEDIATE LANGUAGE AND INTERPRETER FOR THE ASSOL GRAPHICS--ETC(U)

DEC 81 K P ALBERT

UNCLASSIFIED

AFIT/GCS/HA/81D-1

NL

202

010000



END

DATE

FILED

7-82

DTIC

Appendix C
Conditional and Looping Productions

WHILE

<WHILE INSTRUCTION> <WHILE HEAD> DO <STRUCTURE COMMAND LIST>
END WHILE

- (1). Generate JP to begin label.
- (2). Write end label.

<WHILE> ::= WHILE

- (1). Generate end label.
- (2). Generate begin label.
- (3). Write begin label.

<WHILE HEAD> ::= <WHILE> <EXPRESSION>

- (1). Generate JIF to end label.

REPEAT

<REPEAT INSTRUCTION> ::= <REPEAT> <STRUCTURE COMMAND LIST>
<REPEAT END>

No action is taken for this production.

<REPEAT> ::= REPEAT

- (1). Generate begin label.
- (2). Write begin label.

<REPEAT END> ::= UNTIL <EXPRESSION> END REPEAT

- (1). Generate JIF to begin label.

IF-THEN-ELSE

<IF INSTRUCTION> ::= <IF HEAD> <TRUE BRANCH> <FALSE BRANCH>
END IF

No action is taken for this production.

<IF HEAD> ::= IF <EXPRESSION>

- (1). Generate end label.
- (2). Generate else label.
- (3). Generate JIF to else label.

<TRUE BRANCH> ::= THEN <STRUCTURE COMMAND LIST>

No action is taken for this production.

<ELSE> ::= ELSE

- (1). Generate JP to end label.
- (2). Write else label.

<FALSE BRANCH> ::= <ELSE> <STRUCTURE COMMAND LIST>

(1). Write end label.

<FALSE BRANCH> ::= <NULL>

(1). Write else label.

FOR

<FOR INSTRUCTION> ::= <FOR HEAD> DO <STRUCTURE COMMAND LIST>

END FOR

(1). Generate JP to start label.

(2). Write end label.

<FOR START> ::= FOR <SET VARIABLE> = <EXPRESSION>

(1). Store expression result in variable.

(2). Generate start label.

(3). Generate end label.

<WAY> ::= TO <EXPRESSION>

No action taken for this production.

<WAY> ::= DOWN TO <EXPRESSION>

No action taken for this production.

<BY CLAUSE> ::= BY <EXPRESSION>

No action is taken for this production.

<BY CLAUSE> ::= <NULL>

No action is taken for this production.

<FOR HEAD> ::= <FOR START> <WAY> <BY CLAUSE>

- (1). Load variable.
- (2). Load TO expression.
- (3). Test condition.
- (4). Generate JIF to end label.
- (5). Generate JP to start of instructions label
- (6). Write start label.
- (7). Load variable.
- (8). Load BY expression.
- (9). Add or subtract as required.
- (10). Save variable.
- (11). Load variable.
- (12). Load TO expression.
- (13). Test condition.
- (14). Generate JIF to end label.
- (15). Write start of instructions label.

Appendix D
Compiler Routines

SUBROUTINE ADDJUMP(LOC,ADDR)

This routine updates jump instructions.

SUBROUTINE CLRERR

This procedure is called to initialize the error common block.

SUBROUTINE CLRINS

This routine clears the instruction stack and resets the instruction stack pointer.

SUBROUTINE CLRLABL

This procedure clears all of the stacks used in label processing.

SUBROUTINE CLRLEX

This routine initializes the lexical variables in the common block LEXCOM.

SUBROUTINE CLRSMAN

This routine initializes all of the semantic variables in the common block SMANCOM.

SUBROUTINE CLRSTK(NUMBER)

When called by INITIAL(number = 0) this routine clears all the state stacks. When called by DOTRAN only the entry specified by number is cleared.

SUBROUTINE CLRVART

This routine is called to initialize the symbol table and its associated tables.

LOGICAL FUNCTION DIGIT(IBYTE)

This routine is very CDC dependant. It determines if IBYTE is a digit in display code.

