253

Design and Implementation of a Portable Interactive Graphics Language Interpreter

by

MARY CATHERINE NEAL
B.S. Iowa State University, 1973

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University
Manhattan, Kansas

1978

Approved by:

Linda Shapiro
Major Professor

Table of Contents

THIS BOOK CONTAINS NUMEROUS PAGES WITH THE ORIGINAL PRINTING ON THE PAGE BEING CROOKED.

THIS IS THE BEST IMAGE AVAILABLE.

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

List of Figures and Tables

ACKNOWLEDGEMENT

In anticipation of the successful conclusion of the work required to complete the Master's degree in Computer Science, I would like to thank my advisor, Dr. Linda G. Shapiro, for all the guidance and encouragement which she gave to me. The members of my committee, Dr. Tom L. Gallagher and Dr. William J. Hankley, were also helpful in their academic support and their suggestions for improvement in this project. I would also like to thank my husband, David, for his timely assistance, especially in the areas of emotional and system support.

Introduction

PIGLI (Portable Interactive Graphics Language Interpreter) is a high-level interactive graphics system. The PIGLI language allows the programmer to construct two-dimensional line drawings in terms of picture primitives, to apply transformations to existing pictures, and to display pictures and text on a variety of graphics devices. The language also includes general purpose programming commands including type declarations for real or integer variables, assignment statements, IF_THEN_ELSE statements, DO_WHILE statements, and BEGIN_END blocks. Another useful feature is the ability to access external files of PIGLI commands to perform tedious or frequently needed tasks, as well as to accept commands interactively from a console device. PIGLI also provides a general purpose debugging command.

Many of the features of PIGLI are based on concepts taken from ESP3 (Extended SNOBOL Picture Pattern Processor, Shapiro, 1974), a high-level language for two-dimensional graphics and picture pattern recognition. ESP3 was designed to allow users to accomplish two major functions:

1) creating two-dimensional line drawings and manipulating them using transformations, and
2) defining picture patterns and locating subpictures in line drawings that match these patterns.

PIGLI is an interactive implementation of the first function. In this chapter, we will briefly describe the ESP3 graphics language and introduce the PIGLI system.

Related Literature

Historically SKETCHPAD (Sutherland, 1965) was the first widely recognized general purpose graphics system. The SKETCHPAD system consists of a collection of subroutines called interactively through a menu selection process. The system allows pictures to be constructed hierarchically from other pictures and is noted for its use of a ring data structure to store picture descriptions. Kulsrud (1968), Williams (1972), and Giloi (1975) presented models for the definition of a general purpose graphics language. Kulsrud suggested that the first version of the proposed language have written commands and that it later be adjusted to accept input from graphic devices such as light pens and trackballs. The language she described was capable of picture description, manipulation, and analysis. Although it could be used to implement interactive applications programs, it was not an interactive language.

Williams described a language that provided 1) data types with related operations particularly suited to graphical applications, and 2) the ability to add new data types and operations. For example, a "point" could be a

data type, and a specially defined addition operator would operate on that data type. The language was thus highly extensible, but it was not interactive. Giloi proposed a model to be used in constructing either subroutine packages for graphic display applications or graphical extensions to existing languages. In his model, pictures were described as a hierarchy of subpictures and picture primitives. Primitives were defined as anything for which there was a hardware generator in the display processor, placing limits on the device independence of a language developed from his model. An interactive version of the model was developed by extending APL to include graphics capabilities, and a non-interactive version was developed as a FORTRAN subroutine package.

The general purpose graphics systems presented in recent years can be classified as 1) subroutine packages for graphics applications, 2) graphics extensions to existing languages, and 3) new languages possessing graphics capabilities. Graphics subroutine packages are most widely distributed, particularly by manufacturers of graphics display hardware. Some example packages are GINO-F (1975), GPGS (Caruthers, 1977), GRAF (Hurwitz, 1967), DISSPLA (1970), and EXPLOR (Knowlton, 1970). Most packages are limited to the manipulation of picture displays with few programming control or storage capabilities. Where such

abilities are available, they often serve specialized purposes as in WAVE (Robbins, 1975), a package for waveform analysis. An exception is the VIP system (Streit, 1969) where the user is able to combine the available system function subroutines into special purpose functions which can then be used in the same way that the original system functions were used.

Extensions of an existing language, such as Euler-G (Newman, 1971), IMAGE (O'Brien, 1975), APLBAGS (Bassman, 1973), APLG (Giloi, 1974), and PENCIL (van Dam, 1967), provide a programmer with graphics capabilities as well as general programming features. Euler-G has excellant data structure definition facilities. IMAGE, an extension of FORTRAN, cannot provide the data structure description capabilities that are available in Euler-G, but it has the advantage of being based on the most widely distributed host language available. APLBAGS and two versions of APLG, three extensions of APL, and PENCIL, an extension of the MULTILANG on-line programming system (Wexeblat, 1967), are truly conversational languages. GRASP, a PL/1 extension (Wallace, 1974), is a compiled language but it allows dynamic interaction. GRASP also allows the definition of models from which complex pictures can be created hierarchically. ESP3 (Shapiro, 1974), an extension of SNOBOL4, is a non-interactive language from which many of the high-level

concepts found in PIGLI are drawn. ESP3 will be described in more detail later in this section. Language extensions are found mainly in research installations.

Two complete graphics languages developed recently are METAVISU (Boullier, 1972) and GLIDE (Eastman, 1977). Both take characteristics from a base language (PL/1 and ALGOL, respectively) and add capabilities for defining, displaying, and manipulating pictures. Full languages are less widely distributed than subroutine packages or language extensions.

PIGLI is a general purpose graphics language. Its strong points include 1) interactive programming capabilities, 2) simple, but rich syntax and semantics for general purpose graphical programming, 3) hierarchical generation of pictures from a set of desirable primitives, 4) built-in functions for referencing points and values associated with pictures, 5) interactive debugging facilities for exploring (reviewing) the structure of pictures, 6) extensibility through the use of exec files, 7) output device independence, and 8) a high degree of portability.

Graphics extensions of existing languages provide the best computation and programming facilities for the user. PIGLI contains only a small number of such facilities but they were carefully chosen to provide a rich assortment of programming capabilities. PIGLI is interactive in the same

sense as the graphics languages that are extensions of conversational languages, APLBAGS, APLG, and PENCIL, but PIGLI concentrates on providing graphics capabilities with programming support rather than the opposite. The other languages, extensions, or subroutine packages mentioned are useful for writing interactive applications but are not interactive languages.

Pictures are defined hierarchically in PIGLI in a manner similar to the methods used in GRASP, GRIP, and the model described by Williams. Pictures are also constructed from picture primitives as was done in GRASP and GRIP. However PIGLI primitives more closely resemble the primitive graphic objects described by Wallace than the hardware display primitives discussed by Giloi. The hierarchical nature of the picture data structure provides the basis for interactive debugging facilities and built-in functions for retrieving (referencing) the point and value attributes of constructed pictures. These facilities are not offered in most of the languages developed recently.

In the model described by Williams and in the VIP system discussed by Streit, a premium was placed on the extensibility of data types and their related operations. Taking PIGLI primitives to belong in the category of data types and picture transformations to be the related operations, PIGLI is extensible using the provided exec file

facilities. Files of PIGLI commands which describe complex objects and operations may be constructed and stored for use later in the same PIGLI programming session or in a different session.

Output device independence is achieved in PIGLI by describing pictures to be drawn in terms of a device independent picture display command language. This method is used by most languages which possess output device independence. The portability of PIGLI relies on the portability of its base operating system and its output device independence.

ESP3

The ESP3 language (Shapiro, 1975, 1977) was described as an extension of the SNOBOL4 programming language. Some of the important graphics features of ESP3 include the PICTURE and POINT data types, picture valued expressions which evaluate to PICTUREs, built in referencing functions, and user-defined referencing capabilities. An experimental version of ESP3 was implemented under the SPITBOL (Dewar, 1971) compiler on an IBM 360/65 computer with CALCOMP plotter output. In the SPITBOL implementation, the ESP3 commands are implemented as SNOBOL4 functions. Thus a programmer may use ESP3 commands, as well as all the features of SNOBOL4, to accomplish a programming task. The

following is a brief summary of the functions and operators used in the SPITBOL implementation of ESP3.

BUILD is a function which accepts a description of a two-dimensional line drawing and constructs a data structure representing that line drawing. DRAW is a subfunction of BUILD used to define the following primitive picture components:

1) LINE a straight line,
2) CIRCLE a circle,
3) ARC a portion of a circle,
4) FIGURE a set of points connected by straight lines, and
5) CURVE a set of points connected by curved lines.

The character 'c' is a composition operator used in a BUILD command for combining more than one primitive component in a single picture. Primitive components of a picture may be associated with an identifier by using the assignment operator ':'. The following is an example of the construction of a simple picture.

```
      A_LINE_AND_A_CIRCLE = BUILD("A_LINE : DRAW("
     +    " 'LINE','START=PNT(1.0,1.0);END=PNT(4.0,5.0);')"
     +    " c A_CIRCLE:DRAW('CIRCLE','CENTER=PNT(1.0,1.0);"
     +    "RADIUS=.5;')")
```

The above statement causes the creation of a picture named A_LINE_AND_A_CIRCLE. The picture is composed of a straight line, called A_LINE and a circle called A_CIRCLE. A_LINE is defined by the position of its start and end points and A_CIRCLE by the position of its center point and the length (in inches) of its radius. ESP3 also allows several other standard ways to define these primitives.

There are three other subfunctions of BUILD for performing picture transformations; TRANSPIC for translation, TURNPIC for rotation, and SCALEPIC for scaling. The following example illustrates the use of TRANSPIC.

```
NEW_LINE_AND_A_CIRCLE = BUILD(
+   "NEW_LINE : TRANSPIC(A_LINE,'PNT(1.0,1.0)=>"
+   "PNT(1.0,1.5);') c A_CIRCLE")
```

The picture NEW_LINE_AND_A_CIRCLE is similar to A_LINE_AND_A_CIRCLE, but the line component is translated such that the point (1.0,1.0) moves to the point (1.0,1.5).

POINT and VALU are functions for returning defined points and values, respectively, from the picture data structure. A point (PNT) is a defined data type in ESP3, consisting of an x-coordinate and a y-coordinate. Examples of PNTs are the start and end of a line, and the center of a

circle. A VALU is a number or character string associated with a picture. Examples of VALUs are the radius of a circle, the length of a line, and the direction of an arc.

ADDREF is a function which allows association of user-defined reference names with subpictures, points, or values of a picture. REFERENCE is a function for retrieving the subpictures, points, or values associated with a reference name. For example, in the statements

```
ADDREF(A_LINE_AND_A_CIRCLE,'INCLINE',
       VALU(A_LINE,'ANGLE'))
SLOPE = REFERENCE ('INCLINE' #
        A_LINE_AND_A_CIRCLE)
```

the reference name INCLINE is associated in the picture A_LINE_AND_A_CIRCLE with the angle of the line A_LINE, and the variable SLOPE is assigned the (retrieved) value of the INCLINE of A_LINE_AND_A_CIRCLE. The reference operator '#' indicates the picture from which the value is to be returned. A value for INCLINE can also be retrieved for any transformations of A_LINE_AND_A_CIRCLE.

Graphic output from an ESP3 program is performed by the function PLOT which produces plotted output of a picture.

