

C:

DESIGN AND IMPLEMENTATION OF A
HIGH-LEVEL LANGUAGE FOR INTERACTIVE COMPUTER GRAPHICS

by

Albertus Jacobus (Bert) Pieké
Diplom in E.E., ETH-Zürich 1969

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in the Department
of
Electrical Engineering

We accept this thesis as conforming to the
required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1973

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of ELECTRICAL ENGINEERING

The University of British Columbia
Vancouver 8, Canada

Date 17 July 1973.

ABSTRACT

The design and implementation of the interactive graphics language IGL is described. This language not only allows the definition and display of line drawings but also has full facilities for manipulating, naming, identifying and interacting with such drawings. The language has been implemented as an extension to Fortran IV using the XPL compiler generator system. The experience gained so far in the use of the language has already proven a number of advantages over present-day graphics systems. The language is readily learned by users with previous high-level language experience. As no extensive testing and documentation is necessary due to the readability of the program, the time required for the completion of a project is greatly reduced.

TABLE OF CONTENTS

| | <u>Page</u> |
|---|-------------|
| ABSTRACT | i |
| TABLE OF CONTENTS | ii |
| LIST OF FIGURES | iii |
| ACKNOWLEDGEMENT | iv |
| 1. INTRODUCTION | 1 |
| 2. COMPUTER GRAPHICS, A SHORT SURVEY | 3 |
| 3. GRAPHIC LANGUAGES | 6 |
| 4. THE LANGUAGE IGL | 10 |
| 5. IGLCOM, AN IGL COMPILER | 19 |
| 6. IGLRUNLIB, AN IGL RUNTIME LIBRARY | 21 |
| 7. IMPLEMENTATION OF IGL ON A DATA GENERAL SUPERNOVA MINI COMPUTER | 29 |
| 8. EXPERIENCE WITH THE IMPLEMENTED IGL SYSTEM | 35 |
| 9. BIBLIOGRAPHY | 36 |
| REFERENCES | 37 |
| APPENDICES: | |
| A IGL users' guide | 40 |
| B Illustrations from IGL programs | 58 |
| C Listings: IGL-Syntax, IGLCOM | 62 |
| D Device Dependent Support Routines | 73 |

LIST OF FIGURES

| | | <u>Page</u> |
|---|------------------------------|-------------|
| 1 | Sample IGL program | 16 |
| 2 | Display algorithm | 26 |

ACKNOWLEDGEMENT

The author wishes to thank his supervisor, Dr. G.F. Schrack and also Dr. F. Nake (formerly of the Dept. of Computer Science, U.B.C.) for their interest, advice, and encouragement throughout the period of this research. Thanks are also due for the many helpful suggestions by the people in this department who have used the IGL language in its initial stages and to Miss Norma Duggan for typing the thesis.

Financial support for this research was provided by the National Research Council of Canada as a Research Assistantship and a Bursary to the author.

1. INTRODUCTION

Over the last years, an increasing need has been felt in the Electrical Engineering Department at this University for a computer facility to permit computer aided design.

A first attempt was made in the summer of 1970. A program was written to enable a user to enter the topology of an electronic circuit diagram to a PDP-9 computer, using a display screen and light pen as communication interface [19]. This program had a number of remarkable features such as an elegant windowing facility and automatic labelling of elements. It was reasonably well documented and tested and yet proved to be less useful than hoped for.

The reason was simple. Some additional features were needed, some inconsistencies still existed and the interfacing to a circuit analysis program had not been included. So changes had to be made to the program and the author was no longer available. The program was written in assembler and was quite complex. The effort needed for somebody else to familiarize himself with the programming techniques used was considerable and the project was never completed.

This example shows clearly that the design of complex software systems should occur on a more general level if the systems are to be used by many different users and if they are to be expandable to future needs. This is particularly true in a university environment where a regular turnover of students using and updating an existing system takes place.

This thesis describes, after an introduction to the field of computer graphics, the design and implementation of a high-level

language for interactive graphics, the resulting graphics system, and the experience gained with it.

2. COMPUTER GRAPHICS, A SHORT SURVEY

Interactive techniques are widely used in problem solving by computers. The purpose of these techniques is to make efficient use of both man and machine. The computer is used to carry out those actions that can be specified in advance and human reasoning is called upon to direct the course of those actions. To be able to do this, the user must have a clear understanding of the state of his computations and he must have a means of indicating his commands to the computer. A graphics display provides a natural way of communicating information to the user. Line drawings and graphs are much easier understood than tables of numbers. Also a graphical input device such as a light pen or joystick allows the user to transmit his wishes with a maximum of convenience. This kind of interactive computing is called interactive computer graphics.

The earliest attempt at computer graphics was made in 1962 when Ivan Sutherland implemented the now famous "Sketchpad" system on the TX-2 computer at Lincoln Laboratory at MIT [20]. The publication of his work triggered research in this field by many others and led companies like General Motors, Boeing and Lockheed to set up graphics systems for computer aided design (CAD) [18].

All these systems used expensive display terminals (such as the IBM 2250) and required a huge computer to monitor the interaction. Their cost effectiveness has been highly disputed and the economic decline in the aerospace industry has caused the development of many new CAD projects to be abandoned.

In the meantime, a new generation of computer hardware has

appeared and with it many new concepts in computer software [2]. The high cost of computer graphics had been determined by two factors:

- the high cost of the terminal and main computer hardware
- the high cost of developing the necessary programs.

The cost of computer hardware has decreased sharply over the last decade and the availability of mini computers and time-sharing systems has further reduced the cost of interactive computing. Storage tube displays have eliminated the necessity of continuous refreshing of the display screen and thereby made economical terminals for computer graphics a reality.

A great effort has been made to match this progress in hardware with a similar cost cutting progress in software. As the hardware gets less expensive, the software development costs are becoming an ever more dominating factor in any versatile computer system. The previous approach where all software was written in assembly language for maximum runtime efficiency is now being overtaken by an effort to provide the programmer with easy to use computer languages that enable him to write his programs with a maximum of programming time efficiency. A number of special purpose languages have recently appeared in the literature that attempt to make the programming of graphics applications as easy for the programmer as programming numerical programs in conventional algorithmic languages such as Fortran and Algol. The first "Graphic Languages" were published in 1968 [5,10,13] and the development of such languages has since been the subject of a special conference [8].

The advantage of a powerful graphic language is obvious. It allows the user of a graphics system to write the programs necessary

for his special application himself. He can experiment with ideas and make changes to his programs without having to rely on a programmer to do this work for him. Thus he has control over and can be made responsible for his own graphics programs. What is equally important is the readability of programs written in graphic languages. It not only reduces the testing and documentation time considerably, but (as most programs usually end up with inadequate documentation) it allows changes to be made to the program by consulting just a program listing.

The next chapter will describe the nature of graphic languages and attempt to identify some of the points that are important to applications in computer aided design.

3. GRAPHIC LANGUAGES

The variables in graphic languages that are manipulated are pictures, two-dimensional structures. One of the most difficult problems in computer graphics is to find a concise description for such pictures. Even the simplest of two-dimensional drawings contains a great amount of implicit information. For example, a drawing consisting of only two lines contains information about the lengths, the positions, and the angles of the lines relative to a frame of reference and relative to each other; also whether they are solid, dotted, dashed, intersecting, etc. The more complicated the drawing, the greater the amount of such implicit information becomes. Even more confusing is the case where pictures are considered to include grey-levels and raster sizes, common factors where pictures are digitized for processing. To simplify the description, we will look at only generation, not recognition, of pictures and consider all pictures to be line drawings consisting of solid lines of equal thickness.

3.1 Primitives

The primitives of a language are the constants defined in the language that can be broken down no further. In Fortran these are the real, integer, and logical constants, in a graphic language they are the basic picture constants. Line drawings consist of lines, or rather line segments. The dot can be treated either as a line segment of length zero or as a special symbol.

Defining only DOT and LINE as the primitives of a graphic language is sufficient, but treating pictures directly at this primitive level is rather bothersome and so usually some additional primitives

are defined in the language as a help for the programmer. A set could consist of elementary geometric symbols like: DOT, LINE, SQUARE, CIRCLE, TRIANGLE, ARC, etc. The representation of the numbers and the characters of the alphabet are also often defined in the language as picture primitives. Depending on the display hardware other characters may be included.

3.2 Operators

The operators of a language are the symbols that stand for manipulations to be performed on variables and constants. Here our incomplete understanding of pictures becomes evident. We are able to define basic mathematical operators like $+$, $-$, $*$ for manipulating numbers without trouble, as we are so familiar with using these symbols in calculus where they are well defined. A basic set of operators for pictures, however, is not as easily extracted from geometry, the mathematical discipline which studies manipulation of two dimensional structures. Even including other relevant disciplines, as Graph Theory and Set Theory, does not bring us much closer to understanding pictures yet. Research is being done in this direction, however [8].

The easiest way of finding a useful set of operators is by comparing existing graphic languages and picking out features that seem desirable. As an example we shall consider the process of building up pictures:

One operator should allow us to add subimages together to form a picture. This operation can be done as a superposition of two pictures or as a concatenation of two pictures.

(a) In the case of superposition some coordinate system must be known

for both subpictures. This leads to considering each subpicture to be defined in a "frame" [10], and superposition involves superimposing the two frames.

- b) In the case of concatenation, the pictures must contain some specially designated points at which they can be joined together. This leads to considering each subpicture as having a "head" and a "tail" and concatenating pictures head to tail only [13].

Both approaches have their limitations, in (a) it is difficult to keep track of what is now connected to what and in (b) each subpicture is limited to one head and one tail.

Further operators should allow us to delete pictures or parts of pictures, to move pictures, to scale pictures and maybe to display pictures. As no arithmetic operators, apart from + for adding and - for deleting, seem to apply to pictures in a natural manner, most other forms of manipulation are described as functions. Giving these functions mnemonic names makes them easy to remember.

Examples:

SCALE, MOVE TO, ROTATE, WINDOW, DISPLAY ...

or built in constraints like:

PARALLEL, VERTICAL, POINT ON LINE ...

3.3 Assignments

In building pictures it must be possible to give names to subpictures, in order to use these names for building up more complicated pictures. Most languages include the assignment statement in one way or another. The symbol used is the same as in arithmetic assignments (=, :=, ←). Some published languages use instead a procedure oriented approach. Instead of assigning a pictorial expression

to a variable, they assign a procedure to a variable. This procedure is executed whenever the variable is referenced in the program and causes the corresponding picture to be drawn. The decision for or against display procedures will most likely be influenced strongly by the programming language available for the implementation. Procedure oriented languages such as Euler have been used successfully for designing graphic languages [16] and similar results have been achieved using APL and APL-like structures [4,6]. The reason that the language used for the implementation shows through so strongly in the design of a graphic language is the following.

The creation of a complete, versatile computer language can be a huge task if carried out from basics. However, if the graphic language is defined as an extension of an existing algorithmic language, only those constants, operators, and functions that are not already available need to be included. Features such as, e.g. branching, conditional and arithmetic statements and the complete I/O handling facility can be used directly as defined in the algorithmic language which now takes the place of "host-language" for the graphics language. Thus a graphics program is really a mixture of graphics statements and host-language statements, and the designer of such a language will usually choose his graphics syntax definitions to fit in with the overall syntax structure of the host-language.

4. THE LANGUAGE IGL

The acronym IGL stands for Interactive Graphics Language, a computer language for handling line drawings. This language not only allows the definition and display of line drawings but also has full facilities for manipulating, naming, identifying and interacting with such drawings. Through the use of windowing techniques, parts of the drawing can be magnified for closer inspection or scaled down for greater display density. Commands are further provided for saving and restoring drawings on secondary storage.

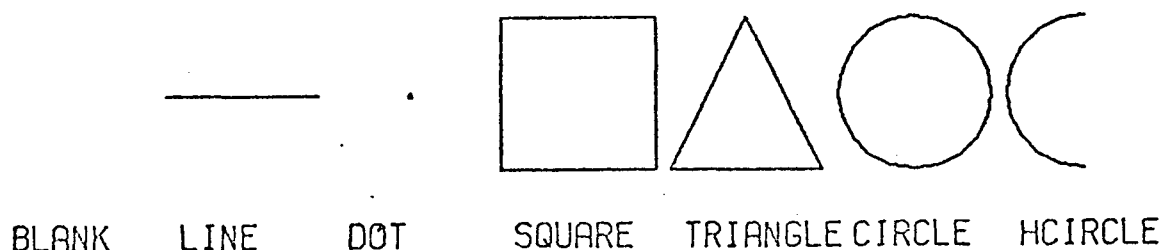
An applications program written in IGL appears to be a mixture of statements for graphical manipulations and host-language statements. The distinction between the two is made on a card-by-card basis, the graphical statements having a special character (*) in the first column of each card. The compilation is executed in two stages: first a compilation from IGL into host-language, secondly a compilation of the host-language into machine language.

Because of its widespread use and support, Fortran IV was chosen both as host-language for IGL and as programming language for the routines in the IGL runtime library. This one-language approach provides almost complete portability from one computer installation to another.

4.1 Syntax

The graphical aspects of the syntax of IGL are based on the work of F. Nake [14]. In addition to existing types of constants and variables in the host-language (e.g. integer, real, etc.), the new type IMAGE is introduced. Image constants are either elements from the set

{BLANK, DOT, LINE, SQUARE, TRIANGLE, CIRCLE, HALFCIRCLE} or strings of literals (keyboard characters). Image variables are defined with the aid of the image assignment operation ($:=$) by image expressions which are strings of image constants and/or variables joined by diadic image operators. Image constants and variables are initially defined on the unit square as follows:

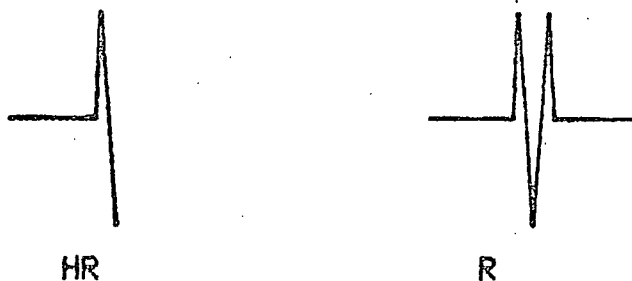


For each use of an image variable, a set of attributes is attached defining coordinates, scale, and angle of rotation of the picture which the variable represents. These attributes can be redefined with the use of unary image operators, hence affine transformations such as translation, scaling, rotation and mirroring can be applied.

For illustration, the following image assignment statements define an image variable to represent the symbol "resistor".

```
*      HR:= LINE FROM 0,.5 TO .4,.5 + LINE FROM .4,.5 TO .425,1
*      + LINE FROM .425,1 TO .5,0;
*      R:= HR + HR VSYM .5;
```

The right hand sides of both statements employ the diadic image operator $+$ (superposition). The first statement defines a temporary image variable HR by superimposing the image constant LINE three times, each time modified with the image operator FROM ... TO ... in the desired manner.



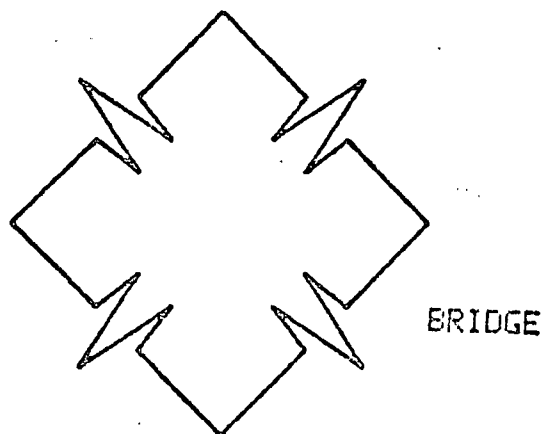
The second statement defines the variable R as a superposition of HR and the reflection of HR on the vertical at $x = .5$, thus completing the symbol "resistor".