SUBROUTINE DORED(IPROD)

This routine performs the reduction specified by IPROD.

SUBROUTINE DOTRAN(ISTA)

The transition to state ISTA is performed by this routine.

SUBROUTINE ENTRVAR(SYM1,ENTRY)

This routine is called to enter an identifier into the symbol table. The routine assumes that LOOKUP was called previously to make sure the identifier does not already exist. At the conclusion of this routine the parameter ENTRY points to the identifier's position.

SUBROUTINE ERRDIAG

This routine prints the errors that have been detected by the parser.

SUBROUTINE ERROR(INT)

This routine is called whenever an error is detected. The error number and line number are saved in the error common block.

SUBROUTINE EXITBLK(LEXICAL)

This routine is called when a block terminates. It is responsible for clearing the variables no longer needed from the symbol table.

SUBROUTINE GETLAB(LABEL)

This routine returns the next free label number.

SUBROUTINE GETLIN

This is the input routine for the parser. It gets a line from the unit inunit and puts it into the area LINEBUF. If there was an error on the previous line, line pointers are first put out to the program file.

FUNCTION IFINDR(ISTATE,ITOKEN)

If a reduction is possible when in state ISTATE looking ahead at symbol ITOKEN, the production number is returned.

FUNCTION IFINDT(IS,IT)

This function determines if a read transition can be performed when in state IS looking at symbol IT.

FUNCTION IFNDTK(IENTRY,LENGTH)

This routine searches the vocabulary for the entry IENTRY and returns its index. If it is not found a 0 is returned.

SUBROUTINE INITIAL

This routine handles all of the initialization for the system.

SUBROUTINE INITPAR

This routine sets up the system parameters.

SUBROUTINE INITTAB

This routine initializes the tables used by the parser.

SUBROUTINE INITTOK

This routine is called to initialize the basic terminals in the system.

FUNCTION IORD(ICHAR)

This function returns the integer value of the character passed to it. Note: this function is CDC dependant.

SUBROUTINE JUMPTO(LABEL)

This routine creates an entry in BRSTACK for a jump instruction.

LOGICAL FUNCTION LETTER(IBYTE)

This routine determines if IBYTE is a letter id display code. Note: This routine is CDC dependant.

SUBROUTINE LOOKUP(SYM1,ENTRY)

This routine receives a name in the parameter SYM1 and searches the symbol table for the "first" occurrence of that name. If a match is found the pointer into the symbol table for that name is returned in the parameter ENTRY; otherwise ENTRY = 0.

SUBROUTINE NEWVAR(PTR)

This routine is called to locate the first free entry in the symbol table.

SUBROUTINE NEXTINS(ARG1,ARG2,ARG3,ARG4)

This routine puts AIL instructions into the instruction stack.

FUNCTION NUMBER(IVAL)

This function returns the binary value of a number in character format. Note: this function is CDC dependant.

SUBROUTINE PARSER

This is the controlling routine for the parser and thus for the entire compiler.

SUBROUTINE POP(LABEL)

This routine removes a label number from the label save stack.

SUBROUTINE PUSH(LABEL)

This routine puts a label number onto the top of the label stack.

SUBROUTINE RESOLVE(LABEL)

This routine removes label references from the BRSTACK array.

SUBROUTINE SCANNER(TOK)

This routine gets the next lexical item from the input stream.

SUBROUTINE TABDUMP

This routine is a diagnostic tool and is used to print the symbol table upon abnormal termination of the parser.

SUBROUTINE WRITLAB(LABEL)

This routine enters a label reference into the LABLSTK array. It then searches BRSTACK and resolves any references to that label.

Appendix E

Interpreter Routines

Since most of the routines in the interpreter execute AIL instructions, the following listing will only list the AIL instruction after the subroutine name for those routines.