Assignment of a picture to the variable PLOTT causes the picture to be plotted. For example, the statement

```
PLOTT = NEW_LINE_AND_A_CIRCLE
```

causes the picture NEW_LINE_AND_A_CIRCLE to be plotted. In the SPITBOL implementation of ESP3, PLOT produces the corresponding plotter commands for NEW_LINE_AND_A_CIRCLE.

Motivation for the Development of PIGLI

Several features found in ESP3 are very desirable in a general purpose graphics language. The language has excellent general programming constructs, a benefit derived from the fact that ESP3 is embedded in SNOBOL4. It also has easily defined primitive units for picture construction and powerful combination facilities. A third major feature is the inclusion of the built-in reference functions POINT and VALU which are especially useful for constructing picture segments relative to other picture segments. Finally, the user defined reference functions ADDREF and REFERENCE give an even greater flexibility to the programmer.

There are other features of ESP3 that are a hindrance to the user of a graphics language, primarily the fact that the existing implementation of ESP3 is a batch processing language with graphical output limited to a pen and ink

plotter. In this situation the user is required either to produce extremely accurate programs in terms of picture construction or to be prepared to submit the program, with its required adjustments, a number of times. The amount of time needed to correct the program might be very large, particularly if there are limits on the availability of the plotter. Because of these drawbacks, the emphasis in the ESP3 language is on picture data structure construction rather than on picture display or picture manipulation.

Another drawback to the current implementation of ESP3 is that it does not provide any convenient means of interrogating the data structure. There is no method for reviewing information that has already been stored in the data structure representing a picture, a capability that would be very useful as a debugging tool in an interactive implementation. Nor is there a method of saving the data structure for future use except by reconstructing it from the original ESP3 program. A final drawback to the SPITBOL implementation of ESP3 is the very complex syntax of its commands which is a result of the SNOBOL4 embedding process.

PIGLI

This paper will present a graphics language designed to correct the inadequacies of ESP3 while retaining the desirable features. The chief difference between ESP3 and

PIGLI is that PIGLI is interactive. It is implemented on an Interdata 8/32 minicomputer as an interpreter written in sequential PASCAL. PIGLI is a stand-alone language and includes a much smaller set of programming constructs than ESP3 embedded in SNOBOL4. The language constructs included are:

1) two declaration statements for real and integer variables,
2) an arithmetic assignment statement,
3) an if-then-else statement,
4) a while-do statement, and
5) a begin-end block.

This small number of constructs provides a fairly powerful base for picture manipulation programs.

The graphics commands included in PIGLI can be divided roughly into two groups, data structure creation commands and display commands. The primary data structure creation command is the BUILD command which corresponds very closely to the BUILD function of ESP3. There are also five commands in the picture display category:

1) the SETSCREEN command which controls clipping and viewport boundaries,
2) the DRAW command for picture display,
3) the ERASE command for picture removal,

4) the HTEXT command for horizontal display of text material, and
5) the VTEXT command for vertical display of text material.

Five utility commands are included in PIGLI as special help for the interactive user. The LIST command is a general query command for reviewing information stored in the picture data structure. The LOAD command is used for dynamically activating output device drivers. The EXECUTE command allows the user to execute previously defined exec files containing sets of PIGLI commands. The LOGON and LOGOFF commands establish a log file of all interactive commands for future use.

The remainder of this paper will describe the PIGLI system in detail. Chapter two contains a description of the language. Chapter three discusses the operating system environment under which PIGLI is implemented. Chapter four discusses the process of implementing PIGLI. Chapter five explains the structure of the various output device drivers that are available in the system.

PIGLI Command Language Description

This chapter describes the PIGLI command language. The first section provides a general overview of the language with short explanations of the purpose of each command. The second section gives an example of a PIGLI terminal session. Appendix A contains a discussion of the precise syntax and semantics of the PIGLI language. The notation used to describe commands in this chapter is similar to that used in the syntax appendix.

A user of PIGLI communicates with the computer by issuing commands from a console device. The commands are processed by the computer, providing feedback to the user in most cases. Information requested by the user and status and error messages are displayed on the console device. Picture displays are produced on a graphics device that is separate from the console device.

A PIGLI command is an ordered sequence of identifiers, keywords, numeric values, and strings, punctuated with special symbols, and terminated by a period. Commands fall into four categories; programming commands, picture construction commands, picture display commands, and special utility commands.

Programming Commands

There are six commands that provide programming

capabilities for the PIGLI user. Arithmetic variable identifiers must be declared before they may be used. The two commands provided for this purpose are

REAL <variable list> and
INTEGER <variable list>

where <variable list> is a list of identifiers separated by commas. Identifiers for pictures and named picture components do not need to be declared.

Example: REAL A.
INTEGER M, MA, MB.

The assignment command has the form

<variable> := <expression>

where <variable> is an identifier and <expression> is an arithmetic expression consisting of identifiers, numeric values, and operators. An identifier used in an assignment command must have been previously declared and, if used in an expression, it must have a defined value. All the elements of the expression must agree in type (real or integer). Legal operations are addition, subtraction, multiplication, and division. Integer division may result in a truncated value or a remainder value. Explicit conversion functions are provided for situations that require type mixing. Multiplication and division operations

take precedence over addition and subtraction operations.

Example: M := (MA + MB) * TRUNC(A).

PIGLI provides two control constructs. The first is an IF-THEN-ELSE command. The general form is

IF <boolean expression> THEN <command> ELSE <command>.

The boolean expression is made up of one or more logical expressions and the boolean operators AND, OR, and NOT. A logical expression is made up of two numeric expressions of the same type (real or integer) compared by one of the logical operators <, >, =, <=, >=, <> (not equal). The commands following THEN and ELSE may be any PIGLI command except declaration commands. A special utility command NULL is provided for branches of the IF-THEN-ELSE command where no operation is desired.

Example: IF MA < MB THEN NULL ELSE M := MB.

The WHILE-DO command provides looping capabilities for the programmer. The general form is

WHILE <boolean expression> DO <command>.

The commands that are allowed in the branches of the IF-THEN-ELSE command are also allowed in the WHILE-DO command.

Example: WHILE MA < MB DO MA := MA + 1.

Because it is desirable to be able to execute several commands in either branch of the IF-THEN-ELSE command or in the body of the WHILE-DO loop, PIGLI includes a compound statement of the form

```
BEGIN
    <command>;
        .
        .
        .
    <command>;
END.
```

The entire block of one or more commands is treated as a single command by the PIGLI interpreter.

```
Example:  WHILE MA < MB DO
              BEGIN
                 M := MB * TRUNC(A);
                 MA := MA + 1;
              END.
```

Picture Construction Commands

The single most important picture construction command is the BUILD command. The BUILD command is used to define a picture. Execution of the BUILD command does not create or alter any graphical output. The interpreter converts a picture definition to a standard format and retains the information in a picture data structure.

Pictures are defined by combining primitive components,

previously defined pictures, and transformations of previously defined pictures. There are five types of picture primitives:

1. LINE a straight line,
2. CIRCLE a circle,
3. ARC a portion of a circle,
4. FIGURE a set of points connected by straight lines, and
5. CURVE a set of points connected by curved lines.

Each of these primitives has a finite set of attributes. For example, the start point, the midpoint, and the end point are attributes of a LINE. Figure 2.1 shows all the specification attributes for each type of picture primitive. To define a primitive, a PIGLI programmer states the values to be assigned to certain of the attributes. In the straight line example, a line could be defined by the start point and the end point, the start point and the midpoint, or the midpoint and the end point. The line could also be defined by the values of all three attributes, but the third point would be redundant at best and possibly not consistent with the other two points. For these reasons only the minimum amount of information necessary to define a primitive is accepted. All the allowable definitions of the five picture primitives are given in the syntax and

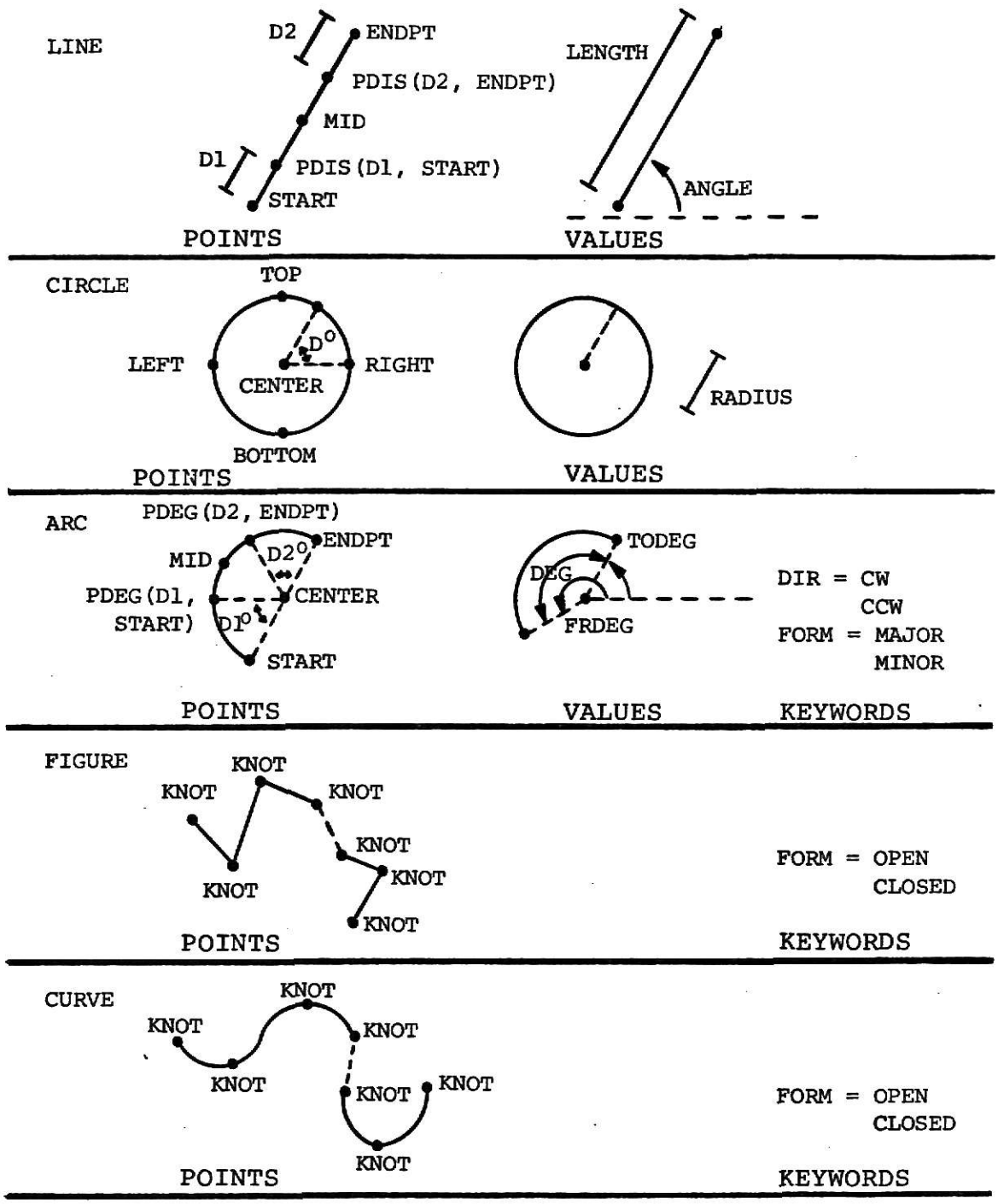


Figure 2.1: The Specification Attributes for Picture Primitives

semantics appendix.