The numeric operands of image operators may be not only numeric constants but can be numerical variables of the host-language as well. Therefore, complicated pictures can be defined with image assignment statements imbedded in statements of the host language, e.g. in loops. For example, the following program portion:

```

      S = .25*SQRT(2.)
      SC = 2.*S
      ALPHA = PI/4.
*      BRIDGE:= BLANK;
      DO 100 I = 1, 4
      X1 = .5 + S * SIN(ALPHA)
      Y1 = .5 - S * COS(ALPHA)
*      BRIDGE:= BRIDGE + R AT X1,Y1 SCALE SC,.1 ANGLE ALPHA;
100    ALPHA = ALPHA + PI/2.
*      DISPLAY BRIDGE;
```

would display a resistor bridge:



The facilities described above allow the creation and naming of items called image variables. For most applications, however, an additional facility is needed to group into logical groupings, items that do not necessarily look alike. As an example consider an electronic circuit diagram. At first glance it may seem to be composed of a limited set of identical symbols. Apart from being in different places in a circuit, two resistors, e.g., may seem to be duplicates of each other. A closer examination, however, shows that the leads of the first have different lengths than the leads of the second and that the labels differ from one another. Such details as the length of the leads and the label are properties of the pictures representing the individual resistors and must be taken into account. A second category of variables, subscripted image variables, is therefore defined in the syntax. Subscripted variables names are names of items that are not necessarily affine transformations of one another, but that are logically members of the same set. All network elements in a circuit can thus be named, e.g., `ELEMENT(I)` regardless of whether they are resistors, capacitors etc. Alternatively, if this distinction is important, all resistors in a circuit can be named `RESISTOR (I)` regardless of the length of their leads or

the letters in their label.

Example:

```
*      RESISTOR (3): = R AT X,Y SCALE S1 + LEAD FROM X1,Y1 TO X2,Y2 +
*      LEAD FROM X3,Y3 TO X4,Y4 + 'R' AT X5,Y5 + VALUE(3) AT X6,Y6;
*      CIRCUIT:= CIRCUIT + RESISTOR(3);
```

The function VALUE, used in this example, allows the conversion of an integer value to an image value. A similar function, TEXT, accepts a variable containing a text string for conversion.

For a more detailed description of the IGL syntax and semantics, the reader is referred to the IGL Users' Guide in Appendix A of this thesis.

4.2 Interaction

Describing the interactive process means defining the response of the system to each input. The response is not only dependent on the type of input, but also on the state of the system at the time the input occurred. W.H. Newman [15] has proposed to treat the system as a finite-state automaton where the response is determined by the state of the program as well as by the action. The actions are inputs to the automaton, which cause it to change its state; reactions are the outputs.

The interaction is then best described in the form of a state diagram, which is used as a guide when writing the program. To facilitate this process, the syntax of IGL allows a close correspondence between the state diagram notation and the formulation in the program.

This aspect of the IGL-system is based on the work of P. Boullier et al. [1]. The program is divided into states which correspond to the states

in the diagram. For each input (e.g. a hit on a menu symbol on the screen) a decision is made within the current state of the program as to which state is to be executed next. For an illustration, see Figure 1.

The division into states also provides a physical segmentation of the program which can be used for paging or overlaying at computer installations with insufficient core storage for the entire program, this will be discussed in greater detail in Chapter 7. Control statements are defined in the language to allow variables to be displayed on the screen, to turn on the cursor or cross-hairs (which can be positioned by the user with the help of some graphical input device such as a joy-stick, tracker ball or light pen) and to detect an interrupt from the user, signalling that he has chosen an item on the screen. A special variable: IHIT is used in the program to indicate the subscript of the item last identified on the screen. This subscript allows the programmer to refer to items that are pointed at by the user when he is interacting with the program. For example the sequence:

```
*      DISPLAY 'IDENTIFY RESISTOR TO BE DELETED' AT .2,.9;
*      CURSOR ON; WAIT FOR INTERRUPT;
*      FOR HIT ON RESISTOR: CIRCUIT:= CIRCUIT - RESISTOR (IHIT);
```

would prompt the user to select the resistor to be deleted and would execute the deletion.

4.3 Semantics and Data Structure

The semantics are defined in the routines in the runtime library that perform the actions specified by the statements written in the program. The control and interaction routines affect the sequence of the execution and the communication with the graphics terminal,

Fig. 1 Sample IGL program

```

* STATE 1:
C
C ==> DEFINE ELEMENTS AND MENUSYMBOLS
C
    ***
    ***
C ==> CAPACITOR
* HALFC := LEAD + LINE FROM .4,0 TO .4,1;
* C := HALFC + HALFC VSYM .5 + MARKER;
    ***
    ***
C ==> MENUCOMMANDS
* DELETE:='DELETE'; SAVE:='SAVE'; ROTATE:='ROTATE';
    ***
    ***
* GOTO STATE 2;
* END STATE 1;

* STATE 3:
C
C ==> STATE 3 : PICKING FROM THE MENU
C
* DISPLAY 'CHOOSE FROM MENU' AT .1,.9;
* CURSOR ON; WAIT FOR INTERRUPT;
* FOR HIT ON MENU:
* FOR HIT ON DELETE: GOTO STATE 4; END;
* FOR HIT ON ROTATE: GOTO STATE 5; END;
* FOR HIT ON SAVE: GOTO STATE 6; END;
    ***
    ***
* END;
    ***
    ***
* END STATE 3;

* STATE 5:
C
C ==> STATE 5 : ROTATING AN ELEMENT
C
* DISPLAY 'PICK ELEMENT TO BE REVERSED' AT .1,.2;
* CURSOR ON; WAIT FOR INTERRUPT;
* FOR HIT ON MENU: GOTO STATE 3; END;
* FOR HIT ON ELEMENT:
* AN = ANGLE(ELEMENT(IHIT));
  PI=3.141593
  AN=AN+PI
* ANGLE(ELEMENT(IHIT)) = AN;
* ERASE SCREEN; DISPLAY CIRCUIT;
* END; GOTO STATE 3;
* END STATE 5;

```

whereas the assignment routines operate on a data structure that represents the current state of the pictorial information. The display is derived algorithmically from this structure and no separate display file is needed. The correspondence between the variable names used in the program and the locations in the data structure is kept through a hash-coded directory where all names are stored together with pointers to the structure. (For a description of hash-coding see [11]). The data structure is implemented as a linked list [9,21] with "brother"-pointers linking items that are connected by the (+) operator and "son"-pointers linking downwards through the structure to subitems and primitives that are used in the definition of items. No "father"-pointers are kept linking upwards through the structure, so it would not be possible to say e.g. in which items the primitive LINE is referenced. This means that the cursor identification on the screen does not follow the usual pattern of determining which line is closest to the location of the cursor and then determining to which item this line belongs. Instead, the syntax of the identification statement: FOR HIT ON <variablename>: <statementlist> END; allows a scan of all items with the name <variablename>, which are linked by additional "buddy"-pointers, to determine whether any of those items were within a tolerance region (specified by the x and y-scale of the item) around the cursor location.

4.4 System Configuration

Two implementations are used at this university. One runs under the MTS timesharing system on an IBM 360/67 with an Adage/10 graphics terminal, Calcomp and line-printer plotters. This system is used mainly for debugging new programs. The other system runs under

DOS on a Data General 20k Supernova with a Tektronix 4010 graphics terminal. Graphics programs can be run on either system without modification.

5. IGLCOM, AN IGL COMPILER

IGLCOM, the IGL compiler used in this implementation was written using the XPL compiler writing system [12].

This is a well documented and easy to use program package written in the XPL dialect of PL/I. To build a compiler using the XPL system, the language syntax in Backus Naur notation (BNF) and the corresponding semantics in XPL must be supplied. These two user components are processed as follows. The syntax is read in, printed, analyzed and a parser is punched out by the XPL program ANALYZER. ANALYZER will check that the grammar is unambiguous and will attempt to modify the grammar if that is not the case. The produced parser is in a format that allows it to be directly inserted into a compiler framework called SKELETON. SKELETON itself is written in XPL and has two open slots, one for accepting the parser from ANALYZER and one for accepting the semantics. When these two components have been added, SKELETON can be compiled by the XPL compiler producing for the input grammar a compiler in object code.

The semantics are written as an XPL procedure named SYNTHESIZE. This procedure is called in SKELETON each time a grammar rule is applied by the parser for reducing the input string. The number of the particular rule applied is passed along as an argument. In SYNTHESIZE the output language is generated. If the output is in a high level language, the output statements will usually consist of calls to semantic routines. If the output is in assembly language, the actions can be generated directly. For passing names from the input to the output language, SYNTHESIZE has access to the contents of the parsing stacks. Pointers

are kept by SKELETON to show the current state of these stacks.

This automation of the compiler-writing process allows changes to the language to be carried out easily when needed. The syntax and the corresponding semantics can be updated independently of one another. Compilers generated by the XPL system provide good error checking facilities and have proven to be quite efficient.

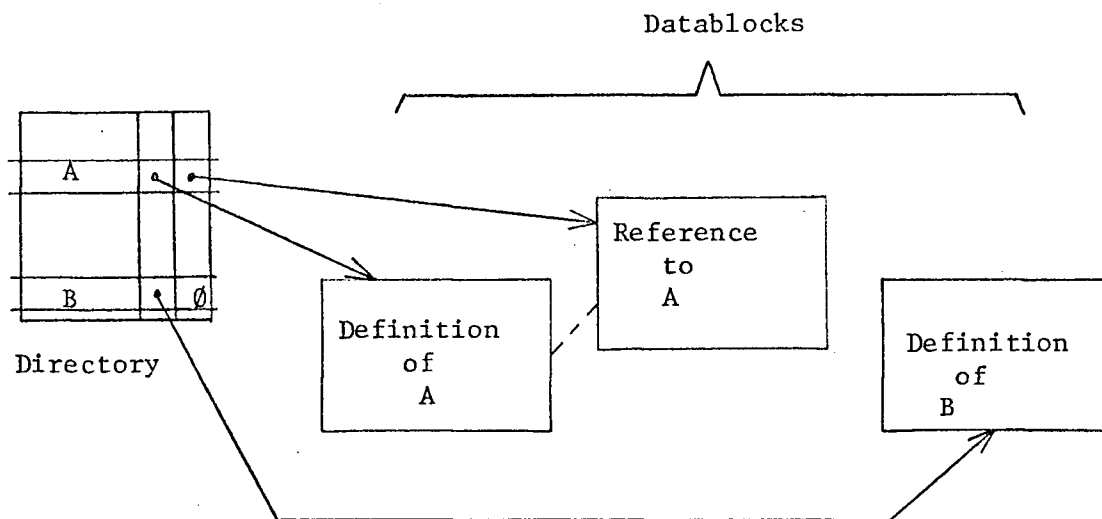
Complete listings of the IGL syntax description in BNF and of the SYNTHESIZE routines can be found in Appendix C of this thesis.

6. IGLRUNLIB, AN IGL RUNTIME LIBRARY

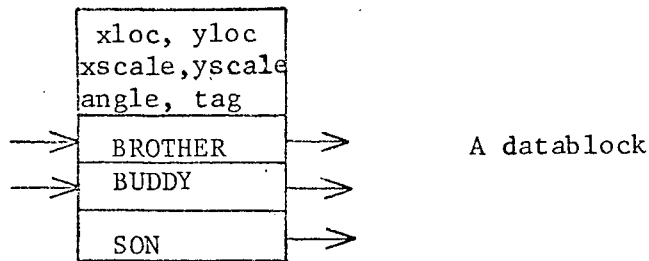
The routines in IGLRUNLIB maintain a datastructure, representing the pictorial information created by the executing IGL program, and handle the communication with the graphics terminal.

6.1 Datastructure

The overall datastructure has been introduced in Chapter 4. It consists of a linked list of datablocks, addressable through a hash-coded directory. The directory contains the name, the location of the first definition datablock and the location of the first datablock that references that definition for every named item. The primitives, defined in the language, are flagged in the directory to prevent their alteration by the program.



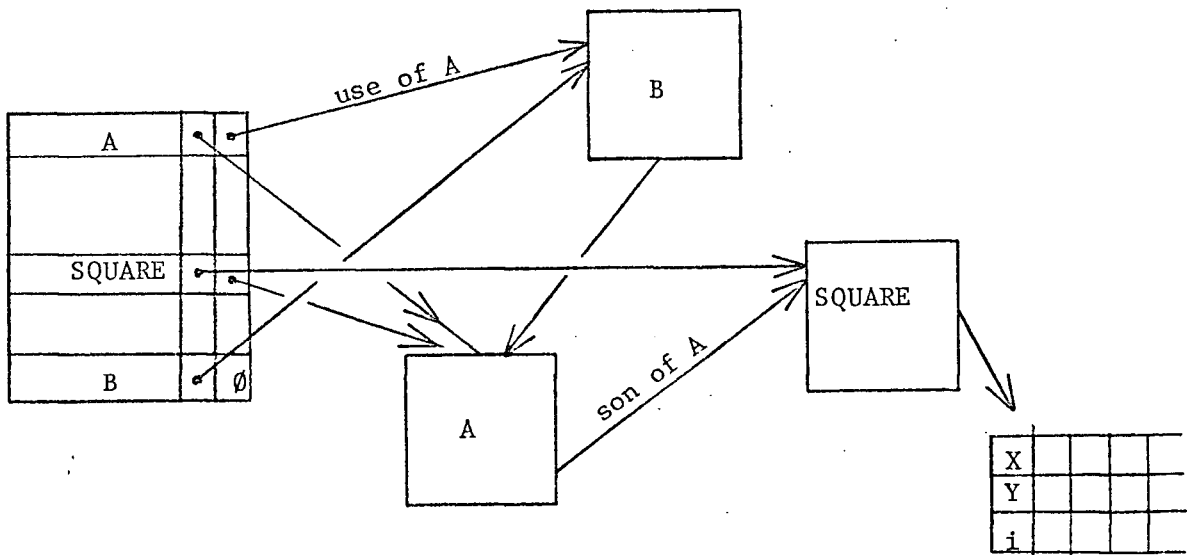
The datablocks contain the location (xloc, yloc), scaling (xscale, yscale), rotation (angle), and subscript (tag) of a picture item, and three pointers as links to other items.



The son-pointer links the datablock to its definition block(s) in the structure. It is a one-way pointer; no pointers link "sons" back to "fathers". The datablocks for the primitives have son-pointers that point to their graphical description in an array of (x,y,ipen) triples. To distinguish this pointer from normal son-pointers, a value of 10,000 is added to its correct value.

Example:

* A:= SQUARE SCALE .1; B:= A ANGLE 45;

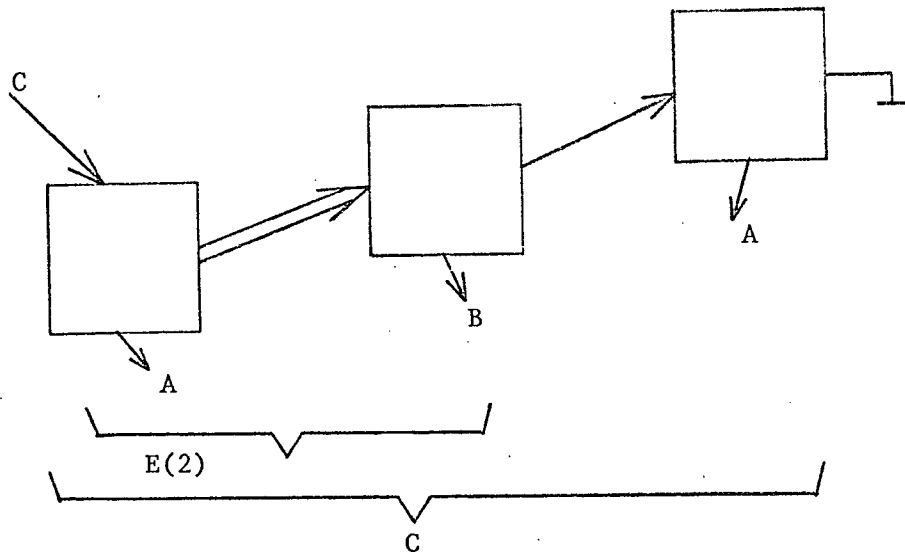


SQUARE is here a son of A, who is in turn a son of B.

The brother pointer links together datablocks that are logically connected by the (+) operator. The end of the link is indicated by a zero entry. Brother pointers exist on two levels: "stronger" links within the definition of subscripted variables and "weaker" links within the definition of unsubscripted variables. This allows a two level hierarchy within a linear list.