SUBROUTINE ABSOL
ABS

SUBROUTINE ADDIT
ADD

SUBROUTINE ANDIT
AND

SUBROUTINE ARROW
ARROW T

SUBROUTINE AVA
AVA B,J,N

SUBROUTINE BE
BE N

SUBROUTINE CONT
CONT

SUBROUTINE INTCOS
COSINE

SUBROUTINE DECS
DECS

SUBROUTINE DECSN
DECS N

SUBROUTINE DIV
DIV

SUBROUTINE EB
EB

SUBROUTINE EQ
EQ C

SUBROUTINE EXP
EXP

SUBROUTINE FRACTN
FRACTION

SUBROUTINE FRAME
FRAME T

SUBROUTINE GETNEXT(NEXT)
This subroutine gets the next instruction from the
instruction stack.

SUBROUTINE GT
GT

SUBROUTINE HALT
HALT

SUBROUTINE IDIV
IDIV

SUBROUTINE INCS
INCS

SUBROUTINE INCSN
INCS N

SUBROUTINE INTEG
INTEGER

SUBROUTINE INTGRID
GRID

SUBROUTINE INVCOS
INVCOS

SUBROUTINE INVSIN
INVSIN

SUBROUTINE INVTAN
INVTAN

SUBROUTINE JIF(NEXT)
JIF L

SUBROUTINE JUMP(NEXT)
JP L

SUBROUTINE LA
LA B,J

SUBROUTINE LC
LC C

SUBROUTINE INTLEG
LEGEND T,L,N,G

SUBROUTINE LINE
LINE T

SUBROUTINE LENGTH
LENGTH

SUBROUTINE LS
LS

SUBROUTINE LV
LV B, J

SUBROUTINE MAS
MAS B, J, N

SUBROUTINE MULT
MULT

SUBROUTINE NEG
NEG

SUBROUTINE BOOLNOT
NOT

SUBROUTINE BOOLOR
OR

SUBROUTINE INTPAGE(PAGENUM)
PAGE D, M, N, L

SUBROUTINE PAGEPOS(XPOS, YPOS, LOCAT, ALENG)
This subroutine calculates the page number position.

SUBROUTINE INTSINE
SINE

SUBROUTINE ST
ST

SUBROUTINE STD
STD B, J

SUBROUTINE SUB
SUB

SUBROUTINE TAN
TAN

SUBROUTINE XOR
XOR

Appendix E

User's Guide

This appendix is intended to serve solely as a users guide and not as a tutorial on the use of ASGOL. It is hoped that by studying the examples of Chapter 6 and by referencing the appropriate instructions in this appendix an understanding of the use of ASGOL can be obtained. To facilitate the use of this guide the instructions have been divided into six sections as follows:

- (1). Block structure instructions,
- (2). Constant and variable declarations,
- (3). Operating instructions,
- (4). Graphing instructions,
- (5). Graph set-up instructions, and
- (6). Arithmetic operations.

Before discussing specific instructions it is necessary to define some common symbols that are used often.

<ID> - This symbol represents a sequence of letters and digits with the following restrictions:

- (1). The first character must be a letter (A-Z),
 - (2). Only letters and digits (0-9) are allowed,
 - (3). The length must be less than or equal to 10,
- and
- (4). It cannot be a reserved word.

<EXP> - This symbol represents an expression and can have either a real, integer, character, or boolean value.

<UNIT> - This symbol can represent one of three forms; <EXP> INCH, <EXP> INCHES, or <EXP> <ID>.

<VARIABLE> - This symbol represents an identifier and can be an array specification.

<INTEGER> - This represents an integer constant: 1, 2, 5, etc.

Block Structure Instructions

```
PROGRAM <ID>  
END PROGRAM <ID> .
```

These two instructions are required for all programs. The first must be at the beginning and the second must be at the end. The <ID> refers to the name of the program and it cannot be used anywhere else within the program.

Note: The <ID> in both instructions must be the same.

SECTION <ID>
END SECTION <ID>

Within a program any number of sections may be defined. Each must have a unique name and each must begin with the first instruction and end with the second. Because of the block structure of the language all data defined in one section will not be known by any other section. This allows constants and variables with only limited usefulness to be defined for only a limited time.

Note: The <ID> in both instructions must be the same.