Attributes may take on values of three different types, point values, numeric values, and keyword values. Some points are fixed with regard to the picture primitive. For example, the MID point of a LINE is always equidistant from the two end points, and the TOP point of a CIRCLE is the point on the circle with the largest y coordinate value. Other points are defined relative to the fixed points. For example, for a CIRCLE, PDEG(D) refers to a point on the CIRCLE a distance of D degrees in the counter clockwise direction from the RIGHT point of the CIRCLE, and for a LINE, PDIS(START,L) is a point on the LINE a distance of L units from the START point. Thus, a LINE may be described by its ENDPT point and a point that is known to be a specific distance from the START point.

Numeric values, such as the LENGTH of a LINE or the RADIUS of a CIRCLE or an ARC, are also used to define picture primitives. Keyword values may be used in the definition of ARCs, CURVEs, and FIGUREs. For example, the DIRection of an ARC can be specified as CCW (counter clockwise) or CW (clockwise). The FORM of FIGUREs and CURVEs may be either OPEN or CLOSED.

Examples:

```
BUILD ALINE:=LINE(START=PNT(1,1),ENDPT=PNT(5,2)).

BUILD ACIRCLE := CIRCLE(CENTER=PNT(3,3),RADIUS=1).

BUILD AN_ARC := ARC(START=PNT(5,8),RADIUS=1,
     ENDPT=PNT(3,5),DIR=CW,FORM=MINOR).

BUILD A_FIGURE := FIGURE(KNOTS(PNT(1,1),PNT(2,5),
     PNT(3,3),PNT(5,8),PNT(3,5)),
     FORM=CLOSED).

BUILD A_CURVE := CURVE(KNOTS(PNT(1,1),PNT(2,5),
     PNT(3,3),PNT(5,8),PNT(3,5)),
     FORM=OPEN).
```

Previously defined pictures, used as components of other pictures, are referred to by the identifier to which they were assigned when initially defined. Previously defined pictures can be used as the bases for transformations. There are three picture transformations:

1. TRANS to translate a picture,
2. TURN to rotate a picture, and
3. SCALE to increase or decrease the size of a picture.

Each transformation is defined by specifying the base picture plus certain other information. For TRANS the necessary information includes the coordinates of a point in the base picture and the new coordinates of the same point in the translated picture. For TURN, the required information includes the coordinates of a point about which to rotate the picture, the number of degrees of rotation

desired, and the direction (clockwise or counter clockwise) of the rotation. For SCALE, the point about which the scaling is to be done, and the factor by which the picture should be scaled must be supplied. The factor is greater than one for increasing size and less than one for decreasing.

Examples:

```
BUILD NEWLINE := TRANS(ALINE,PNT(4,.5)=>PNT(4,1)).

BUILD NEWCIRCLE := SCALE(ACIRCLE,FACTOR=2,
     ABOUT=PNT(3,3)).

BUILD NEWARC := TURN(AN_ARC,DEG=15,DIR=CCW,
     ABOUT=PNT(3,5)).
```

Pictures may be composed of several primitive and non-primitive elements. The BUILD command can be used to construct a picture made up of several components. The components specified in the BUILD command are separated by the operator '&' indicating composition. When several primitive elements or transformations are defined within one BUILD command, each element may be assigned a distinct name to allow later access to that element. This is done by preceding the definition of the primitive or the transformation by a unique identifier followed by the immediate assignment operator ':'. If an element is not named, it cannot be referenced individually at a later time.

The general form of the BUILD command is.

```
BUILD <picture identifier> := <picture element>
        { & <picture element> }.
```

Picture elements may be named or unnamed picture primitives, named or unnamed transformations, or the identifiers of previously defined pictures.

Example:

```
BUILD BARBELL := ACIRCLE &
     LINE(START=PNT(3,3),ENDPT=PNT(7,3)) &
     TRANS(ACIRCLE,PNT(3,3)=>PNT(7,3)).
```

To change the definition of a picture it is only necessary to rebuild that picture. However, unless the old definition of the picture is included in the new definition, the old definition will be lost. Pictures that were based on the original picture are based on the new picture after it is defined. Care should be taken to avoid redefining a picture in terms of another picture that is based on the first. This could create a circular definition that is impossible to draw. The following example assumes that a picture called SPOTS has been constructed. In this BUILD command, the picture SPOTS is translated from point (2,1) to point (3,1). Then the original picture SPOTS is composed with the translated picture, which is left unnamed, to become the new picture SPOTS.

Example:

```
BUILD SPOTS := TRANS(SPOTS,PNT(2,1)=>PNT(3,1)) &
     SPOTS.
```

Deleting picture descriptions after they are no longer wanted is done using the DELETE command, which has the form

```
DELETE <picture identifier>.
```

Deleting a picture removes the definition of that picture from the picture data structure where it was stored. This command does not initiate any checking to prevent removing pictures on which other pictures are based.

Picture Display Commands

There are five PIGLI commands for manipulating a graphics display device. Four are for displaying text and pictures. One is for specifying clipping and viewport boundaries.

HTEXT and VTEXT display character strings horizontally and vertically, respectively. The general form of the HTEXT and VTEXT commands is

```
HTEXT
      <orientation point> , <text list>.
VTEXT
```

The orientation point indicates the position of the first character of the text to be displayed. Following the orientation point are one or more text items. Each item may be a character string enclosed in single quotes or an integer or real valued expression.

Example:

```
HTEXT PNT(3,.5),'EXAMPLES OF ',5,' PRIMITIVES'.
```

The DRAW and ERASE commands are used for displaying and erasing pictures, respectively. The form of these commands is

```
DRAW
      <picture identifier>.
ERASE
```

DRAW causes a two-dimensional line drawing that represents the picture to be transmitted to the display device. A sequence of DRAW commands may be used to cause several pictures to be displayed together. For display devices with a selective erase feature, ERASE removes the portions of the line drawing display corresponding to the picture named in the command. The command

```
ERASE SCREEN
```

clears the entire display.

The fifth picture display command is used for specifying viewports and clipping boundaries. The SETSCREEN command sets display conditions for all DRAW and ERASE commands until those conditions are changed by another SETSCREEN command. The general form of the command is

SETSCREEN <orientation point>, XLEN=<expression>,
YLEN=<expression>.

The orientation point is the picture point that will be mapped to the lower lefthand corner of the output device when a picture is displayed. The XLEN and YLEN values establish clipping boundaries based on the orientation point. If the orientation point has the coordinates (A,B), then all displayed points will be clipped between A and A + XLEN for x values, and between B and B + YLEN for y values. As an example, consider the triangle in Figure 2.2-a. If the triangle is drawn after issuing the command

SETSCREEN PNT(2,1), XLEN=8, YLEN=8.

then the entire triangle is drawn with the lower left corner of the triangle at the origin of the display screen, (Figure 2.2-b). If it is drawn after issuing the command

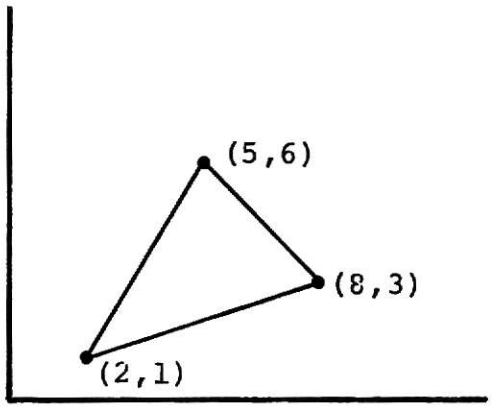
SETSCREEN PNT(3,1), XLEN=4, YLEN=4.

then only portions of the three sides are visible on the display screen, (Figure 2.2-c).

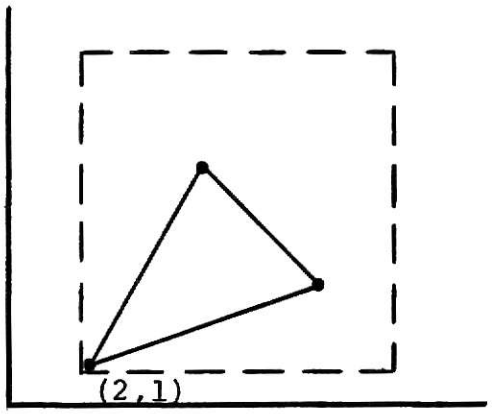
Special Utility Commands

PIGLI has several commands which fall into the category of utility commands.

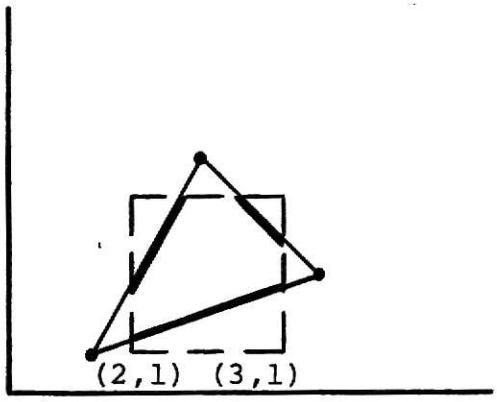
The HALT command is used to terminate the PIGLI interpreter at the end of a programming session. This command is implemented primarily to insure that all the



a. Triangle in the real world



b. Viewport after SETSCREEN PNT(2,1), XLEN = 8, YLEN = 8



c. Viewport after SETSCREEN PNT(3,1), XLEN = 4, YLEN = 4

Figure 2.2: The Effects of Changes of the SETSCREEN Parameters on Picture Displays

actions are taken for normally terminating a program under the SOLO operating system. The form of the command is simply

HALT.

The command

NULL

mentioned earlier in conjunction with the IF_THEN_ELSE command, is used in branches of that command where no operation is desired.

The LOAD command is used to replace the display device driver that is currently executing with another device driver. Each implemented device driver is an independent program available to the SOLO operating system. The general form is

LOAD <device driver identifier>.

The LOAD command closes and unloads the running device driver and loads and initializes the device driver specified. (Chapter five will discuss individual device drivers.)

One of the desirable features of an interactive language that were discussed in chapter one was the ability to query

the picture data structure to recall information processed earlier in the programming session. The LIST command is used for retrieving this information. There are several variations of LIST designed to retrieve different types of desired information. The simple command

LIST

retrieves and displays on the programming console all the symbol table entries which are the names of pictures. The command

LIST SCREEN

retrieves the identifiers that name the pictures that are currently visible on the graphics display device. The command

LIST <primitive or transform>

where primitive or transform may be LINE, CIRCLE, ARC, FIGURE, CURVE, TRANS, TURN, or SCALE, returns the identifiers of all the picture elements of the desired type. The command

LIST DEF <picture identifier>

retrieves information about the definition of the picture named by the picture identifier. If the picture identifier

names a picture of more than one component, then the information returned is a list of the names of the components of that picture. If the picture identifier names a primitive or a transformation, the information listed is the definition of that element. Using the LIST commands, the programmer may retrace the structure of any picture that has been defined.

Another feature that is desirable in an interactive language is the ability to construct and access files of commands which perform tasks that are needed frequently or tasks that are tedious. For special implementations, these exec files can also be used to increase the number of picture primitives. To create an exec file from an ongoing PIGLI programming session, the command

LOGON <file identifier>

is used. This command causes all subsequent commands to be recorded in an external file. To stop the logging of PIGLI commands the command

LOGOFF

is used. The LOGON command loads and opens an existing file with the desired file name and the LOGOFF command closes it. Only one LOGON command may be in effect at a time.

To access exec files, PIGLI uses the EXECUTE command which has the form

EXECUTE <file identifier>.