Example:

* $E(2) := A + B; C := E(2) + A;$



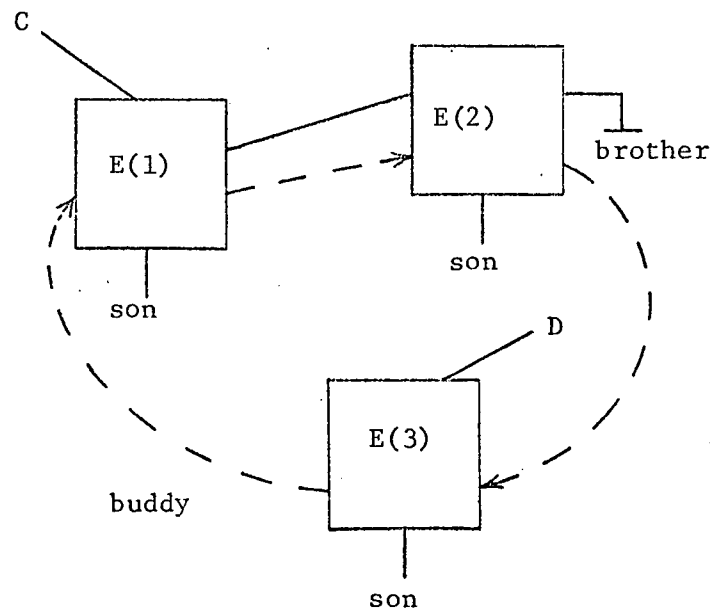
The statement: * $C := C - E(2);$ would now cause the deletion of both of the first two blocks in C, as they are linked by the stronger links. The third block, however, will be retained as it is linked by a weaker link, meaning that although it is part of C, it is not a part of E(2).

The buddy pointer links together datablocks that have the same name. They are essential for access to any item given by name and

subscript. This access is accomplished as follows: The name is looked up in the directory, where the location of the first definition and referencing block is found. A sequential search of all blocks with that name is made by following the buddy pointers. A comparison between the stored "tag" in the datablock and the subscript given allows the item to be identified. As the buddy pointers are linked in a closed ring, the end of an unsuccessful search is determined when returning to the first block.

Example:

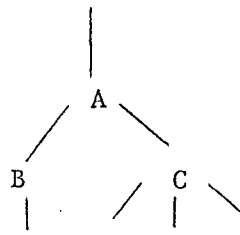
* C:= E(1) + E(2); D:= E(3);



6.2 Display Algorithm

For displaying a picture, the information in the data structure has to be transformed into a linear sequence of beam or pen movements on the display device.

In the following the terms father, son and brother are defined as shown:



A is the father of B and C

B is the first son of A

C is B's brother

The algorithm is the following: Starting at the image to be displayed, the first son (subimage) is taken and the valuation of the father applied. The valuation is the set of attributes: (XLOC, YLOC, XSCALE, YSCALE, ANGLE). This is carried out repetitively until a primitive is encountered, the intermediate steps being saved in a push-down stack. Then the primitive is displayed using the accumulated transformations.

After this, the top item in the push-down stack is popped up and it is treated as a new starting image. This process is stopped when the stack is empty and all brothers of the original father have been considered. The process is illustrated by the flowchart in Fig. 2.

In order to be able to apply a father's valuation to his son we must consider the following:

The valuation involves a SCALING, ROTATION and TRANSLATION of some point P (X,Y) in a coordinate system. The equations for this transformation are:

$$\bar{X} = (X - 0.5) XSCALE \cdot \cos\alpha - (Y - 0.5) YSCALE \cdot \sin\alpha + XLOC$$

$$\bar{Y} = (X - 0.5) XSCALE \cdot \sin\alpha + (Y - 0.5) YSCALE \cdot \cos\alpha + YLOC$$

or:

$$\bar{P} = [A] \cdot P + [B] \quad \text{where:}$$

$$[A] = \begin{bmatrix} XSCALE \cdot \cos\alpha & -YSCALE \cdot \sin\alpha \\ XSCALE \cdot \sin\alpha & YSCALE \cdot \cos\alpha \end{bmatrix}$$

DISPLAY 'NAME'

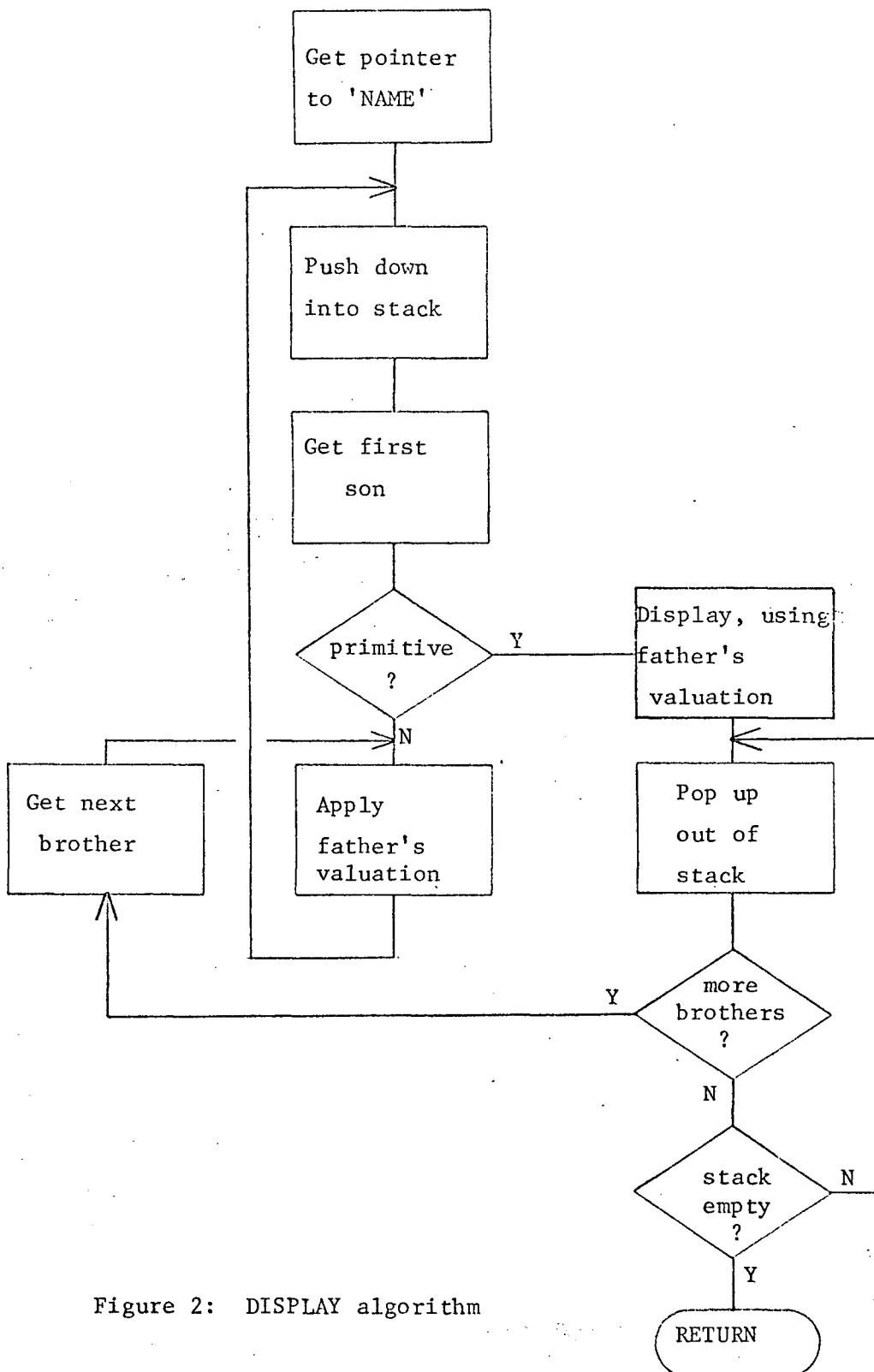


Figure 2: DISPLAY algorithm

$$[B] = \begin{bmatrix} XLOC + \frac{YSCALE \cdot \sin \alpha - XSCALE \cdot \cos \alpha}{2} \\ YLOC - \frac{XSCALE \cdot \sin \alpha + YSCALE \cdot \cos \alpha}{2} \end{bmatrix}$$

Storing the valuation as ([A], [B]) rather than as (XLOC, YLOC, XSCALE, YSCALE, α) allows us to apply one valuation (A_f, B_f) to another (A_s, B_s).

$$\bar{P} = A_f \bar{P} + B_f = A_f (A_s P + B_s) + B_f$$

$$= A_f A_s P + A_f B_s + B_f = \bar{A} P + \bar{B}$$

$$\rightarrow \bar{A} = A_f \cdot A_s$$

$$\bar{B} = A_f B_s + B_f$$

For this reason the internal representation of the valuation components is a sextuplet ($A_{11}, A_{21}, A_{12}, A_{22}, B_1, B_2$) rather than the quintuplet used in the IGL-syntax.

6.3 Terminal Interface

The communication with the graphics terminal takes place on a logical level, using a coordinate system of 1.0 by 1.0 as defined in the language. Keeping this interface on a logical level, rather than on a physical level, makes it relatively easy to interface to different terminals.

The logical terminal is assumed to be capable of four things:

- erase the screen
- move the beam or draw a vector from the current beam position to location X,Y
- display a textstring of n characters, starting at location X,Y
- display a cursor on the screen, wait for the user to position the cursor, return the X,Y coordinates of the cursor, and the

charactercode of the button pushed by the user to indicate his interrupt.

The calling sequences are:

| | |
|---------------------------|-----------------------|
| CALL ERASE | IPEN = 0: move beam |
| CALL VECTOR (X,Y,IPEN) | IPEN = 1: draw vector |
| CALL DISPS (ITEXT, X,Y,N) | |
| CALL WAITIN | |

COMMON/CURSOR/IDUMMY, X,Y, KEYHIT

Any other input/output can be programmed using conventional Fortran WRITE and READ statements.

7. IMPLEMENTATION OF IGL ON

A DATA GENERAL SUPERNOVA MINICOMPUTER

Although, during the implementation phase, every effort was made to keep the programs hardware independent, it was unavoidable that the particular computer system used had some influence on the decisions that were made.

The most far-reaching decision was made in choosing Fortran as programming language for the runtime library. Although the language Fortran IV was standardized by ASA in 1964 [7], a combination of practical considerations and the ambiguities in the published description of the language have led to the present situation, where the Fortran compilers supplied by computer manufacturers include most (if not all) features of ASA Fortran and, to make up for any deficiencies, usually offer added on special features not found in ASA Fortran. The average user of such a compiler will tend to regard his particular expanded subset of ASA Fortran as the "only Fortran" and distrust any other implementations of the language. This is particularly the case if the implementation used is the IBM Fortran IV G-level compiler, regarded by many as the ultimate definition of the Fortran language.

The runtime library was written in a subset of Fortran common to the IBM and Data General compilers. These routines can thus be executed on either computer without modification. This fact does not insure, however, that these routines will execute on any computer due to the above reasons.

Only the routines that communicate with the graphics terminal differ in the two implementations. The routines for the Tektronix 4010 terminal are written in assembler. They were supplied with the terminal

and are documented by Tektronix. The routines for the Adage/10 terminal are written in Fortran using the UBC AGT:BASIC support package and can be found in Appendix D of this thesis.

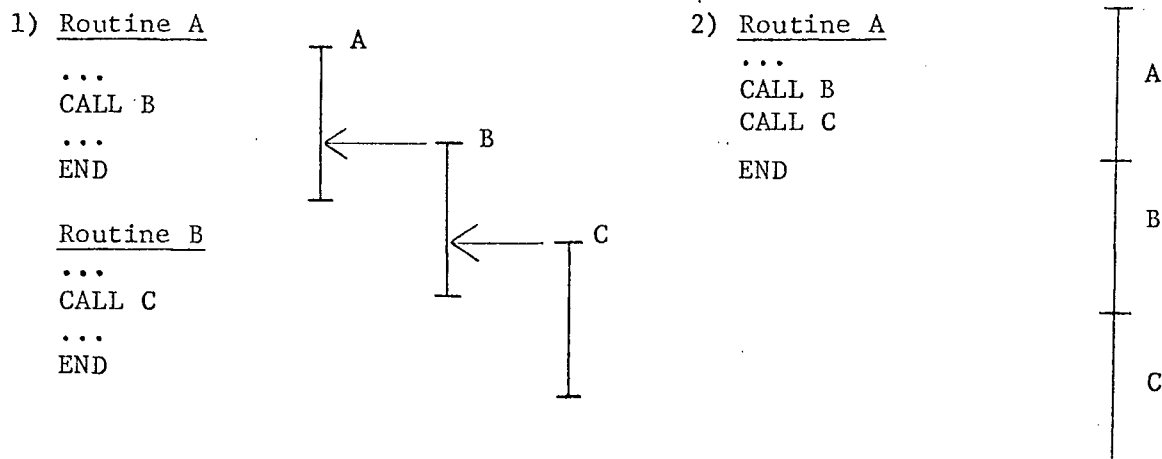
The same restrictions that apply to the runtime library apply to any user program written in IGL. A program that will execute correctly on the IBM 360 may not do so on the Supernova. The Users' Guide lists some of the points to be watched when writing a program, most of which need not concern us here. There are a few points, though, that are of more general interest and may apply to a number of computers.

Wordlength - Many third generation computers have followed the example of the IBM 360 system, using a wordlength of 32 bits (4 bytes), divided into 2 halfwords of 16 bits (2 bytes). Fortran compilers on these machines typically allow integer wordlengths of one word (INTEGER*4) or of one halfword (INTEGER*2). For numerical algorithms this fact is of no great significance as long as it is kept in mind that the largest allowable INTEGER*2 value is 32767, which may be quite low for many problems. More noticeable is the effect on the character handling capabilities, which in Fortran are linked to the integer wordlength. Many minicomputers are organized around a wordlength of 16 bits, which leaves them with the restriction of allowing only INTEGER*2 values. The Supernova is an example. To be compatible between the two machines, only 16 bit integers should be used for character manipulation. These facts naturally limit the portability of any Fortran program from one computer installation to another, but interactive programs especially are dependent on character handling for evaluating responses and values typed in by the user.

Storage - 16 bit machines are designed to use two computer words for storing a floating point value. One word would not permit sufficient accuracy. This means that the storage of REAL values uses twice the amount of space needed for storing the same number of INTEGER values. For this reason, all arrays representing the IGL datastructure are stored as integer values. This is at a cost in accuracy (4-5 versus 7 decimal places), a cost in conversion for arithmetic operations, and at the risk of integer overflow if unexpected large values should occur.

Another problem in storage on a minicomputer is the very finite size of the main memory. Designers of operating systems and programmers of large programs have resorted to various techniques of using available external storage space (fixed or moving head disk) to augment the power of the main (core or semiconductor) memory. The simplest technique is to keep part of the data used on external storage and read it in every time it is to be used. Another possibility is to keep part of the program to be executed on external storage and read it into the main memory when needed; this scheme is generally referred to as overlaying.

The Data General disk operating system allows up to 6 levels of overlays and in addition allows chaining of segments. To understand the difference between these two methods, we consider the following example.



We assume that any of the routines A-D will fit into the main memory separately, but no two routines will fit in together. Each routine is now considered a separate program segment. In case 1), the execution of segment A must be suspended for segment B to be brought into core. When B has finished, A can continue. As A, however, would be overwritten by B, it must first be saved.

An overlay thus involves:

- saving the current segment A
- loading the needed segment B
- starting the execution of B.

A return from overlay involves:

- reloading the previous segment A
- restarting the execution of A at the location where the execution was suspended.

In case 2), the execution of segment A is finished when segment B is needed. There is thus no need to save A, and B can be loaded directly. This process is called chaining.

A chaining involves:

- loading the needed segment B

- starting the execution of B.

A return from chaining does not exist.