PAGE <ID> (<DIRECTION> , <MARGIN> <NUMBER> ,
<LOCATION>)
END PAGE <ID>

Within a section if any are defined or alone otherwise any number of pages may be defined. Every page must have a unique name and must begin with the first instruction. This instruction has four parameters that are defined as follows:

<DIRECTION> - Determines the physical size of the page.

VERTICAL - 8 1/2 X 11 INCHES

HORIZONTAL - 11 X 8 1/2 INCHES

<MARGIN> - This parameter is optional and is used to determine where on the page the grace (binding) margin is to be located. If used this parameter must be followed by a comma.

LEFT RESET - 1" margin on upper, lower, and right sides; 1 1/2" on left side.

RIGHT RESET - 1" margin on upper, lower, and left sides; 1 1/2" on right side.

CENTER - 1" margin on all sides.

<NUMBER> - This parameter can be either an integer value, a real value, or a character string. It is the number that is printed on the page.

Note: (1). If an integer value 0 is used no number will be printed.

(2). If a character string is used a maximum of 20 characters is allowed. If more than 20 are used only the first 20 will be printed.

<LOCATION> - This parameter determines where on a page the page number will be printed.

LEFT TOP - The number will begin directly above the left margin.

RIGHT TOP - The number will end directly above the right margin.

INSIDE TOP - The number will either begin or end directly above the binding margin depending on the side that the margin is on.

OUTSIDE TOP - The number will either begin or end directly above the margin opposite the binding margin.

TOP - The number will be centered at the top of the page.

LEFT BOTTOM - The number will begin directly below the left margin.

RIGHT BOTTOM - The number will end directly below the right margin.

INSIDE BOTTOM - The number will either begin or end directly below the binding margin depending on the side the margin is on.

OUTSIDE BOTTOM - The number will either begin or end directly below the margin opposite the binding margin.

BOTTOM - The number will be centered at the bottom of the page.

Note: (1). If the margin parameter is omitted the default is LEFT RESET.

(2). Even though a 0 is entered to indicate no page number is to be printed, one of the location options must still be specified.

(3). INSIDE TOP, OUTSIDE TOP, INSIDE BOTTOM, and OUTSIDE BOTTOM cannot be used with the margin parameter CENTER.

(4). Numbers printed at the top of the page are 12" above the top margin and numbers printed at the bottom of the page are 12" below the bottom margin.

All pages must end with the instruction

END PAGE <ID>.

SEGMENT <ID> <PARAMETERS>

END SEGMENT <ID>

Within each page any number of segments can be defined. Like the PROGRAM, SECTION, and PAGE blocks they must each have a unique name. The segment is the lowest level that can be obtained and it must begin with the first instruction. This instruction can have up to four parameters as follows:

<PARAMETERS> - This parameter is used to define the amount of the page that the segment is to cover.

(ALL) - The segment area is the same as the page area.

(<UNIT> , <UNIT> , <UNIT> , <UNIT>) - These four values represent the area of the segment by defining the distance from the page origin to the beginning and the end

of the area in each direction. The first value is the distance to the start in the X direction and the second value is the distance to the end. Similarly the third parameter is the distance to the start in the Y direction and the fourth is the distance to the end.

Note: (1). Value 2 must be greater than value 1 and value 4 must be greater than value 3.

(2). All values must be greater than 0 and must be contained within the dimensions of the page.

All segments must end with the instruction

END SEGMENT <ID>

where <ID> is the same as the name specified when the segment was defined.

Constant and Variable Declarations

DECLARE

This instruction must begin any declaration of constants or variables. It can be used only once per block.

CONSTANT

This instruction is used when constants are desired. It must follow the DECLARE statement and it also can only be used once per block.

`<ID> = <TYPE>`

This statement is the form of the constant declarations. Any number of these instructions can follow the CONSTANT statement. If, however, more than one constant is declared the names must be unique. The value of this form is that constants cannot be modified within the program while variables can.

<TYPE> - This parameter represents the value that the constant is to have. It can be an integer value, a real value, a character string, or the boolean values TRUE or FALSE.

VARIABLE

This instruction is used in the definition of variables. It must follow all constant declarations, if any, and can only be used once per block.