This command causes the PIGLI interpreter to stop accepting command input from the console device and instead accept commands from the external file named by the file identifier. Commands will be accepted from the exec file until the file is exhausted. All the commands in the exec file are executed exactly as if they were issued interactively at the programming console. However, the EXECUTE command is not allowed in an exec file.

A Sample Terminal Session

In this example there is a simple exec file called TRIPIC which is composed of the following PIGLI command.

```
BUILD TRI = QL1 : LINE (START = PNT
    (TRITOPX,TRITOPY), ANGLE = 240, LENGTH =
    TRILENGTH) & QL2 : LINE (START = POINT (QL1,
    ENDPT), ANGLE = 0, LENGTH = TRILENGTH) & QL3 :
    LINE(START = POINT (QL2, ENDPT), ENDPT = POINT
    (QL1, START)).
```

The user is aware of this exec file and proceeds to interactively execute the following PIGLI program. (Responses to the user from the PIGLI system are marked by asterisks.)

```
PIGLI
REAL TRITOPX, TRITOPY, TRILENGTH, CENTX, CENTY.
TRITOPX := 3.0.  TRITOPY := 3.0.
```

```
TRILENGTH := 2.0.
EXECUTE TRIPIC.
BUILD THISTRI := TRI.
CENTX := TRITOPX.
CENTY := (1.0 / 1.73) + YVAL(QL1, ENDPT).
BUILD BIGTRI := SCALE (THISTRI, ABOUT = PNT
    (CENTX, CENTY), FACTOR = 20.0).
BUILD SMALLTRI := SCALE (THISTRI, ABOUT = PNT
    (CENTX, CENTY), FACTOR = 0.5).
BUILD TRIS := SMALLTRI & BIGTRI.
DRAW TRIS.
"NOTE THAT THE PICTURE EXTENDED OFF THE SCREEN!"
LIST SCREEN "TO SEE WHAT WAS DRAWN" .
*TRIS          PICTURE
LIST DEF TRIS "TO GET ITS DEFINITION" .
*NAME        TYPE    DEFINITION
*SMALLTRI    SCALE
*BIGTRI      SCALE
LIST DEF BIGTRI.
*NAME        TYPE    DEFINITION
*            SCALE   BASE=TRIS,
*                    ABOUT=PNT(3.0,1.85),
*                    FACTOR=20.0
"NOTE THE FACTOR WAS TOO BIG"
BUILD BIGTRI := SCALE (THISTRI, ABOUT = PNT(3.0,
    1.85), FACTOR = 2.0).
ERASE SCREEN.
DRAW TRIS.
"THIS IS THE RIGHT PICTURE" ERASE SCREEN.
INTEGER I. I := 0 .
WHILE I < 5 DO
  BEGIN
    BUILD TEMPTRIS := TURN (TRIS, ABOUT = PNT
        (CENTX, CENTY), DEG = I * 72);
    DRAW TEMPTRIS;
    I := I + 1;
  END.
LOAD PLOTTER.  I := 0 .
WHILE I < 5 DO
  BEGIN
    BUILD TEMPTRIS := TURN (TRIS, ABOUT = PNT
        (CENTX, CENTY), DEG = I * 72;
    DRAW TEMPTRIS;
    I := I + 1;
  END.
HALT.
*NORMAL TERMINATION
```

In the preceding PIGLI terminal session the user

declared and initialized the information needed by the exec file and invoked it in the first four lines of the program. Then the picture produced by the exec file was stored in the picture THISTRI. In the next five commands, the user calculated the center of THISTRI, built two transformed pictures from the original, and composed those pictures in the picture TRIS. When TRIS was drawn the user noted that the image was too large for the display screen and therefore used the LIST debugging facilities to determine exactly what should be corrected. BIGTRI was rebuilt and, after erasing the display screen, TRIS was redrawn with no distortion. The user then attempted to draw multiple copies of the original with each copy being a unique rotation of TRIS. Notice the use of a BEGIN_END block to allow the presence of more than one statement in the WHILE_DO command. After interactively constructing the picture, the user loaded a new output device driver called PLOTTER and repeated the WHILE_DO command to produce a hard copy of the picture. Figure 2.3 is a representation of the plot produced by the sample terminal session.

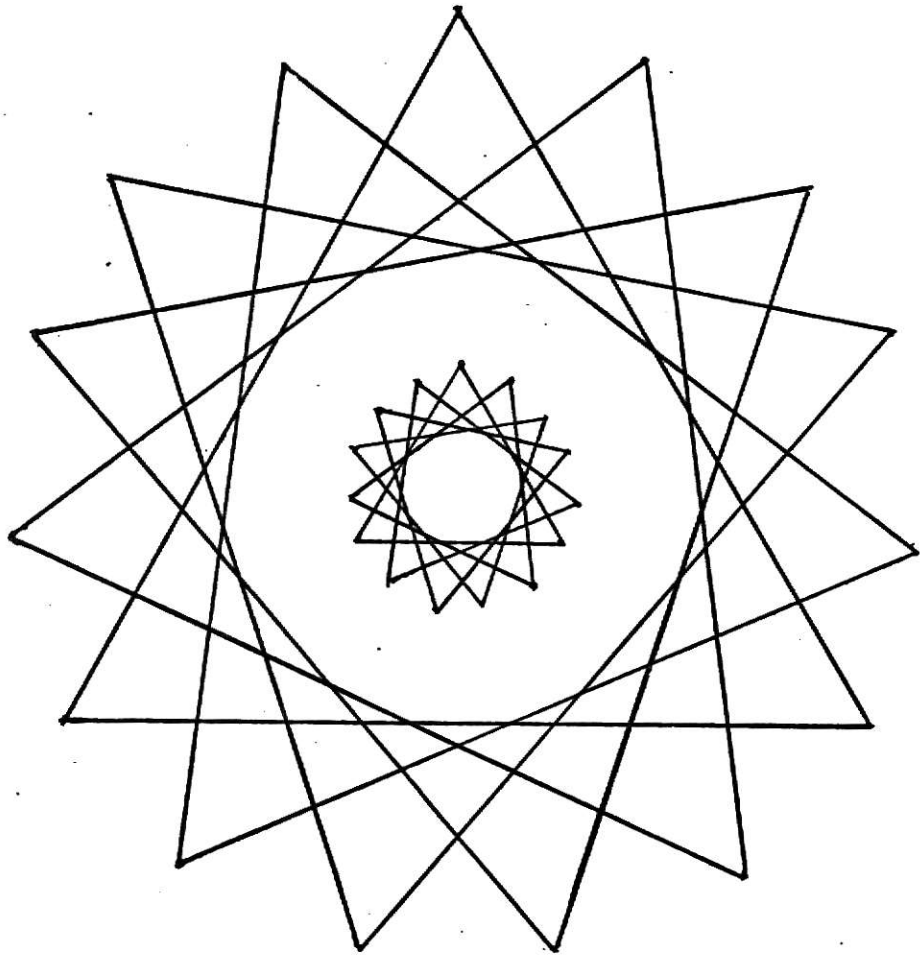


Figure 2.3: Drawing of the Plotted Output Produced by the Sample Terminal Session

Operating System Considerations

The PIGLI interpreter must accomplish two major tasks. It must accept user input commands and perform the actions specified in those commands. Accepting commands includes receiving command input, scanning the input for tokens, and parsing the command. Performing commands involves several different types of execution including expression evaluation, data storage and retrieval, and picture and text display. PIGLI is implemented in sequential PASCAL under the SOLO operating system which provides the support necessary to accomplish these tasks.

Three aspects of SOLO are particularly helpful in implementing an interpreter.

1) The system was designed to be portable.
2) The virtual machine environment provided by the system promotes modular design.
3) The system has flexible input and output procedures and inter-module communication.

The SOLO System

The SOLO operating system was designed to be an easily portable system for compiler writing (Brinch Hansen, 1976). It was developed and written in Concurrent PASCAL on a PDP 11/45 at the California Institute of Technology by Per Brinch Hansen. Its portability has been shown to be

possible and relatively simple, even to machines with very different architectures (Neal, 1977). The system is currently running on an Interdata 8/32 minicomputer at Kansas State University. It supports several online users, providing each user with a virtual machine in which to work. By implementing PIGLI under a portable operating system, it is hoped that the task of porting PIGLI will be minimal.

Under SOLO, each user's virtual machine is composed of three process partitions; one for a job process and two smaller ones for input and output processes. (See Figure 3.1) The input and output processes are designed to handle input and output data manipulation such as blocking and unblocking, while the job process is used for relatively complex processing. The SOLO system loads a user's task in the job process partition and general input and output tasks in the input and output process partitions. Thereafter the job process may load other tasks in the input and output partitions or invoke other tasks to run in the job process partition in its place.

Tasks running in any of the three partitions provided by SOLO must be written in PASCAL and begin with a standard prefix. The prefix contains constant definitions, type definitions, and interface routine definitions. These definitions are the basis for interface routine reference checking that is done by the compiler. The interface

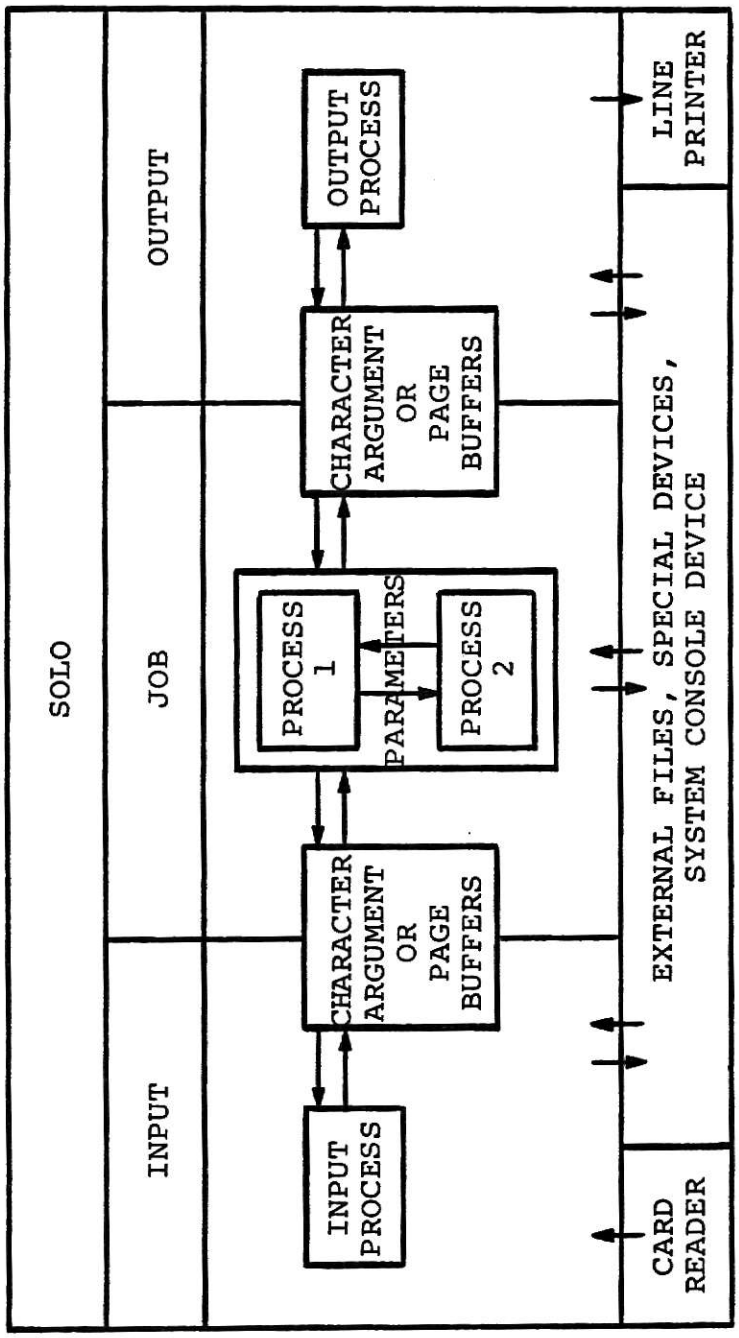


Figure 3.1: The SOLO Environment with Information Passing Channels

routines are procedures available to the user tasks which allow the task to communicate with tasks in other partitions and with external files and devices. For example, the job process may receive information from and send information to both the input process and the output process. This information may be passed one byte (a character), twelve bytes (an argument), or 512 bytes (a page) at a time. The input process may receive data from a cardreader and similarly the output process may send data to a printer. All processes can communicate with external files, special devices such as graphics devices, and the user's virtual machine console device.