The communication between segments is only possible through the BLANK COMMON area, the only part of core that is neither saved nor restored when executing an overlay. (Communication by saving relevant data on external storage is of course always possible). The present version of DOS (release 5) will, however (contrary to its description), store and load all of core, including blank common. This can be prevented by calling the initialization routine INCOM, described in Appendix D.

The division of an IGL program into states makes an automatic overlaying scheme feasible. Each state corresponds to a separate overlay segment. As no return to a previous state is provided for in the IGL syntax, segments can be chained. Apart from saving time (chaining: 3 sec., overlay: 7 sec. in our system), the chaining allows the execution to proceed at the same level in DOS, thus setting no limit to the number of states to be executed. The IGL compiler translates the goto statement: GOTO STATE n into: CALL CHAIN ('STAT \bar{n} .SV'), where n is the number of the state to be executed. ($\bar{n} = n$ for $n > 9$, $\bar{n} = 0 n$ for $n \leq 9$).

Because of the limited storage size of 20k words main memory, it was initially found that the runtime library was taking up too much space, leaving only a miniscule amount for the user program. The solution has been to move the display and error routines to separate overlay segments. This is regrettable, as the display routines are called often, and executing an overlay for each display command causes a very noticeable pause in the interaction.

For an analysis of this situation, it is useful to examine the storage allocation within the main memory and see which parameters can be varied. With an IGL program executing, the main memory of the computer is roughly divided up as follows:

| | |
|---------------------------------|------------|
| Page zero and Fortran addresses | .5 k |
| IGL user program | 1.5 k * |
| IGLRUNLIB | 4.5k(*) |
| Fortran runtime library | 4 k |
| DOS stacks | 1 k |
| BLANK COMMON | 2.5 k * |
| DOS SYSTEM | <u>6 k</u> |
| | 20 k |

Of these sections, only the starred values are variable, all others are fixed, regardless of program size.

What would now be the effect of increasing the memory size by 4k to 24 k words? The following changes could be made to the current implementation:

- The display routines are added to the main segment (1.5 k)
- The BLANK COMMON area and user program are allowed an additional 2.5 k.

This would mean instant response to a display command, more data storage and larger (and hopefully more interesting) applications.

The present implementation is successful, in as far as it proves that an IGL system can be implemented and used on a moderately small machine. For the system to be truly convenient and sufficiently powerful for real world applications, however, a minimum storage size of 24 k words is absolutely necessary.

8. EXPERIENCE WITH USING THE IMPLEMENTED IGL SYSTEM

The flexibility and ease of use make the language IGL ideal for experimenting with new graphics techniques and ideas. To gain experience with the system, several applications were programmed by different users. (See Appendix B). First projects were a program for drawing electronic circuits and a load flow program for the analysis of power systems [3]. In two 12 week systems lab projects, groups of fourth year students in this department implemented complete interactive systems; one for interactive project scheduling using CPM methods and one for interactive nonlinear circuit analysis, [17]. Both systems run on a minicomputer in this department and interface to analysis programs on the IBM 360 installation in the Computer Centre. A voice grade data link is used for transmitting data from one computer to another. The experience gained so far in the use of the language has already proven a number of advantages over present-day graphics systems.

Due to the definition of the syntax and the semantics, changes of these are easily incorporated as experience is gained in the use of the language. The system has been found far easier to use than the graphical subroutine packages found on most computer installations. The language is readily learned by users with previous high-level language experience. As no extensive testing and documentation is necessary due to the readability of the program, the time required for the completion of a project is greatly reduced. Furthermore, the availability of a new powerful tool stimulates the imagination of the user to tackle problems previously considered out of reach.

9. BIBLIOGRAPHY

Until recently, little interaction has taken place between the different workers in the field of Computer Graphics. No textbooks existed on graphics and articles were scattered throughout various publications such as Communications of the ACM, Computing Surveys, AFIPS conference proceedings etc.

However, in 1970 a first conference was held on Computer Graphics at Brunel University. In 1972, the journal "Computer Graphics and Image Processing" appeared and a working conference on Graphic Languages was held by IFIPS at the University of B.C. Finally, an excellent textbook has appeared in 1973, "Principles of Interactive Computer Graphics", by W.H. Newman, that contains the essence of a way of thinking about computer graphics that was developed over the last 10 years.

REFERENCES

1. P. Boullier et al., "METAVISU, A general purpose graphic system", in F. Nake, A. Rosenfeld, Eds., Graphic Languages, Amsterdam: North-Holland Publ. Co., 1972.
2. I.W. Cotton, R.S. Grestorex, "Data structures and techniques for remote computer graphics", AFIPS Proc. FJCC, 1968.
3. B.A. Dixon, "Interactive Graphical Load Flow", submitted to: International Electrical, Electronics Conference and Exposition, Toronto, Oct. 1973.
4. J.D. Duffin, "A language for line drawing", Tech. Rep. #20, Comp. Sc. Dept., U. of Toronto, May 1970.
5. A.J. Frank, "B-line, Bell line drawing language", AFIPS Proc. FJCC, 1968.
6. "Grapple - Graphical Programming Language Reference Manual", Bell-Northern Research, Feb. 1972.
7. "History and Summary of FORTRAN standardization development for the ASA". Comm. ACM, Oct. 1964.
8. IFIP Working Conference on Graphic Languages, in F. Nake, A. Rosenfeld Eds., Graphic Languages, Amsterdam: North-Holland Publ. Co., 1972.
9. D.E. Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley, 1969.
10. H.E. Kulsrud, "A general purpose Graphic Language", Comm. ACM, April 1968.
11. W.D. Maurer, "An improved Hash Code for scatter storage", Comm. ACM, January 1968.
12. W.M. McKeeman et al., A Compiler Generator, Prentice-Hall, 1970.
13. W.F. Miller, A.C. Shaw, "Linguistic methods in picture processing - A survey", AFIPS Proc. FJCC, 1968.
14. F. Nake, "A proposed language for the definition of arbitrary two-dimensional signs" in O.I. Grusser, R. Klinke, Eds., Zeichenerkennung in biologischen und technischen Systemen. Berlin: Springer, 1971.
15. W.M. Newman, "A system for interactive graphical programming", AFIPS Proc. SJCC, 1968.
16. W.M. Newman, R.F. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill, 1973.

17. B. Pieké, G.F. Schrack, "Interactive circuit analysis using a high-level graphics language", 16th Midwest Symposium on Circuit Theory, Waterloo, 1973.
18. M.D. Prince, Interactive Graphics for Computer-Aided Design, Addison-Wesley, 1971.
19. S. Semrau, "PDP-9 circuit input program", E.E. Dept., U. of British Columbia, 1971.
20. I.E. Sutherland, "SKETCHPAD - A man-machine graphical communication system", AFIPS Proc. SJCC, 1963.
21. R. Williams, "A survey of data structures for computer graphics systems", Computing Surveys, March 1971.

APPENDICES

APPENDIX A: IGL USERS' GUIDE

1. Introduction

IGL is an acronym for Interactive Graphics Language, a computer language for writing programs for handling line drawings. This language not only allows the definition and display of line drawings, but also has full facilities for manipulating, naming, identifying and interacting with such drawings.

The IGL system consists of IGLCOM, the IGL compiler, and IGLRUNLIB, the IGL runtime library.

2. The IGL Language

The definition of the IGL language has been formally given in BNF notation, but this definition should not be required reading for the average user. Instead, this users' guide will attempt to introduce the language in a less formal manner, using examples to explain the use of IGL statements.

IGL has been implemented as an extension to Fortran IV, allowing the user to mix Fortran IV and IGL statements in the same program. IGL statements are recognized by the IGL compiler by examining the first column of each line. The characters {`—`, 0, 1...9, C, A, X} indicate a Fortran statement, any other character an IGL statement. In all examples in this manual, the character `*` will be used to indicate IGL statements.

The rules for the coding of IGL statements do not follow the Fortran convention. IGL statements can be written in free format, with no regard for card boundaries. One rule should be followed, however. Keywords in either Fortran or IGL are considered to be reserved words.

They cannot be used as variable names and must be written without imbedded blanks. Keywords should be separated by at least one blank.

Example: READ, which is legal in IBM Fortran IV is not legal in an IGL program.

3. Writing an IGL Program

An IGL program is divided up into states, each state corresponds to a logical state in the interaction.

Consider the following example:

A 'rubber-band' line can be created by means of a joystick and a push-button in a sequence of five operations.

- 1) press button to turn on cursor;
- 2) move cursor, using joystick, to starting point of line;
- 3) press button to fix starting point;
- 4) move cursor to end point of line;
- 5) press button to fix end point.

A state diagram might look as seen on the next page.

Here a feature is added to allow deletion of lines and to continue drawing lines from the last endpoint. Each of the 5 states shown represents a different stage in the interaction. This can be seen by the meaning of the push-button in each state.

in state 2 it means: Fix the starting point of a new line,
and the end point of the line

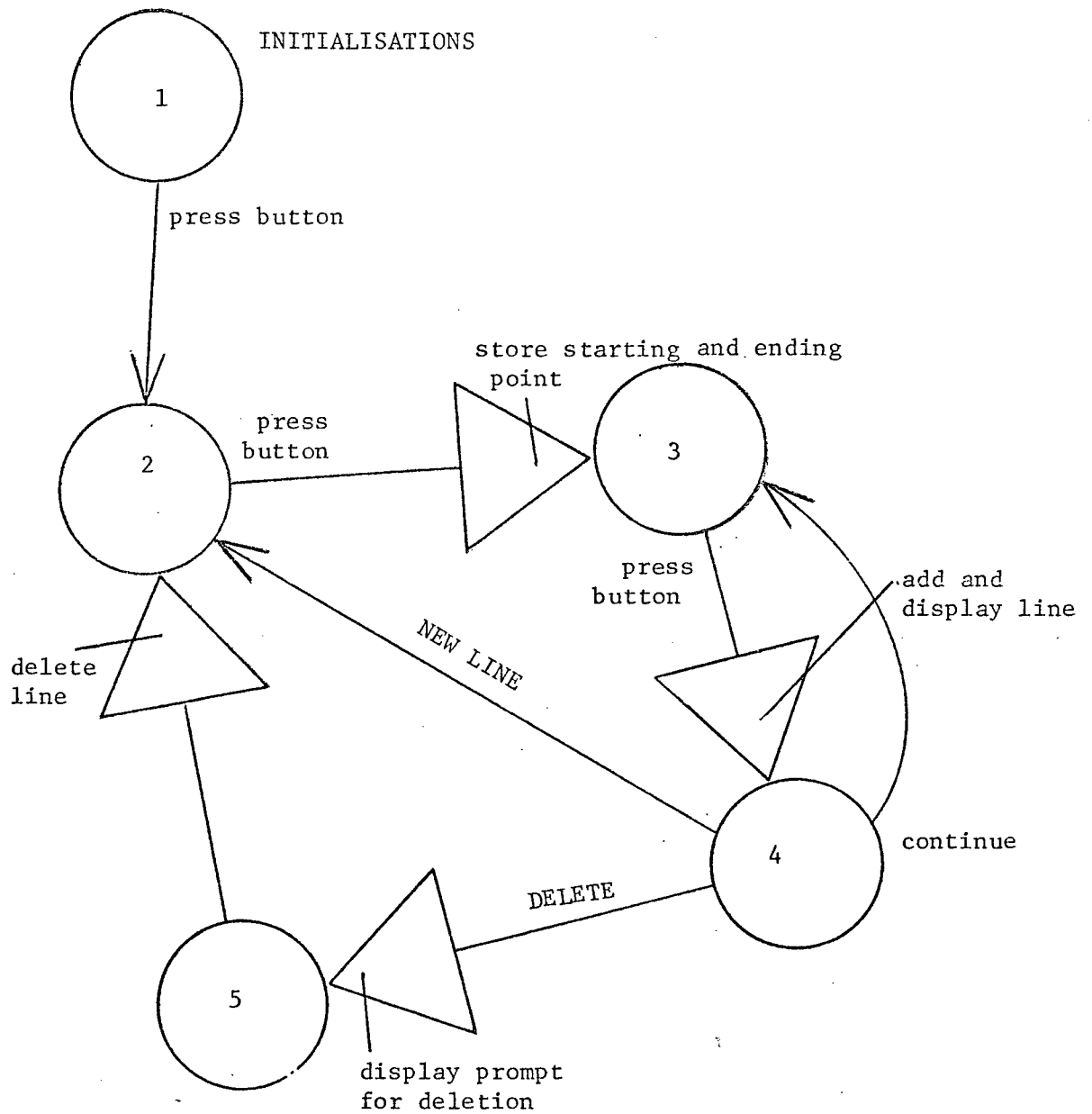
in state 4 : Execute a command from the menu

in state 5 : Delete the line that is pointed to

An IGL program that corresponds to this state diagram would look as follows:

STATE DIAGRAM

(creating a 'rubber-band' drawing)




```

* STATE 1:
C
C --- INITIALISATIONS
C
COMMON X1,X2,Y1,Y2,I
* NEWLINE:='NEW LINE'; DELETE:='DELETE';
* DRAWING:=NEWLINE SCALE .1,.1 AT 1,15,.8 + DELETE SCALE .1,.1 AT 1,15,.6;
I=0
X1=0.
Y1=0.
PAUSE 'PRESS BUTTON'
* ERASE SCREEN; DISPLAY DRAWING; GOTO STATE 2; END STATE 1;

* STATE 2:
C
C --- STORING STARTING AND ENDING POINT OF A LINE
C
COMMON X1,X2,Y1,Y2,I
* CURSOR ON; WAIT FOR INTERRUPT;
X1=XHIT
Y1=YHIT
* DISPLAY DOT AT X1,Y1;
* CURSOR ON; WAIT FOR INTERRUPT;
X2=XHIT
Y2=YHIT
* GOTO STATE 3; END STATE 2;

* STATE 3:
C
C --- ADDING LINE TO DRAWING
C
COMMON X1,X2,Y1,Y2,I
I=I+1
* L(I):=LINE FROM X1,Y1 TO X2,Y2; DISPLAY L(I);
* DRAWING:=DRAWING + L(I);
* GOTO STATE 4; END STATE 3;

* STATE 4:
C
C --- CHOOSING FROM MENU
C
COMMON X1,X2,Y1,Y2,I
* CURSOR ON; WAIT FOR INTERRUPT;
* FOR HIT ON NEWLINE: GOTO STATE 2; END;
* FOR HIT ON DELETE: GOTO STATE 5; END;
X1=X2
Y1=Y2
X2=XHIT
Y2=YHIT
* GOTO STATE 3; END STATE 4;

* STATE 5:
C
C --- DELETING A LINE
C
COMMON X1,X2,Y1,Y2,I
* DISPLAY 'CHOOSE LINES TO BE DELETED' AT .2,.9;
1 CONTINUE
* CURSOR ON; WAIT FOR INTERRUPT;
* FOR HIT ON L: DRAWING := DRAWING-L(IHIT);
GOTO 1
* END;
* ERASE SCREEN; DISPLAY DRAWING; GOTO STATE 2;
* END STATE 5;
* EOF EOF EOF

```

This example shows how the interaction can be programmed in IGL and already uses most of the statement types to be described below. A number of points should be noted here, however.

- The program is divided into states of which the first to be executed must be STATE 1.

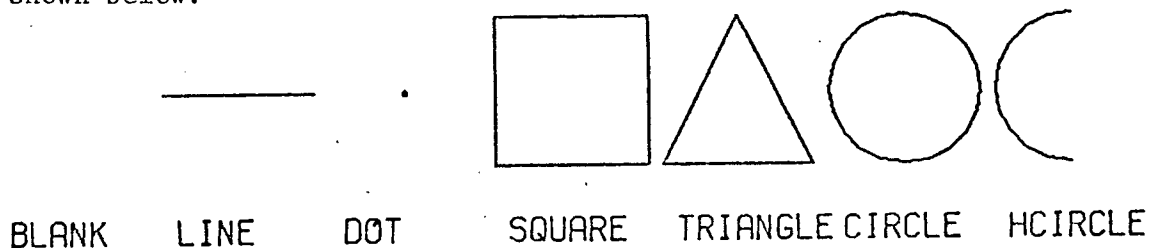
- Fortran variables can be passed between states using BLANK COMMON only. The length of BLANK COMMON areas must be declared equal in all states.