<ID> : <TYPE>

<ID>, <ID> : <TYPE>

These two forms are used to define variables. Any number of statements may be used and the two forms can be used in any combination. The first form is used for a single variable while the second is used if several variables of the same type are desired. These instructions must follow a VARIABLE statement.

<TYPE> - This parameter determines what kind of variable is being defined and can have any of the following values:
INTEGER; REAL; BOOLEAN; CHARACTER; AXIS; UNIT; STRING(size);
or ARRAY() OF INTEGER, REAL, BOOLEAN, or CHARACTER.

Note: Constants and variables can be defined in any of the segments, pages, sections, or even in the program block. Wherever defined, they can only be used in the block in which they are defined and in any block defined within that block. For example, the sequence

PROGRAM TEST

DECLARE CONSTANT A = 10.5

PAGE EXAMPLE

would result in the constant A being defined. Further since the page EXAMPLE is a block within the program block A could be used by the block.

Operating Instructions

These instructions are responsible for the manipulation of data within the ASGOL program. Whenever the representation <COMMANDS> is used it refers to one of these instructions.

MARGIN

MARGIN (<UNIT>,<UNIT>)

This instruction can have one of two forms and is used to set a margin relative to the page or segment area. The first form will result in a margin value of 5% of the area dimension in both directions. If specific values are desired the second form allows the exact values to be specified for both the X and the Y directions. The first

value is the X margin while the second is the Y margin.

FRAME

FRAME (value)

This instruction is used to draw a frame around the page or segment area. If the first is used the frame will have a thickness of one line. If a greater thickness is needed, the second form is used. The value or expression result in parentheses represents the number of lines in the frame. This is an integer value and can be any number in the range 0-25.

SET <VARIABLE> = <EXP>

This instruction is used to modify any variables other than UNIT, STRING, or AXIS. <VARIABLE> can be any other type including array if desired. <EXP> represents an arithmetic expression, a variable or a constant.

SET STRING <ID> = <EXP>

This instruction is used only to modify string variables. <ID> must be the name of a string and <EXP> must be a string expression, a string variable, or a string constant.

SET UNIT <ID> = <UNIT>

This instruction is used to define UNIT variables. These variables can be defined in terms of INCHES or of another unit variable. For example, if the unit variable FEET had been declared as 12 INCHES then the following statement could be used to define METERS.

SET UNIT METERS = 3.281 FEET

Through this type of definition any units that are needed can be declared removing the need for manual conversions.

SET AXIS <ID> = <DEF>

This instruction is used to define or modify an axis variable. <DEF> is defined as follows.

(<TITLE>,<TYPE>,<MIN>,<MAX>,<DELTA>, <TICKS>) - This form is used if all six parameters of the axis are defined at the same time.

<TITLE> - A character string or string variable that is 80 characters or less.

<TYPE> - One of four values LINEAR, LOG, LOGARITHMIC, or MONTH, that describes the values defined by the axis.

<MIN> - This value is either an expression representing the starting value for LINEAR or LOG axes or the name of a month (JAN-DEC) for month type axes.

<MAX> - This value has the same type as the MIN value and it represents the end point of the axis.

<DELTA> - This is an expression representing the number of incremental steps of the axis starting with 1 at the minimum value.

<TICKS> - This value is an expression representing the number of marks placed evenly between each incremental step of the axis.

```
SET AXIS <ID> . TITLE = <TITLE>
SET AXIS <ID> . TYPE = <TYPE>
SET AXIS <ID> . MIN = <MIN>
SET AXIS <ID> . MAX = <MAX>
SET AXIS <ID> . DELTA = <DELTA>
SET AXIS <ID> . TICKS = <TICKS>
```

These instructions are used to define or modify a particular parameter of an axis variable. The values have the same meaning as was discussed in the previous instruction.

```
IF <EXP> THEN <COMMANDS> END IF
IF <EXP> THEN <COMMANDS> ELSE <COMMANDS> END IF
```

The IF statement can take one of two forms depending on whether the ELSE clause is desired or not. If the expression is true the THEN clause is executed, otherwise the ELSE clause, if present, is executed. If no ELSE clause is present execution will continue with the next instruction.