GRAPHSOLO

To adapt the operating system to the requirements of the PIGLI system, three changes were made to SOLO. Normally, the input and output process partitions are the same size and fairly small. For GRAPHSOLO, the size of the input process partition was increased. Due to this change, the scanner module, which is solely responsible for command input, fits completely in the input process partition. The second change is in the combined size of all three process partitions. Normally, SOLO runs in 82K bytes of memory (under OS 32 MT on the Interdata 8/32). However, to accommodate the larger input process partition and to allow

for sufficient data space for internal command storage and picture storage it is necessary to run GRAPHSOLO in 110K bytes of memory. Finally, the standard SOLO prefix has been expanded to include constant definitions and type definitions for the data structures shared by the parser and the executor. (These shared data structures will be discussed more fully in chapter four.) Adding these definitions to the prefix made passing the shared information between the two modules much simpler. The expanded version of the prefix is given in Appendix B.

The PIGLI system is divided into four modules; the scanner, the parser, the executor, and the output device driver. It resides in the GRAPHSOLO operating system environment as shown in Figure 3.2. The scanner portion of the interpreter is placed in the input partition. The scanner uses system procedures to accept commands either from an external exec file or from the console device, to record commands in an external log file, and to echo exec file commands on the console device if desired. When tokens are isolated, the scanner passes them to the job process using the argument buffer.

Both the parser and the executor reside in the job process partition, although both do not necessarily occupy that partition at one time. The parser receives its token input from the scanner via the argument buffer. If a

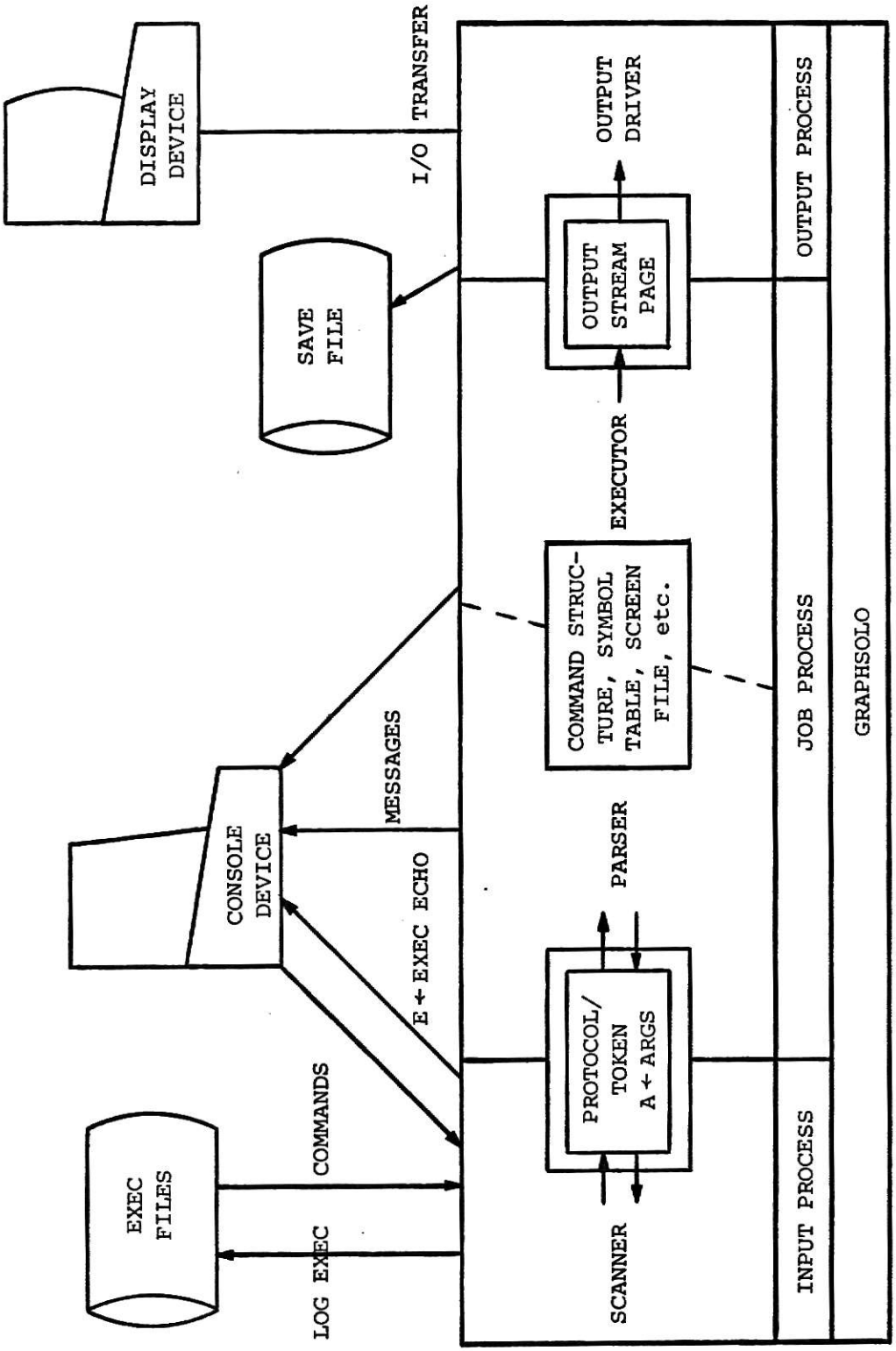


Figure 3.2: PIGLI System under GRAPHSOLO Showing Peripherals

command is completely parsed, error free, then the parser calls the executor to run in the job process partition using the RUN procedure and parameter passing capabilities. For implementations with ample machine size both the parser and the executor may reside in the job process partition at the same time, but for smaller systems these two procedures may be overlayed.

When either the parser or the executor encounters an error, or when a command is completed, a message is displayed on the console device using SOLO system procedures. When a picture display command is executed, the executor produces a sequence of device independent output commands which are sent to the output process partition using the page buffer. The output process partition contains a device dependent output device driver to translate the display commands into graphic display commands for a particular device.

Splitting the PIGLI interpreter into three partitions provides benefits in flexibility and portability. The modularity of the interpreter will aid in future expansion and revision by reducing the impact on static modules of changes in updated modules. For instance, the scanner may be expanded to allow different command sources without affecting the parser. Additional commands and graphics

capabilities may be added to the parser and the executor, such as three-dimensional and perspective transformations, without making major changes to the input or output processes. It is especially easy to adapt the output process to different graphics devices. These considerations, added to the relative ease with which the SOLO operating may be ported, makes PIGLI a highly portable system.

Implementation Considerations

The two basic functions of the PIGLI interpreter are 1) to accept user input commands and 2) to perform the actions specified by those commands. These tasks are accomplished by three modules: the parser, the scanner, and the executor. The parser parses commands using a top-down, recursive approach, and requests the scanner to provide tokens as required. The executor executes commands, calling on the display device driver when necessary.

When the PIGLI interpreter is initiated, the parser is loaded in the job process partition. (See Figure 4.1) The parser initializes the scanner and the Computek device driver in the input process partition and the output process partition, respectively. These two modules are said to run concurrently with the parser, although protocol in the interpreter prevents the parser and the scanner from executing concurrently most of the time.

When the parser has parsed a syntactically correct command, it invokes the executor to run in the job process partition in place of the parser. After the executor has finished executing the command, control of the job process partition returns to the parser. The parser is then ready to begin parsing another command.

In this chapter, the implementation of the interpreter will be discussed. Each of the major modules will be