- Fortran statement numbers are local to states.

- All statements in IGL are terminated using a semicolon (;).

Constants

There are two types of image constants in IGL: picture constants and literal constants. Picture constants form the set {BLANK, LINE, DOT, SQUARE, TRIANGLE, CIRCLE, HCIRCLE} and are defined as shown below.



Literal constants are textstrings, delimited by quotemarks. They can have a maximum length of 10 characters.

Functions are provided to convert Fortran values to corresponding IGL image values.

VALUE converts an integer value

TEXT converts an integer variable containing a textstring.

(The string is assumed to be 10 characters long).

Example:

'DELETE', 'HI, THERE', VALUE (3) or TEXT (ITEXT) are literal constants.

Variables

Image variables stand for pictures and/or literals and can be assigned image values. They are named according to the Fortran rules for variable names, but can have up to 8 characters.

Example:

DELETE or RESISTOR are legal variable names.

Assignments

Image variables are assigned values using an assignment statement. The image assignment operator is the (:="). All values on the right hand side of the assignment must have been previously defined.

Example:

* DELETE:= 'DELETE'; A:= SQUARE; B:= VALUE (I3);

Modifiers

Constants and variables can be modified, using valuation modifiers. All affine transformations can be applied. The modifiers are:

- translation: AT x,y
- scaling: SCALE xs, ys or SCALE s
- rotation: ANGLE alpha
- mirroring: VSYM axis or HSYM axis

(the lower case names stand for Fortran REAL values)

Any combination of these can be specified and will be applied in the following order:

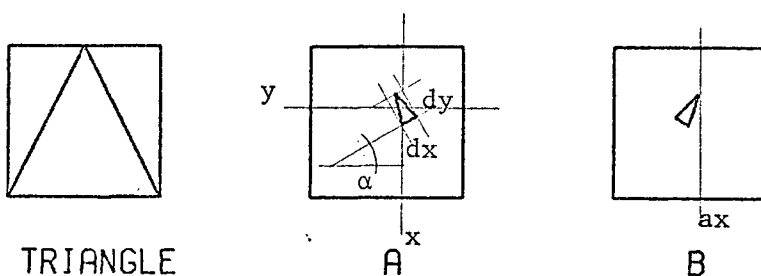
Scaling, rotation, translation, mirroring

As all picture variables are defined within a unit square, it helps to visualise what happens to this square when the modifiers are applied.

Example:

```
* A:= TRIANGLE SCALE dx, dy ANGLE alpha AT x,y;
```

```
* B:= A VSYM ax;
```



For line segments, the modifiers as given are awkward to use. The following can be used:

```
-FROM x1, y1 TO x2, y2
```

This corresponds to:

```
SCALE xs, ys ANGLE alpha AT x,y
```

where: $XS = \text{SQRT}((X2-X1)^2 + (Y2-Y1)^2)$

$YS = .1 * XS$

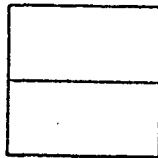
$ALPHA = \text{ARCTAN}((Y2-Y1)/(X2-X1)) * 180/\pi$

$X = (X1 + X2)/2$

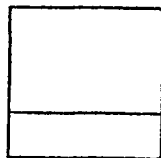
$Y = (Y1 + Y2)/2$

Example:

```
* HORIZON:= LINE FROM 0,.3 TO 1,.3;
```



LINE

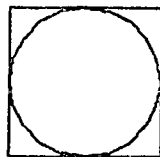


HORIZON

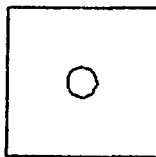
Note: A trailing decimal point need not be written. No value larger than 3.0 should be used. (Larger values may produce integer overflow at runtime). If the scaling factors for X and Y are identical, only one value need be given.

Example:

```
* SUN:= CIRCLE SCALE .1;
```



CIRCLE



SUN

Default values are assigned for any modifiers that are not specified.

The default values are:

```
SCALE 1,1  ANGLE 0  AT .5,.5
```

Expressions

The right hand side of an assignment can be an image expression.

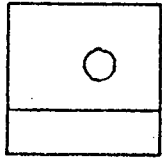
The operators are (+) for superposition and (-) for deletion.

Example:

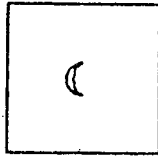
```
* SUNSET:= HORIZON + SUN AT .6,.6;
```

```
* MOON:= HCIRCLE SCALE .1 + HCIRCLE SCALE .05,.1;
```

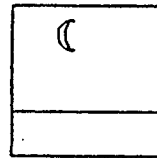
```
* NIGHT:= SUNSET - SUN + MOON AT .4,.8;
```



SUNSET



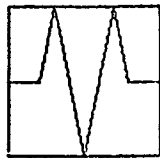
MOON



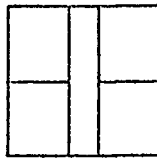
NIGHT

Subscripted variables

It may be useful to assign the same variable name to pictures that are not affine transformations of one another. Assume that R and C are defined as follows:



R



C

Both may be referred to as ELEMENTS (E):

```
* E(1) := R; E(2) := C;
```

```
* DIAGRAM := E(1) SCALE dx, dy AT x1, y1 + E(2) SCALE dx, dy
               AT x2, y2;
```

or alternatively:

```
* E(1) := R SCALE dx, dy AT x1, y1;
```

```
* E(2) := C SCALE dx, dy AT x2, y1;
```

```
* DIAGRAM := E(1) + E(2);
```

Identification and Hitarea

An important feature of each picture item is the size of the area surrounding it which is identified on the screen as being part of the item. This area is called the hitarea of the item. The size of

the hitarea is defined by the XSCALE and YSCALE attributes of the item. This means that the hitarea corresponds to the original unit square in which the item was defined. A distinction must be made between item definitions and uses of items:

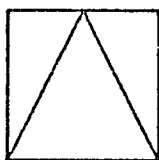
A definition:

```
P:= TRIANGLE SCALE .5;
```

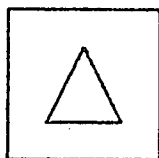
defines the item P. Every time the item P is referenced in defining other items, a use of the item P is created. Its hitarea is the size of the unit square of P, modified according to the attributes of this use of P.

Example:

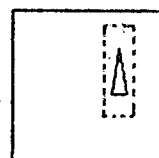
```
* PICTURE:= P SCALE .2,.6 AT .7,.6;
```



TRIANGLE



P



PICTURE

The dotted square of size .2 by .6 is the hitarea of P. Because P was defined as a triangle, scaled by a factor $\frac{1}{2}$, the hitarea of all uses of P will be rectangles double the size of the triangle displayed.

Note: For hitarea size, the modifiers applied to subscripted variables are cumulative.

Example:

```
* Q(3):= SQUARE SCALE .5; PICTURE:= Q(3) SCALE .2,.6;
```

The hitarea for Q(3) is $.5 \cdot (.2 \text{ by } .6) = .1 \text{ by } .3$. The reason for this is that there is only one item Q(3). If it is modified, its attributes are changed. In contrast, there may appear many uses of P,

each with its own hitarea.

Interaction

Identification on the screen is accomplished as follows:

The statements: `CURSOR ON; WAIT FOR INTERRUPT;` cause the cursor (crosshairs) to be displayed on the screen and cause the program to wait for a key to be pressed on the keyboard.

When a key is pressed, the X and Y coordinates of the cursor are stored in the global Fortran variables `XHIT` and `YHIT` and the literal code of the key in `KEYHIT`. (On the Nova: ASCII code of key in left byte, right byte = null; on the 360: EBCDIC code of key is left byte, right bytes = blanks).

The statement:

```
FOR HIT ON <variable>: <statement list> END;
```

allows the user to determine whether the cursor was within the hitarea of any item named <variable>. Is this the case, the statements in <statement list> are executed and the subscript of the item hit is stored in the Fortran variable `IHIT`. If there was no hit, the program will branch to the statement following the `END`. For-statements can be nested but the nesting has to close within each state. If an unsubscripted variable is identified on the screen, the variable `IHIT` contains the value 0.

Example:

```
* FOR HIT ON MENUAREA:
* FOR HIT ON DELETE: GOTO STATE 5; END:
* FOR HIT ON ROTATE: GOTO STATE 4; END;
  :
  :
* END;
```



```
* FOR HIT ON E:
```

```
* CIRCUIT:= CIRCUIT - E(IHIT);
```

```
.....
```

```
* END
```

```
* CURSOR ON; WAIT FOR INTERRUPT;
```

```
* CIRCUIT:= CIRCUIT + NODE SCALE .03 AT XHIT, YHIT;
```

Attribute functions

The functions XLOC, YLOC, XSCALE, YSCALE, ANGLE and TAG allow access to the attributes given to any picture item. Note the use of the equal sign (=).

Example:

```
* X1 = XLOC (ELEMENT (IHIT));
```

```
* ANGLE (E(IHIT)) = ALPHA2;
```

```
* TAG (Q(3)) = 4;
```

The last statement will cause the item previously called Q(3) to be renamed Q(4). Q(3) now no longer exists.

Jumps

The GOTO STATE n statement is used to transfer control to the beginning of another state. Within a state, Fortran GOTO's and labels (statement numbers) should be used.

Example:

```
* GOTO STATE 2; or GOTO STATE 2;
```

Display Statements

A number of commands allow the user control over the graphics terminal.

A display statement causes an item to be displayed on the screen.

Example:

```
* DISPLAY CIRCUIT; DISPLAY R AT X1, Y1;
```

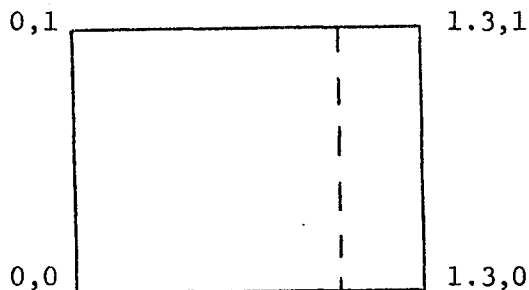
```
* DISPLAY 'GOOD MORNING' AT .7,.8;
```

The erase statement: ERASE SCREEN will cause the screen to be erased.

Partial erasure of the screen is not possible.

The Tektronix 4010 screen

The unit square is mapped onto the rectangular screen as follows:



The screen is physically addressable by 781 x 1024 points, corresponding to a 1 x 1.3 ratio. The character size on the screen is fixed (approx. 14 x 22 points). Textstrings are neither scaled nor rotated. Such values, if given, are ignored for display purposes.

Writing to and reading from the terminal can further be programmed in Fortran. The terminal is addressed as a teletype (output: WRITE (10), input: READ (11)). Vectors can be drawn on the screen using the IGL statement: DRAW FROM x1, y1 to x2, y2. Such vectors are not stored in the data structure.

Example:

```

C --- DRAW A GRID, RASTER SIZE = .05

DO 1 I = 1, 21

  V = 0.5 * (I-1)

  * DRAW FROM 0, V to 1, V; DRAW FROM V,0 TO V,1;

1    CONTINUE

```

Windowing

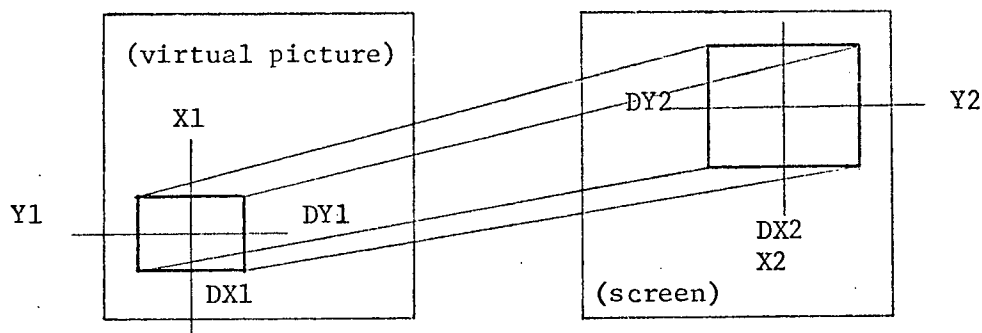
The display command allows a window to be set to show only part of a picture on the screen.

Example:

```

* DISPLAY CIRCUIT WITHIN X1, Y1 DX1, DY1 ONTO X2, Y2 DX2, DY2;

```

Saving and restoring on secondary storage

The statement:

```

* STORE nm,n;

```

will cause the complete IGL data structure and in addition a following blank common area of n words to be stored on a disk file with the name as specified in the variable nm . This file will be created if it does not already exist.

The corresponding statement:

```

* RESTORE nm,n;

```

will cause the file as specified in nm to be read n and used as the current data structure.

Differences between the IBM 360 version and the Data General Supernova version of IGL

The differences between the two implementations are twofold:

- 1) The procedure for compiling and running IGL is different.
- 2) The two Fortran implementations are different.

Procedures for compiling and executing IGL programs

IBM 360

Datagen Supernova

First card in program:

* 360 VERSION

* NOVA VERSION

Compiling program:

\$R ELEC:XPL 0 = ELEC:IGLCOM 2 = -OUT SCARDS = igl-source

SPRINT = igl - listing

Compiling intermediate Fortran program

\$R * FORTRAN SCARDS = -OUT

SPUNCH = igl-object

or

\$R* WATFIV PAR=SIZE=40

SCARDS= -OUT+ELEC:IGLRUNLIB.S

Transfer -OUT to Supernova,
split up X into individual
programs, one for each state,
named STAT01...STATn
FORT STAT01; FORT STAT02 ...
R1DR STAT01 IGLRUNLIB; ...

Execute Program

\$R igl-object+ELEC:IGLRUNLIB 9=plotfile

(plots go to plotfile)

or

\$R igl-object+ELEC:IGLAGTLIB

(interaction through Adage terminal)

IGLGO
(interaction through Tek-
tronix 4010 terminal).

Differences in the Fortran implementation on the IBM 360 and the Datagen Supernova

Below, a list is given of the major variations of Datagen Fortran from the IBM Fortran IV definition.

- All variables not stored in COMMON are placed on a run-time stack. Any program that does not alter COMMON storage is therefore a reentrant program. (Variables in subroutines (or states) are not saved from one call to the next!)

- Program units must be ordered as follows:

FUNCTION, SUBROUTINE statement.

- Declaration statements. These begin with the keywords COMMON, COMPLEX, DIMENSION, DOUBLE, EQUIVALENCE, EXTERNAL, INTEGER, LOGICAL or REAL.

- Statement functions and internal subprograms (FORMAT statements and DATA initialization statements may be given in this area).

- Executable statements. (FORMAT statements and DATA initializations may be given in this area).

- Imbedded blanks are significant except when they appear in the name of a program variable or in the statement identifier GOTO (GOTO).

- Statement identifiers, operator names, and names of library functions are reserved and cannot be used as program variables. The reserved names are:

| | | | | |
|---------|------------|------------------|-------------|------------|
| .AND. | ASSIGN | DATAN | DREAL | LOGICAL |
| .EOT. | ATAN | DATAN2 | DSIGN | MAXØ |
| .EQ. | ATAN2 | DATN2 | DSIN | MAX1 |
| .FALSE. | BACKSPACE | DBLE | DSQRT | MINØ |
| .GE. | BINARY | DCABS | DTAN | MIN1 |
| .GT. | BLOCK DATA | DCCOS | DTANH | MOD |
| .LE. | DABS | DCEXP | ENDFILE | PARAMETER |
| .LT. | CALL | DCLOG | ENTRY | PAUSE |
| .NE. | CCOS | DCMPLX | EQUIVALENCE | READ |
| .NOT. | CEXP | DCOS | EXP | REAL |
| .OR. | CLOG | DCSIN | EXTERNAL | RETURN |
| .TRUE. | CMPLX | DCSQRT | FLOAT | REWIND |
| ABS | COMMON | DEXP | FORMAT | SIGN |
| ACCEPT | COMPILER | DFLOAT | FUNCTION | SIN |
| AIMAG | COMPLEX | DIM | GOTO | SINH |
| AINØ | CONJG | DIMENSION | IABS | SNGL |
| ALOG | CONTINUE | DLOG | IDIM | SQRT |
| ALOG1Ø | COS | DLOG1Ø | IDINT | STOP |
| AMAXØ | CSIN | DMAX1 | IF | SUBROUTINE |
| AMAX1 | CSQRT | DMIN1 | IFIX | TAN |
| AMINØ | DABS | DMOD | INT | TANH |
| AMIN1 | DAIMAG | DO | INTEGER | TYPE |
| AMOD | DATA | DOUBLE PRECISION | ISIGN | WRITE |

- Names identical to DGC extended assembler mnemonics are not available for use as subprogram names.