WHILE <EXP> DO <COMMANDS> END WHILE

As long as the expression is true in this instruction the commands will be executed.

FOR <VARIABLE> = <EXP> <WAY> <EXP> <BY> DO <COMMANDS>
END FOR

This statement is used to execute a loop a specific number of times.

<WAY> - This parameter determines whether the loop variable is incremented or decremented. It can have one of the following two values.

TO - The BY expression is added to the variable on each iteration of the loop.

DOWN TO - The BY expression is subtracted from the variable on each iteration of the loop.

<BY> - This parameter is optional. If omitted a value of 1 is assumed.

BY <EXP> - This expression is used as the value for each loop iteration. It is either added to or subtracted from the loop variable depending on the value of the parameter <WAY>.

REPEAT <COMMANDS> UNTIL <EXP> END REPEAT

This instruction will repeat the commands until the expression is true.

CASE <VARIABLE> OF

This instruction provides a multi-path branching capability for ASGOL. This must be the first statement.

<INTEGER>,<INTEGER> : <COMMANDS>

<CHARACTER STRING> : <COMMANDS>

OTHERS : <COMMANDS>

These instructions are used to define the branches of the CASE instruction. If the integer constant form is used a string of integers can be put into one instruction. Any number of instructions can be used in one CASE statement. If the value of the variable is equal to one of the integers that instruction is executed, otherwise the OTHERS statement, if included, is executed.

Note: (1). Integer and character strings cannot be used together within the same CASE statement.

(2). Only one character string is allowed per instruction.

(3). OTHERS instruction can be used with either form.

END CASE

This instruction must be the last statement in the CASE instruction.

INPUT <ID> : <SOURCE>

This instruction is used to input data to the program. the source for this data as follows.

TERMINAL - A value is read from the terminal.

TAPE <INTEGER> - A value is read from the tape file specified by <INTEGER>.

<DIRECT INPUT> - This parameter allows data to be specified in the in the instruction and has the value

DATA /<VALUES>/

where <VALUES> can be integer, real, boolean, or character constants. These values can be in the form of a string (10, 5, 4, 3) or if multiple copies of one value is desired the

following format can be used:

DATA 10.1,5.2:3

In this case four values are defined 10.1, 5.2, 5.2, and 5.2.

OUTPUT <EXP> : <DESTINATION>

This instruction is used to write the results of the expression to the destination specified by <DESTINATION>. This parameter can have one of the following values.

TERMINAL - The value is written to the terminal.

TAPE <INTEGER> - The value is written to the tape file specified by <INTEGER>.

OUTPUT - The value is written to the standard output printer file.

CHANGE <FROM> TO <TO>

This instruction is used to change one of the six character sets that DISSPLA allows to be active at one time. DISSPLA has other character sets that could be useful. The two sets are defined as follows:

<FROM> - UPPER ROMAN, LOWER ROMAN, UPPER ITALIC,, LOWER ITALIC, UPPER SCRIPT, LOWER SCRIPT.

<TO> - SPECIAL, MATH, UPPER GREEK, LOWER GREEK,

UPPER RUSSIAN, LOWER RUSSIAN, HEBREW.

Graphing Instructions

DRAW ARROW <STYLE> (<UNIT>,<UNIT>,<UNIT>,<UNIT>)

This instruction is used to draw arrows anywhere within a PAGE or SEGMENT area. Any number of these arrows can be drawn in one plot. The first two unit parameters represent the X and Y position of the starting point and the last two parameters represent the ending point of the arrow. <STYLE> represents an integer value used to indicate the type and position of the arrow heads as shown in the DISSPLA manual (Ref 2:PART B 21-7).

DRAW LINE (<UNIT>,<UNIT>,<UNIT>,<UNIT>)

This instruction is used to draw lines in the plotting area. Like the ARROW instruction the first two parameters are the X and Y coordinates of the starting point and the last two parameters are the coordinates of the ending point of the line. There is no restriction to the number of LINE instructions that can be included within a plot.

Note: (1). Both endpoints must be within the plotting area.

(2). Any combination of ARROW and LINE instructions is valid.