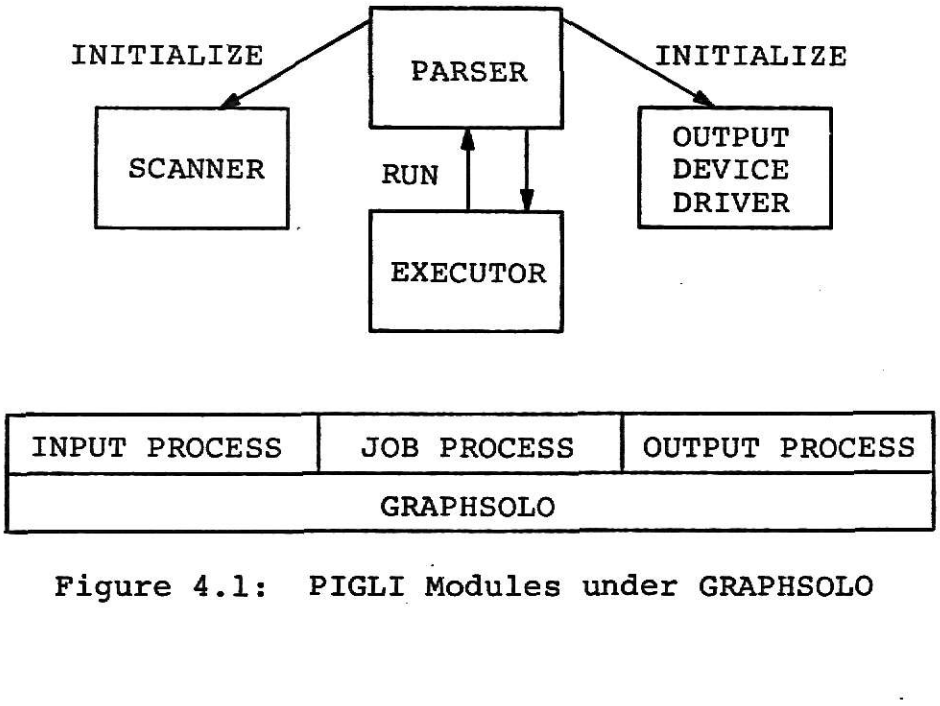


Figure 4.1: PIGLI Modules under GRAPHSOLO

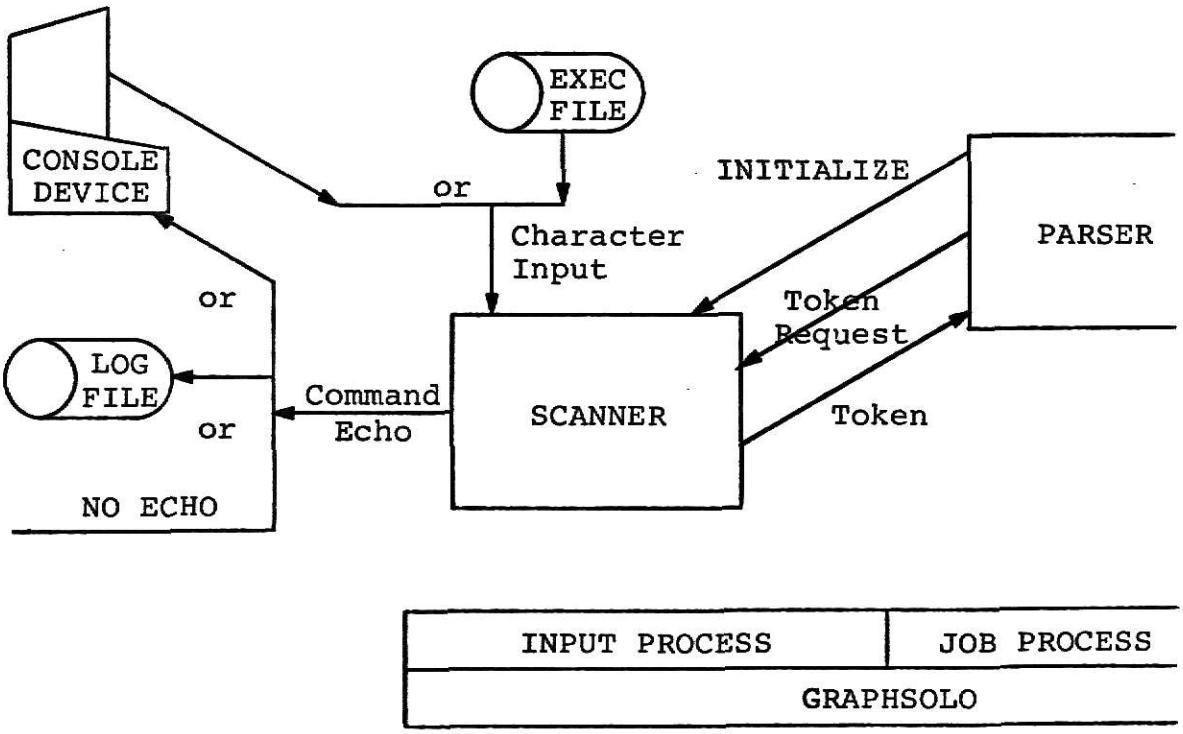


Figure 4.2: Closeup of Scanner Module Activity

described, with emphasis on data structures and algorithms.

The Scanner Module

Scanning Process

The scanner resides in the input process partition under the SOLO operating system (Figure 4.2). It is invoked by the parser during the initialization procedure. After the scanner is invoked, it begins its own initialization process, primarily to create and initialize the name table. After initialization, the scanner enters a loop and identifies tokens on request until the interpreter terminates execution. The activity within the loop is summarized by the following steps.

1) Wait for a token request from the parser.
2) Analyze the token request to determine whether or not to HALT scanning.
3) Scan the character input and construct a token.
4) Return the token to the parser.

Each time the parser needs a new token, it sends a request for a token to the scanner. The request indicates where the token is to be found, either at the PIGLI system console or in an external exec file. It indicates where the token is to be written; to an external logging file, to the system console in the case of exec file input, or to no

special place. The request also indicates control functions for error handling or system termination. When the parser encounters a syntax error, the scanner bypasses all tokens until it locates the command terminator, a period. When the parser request specifies that the scanner should halt, it exits the loop to perform wrap up activities before terminating.

The method used to scan for tokens generally follows the method used in the first pass of the Concurrent PASCAL compiler written by A. C. Hartmann (1976). Tokens fall into five categories; comments, identifiers, strings, numbers, and special symbols. Each category has a character or set of characters with which tokens in that category may start and which determines the subsequent scanning activity. For example, encountering a digit as the first character of a token indicates that the token is a number token. Thereafter, only digits and a decimal point are expected characters. Any other character encountered stops the scanning of the number token. Any ASCII character which is not a member of one of the token starting sets is ignored as an invalid character when it is encountered at the beginning of a token. Table 4.1 shows the starting character sets for each category of token.

After a token has been found, it is passed to the parser. Tokens are represented by an integer token code

TOKEN	CHARACTER SET
Comments	"
Strings	'
Numbers	0 1 2 3 4 5 6 7 8 9
Identifiers	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _
Special Symbols	. : < > = ( ) , ; * / + - &

Table 4.1: Starting Character Sets for PIGLI Tokens

accompanied by information which is dependent upon the type of token. The token code is always passed to the parser first so that the method of receiving the rest of the information may be determined. There are unique token codes for all special symbols, for all keywords, and for integer numbers, real numbers, new identifiers, old identifiers, and strings. Table 4.2 shows the codes for each kind of token.

For keywords and special symbols, the token code is sufficient information to be passed to the parser. For new identifiers, that is, identifiers that have not been encountered previously, the scanner passes the token code representing "new identifier", the spelling index (a unique integer by which the identifier will be known), and the string of characters that make up the identifier. For old identifiers the scanner passes the token code representing "old identifier" and the spelling index for that identifier. For integer and real numbers, the token code and the value of the number are passed. Strings require more complex information because they may have variable lengths. The scanner passes the token code, the length of the string, and the characters of the string. Strings are divided into blocks of twelve characters to decrease the amount of parameter passing needed. Table 4.3 summarizes the information passed to the parser for each type of token.

KEYWORD TOKEN CODES

BEGIN	0	SAVE	22	CIRCLE	72	DEG	93
REAL	1	LOAD	23	ARC	73	ANGLE	94
INTEGER	2	NOT	24	FIGURE	74	FRDEG	95
IF	3	AND	25	CURVE	75	TODEG	96
WHILE	4	OR	26	TRANS	76	FORM	97
EXECUTE	6	DIV	37	TURN	77	OPEN	98
ENTER	7	MOD	38	SCALE	78	CLOSED	99
EXIT	8	CONV	40	START	79	DIR	100
LOGON	9	TRUNC	41	MID	80	CW	101
LOGOFF	10	VALU	42	ENDPT	81	CCW	102
NULL	11	XVAL	43	CENTER	82	POINT	103
HALT	12	YVAL	44	TOP	83	PNT	104
BUILD	13	THEN	50	BOT	84	LTOP	105
CHANGE	14	ELSE	51	LEFT	85	RTOP	106
DELETE	15	DO	52	RIGHT	86	LBOT	107
SETSCREEN	16	END	53	PDIS	87	RBOT	108
DRAW	17	XLEN	65	PDEG	88	TRIGHT	109
ERASE	18	YLEN	66	ABOUT	89	BRIGHT	110
LIST	19	DEF	68	LENGTH	90	TLEFT	111
HTEXT	20	SCREEN	69	RADIUS	91	BLEFT	112
VTEXT	21	LINE	71	FACTOR	92		

SYMBOL TOKEN CODES

>	27	-	33	;	49	)	60
<	28	+	34	EOM	54	.	61
>=	29	/	35	'	55	=>	62
<=	30	*	36	(	56	:	63
=	31	UMINUS	39	:=	59	&	67
<>	32						

OTHER TOKEN CODES

OLD_IDENTIFIER	5	REAL_NUMBER	45
NEW_IDENTIFIER	47	INTEGER_NUMBER	46
STRING	48		

Table 4.2: Token Codes for all Keywords, Tokens, and Special Symbols

TOKEN	PASSING PARAMETERS		
KEYWORDS	TOKEN CODE		
SPECIAL SYMBOLS	TOKEN CODE		
OLD IDENTIFIER	TOKEN CODE	SPELLING INDEX	
NEW IDENTIFIER	TOKEN CODE	SPELLING INDEX	NAME
INTEGER	TOKEN CODE	INTEGER VALUE	
REAL	TOKEN CODE	REAL VALUE	
STRING	TOKEN CODE	INTEGER LENGTH	STRING BLOCKS

Table 4.3: Parameters for Passing Tokens to the Parser

Scanner Data Structures

The only data structure used by the scanner is the name table. There is an entry in the name table for every PIGLI keyword and for every new identifier encountered during the course of a PIGLI programming session. Each entry consists of the name of the keyword or identifier and the spelling index, an integer which uniquely identifies the keyword or identifier.

Two methods are used to determine the spelling indices which will be associated with identifiers; one method for keywords and one method for ordinary identifiers. For a keyword, the spelling index is found by negating the token code for that keyword making the spelling indices for all keywords unique negative integers. For example, the token code for the keyword BUILD is 13, (see Table 4.2) therefore the spelling index for the keyword BUILD is -13. For ordinary identifiers, the spelling indices are unique positive integers. The first identifier encountered in a PIGLI programming session is assigned integer 1 as the spelling index. Each new identifier found thereafter is assigned the next consecutive positive integer as the spelling index. For example, if the first command of a PIGLI program is

```
REAL X, Y, Z.
```

the three new identifiers X, Y, and Z are assigned integers

1, 2, and 3, respectively, as spelling indices.

Keywords are entered into the name table during the scanner's initialization process. To determine where to enter a keyword, the following hashing procedure is used.

```
PROCEDURE HASH_KEY(ID);
"ID IS AN ARRAY OF AT MOST MAX_ID_LENGTH CHARACTERS"
   BEGIN
      HASH_KEY := 1;
      FOR CHAR_INDEX := 0 TO MAX_ID_LENGTH DO
         IF ID[CHAR_INDEX] <>
         THEN BEGIN
            NUM_EQUIV := (ORD(ID[CHAR_INDEX])
               MOD NUMBER_POSSIBLE_CHARS + 1);
            HASH_KEY := HASH_KEY * NUM_EQUIV
               MOD HASH_MAX_PRIME;
         END;
      WHILE NAME_TABLE[HASH_KEY] <> EMPTY DO
         HASH_KEY := (HASH_KEY + 1)
            MOD HASH_MAX_PRIME;
   END;
```

This procedure calculates the HASH_KEY which is the index into the name table. The NUM_EQUIV, an integer in the range of 1 to 27, is obtained for each character in the identifier; 1 is equivalent to an A, 2 is equivalent to a B, and so on. The function ORD returns the ordinal equivalent of its argument, in this case a character of the identifier. The numeric equivalents for all the characters are multiplied together, and the product is modded with the maximum number of entries allowed in the table (HASH_MAX_PRIME), to produce the initial HASH_KEY. If the entry indicated by HASH_KEY is not empty, succeeding entries are tried until an empty entry is found.

When an identifier is encountered during the scanning process, the hashing function is used to determine the appropriate entry in the name table to begin searching for that identifier. If the entry is empty, the identifier is a new identifier. It is inserted in the empty entry of the name table and given a spelling index. If the entry contains an identifier which does not match the current token, the scanner looks at the next consecutive entry until it locates the correct identifier or an empty entry. If the entry does match the current token, the spelling index is used to classify the token as a keyword or an old identifier.

The Parser Module

Parsing Process

The major processing of the parser can be described by the following steps.

1) Parse a command
2) If the command is syntactically correct, invoke the executor; otherwise identify the error for the programmer.
3) Reinitialize to begin processing a new command.
4) If the command was HALT, terminate the system.

PIGLI uses the method of recursive descent to parse a

command. Each command is distinguished by a unique keyword. After a command keyword is found, a procedure for that specific command is invoked to continue the parsing process. Whenever a new token is needed, a scanning procedure requests a token from the scanner module and interprets the returned information. At each step in the parse, there are a finite number of tokens which may legally be encountered and which determine the subsequent direction of the parsing process.

Encountering an unexpected token indicates a syntax error and causes the parsing to stop. Error recovery consists of recording the illegal token and the token type expected, and requesting the scanner to begin error processing. The scanner bypasses all tokens in the remainder of the command up to the command termination symbol. The parser displays an appropriate message on the PIGLI system console identifying the error and prepares to accept a new command.