- DATA initialization is provided for labeled COMMON only.

- Only COMMON variables can be EQUIVALENCed.

- DATA initialization of labeled COMMON is possible in any Fortran program or subprogram.

- Subprogram names must be unique within the first five characters (ANSI standard is six).

- The characters > and < cannot be used in Hollerith strings.

- All integers are of length 2 bytes
- All reals are length 4 bytes
- Data initialization statements use the null character as fill character. S Format uses the null character as fill character (A Format uses the blank character, as on the IBM 360)
- Read and write statements use unit numbers 11 and 10.

Example:

```
DATA IA/'A'/'      AO
DATA IA/'A '/      AL
FORMAT(A1)         AL
FORMAT(S1)         AO
```

For complete information on IBM Fortran, consult:

IBM System/360 FORTRAN IV Language, document #C28-6515-7

For Data General Fortran:

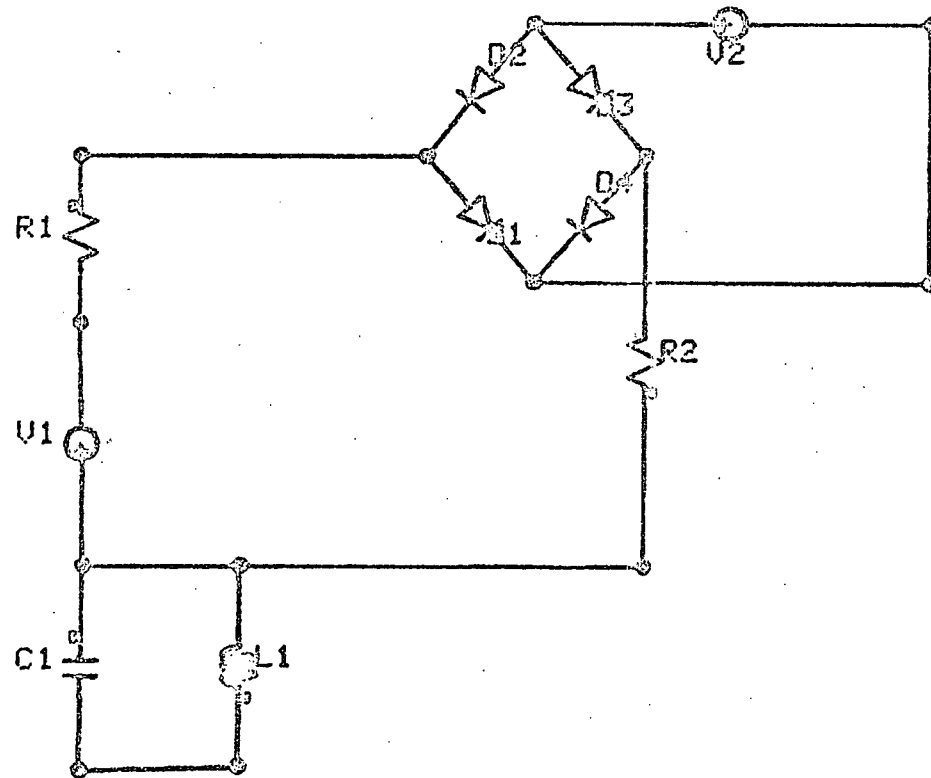
DGC FORTRAN IV USER'S MANUAL, #093-000053.

APPENDIX B

ILLUSTRATIONS FROM IGL PROGRAMS:

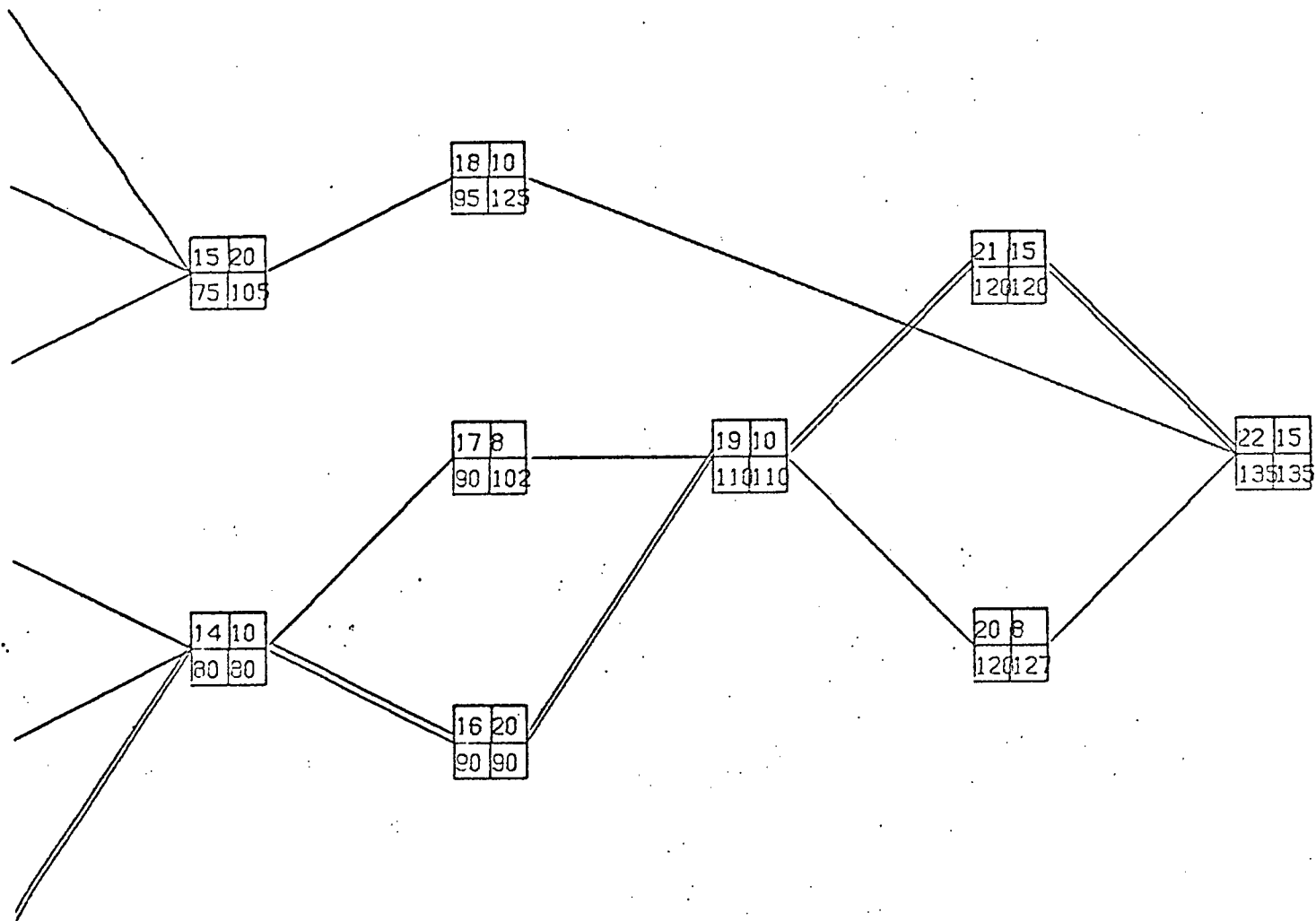
NETWORK ANALYSIS, CPM, LOAD FLOW

PICK TWO NODES AND AN ELEMENT, LAST NODE IS SAVED

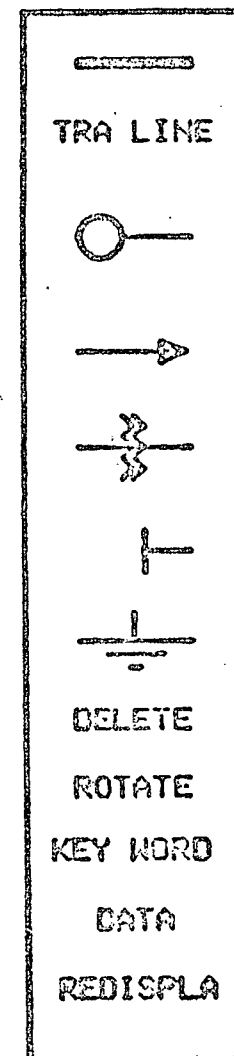
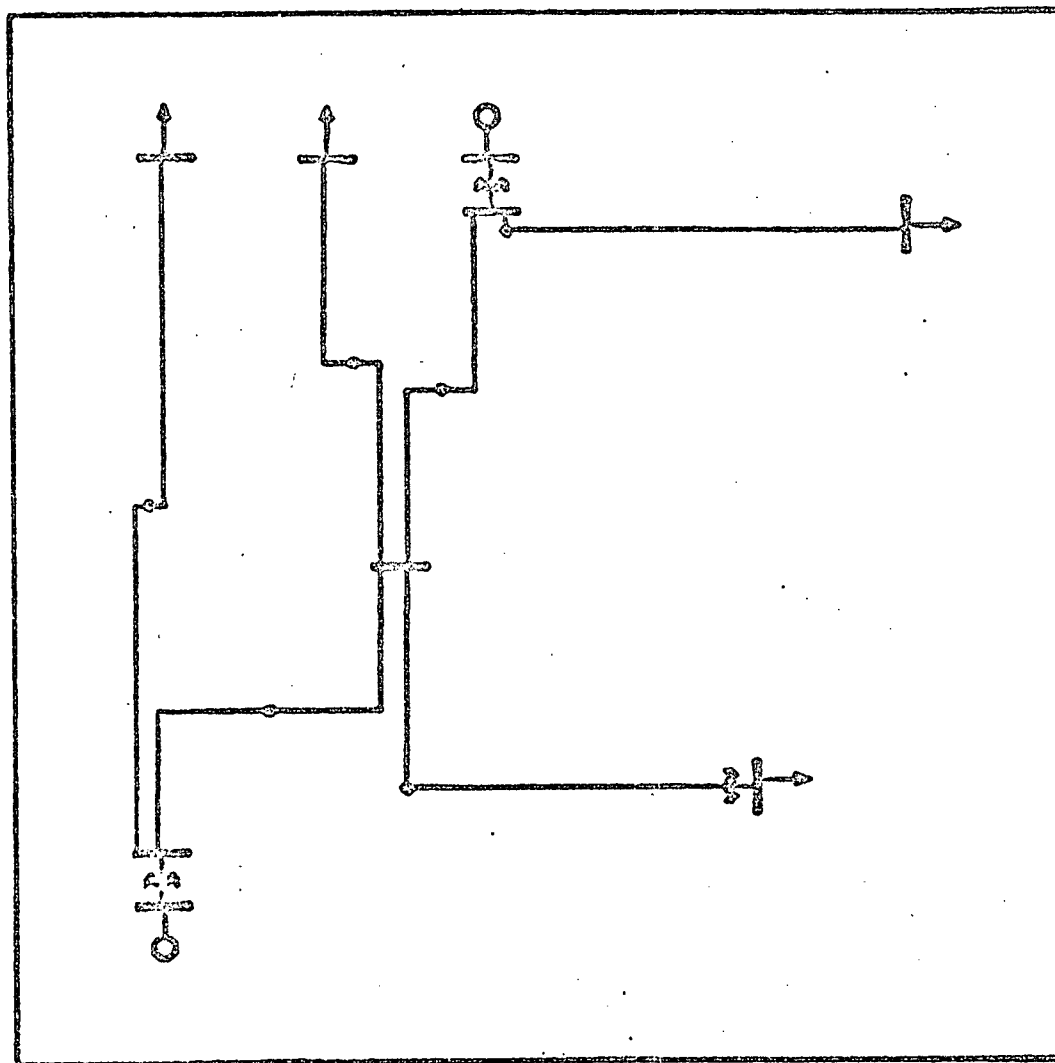


ROTATE
DELETE
LABEL
VALUES
SAVE
REDRAW
ANALYZE
PLOT
RESTART





INSERT
DELETE
ACTIVITY
RELATION
REDRAW
MOVE



APPENDIX C

LISTINGS: IGL-SYNTAX, IGLCOM

(Only routines that differ from the
published XPL description [12] are shown).

P R O D U C T I O N S

```

1  <PROGRAM> ::= <STATES>
2  <STATES> ::= <STATE>
3              | <STATES> <STATE>
4  <STATE> ::= <STATE HEAD> : <STATEMENTLIST> <STATE ENDING> ;
5  <STATE HEAD> ::= STATE <NUMBER>
6  <STATEMENTLIST> ::= <STATEMENT>
7                      | <STATEMENTLIST> <STATEMENT>
8  <STATE ENDING> ::= END STATE <NUMBER>
9  <STATEMENT> ::= <PICTORIAL ASSIGNMENT> ;
10                | <NUMERICAL ASSIGNMENT> ;
11                | <DISPLAY STATEMENT> ;
12                | <FOR CONSTRUCT> ;
13                | <GOTO STATEMENT> ;
14                | <WAIT STATEMENT> ;
15                | <TRACKING STATEMENT> ;
16                | <ERASE STATEMENT> ;
17                | <PLOT STATEMENT> ;
18                | <FILE STATEMENT> ;
19                | ;
20 <PICTORIAL ASSIGNMENT> ::= <VARIABLE> <REPLACE> <EXPRESSION>
21 <NUMERICAL ASSIGNMENT> ::= <FORTRAN VARIABLE> = <FORTRANVARIABLE>
22 <REPLACE> ::= : =
23 <DISPLAY STATEMENT> ::= DISPLAY <VIRTUAL IMAGE>
24                | DISPLAY <VIRTUAL IMAGE> ONTO <VALUEPAIR> <VALUEPAIR>
25 <VIRTUAL IMAGE> ::= <IMAGE>
26                | <IMAGE> <VALUATION>
27 <IMAGE> ::= <VARIABLE>
28                | <VARIABLE> WITHIN <VALUEPAIR> <VALUEPAIR>
29                | <STRING>
30 <FOR CONSTRUCT> ::= <FOR STATEMENT> : <STATEMENTLIST> <END STATEMENT>
31 <FOR STATEMENT> ::= FOR HIT ON <VARIABLE>
32 <END STATEMENT> ::= END

```

```

33 <GOTO STATEMENT> ::= GOTO STATE <NUMBER>
34                     | GO TO STATE <NUMBER>

35 <WAIT STATEMENT> ::= WAIT FOR INTERRUPT

36 <TRACKING STATEMENT> ::= CURSOR ON
37                     | READ CURSOR <VALUEPAIR>
38                     | READ KEY <FORTRAN VARIABLE>

39 <ERASE STATEMENT> ::= ERASE SCREEN

40 <PLOT STATEMENT> ::= PLOT <COMPONENTPAIR>

41 <FILE STATEMENT> ::= STORE <FORTRAN VARIABLE>
42                     | RESTORE <FORTRAN VARIABLE>

43 <EXPRESSION> ::= <PRIMARY>
44                     | <EXPRESSION> <DIADIC OPERATOR> <PRIMARY>

45 <DIADIC OPERATOR> ::= +
46                     | -

47 <PRIMARY> ::= <VALUATED VARIABLE>
48                     | <VALUATED VARIABLE> <UNARY OPERATOR>

49 <UNARY OPERATOR> ::= VSYM <FORTRAN VARIABLE>
50                     | HSYM <FORTRAN VARIABLE>

51 <VALUATED VARIABLE> ::= <ITEM> <VALUATION>
52                     | <ITEM>

53 <ITEM> ::= <VARIABLE>
54         | <STRING>
55         | VALUE ( <FORTRAN VARIABLE> )
56         | TEXT ( <FORTRAN VARIABLE> )

57 <VARIABLE> ::= <IDENTIFIER>
58         | <TAGGED VARIABLE>

59 <TAGGED VARIABLE> ::= <IDENTIFIER> ( <FORTRAN VARIABLE> )

60 <FORTRAN VARIABLE> ::= <IDENTIFIER>
61                     | <NUMBER>
62                     | <COORDINATE FUNCTION>

63 <COORDINATE FUNCTION> ::= <FUNCTIONHEAD> <VARIABLE> )

64 <FUNCTIONHEAD> ::= <FUNCTIONNAME> (

65 <FUNCTIONNAME> ::= XLOC
66                 | YLOC
67                 | XSCALE
68                 | YSCALE
69                 | ANGLE
70                 | TAG

71 <VALUATION> ::= <COMPONENTS>
72                 | <COMPONENTPAIR>

73 <COMPONENTS> ::= <COMPONENT>