GRAPH <TITLE> (<STACK><FRAME><TYPE>,
<POINTS>,<XARRAY><YARRAY>)

This instruction is the main graphing instruction for ASGOL. The parameters are defined below. Only one GRAPH instruction is allowed for a plot.

<TITLE> - This parameter is optional but if used it is a string name or a string literal of 20 characters or less.

<STACK> - This parameter is also optional. If used the form is

STACK OF <INTEGER>,

This feature allows up to 5 plots to be stacked within one PAGE or SEGMENT.

<FRAME> - Another optional parameter that when used has the form

FRAMED ,

It results in a frame being drawn along the axes of the plot.

<TYPE> - This parameter specifies the kind of plot that is desired as follows:

LINEAR - Both the X and Y axes are linear.

STEP - Steps are plotted at each data point.

BAR - A bar graph is plotted.

STACKED BAR - If more than one set of bars is plotted, they will be stacked to save space.

SPLINE - Spline smoothing will be used on the data points.

SMOOTH - Straight line smoothing will be used on the data points.

PIE - A pie graph will be drawn.

<POINTS> - This parameter is an expression representing the number of data points to be plotted.

<XARRAY> - Either an array name or direct input as defined in the INPUT instruction that represents the X coordinates of the data points to be plotted.

<YARRAY> - This parameter is required for all plot types except PIE plots. It is defined the same as <XARRAY> and contains the Y coordinates of the data points.

TEXT (<JUSTIFICATION>,<STYLE>,<NAME>,
<START>,<LENGTH>,<ANGLE><SIZE>)

This is the text processing instruction of ASGOL. The parameters are defined as follows:

<JUSTIFICATION> - This parameter specifies the justification of the text according to the parameters below.

LEFT JUSTIFIED - All lines except for paragraph indentations are flush with the left margin.

RIGHT JUSTIFIED - All lines are flush with the right margin.

L-R JUSTIFIED - All lines are flush with both margins with blanks inserted as needed.

TOP CENTERED - All lines will be centered with the first line on the top margin.

BOTTOM CENTERED - This is the same as TOP CENTERED except the last line ends on the bottom margin.

CENTERED - All lines will be centered with the first line starting on the top margin. All other lines will be spaced to allow the last line to end on the bottom margin.

<STYLE> - This parameter is used to select the print style from the following: SIMPLE, CARTOG, COMPLX, DUPLX, GOTHIC, SCmplX, SIMPLX, and TRIPLX.

<NAME> - This parameter is either a character string or the string name of the text to be plotted.

<START> - One of two forms is valid for this parameter, <EXP> or NEXT. It indicates the position in the string to start printing. <EXP> is an expression between 1 and the length of the string. When NEXT is specified printing will start with the next character if part of the string was previously printed, otherwise printing starts with the first character.

<LENGTH> - This is an indicator of the number of characters to be printed. If an expression is used it defines the exact number of characters. If, however, the value CONTINUE is used the string will be printed until the end is reached or until the end of the plot area is encountered. This facility removes the necessity for counting strings.

<ANGLE> - This parameter is an expression used to rotate the text relative to the origin of the plot.

<SIZE> - This is an optional parameter that specifies a character height for the string. If omitted, the default height of .14 inches is used.

Graph Set Up Instructions

GRID (<EXP>,<EXP>)

This instruction is used to draw a background grid on a plot. The first expression is the number of lines per inch in the X direction and the second expression is the number of lines in the Y direction. Only one GRID instruction is allowed per plot.

LEGEND <TITLE> (<POSITION>,<LINE>,<LINE>)

This instruction defines a legend for a plot. The parameters are defined below.

<TITLE> - This is an optional parameter which can be used to change the title of the legend block. If omitted, the title "LEGEND" will be used otherwise the character string or string name will be used.

<POSITION> - This position determines where in the plot the legend block will be. Care should be taken with this placement since no points are plotted within the block. The following positions are valid: LEFT TOP, RIGHT TOP, INSIDE TOP, OUTSIDE TOP, TOP, LEFT BOTTOM, RIGHT BOTTOM, INSIDE BOTTOM, OUTSIDE BOTTOM, and BOTTOM.