If a complete, syntactically correct command is identified, the parser passes an internal representation of the command and control of the job process partition to the executor module, using the PASCAL statement

```
RUN('EXECUTOR    ',LIST,LINE,RESULT).
```

The parameters of the RUN statement are defined as follows:

1) 'EXECUTOR ', the twelve character identifier, enclosed in quotes, of the process to be run,
2) LIST, an array of up to ten arguments to be passed to or returned from the process,
3) LINE, the returned line number of an erroneous statement, if the process does not terminate normally, and
4) RESULT, the return code for the process.

(LINE and RESULT are used to indicate failure of the SOLO operating system to properly execute the RUN statement. In the event of such failure, the PIGLI system crashes.)

The PIGLI parser passes six arguments to the executor. The first argument is a return code that indicates whether or not the executor encountered an execution error. The second argument is a pointer to the beginning of the structure which holds the internal representation of a command. The third argument is a pointer to the beginning of the screen file, which is a list of the identifiers of the pictures that are being displayed at a given time. The fourth argument is a pointer to the location of the symbol table. The symbol table contains the stored definitions of all pictures and variables. The fifth argument is a pointer to the free area left in the data space. Free spaces are

needed by the parser when it constructs the internal command structure and by the executor when it builds the definition of a picture. The sixth argument is a pointer to a record which contains the system status indicator and several miscellaneous data items which are shared by both the parser and the executor. The system status indicator may take on integer values between 0 and 5 to indicate command input location, changes in logging, or execution of a HALT command. (Table 4.4 gives the meaning of the six possible system status values.) Shared data items include indices into the symbol table bounding the unused portion of the table.

The arguments passed between the parser and the executor serve two purposes. The first and last arguments contain command execution status information; one informs the parser of incomplete or incorrect execution of the previous command, and the second provides information which the parser needs after the execution of a HALT, LOGON, LOGOFF, or EXECUTE command. After one of these commands, the parser may need to know where to locate the next command input, where to echo that input, or whether or not it should terminate the system. The other four arguments pass the locations of information shared by both the parser and the executor, (the internal command structure, the symbol table, and free data space) and the locations of data structures

VALUE	MEANING
0	Input from console - no logging desired
1	Input from console - commands logged
2	Input from exec file - no logging desired
3	Input from exec file - commands logged
4	HALT processing
5	Error processing

Table 4.4: Possible Values of the System Status Indicator

that must remain intact through the entire PIGLI programming session (the symbol table and the screen file).

When the executor has finished processing the command, control of the job process partition returns to the parser. If one of the commands LOGON, LOGOFF, or EXECUTE was executed, the parser resets its status indicator to the appropriate condition. For example, after the execution of LOGOFF, the parser will no longer need to request the scanner to log PIGLI commands, and will change the request indicator to reflect this. The parser next dismantles the internal representation of the executed command and returns the data space it occupied to the free data space structure. Finally, if a HALT command was executed, the parser terminates the modules in the input and output process partitions and ends. If HALT was not executed the parser begins parsing a new command.

Parser Data Structures

The parser makes use of three data structures in its processing, the internal command structure, the free data space structure, and, to a lesser degree, the symbol table. (The PASCAL definitions of these three data structures are given in Appendix B.) The internal command structure is constructed by the parser as it parses a command. It contains all the information which the executor needs to

execute that command. The free data space structure contains linked lists of all free data space. The symbol table is the primary structure for retaining information about variables, pictures, and external files.

The Internal Command Structure

The internal command structure is implemented in PASCAL as a statement record pointed to by a statement record pointer called COMMAND. The statement record has a definition variant for each of eighteen types of command. Since PASCAL only allows sixteen variants of a given record, the eighteen commands are first divided into two groups, picture statements and programming statements, and then subdivided into individual commands. Picture statements include variants for the BUILD, DELETE, DRAW, ERASE, HTEXT, VTEXT, and SETSCREEN commands. Programming statements include variants for the BEGIN, IF, WHILE, assignment, HALT, NULL, LOGOFF, LOGON, LOAD, EXECUTE, and LIST commands. Figure 4.3 shows the first two levels of variation in the internal command structure.

The basic components of the internal structure for each command type are variant tags, keywords, identifiers, expressions, and points. A variant tag identifies which case of two or more possible cases is to be used to construct the internal representation of a particular command. For example, PIC_STMT and PROG_STMT are tags to

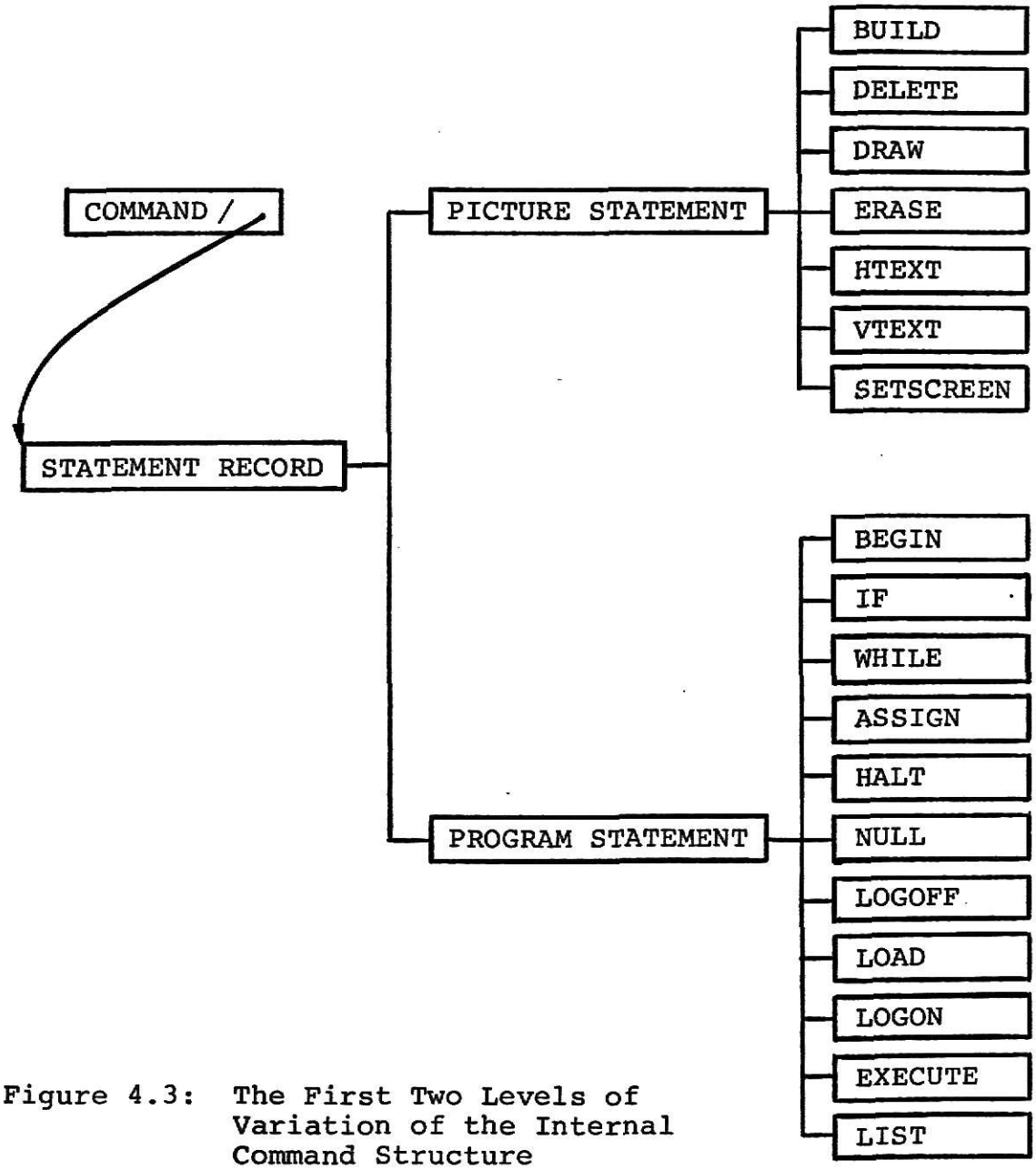


Figure 4.3: The First Two Levels of Variation of the Internal Command Structure

distinguish between the two major groups of commands represented in a statement record. There is also a special tag for each type of command within each group.

Internal command structure keywords are a subset of the keywords of the PIGLI command language. They are used to indicate which of several items or values are appropriate for a particular part of the internal command structure. For example, the FORM of a FIGURE may be OPEN or CLOSED. (See Appendix A) It is convenient to represent the value of the FORM specification of a FIGURE by the keywords OPEN or CLOSED.

Identifiers are the names of variables, pictures, picture components, and external files that have been stored in the symbol table. Identifiers used in the internal command structure, for example, to indicate the target of an assignment command, are represented by the symbol table index for that identifier entry.

Expressions are sequences of operators and operands representing numeric or boolean values which may be encountered in several commands. As an expression is parsed, it is converted from infix notation to postfix notation. Each operator and operand is stored in a linked list called an expression record. Whenever an expression is encountered while parsing a command, an expression record for that expression is constructed. Each element of the

linked list of the expression record may be one of two variants, an operator defined by the keyword associated with it, (including arithmetic and boolean operators) or an operand. An operand is defined by one of three variants, a variable identifier, a numeric constant or a system value reference definition. A constant may be either real or integer. A system value reference is defined by two keywords to indicate which value reference is desired and an identifier to indicate which picture or picture component is to be referenced. After the expression record is constructed, a pointer to the record is added to the internal command structure. Figure 4.4 is a graphical representation of the general structure of an expression record.

Points are encountered during the parsing of a BUILD command. When a point is parsed, the parser constructs a point record to define the point and adds a pointer to the point record to the internal command structure of the BUILD command. A point record, which has two variants, consists of either pointers to two expressions, one each for the x coordinate and the y coordinate of the point, or a system point reference. A system point reference includes a keyword to indicate which point reference is desired, and an identifier to indicate which picture or picture component is to be referenced. Figure 4.5 is a graphical representation