```

```
74      | <COMPONENTS> <COMPONENT>
75 <COMPONENT> ::= AT <VALUEPAIR>
76      | SCALE <VALUEPAIR>
77      | SCALE <FORTRAN VARIABLE>
78      | ANGLE <FORTRAN VARIABLE>
79      | TAG <FORTRAN VARIABLE>
80 <COMPONENTPAIR> ::= FROM <VALUEPAIR> TO <VALUEPAIR>
81 <VALUEPAIR> ::= <FORTRAN VARIABLE> , <FORTRAN VARIABLE>
```

```

1      /*                                CARD IMAGE HANDLING PROCEDURE  $L                                *
2
3
4
5  GET_CARD:
6      PROCEDURE;
7          /* DOES ALL CARD READING AND LISTING                                */
8          DECLARE I FIXED, (TEMP, TEMPO, REST) CHARACTER, READING BIT(1);
9          DECLARE FIRST_CARD BIT(1) INITIAL (TRUE);
10         READ: BUFFER = INPUT;
11         IF LENGTH(BUFFER) = 0 THEN
12             DO; /* SIGNAL FOR EOF */
13                 CALL ERROR ('EOF MISSING OR COMMENT STARTING IN COLUMN 1,1,1)
14                 BUFFER = PAD ('* /*'/* */ EOF;END;EOF', 80);
15             END;
16
17         ELSE CARD_COUNT = CARD_COUNT + 1; /* USED TO PRINT ON LISTING */
18         /* OUTPUT FIRST LINE BEFORE FIRST CARD */
19         IF FIRST_CARD=TRUE THEN DO;
20             FIRST_CARD=FALSE; END;
21         /* ELIMINATE FORTRAN CARDS */
22         DO J=0 TO 13;
23             IF BYTE(BUFFER,0)=BYTE(' C0123456789XA',J) THEN
24                 DO;
25                     IF CONTROL(BYTE('M')) THEN OUTPUT=BUFFER;
26                     ELSE IF CONTROL(BYTE('L')) THEN
27                         OUTPUT = I_FORMAT (CARD_COUNT,4) ||| |||BUFFER;
28                         OUTPUT(2)=BUFFER;
29                         GOTO READ;
30                     END;
31                 END;
32         IF MARGIN_CHOP > 0 THEN
33             DO; /* THE MARGIN CONTROL FROM DOLLAR | */
34                 I = LENGTH(BUFFER) - MARGIN_CHOP;
35                 REST = SUBSTR(BUFFER, I);
36                 BUFFER = SUBSTR(BUFFER, 0, I);
37             END;
38         ELSE REST = '';
39         S=SUBSTR(BUFFER,0,1);
40         BUFFER=SUBSTR(BUFFER,1);
41         TEXT = BUFFER;
42         TEXT_LIMIT = LENGTH(TEXT) - 1;
43         IF CONTROL(BYTE('M')) THEN OUTPUT = S |||BUFFER;
44         ELSE IF CONTROL(BYTE('L')) THEN
45             OUTPUT = I_FORMAT (CARD_COUNT, 4) ||| ||| S ||| BUFFER ||| REST;
46         CP = 0;
47         END GET_CARD;

```



```

1  ADD_FORTTRAN_VAR:
2      PROCEDURE; /* KEEP A LIST OF FORTRAN VARIABLES USED */
3
4  END ADD_FORTTRAN_VAR;
5  RESET:
6      PROCEDURE; /* RESET VALUATION TO DEFAULT VALUES */
7          LOC='1,5,,5';
8          SCALE='1,,1,1';
9          ANGLE='10.1';
10         TAG='101';
11     RETURN;
12 END RESET;
13 PAD_8:
14     PROCEDURE; /* PAD VARIABLENAME WITH BLANKS */
15     I=LENGTH(VAR(MP));
16     IF I>8 THEN VAR(MP)=SUBSTR(VAR(MP),0,8);
17     IF I<8 THEN
18         DO;
19             S=SUBSTR(' ',I);
20             VAR(MP)=VAR(MP)||S;
21         END;
22     RETURN;
23 END PAD_8;
24 EXOUT:
25     PROCEDURE(VARTYPE,PRCODE,EXCODE);
26     DECLARE(VARTYPE,PRCODE,EXCODE) FIXED;
27     DECLARE FILEOUT LITERALLY ' OUTPUT(2) = ' CALL '||||' ;
28     DECLARE CONTINUATION LITERALLY 'OUTPUT(2)= ' *'||||';
29     IF EXCODE=3 THEN FILEOUT 'ASSIG('||VAR(MP-2)||')!';
30     DO CASE VARTYPE;
31     ;
32         FILEOUT 'PRIMA('||PRCODE||',||EXCODE||',||VAR(MP)||',||LOC||
33             ',||SCALE||',||ANGLE||',||TAG||)';
34         FILEOUT 'PRIMS('||PRCODE||',||EXCODE||',||LOC||',||VAR(MP)||')!';
35         FILEOUT 'PRIMC('||PRCODE||',||EXCODE||',||LOC||',||VAR(MP)||')!';
36         FILEOUT 'PRIMS('||PRCODE||',||EXCODE||',||LOC||',||VAR(MP)||')!';
37     END;
38     CALL RESET;
39     RETURN;
40 END EXOUT;
41 SYNTHESIZE:
42     PROCEDURE(PRODUCTION_NUMBER);
43     DECLARE PRODUCTION_NUMBER FIXED;
44     DECLARE LABEL_STACK (100) CHARACTER , LP FIXED INITIAL (0);
45     DECLARE CARDNO_STACK(100) FIXED;
46     DECLARE SIMIND FIXED INITIAL (1000);
47     DECLARE UNARY_OP CHARACTER;
48     DECLARE (NUM_VALUE,VARTYPE,FTYPE) FIXED;
49     DECLARE IM_VALUE FIXED;
50     DECLARE (EXCODE,PRCODE) FIXED;
51     DECLARE (FNAME,VARNAME,VARTAG) CHARACTER;
52     DO CASE PRODUCTION_NUMBER;
53     ; /* CASE 0 IS A DUMMY */
54     /* <PROGRAM> ::= <STATES> */
55     IF MP=2 THEN

```

```

56      DO;
57          CALL ERROR('EOF AT INVALID POINT',1);
58          CALL STACK_DUMP;
59      END;
60      ELSE DO;
61          IF LP=0 THEN DO I=1 TO LP;
62              CALL ERROR('UNMATCHED FOR-STATEMENT, LINE: ||ICARDNO_STACK(LP),1);
63          END;
64          COMPILING=FALSE;
65      END;
66      /* <STATES> ::= <STATE>      */
67      ;
68      /* <STATES> ::= <STATES> <STATE>      */
69      ;
70      /* <STATE> ::= <STATE HEAD> : <STATEMENTLIST> <STATE ENDING> ;      */
71      IF VAR(MP)≠VAR(MPP1+2) THEN CALL ERROR('LABEL ||IVAR(MP)|| AND END ||I
72          VAR(MPP1+2)|| DON'T MATCH.',0);
73      /* <STATE HEAD> ::= STATE <NUMBER>      */
74      DO;
75          IF LENGTH(VAR(SP)) = 1 THEN S='0||IVAR(SP);
76          ELSE S=VAR(SP);
77          IF CONTROL(BYTE('D')) THEN
78              OUTPUT(2)=1      SUBROUTINE STAT||IS;
79              OUTPUT(2)=1C      ----- STATE ||IVAR(MPP1)|| -----!;
80              OUTPUT(2)=1      LOGICAL HITON!;
81              OUTPUT(2)=1      COMMON IDUMMY(3376)!;
82          VAR(MP)=VAR(MPP1);
83      END;
84      /* <STATEMENTLIST> ::= <STATEMENT>      */
85      ;
86      /* <STATEMENTLIST> ::= <STATEMENTLIST> <STATEMENT>      */
87      ;
88      /* <STATE ENDING> ::= END STATE <NUMBER>      */
89      DO;
90          VAR(MP)=VAR(SP);
91          OUTPUT(2)=1C      ----- END STATE ||IVAR(SP)|| -----!;
92          IF CONTROL(BYTE('D')) THEN
93              OUTPUT(2)=1      RETURN!;
94              OUTPUT(2)=1      END!;
95          EJECT_PAGE;
96          OUTPUT=1      INTERACTIVE GRAPHICS COMPILER      (FEB 73 VERSION)!;
97          DOUBLE_SPACE;
98          CARD_COUNT=0;
99      END;
100     /* <STATEMENT> ::= <PICTORIAL ASSIGNMENT> ;      */
101     ;
102     /* <STATEMENT> ::= <NUMERICAL ASSIGNMENT> ;      */
103     ;
104     /* <STATEMENT> ::= <DISPLAY STATEMENT> ;      */
105     ;
106     /* <STATEMENT> ::= <FOR CONSTRUCT> ;      */
107     ;
108     /* <STATEMENT> ::= <GOTO STATEMENT> ;      */
109     ;
110     /* <STATEMENT> ::= <WAIT STATEMENT> ;      */
111     ;
112     /* <STATEMENT> ::= <TRACKING STATEMENT> ;      */
113     ;
114     /* <STATEMENT> ::= <ERASE STATEMENT> ;      */
115     ;

```

```

116  /* <STATEMENT> ::= <PLOT STATEMENT> ; */
117  ;
118  /* <STATEMENT> ::= <FILE STATEMENT> ; */
119  ;
120  /* <STATEMENT> ::= ; */
121  ;
122  /* <PICTORIAL ASSIGNMENT> ::= <VARIABLE> <REPLACE> <EXPRESSION> */
123  ;
124  /* <NUMERICAL ASSIGNMENT> ::= <FORTRAN VARIABLE> = <FORTRAN VARIABLE> */
125  IF BYTE(VAR(MP)) = BYTE(' ') THEN DO;
126  K=LENGTH(VAR(MP))-9;
127  VAR(MP)=SUBSTR(VAR(MP),8,K);
128  FILEOUT 'CHCHOR('||VAR(MP)||','||VAR(SP)||')';
129  END;
130  ELSE OUTPUT(2)= ' ||VAR(MP)|| = ||VAR(SP);
131  /* <REPLACE> ::= : = */
132  ;
133  /* <DISPLAY STATEMENT> ::= DISPLAY <VIRTUAL IMAGE> */
134  DO;
135  DO CASE IM_VALUE;
136  FILEOUT 'DISPL('||VAR(SP)||','||LOC||','||SCALE||','||ANGLE||')';
137  DO;
138  IF LENGTH(VAR(SP))>50 THEN VAR(SP)=SUBSTR(VAR(SP),0,50);
139  FILEOUT 'DISPS('||VAR(SP)||','||LOC||','||SCALE||','||ANGLE||')';
140  CONTINUATION LOC||','||LENGTH(VAR(SP))||')';
141  END;
142  END;
143  CALL RESET;
144  END;
145  /* <DISPLAY STATEMENT> ::= DISPLAY <VIRTUAL IMAGE> ONTO <VALUEPAIR> <VALUEPAIR> */
146  /*
147  IF IM_VALUE=0 THEN DO;
148  FILEOUT 'DISPO('||VAR(MP)||','||VAR(MP+2)||','||VAR(SP)||')';
149  END;
150  /* <VIRTUAL IMAGE> ::= <IMAGE> */
151  ;
152  /* <VIRTUAL IMAGE> ::= <IMAGE> <VALUATION> */
153  ;
154  /* <IMAGE> ::= <VARIABLE> */
155  IM_VALUE=0;
156  /* <IMAGE> ::= <VARIABLE> WITHIN <VALUEPAIR> <VALUEPAIR> */
157  DO;
158  S=VAR(SP-1)||','||VAR(SP);
159  FILEOUT 'WITHIN('||S||')';
160  IM_VALUE=0;
161  END;
162  /* <IMAGE> ::= <STRING> */
163  IM_VALUE=1;
164  /* <FOR STATEMENT> ::= FOR HIT ON <VARIABLE> */
165  DO;
166  OUTPUT(2)= '0||LABEL_STACK(LP)|| CONTINUE';
167  LP=LP-1;
168  END;
169  /* <FOR CONSTRUCT> ::= <FOR STATEMENT> ; <STATEMENTLIST> <END STATEMENT> */
170  DO;
171  LP=LP+1;
172  STMTNO=STMTNO+1;
173  LABEL_STACK(LP)=STMTNO;
174  CARDNO_STACK(LP)=CARD_COUNT;
175  OUTPUT(2)= ' IF(,NOT, HITON(' ||VAR(MPP1+2)|| ' ')) GOTO 0||

```

```

176         LABEL_STACK(LP);
177     END;
178     /* <END STATEMENT> ::= END      */
179     ;
180     /* <GOTO STATEMENT> ::= GOTO STATE <NUMBER>      */
181     DO;
182         IF LENGTH(VAR(SP)) = 1 THEN VAR(SP)='0' || VAR(SP);
183         IF CONTROL(BYTE('D')) THEN
184             OUTPUT(2)='      ' CALL STAT || VAR(SP);
185         ELSE
186             FILEOUT 'CHAIN(STAT || VAR(SP) || ',SV)';
187     END;
188     /* <GOTO STATEMENT> ::= GO TO STATE <NUMBER>      */
189     DO;
190         IF LENGTH(VAR(SP)) = 1 THEN VAR(SP)='0' || VAR(SP);
191         IF CONTROL(BYTE('D')) THEN
192             OUTPUT(2)='      ' CALL STAT || VAR(SP);
193         ELSE
194             FILEOUT 'CHAIN(STAT || VAR(SP) || ',SV)';
195     END;
196     /* <WAIT STATEMENT> ::= WAIT FOR INTERRUPT      */
197     FILEOUT 'WAITIN';
198     /* <TRACKING STATEMENT> ::= CURSOR ON      */
199     FILEOUT 'CURSOR';
200     /* <TRACKING STATEMENT> ::= READ CURSOR <FORTRAN PAIR>      */
201     FILEOUT 'RDCUR(' || VAR(SP) || ')';
202     /* <TRACKING STATEMENT> ::= READ KEY <FORTRAN VARIABLE>      */
203     FILEOUT 'RDKEY(' || VAR(SP) || ')';
204     /* <ERASE STATEMENT> ::= ERASE SCREEN      */
205     FILEOUT 'ERASE';
206     /* <PLOT STATEMENT> ::= PLOT <COMPONENTPAIR>      */
207     FILEOUT 'DRAW(' || LOC || ', ' || SCALE || ', ' || ANGLE || ')';
208     /* <FILE STATEMENT> ::= STORE <FORTRAN VARIABLE>      */
209     FILEOUT 'SAVE(' || VAR(SP) || ')';
210     /* <FILE STATEMENT> ::= RESTORE <FORTRAN VARIABLE>      */
211     FILEOUT 'RESTOR(' || VAR(SP) || ')';
212     /* <EXPRESSION> ::= <PRIMARY>      */
213     DO;
214         EXCODE=3;
215         CALL EXOUT(VARTYPE,PRCODE,EXCODE);
216     END;
217     /* <EXPRESSION> ::= <EXPRESSION> <DIADIC OPERATOR> <PRIMARY>      */
218     DO;
219         VAR(MP)=VAR(SP);
220         CALL EXOUT(VARTYPE,PRCODE,EXCODE);
221     END;
222     /* <DIADIC OPERATOR> ::= +      */
223     EXCODE=1;
224     /* <DIADIC OPERATOR> ::= -      */
225     EXCODE=2;
226     /* <PRIMARY> ::= <VALUATED VARIABLE>      */
227     PCODE=0;
228     /* <PRIMARY> ::= <VALUATED VARIABLE> <UNARY OPERATOR>      */
229     DO;
230         FILEOUT 'UNARY(' || UNARY_OP || ')';
231         PCODE=1;
232     END;
233     /* <UNARY OPERATOR> ::= VSYM <FORTRAN VARIABLE>      */
234     UNARY_OP='1, ' || VAR(SP);
235     /* <UNARY OPERATOR> ::= HSYM <FORTRAN VARIABLE>      */