<LINE> - For each line in the legend a parameter must be included in the order to be printed. This parameter can either be a character string or a string variable and must be 20 characters or less.

Note: For PIE graphs these parameters represent the titles that are written in each slice of the pie.

<AXIS> = <DEF>

This instruction is used to define the X and Y axes for the plot. The parameters are defined as follows.

<AXIS> - This parameter has one of two values, XAXIS or YAXIS, depending on which axis is being defined.

<DEF> - There are two forms to this parameter. It can be an axis variable name or it can be a specific definition of all the axis parameters as shown in the SET AXIS instruction.

Arithmetic Operations

Arithmetic operators - The following operators are valid in arithmetic expressions:

+, -, *, /, MOD, REM, **

Functions - Several functions are available as defined below.

FLOAT (<EXP>) - Converts an integer number to a real number.

INTEGER (<EXP>) - Converts a real number to an integer number.

FRACTION (<EXP>) - Takes a real number and returns the fractional portion.

SIN (<EXP>) - Computes the sine of the expression.

COS (<EXP>) - Computes the cosine of the expression.

TAN (<EXP>) - Computes the tangent of the expression.

INV SIN (<EXP>) - Computes the inverse sine of the expression.

INV COS (<EXP>) - Computes the inverse cosine of the expression.

INV TAN (<EXP>) - Computes the inverse tangent of the expression.

ABSOLUTE (<EXP>) - The absolute value of the expression is returned.

STRING POINT (<ID>) - The position of the next character to be printed in a string is calculated.

LENGTH (<ID>) - This function returns the length of the string denoted by <ID>.

Logic Operators - The following logic operators are valid in conditional expressions:

<, <=, =, /=, >=, >.

Boolean Operators - Four boolean operators are allowed in ASGOL. They are:

AND, OR, XOR, and NOT.

VITA

Kevin P. Albert was born on 31 July 1951 in Houlton, Maine. He graduated from Houlton High School in 1969 and attended the University of Maine, Orono, Maine, from which he received a Bachelor of Arts degree in Mathematics in January 1973. After graduation he was employed as a computer programmer for Dun & Bradstreet, Inc., New York, New York. He entered the Air Force in December 1975, and received his commission from Officers Training School in May, 1977. He served as a computer programmer for Detachment III of the Air Force Flight Test Center until entering the School of Engineering, Air Force Institute of Technology, in June 1980. He is a member of Tau Beta Pi.

Permanent Address: 2 Brook Street

Houlton, Maine 04730

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS
BEFORE COMPLETING FORM |
|---|--------------------------------------|--|
| 1. REPORT NUMBER
GCS/MA/81D-1 | 2. GOVT ACCESSION NO.
AD A115 546 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)
AN INTERMEDIATE LANGUAGE AND
INTERPRETER FOR THE ASGOL
GRAPHICS LANGUAGE | | 5. TYPE OF REPORT & PERIOD COVERED
MS THESIS |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)

Kevin P. Albert, CAPT, USAF | | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS
Air Force Institute of Technology
Wright-Patterson AFB, Ohio 45433 | | 10. PROGRAM ELEMENT, PROJECT, TASK
AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS
Air Force Flight Dynamics Laboratory
Wright-Patterson AFB, Ohio 45433 | | 12. REPORT DATE
December 1981 |
| | | 13. NUMBER OF PAGES
138 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)
UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING
SCHEDULE |
| 16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited. | | |
| 17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

15 APR 1982
Dean for Research and
Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433 | | |
| 18. SUPPLEMENTARY NOTES
Approved for public release; IAW AFR 190-17

F. C. Lynch, Maj, USAF
Director of Information | | |
| 19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Intermediate Language
DISSPLA Software Package
LR(1) Parsing System
Graphics Language | | |
| 20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
In an effort to make ASGOL (ALGOL-Structured Graphics
Oriented Language) more powerful, conditional and looping
instructions were added to the language. To do this the
existing interpreter system was converted to a
compiler/interpreter system. An intermediate language was
designed as the compiler output and thus the source language
for the interpreter. | | |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

132