```

```

236     UNARY_OP='2','1'VAR(SP);
237 /* <VALUATED VARIABLE> ::= <ITEM> <VALUATION>      */
238 ;
239 /* <VALUATED VARIABLE> ::= <ITEM>      */
240 ;
241 /* <ITEM> ::= <VARIABLE>      */
242     VARTYPE=1;
243 /* <ITEM> ::= <STRING>      */
244     DO;
245         I=LENGTH(VAR(MP));
246         IF I > 16 THEN VAR(MP)=SUBSTR(VAR(MP),0,16);
247         IF I < 16 THEN DO;
248             S=SUBSTR(' ',I);
249             VAR(MP)=VAR(MP) || S;
250         END;
251         VARTYPE=2;
252     END;
253 /* <ITEM> ::= VALUE ( <FORTRAN VARIABLE> )      */
254     DO;
255         VARTYPE=3;
256         VAR(MP)=VAR(MPP1+1);
257     END;
258 /* <ITEM> ::= TEXT ( <FORTRAN VARIABLE> )      */
259     DO;
260         VARTYPE=4;
261         VAR(MP)=VAR(MPP1+1);
262     END;
263 /* <VARIABLE> ::= <IDENTIFIER>      */
264     DO;
265         CALL PAD_8;
266         VARNAME=VAR(MP);
267         VARTAG='10';
268         VAR(MP)='''''||VARNAME||''','''||VARTAG;
269     END;
270 /* <VARIABLE> ::= <TAGGED VARIABLE>      */
271 ;
272 /* <TAGGED VARIABLE> ::= <IDENTIFIER> ( <FORTRAN VARIABLE> )      */
273     DO;
274         CALL PAD_8;
275         VARNAME=VAR(MP);
276         VARTAG=VAR(MPP1+1);
277         K=LENGTH(VARTAG)-1;
278         IF BYTE(VARTAG,K) = BYTE(' ',1) THEN
279             VARTAG=SUBSTR(VARTAG,0,K);
280         VAR(MP)='''''||VARNAME||''','''||VARTAG;
281     END;
282 /* <FORTRAN VARIABLE> ::= <IDENTIFIER>      */
283     CALL ADD_FORTRAN_VAR;
284 /* <FORTRAN VARIABLE> ::= <NUMBER>      */
285     /* CHECK FOR DECIMAL DOT IN NUMBER */
286     DO;
287         K=0;
288         DO I=0 TO LENGTH(VAR(SP))-1;
289             IF BYTE(VAR(SP),I)=BYTE('.',1) THEN K=1;
290         END;
291         IF K=0 THEN VAR(SP)=VAR(SP) || ' ',1;
292     END;
293 /* <FORTRAN VARIABLE> ::= <COORDINATE FUNCTION>      */
294     IF FTYPE = 6 THEN VAR(MP)=' ITAGFN('||VAR(MP)||')';
295     ELSE

```

```

296     VAR(MP)= ' CORFUN(' || VAR(MP) || ')';
297     /* <COORDINATE FUNCTION> ::= <FUNCTIONHEAD> <VARIABLE> ) */
298     IF FTYPE = 6 THEN VAR(MP)=VAR(MPP1); ELSE
299     VAR(MP)=FTYPE || ' ' || VAR(MPP1);
300     /* <FUNCTIONHEAD> ::= <FUNCTIONNAME> ( */
301     ;
302     /* <FUNCTIONNAME> ::= XLOC */
303     FTYPE = 1;
304     /* <FUNCTIONNAME> ::= YLOC */
305     FTYPE = 2;
306     /* <FUNCTIONNAME> ::= XSCALE */
307     FTYPE = 3;
308     /* <FUNCTIONNAME> ::= YSCALE */
309     FTYPE = 4;
310     /* <FUNCTIONNAME> ::= ANGLE */
311     FTYPE = 5;
312     /* <FUNCTIONNAME> ::= TAG */
313     FTYPE = 6;
314     /* <VALUATION> ::= <COMPONENTS> */
315     ;
316     /* <VALUATION> ::= <COMPONENTPAIR> */
317     ;
318     /* <COMPONENTS> ::= <COMPONENT> */
319     ;
320     /* <COMPONENTS> ::= <COMPONENTS> <COMPONENT> */
321     ;
322     /* <COMPONENT> ::= AT <VALUEPAIR> */
323     LOC=VAR(SP);
324     /* <COMPONENT> ::= SCALE <VALUEPAIR> */
325     SCALE=VAR(SP);
326     /* <COMPONENT> ::= SCALE <FORTRAN VARIABLE> */
327     SCALE=VAR(SP) || ' ' || VAR(SP);
328     /* <COMPONENT> ::= ANGLE <FORTRAN VARIABLE> */
329     ANGLE=VAR(SP);
330     /* <COMPONENT> ::= TAG <FORTRAN VARIABLE> */
331     DO;
332     TAG=VAR(SP);
333     K=LENGTH(TAG)-1;
334     IF BYTE(TAG,K) = BYTE(' ',1) THEN
335     TAG=SUBSTR(TAG,0,K);
336     END;
337     /* <COMPONENTPAIR> ::= FROM <VALUEPAIR> TO <VALUEPAIR> */
338     DO;
339     LOC=VAR(MPP1);
340     SCALE=VAR(SP);
341     ANGLE='1-999.1';
342     END;
343     /* <VALUEPAIR> ::= <FORTRAN VARIABLE> , <FORTRAN VARIABLE> */
344     VAR(MP)=VAR(MP) || ' ' || VAR(SP);
345     END;
346     END SYNTHESIZE;

```

APPENDIX D

Device Dependent Support Routines:

Supernova: INCOM, RETUR, OVERL, CHAIN.

Adage /10: AGTLIB (programmed by H. Rydzik).

```

;S/R TO CHANGE VALUE OF #NMAX#
;TO POINT TO THE BOTTOM OF BLANK COMMON AREA
;TO CALL :      CALL INCOM
;
      .TITL      INCOM
      .NREL
      .ENT       INCOM
      .EXTD      .CPYL,.FRET
      .EXTN      ERROR
;
      FS.=0
;
      FS.
INCOM: JSR      @.CPYL      ;
      LDA      0,CSIZE    ;GET SIZE OF BLANK COMMON AREA
      NEG      0,0        ;MAKE DECREMENT
      .SYSTEM
      .MEMI
      JMP      @.EROR
      JSR      @.FRET      ;&RETURN TO CALLING PROGRAM
;
;EROR:  ERROR
;
SIZE:  000410      ;COMMON SIZE
;
      .END

```

```

;S/R TO RETURN FROM OVERLAY SEGMENT
;TO CALL :      CALL RETUR
;
      .TITL      RETUR
      .NREL
      .ENT       RETUR,ERROR
;
RETUR: .SYSTEM
      .RTN
ERROR: .SYSTEM
      .ERTN
      JMP
;
      .END

```



```

;S/R TO OVERLAY FORTRAN PROGRAMS
;TO CALL :      CALL OVERL ('NAME OF SEGMENT')
;
      .TITL     OVERL
      .NREL
      .ENT      OVERL
      .EXTD     .CPYL,.FRET
      .EXTN     ERROR
;
      TEXT=-167
      FS.=1
;
      FS.
OVERL: JSR      @.CPYL      ;GET POINTER TO NAME
      LDA      0,TEXT,3;MAKE BYTE POINTER
      MOVZL    0,0        ;IN AC0
      SUB      1,1        ;CLEAR AC1
      .SYSTEM  ;CALL IN SEGMENT
      .EXEC
      JMP      @.EROR      ;
      JSR      @.FRET      ;RETURN TO CALLING PROGRAM
;
.EROR: ERROR          ;
;
      .END

```

```

;S/R TO CHAIN OVERLAY SEGMENT
;TO CALL :      CALL CHAIN ('NAME OF SEGMENT')
;
      .TITL     CHAIN
      .NREL
      .ENT      CHAIN
      .EXTD     .CPYL,.FRET
      .EXTN     ERROR
;
      TEXT=-167
      FS.=1
;
      FS.
CHAIN: JSR      @.CPYL      ;GET POINTER TO NAME
      LDA      0,TEXT,3;FORM BYTE POINTER IN AC0
      MOVZL    0,0        ;
      SUBZL    1,1        ;SET AC1 BIT 0
      .SYSTEM  ;
      .EXEC
      JMP      @.EROR      ;
      JSR      @.FRET      ;&RETURN TO CALLING PROGRAM
;
.EROR: ERROR          ;
;
      .END

```

```

1  C---INTERACTIVE ADAGE PACKAGE FOR PIEKE'S IGL SYSTEM---
2  C---OBJECT DECK CONTAINS $CONTINUE WITH AGT:BASIC THEREBY
3  CONCATENATION OF THAT FILE IS NOT REQUIRED,---
4  SUBROUTINE WAITIN
5  C PROVIDES THE INTERACTION - READS KEYBOARD AND CROSSHAIR LOCATION
6  LOGICAL FLAG /.TRUE./
7  INTEGER PLOT/'PLOT'//,YES/'Y' /
8  COMMON/CURSOR/IHIT,XHIT,YHIT,KEYHIT
9  REAL D(6)
10 IF (FLAG) WRITE(6,30)
11 30  FORMAT(' A KEYBOARD ENTRY OF PLOT WILL AUTOMATICALLY'//
12 *' COPY THE DISPLAY AS A PLOTFILE ON LOGICAL UNIT 9, '//'
13 *' UNSPECIFIED LOGICAL UNIT PUTS PLOTFILE IN -PLOT#, ' )
14 FLAG=,FALSE,
15 1  CALL AGTMV(5925,1,37)
16 CALL DISPLAYS READY MESSAGE ON SCREEN
17 READ(5,10)KEYHIT
18 10  FORMAT (A4)
19 CALL AGTMV(5888,1,37)
20 CALL DISPLAYS WAIT! MESSAGE ON SCREEN
21 C---A KEYBOARD ENTRY OF PLOT WILL AUTOMATICALLY
22 COPY THE DISPLAY AS A PLOTFILE ON LOGICAL UNIT 9,---
23 IF(KEYHIT.EQ.PLOT) GO TO 100
24 CALL DIALS(D)
25 XHIT=(D(4)+1,)*512./780,
26 YHIT=(D(1)+1,)*512./780,
27 RETURN
28 100 NFLAG=0
29 WRITE(6,20)
30 20  FORMAT(' ', 'IS THIS TO BE THE LAST PLOT?')
31 READ(5,21)IREPLY
32 21  FORMAT(A1)
33 IF(IREPLY.EQ.YES)NFLAG=1
34 CALL SAVE(NFLAG)
35 GO TO 1
36 END
37 C - - - - -
38 SUBROUTINE ERASE
39 C ERASES SCREEN AND INITIALIZES THE TERMINAL
40 INTEGER DISP(112)
41 LOGICAL FLAG/,FALSE,/
42 COMMON/POINTR/NV
43 IF(FLAG) GO TO 200
44 FLAG=,TRUE,
45 CALL AGTCVT(DISP(1),0,,0,,0,1)
46 CALL GENERATES AN EOF CONDITION TO KEEP STORED MESSAGES FROM BEING DISPLAYED
47 CREATE MESSAGES
48 CALL AGTEXT(-5,,7,,1,, 'WAIT!!',0,,5,DISP(2),NVD)
49 DO 100 J=33,38
50 100 DISP(J)=0
51 CALL AGTEXT(-5,,7,,1,, 'READY!',0,,5,DISP(39),NVD)
52 CREATE BLANK MESSAGE
53 DO 300 J=76,112
54 300 DISP(J)=0
55 200 CALL AGTDSP(DISP(1),112,5887,,TRUE,,0)

```

```

56      CALL AGTMOV(5888,1,37)
57      CALLS WAIT! MESSAGE
58      NV=37
59      RETURN
60      END
61      C - - - - -
62      SUBROUTINE VECTOR(X1,Y1,IDRAW)
63      C   DRAWS A VECTOR FROM PREVIOUS POSITION TO NEW X1,Y1 WITH PEN
64      C   UP OR DOWN (IDRAW=0 OR 1)
65      LOGICAL FLAG/,TRUE,/
66      INTEGER DISP
67      COMMON/POINTR/NV
68      IF (FLAG) CALL CHECK
69      FLAG=,FALSE,
70      X=X1
71      Y=Y1
72      CALL TSCALE(X,Y)
73      NV=NV+1
74      CREATE VECTOR AND SEND TO AGT
75      CALL AGTCVT(DISP,X,Y,IDRAW,0)
76      CALL AGTDSP(DISP,1,NV,,FALSE,,0)
77      RETURN
78      END
79      C - - - - -
80      SUBROUTINE DISPS(IS,X1,Y1,NC)
81      C   THIS ROUTINE DYNAMICALLY ALLOCATES SUFFICIENT STORAGE FOR DTEXT,
82      CALLS DTEXT, THEN DROPS STORAGE IMMEDIATELY.
83      EXTERNAL DTEXT
84      LOGICAL FLAG/,TRUE,/
85      INTEGER*2 IS(1)
86      IF(FLAG) CALL CHECK
87      FLAG=,FALSE,
88      N=NC*10
89      CALL GSPACE(PDISP,N*4)
90      CALL CALLER(DTEXT,PDISP,IPTR(N),IPTR(IS),IPTR(X1),IPTR(Y1),
91      1IPTR(NC))
92      CALL FSPACE(PDISP)
93      RETURN
94      END
95      C - - - - -
96      SUBROUTINE DTEXT(DISP,N,IS,X1,Y1,NC)
97      C   DISPLAYS A CHARACTER STRING OF NC CHARACTERS AT X1,Y1
98      INTEGER DISP(N),IS*2(1)
99      COMMON/POINTR/NV
100      NV=NV+1
101      X=X1
102      Y=Y1
103      CALL TSCALE(X,Y)
104      CALL AGTEXT(X,Y,78./256.,,IS,0.,NC,DISP(1),NVG)
105      IF (NVG.EQ.0) GO TO 100
106      CALL AGTDSP(DISP(1),NVG,NV,,FALSE,,0)
107      100 NV=NV+NVG-1
108      RETURN
109      END
110      C - - - - -
111      SUBROUTINE TSCALE(X,Y)
112      C   LIMITS X AND Y VALUES AND SCALES FROM IGL TO AGT
113      IF(X,LT,0,) X=0.
114      IF(X,GT,1024./780,) X=1024./780.
115      IF(Y,LT,0,) Y=0.

```

```

116         IF(Y,GT,1,) Y=1,
117         X=X*780./51.2-10,
118         Y=Y*780./51.2-10,
119         RETURN
120     END
121 C - - - - -
122     SUBROUTINE SAVE(NFLAG)
123 C   READS THE BUFFER AT THE AGT AND SENDS DISPLAYED IMAGE TO LOGICAL UNIT 9,
124     COMMON/POINTR/NV
125     CALL AGTMOV(5962,1,37)
126     CALLS BLANK MESSAGE
127     CALL AGPLOT(10.,NV)
128     CALL AGTMOV(5888,1,37)
129     CALLS WAIT! MESSAGE
130     IF(NFLAG,EQ,1) CALL PLOTND
131     RETURN
132     END
133 C - - - - -
134     SUBROUTINE CHECK
135 C   ENSURES THAT ERASE HAS BEEN CALLED BEFORE VECTOR AND DISPS TO
136 C   INITIALIZE SCREEN,
137     LOGICAL FLAG/,TRUE,/
138     IF(FLAG) CALL ERASE
139     FLAG=,FALSE,
140     RETURN
141     END

```