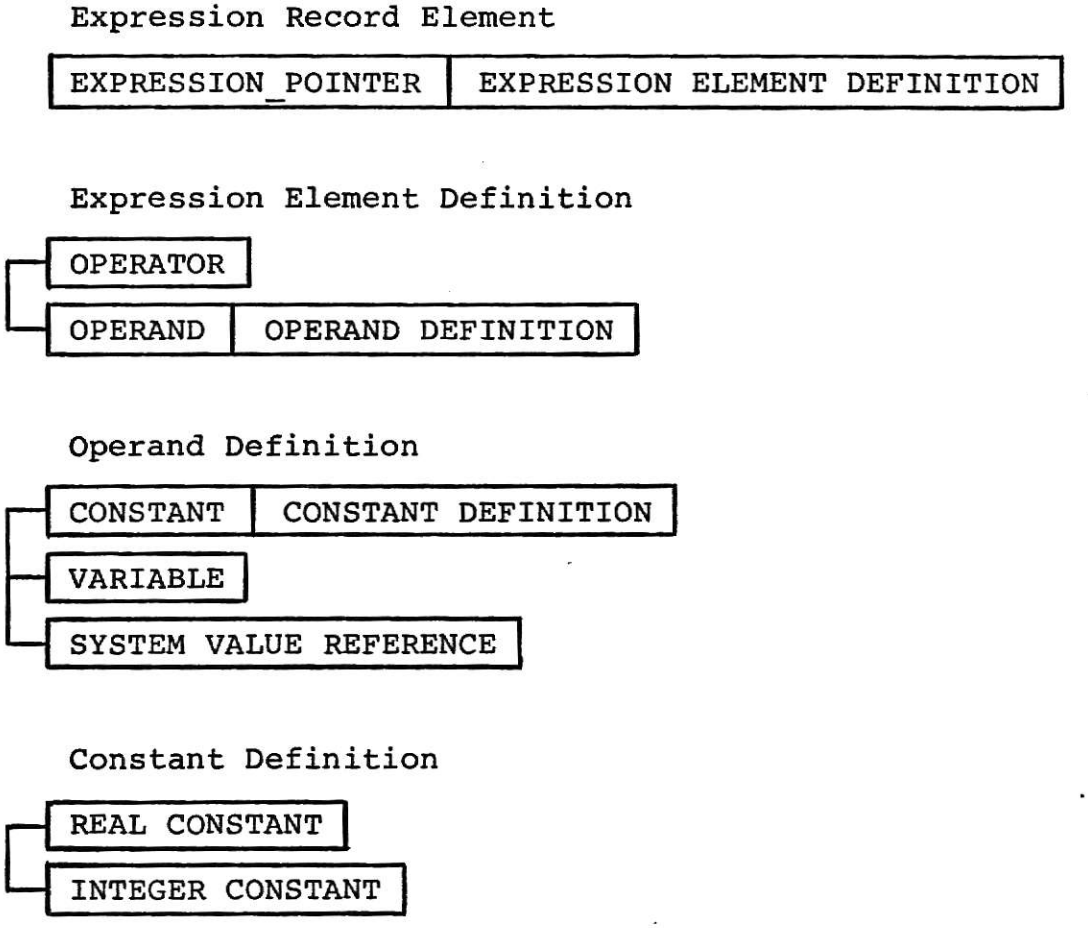


Figure 4.4: Internal Structure of an Expression Record

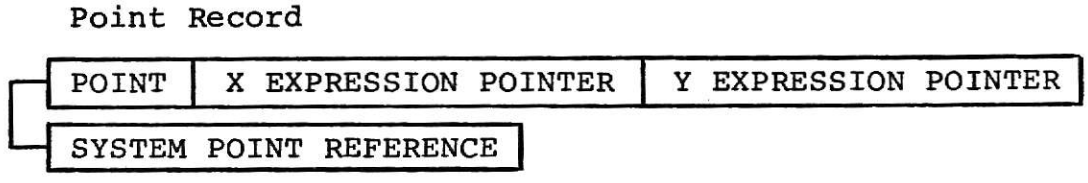


Figure 4.5: Internal Structure of a Point Record

of the general structure of a point record.

The values in each statement record are defined as the command is parsed and pertinant information is collected. The command keyword that begins each command determines a) whether the command structure will use the programming statement variant or the picture statement variant, and b) which particular command variant will be used. Two tags in the internal command structure are set to the appropriate values and, while the remainder of the command is parsed, the information specific to each type of command is stored in the internal command structure. Figure 4.6 shows a graphical representation of the internal command structure of each command.

The internal form of the BUILD command includes the picture identifier to be defined and a pointer to a chain of picture component definitions. New elements of the picture chain are constructed as the parser encounters new components of the picture being built. Each element of the picture chain defines one component of the picture. The definition includes the component's identifier, if one was specified, a pointer to the next element in the picture chain, and one of three component definition variants, depending on how the component is described. One possible variant (SCREEN) indicates that the component is the combination of all the pictures in the current display file.

BUILD Command

BUILD	PICTURE IDENTIFIER	PICTURE CHAIN POINTER

Picture Chain Element

COMPONENT IDENTIFIER	PICTURE CHAIN POINTER
COMPONENT DEFINITION	

Component Definition

SCREEN	OLD PICTURE IDENTIFIER
OLD PICTURE	OLD PICTURE IDENTIFIER
PRIMITIVE	PRIMITIVE TAG	PRIMITIVE SPEC CHAIN POINTER

Primitive Spec Chain Pointer

SPEC TAG	PRIMITIVE SPEC CHAIN POINTER	SPEC DEFINITION

Spec Definition

SPEC POINT	POINT POINTER
SPEC PARM	KEYWORD
SPEC PICTURE	PICTURE IDENTIFIER
SPEC VALUE	EXPRESSION POINTER

DELETE Command

DELETE	PICTURE IDENTIFIER

Figure 4.6: Internal Structures of each Type of Command

DRAW Command

DRAW	PICTURE IDENTIFIER

ERASE Command

ERASE	PICTURE IDENTIFIER

HTEXT and VTEXT Commands

HTEXT	HV_POINT	TEXT CHAIN POINTER
VTEXT	HV_POINT	TEXT CHAIN POINTER

Text Chain Element

TEXT CHAIN POINTER	TEXT DEFINITION

Text Definition

STRING TEXT	STRING
VALUE TEXT	VALUE EXPRESSION POINTER

SETSCREEN Command

SETSCREEN	SS_POINT	X LENGTH EXPRESSION POINTER
Y LENGTH EXPRESSION POINTER		

(Figure 4.6, Continued)

BEGIN Command

BEGIN STATEMENT CHAIN POINTER

Statement Chain Element

STATEMENT CHAIN POINTER	STATEMENT POINTER

IF_THEN_ELSE Command

IF	IF BOOLEAN EXPRESSION POINTER	
IF_TRUE STATEMENT POINTER		IF_FALSE STATEMENT POINTER

WHILE_DO Command

WHILE	WHILE BOOLEAN EXPRESSION POINTER
WHILE_TRUE STATEMENT POINTER	

Assignment Command

ASSIGN	TARGET IDENTIFIER	SOURCE EXPRESSION POINTER

HALT, NULL, and LOGOFF Commands

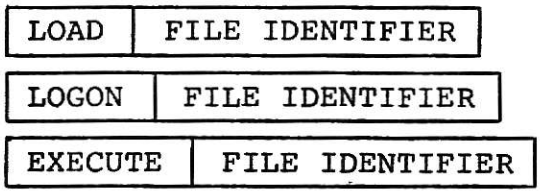
HALT

NULL

LOGOFF

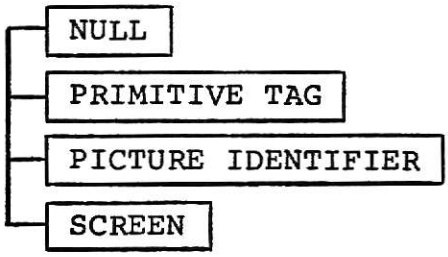
(Figure 4.6, Continued)

LOAD, LOGON, and EXECUTE Commands



LIST Command

List Spec



(Figure 4.6, Continued)

The second variant (OLD_PIC) indicates that the component is a single, previously defined picture. The third variant (PRIMITIVE) defines the component as a picture primitive or a transformation of a previously defined picture. The definitions of the first and second variants are pointers to entries in the symbol table for the SCREEN file or the appropriate picture identifier. The definition of the third variant includes a keyword to distinguish which kind of primitive or transformation the component is, and a pointer to a chain of primitive or transformation specifications. Each element of the specification chain includes a tag indicating which specification item is being defined, a pointer to the next element of the specification chain, and one of four variants that may define the specification item, either a point (SPEC_POINT), a keyword (SPEC_PARM), a picture identifier (SPEC_PIC), or a numeric value (SPEC_VALUE).

The internal forms of the DELETE, DRAW, and ERASE commands are all similar and much simpler than that of the BUILD command. The definitions include only the picture identifier of the picture, or named picture component, to be deleted, drawn, or erased. The identifier is represented by the symbol table index of the appropriate symbol table entry.

The internal form of the HTEXT and VTEXT commands are

also similar to one another. Their definitions include a pointer to the definition of the orientation point, and a pointer to a chain of text items. Each text item includes a pointer to the next element in the text chain, and one of two variant definitions of a text item, either a string of characters (STRING_TEXT), or a numeric value (VALUE_TEXT). The numeric value is represented by a pointer to an expression record.

The SETSCREEN command is represented internally by a pointer to the definition of the orientation point, and two numeric values, XLEN and YLEN. Both values are represented by a pointer to an expression record.

The internal form of the BEGIN block is a chain of statement records within the original BEGIN statement record. Each element of the chain contains a pointer to a separate statement record for one of the commands in the BEGIN block and a pointer to the next element of the statement record chain. After the BEGIN keyword is encountered by the parser, it parses a compound statement, that is, it recursively parses more commands, building an element of the statement record chain for each command, and linking them together in the statement chain.

The internal representation of the IF command includes a pointer to a boolean expression and two pointers to statement records, one for the command to be executed when

the boolean expression evaluates to true, and one for the command to be executed when it evaluates to false. After both keywords, THEN and ELSE, are encountered, the parser recursively begins to construct a new statement record for the commands in the branches. The WHILE command is similar to the IF command. Its internal representation also includes a pointer to a boolean expression record, but it contains a pointer to only one statement record, for the command encountered after the keyword DO, which will be executed repeatedly while the boolean expression evaluates to true.

The internal form of the assignment command is composed of an identifier (represented by the index of the symbol table entry for that identifier), which is the target of the assignment, and a pointer to an expression record whose value will be assigned to the target.

The HALT, NULL, and LOGOFF commands are similar to one another. They do not include any information in their internal forms other than the tag indicating the type of the command.

The LOAD, LOGON, and EXECUTE commands are also similar to one another. Each command includes the identifier of an external file, represented by the index to the symbol table entry for that identifier.

The LIST command is defined internally by one of three

variant definitions. One variant (PIC_LIST) requires no information. It is used when the simple command LIST is issued to retrieve from the symbol table the names of all pictures composed of more than one picture element. The second variant (PRIM_LIST) includes a keyword to indicate that the identifiers of all the named picture elements, of the type specified by the keyword, are to be retrieved from the symbol table. The keyword may also indicate that the names of displayed pictures be listed. The keyword may be LINE, CIRCLE, ARC, FIGURE, CURVE, TRANS, TURN, SCALE, or SCREEN. The third variant (DEF_LIST) includes the identifier of a symbol table entry, either a variable, an external file, a picture, or a named picture component. This form is used when the definition of an identifier is desired.

After a syntactically correct command has been parsed and stored in the internal command structure, it is passed to the executor to be executed. To pass the command, the pointer COMMAND, which points to the beginning of the statement record, is passed to the executor as the third argument of the argument list. The command is organized in such a manner that the executor, receiving a pointer to that structure, can locate all pertinant data as it is needed to execute the command. Consider the following example. In the command

IF A=B THEN A := A+1 ELSE NULL.

the keyword IF indicates an IF_THEN_ELSE command, therefore two tags in the statement record are set to show that the command is a programming statement and that the IF variant of the record will be used. (See figure 4.7) In the IF variant of a statement record there are three data items, IF_BOOL, IF_TRUE, and IF_FALSE. IF_BOOL is a pointer to the internal representation of a boolean expression. IF_TRUE and IF_FALSE are pointers to the internal forms of the statements to be executed if the boolean expression evaluates to true or false respectively.

Parsing the remainder of the command includes parsing the boolean expression, A=B, and two statement clauses, THEN A := A+1 and ELSE NULL. The boolean expression is converted to postfix notation and stored in an expression record pointed to by IF_BOOL. Each branch of the IF_THEN_ELSE command is parsed recursively and two additional statement records are created. IF_TRUE points to a new statement record for the assignment statement A := A+1 and IF_FALSE points to a statement record for a NULL statement.

The Free Data Space Structure

The free data space structure is used to economically manage the data space allocated to the PIGLI interpreter by the SOLO operating system. Dynamic space allocation is

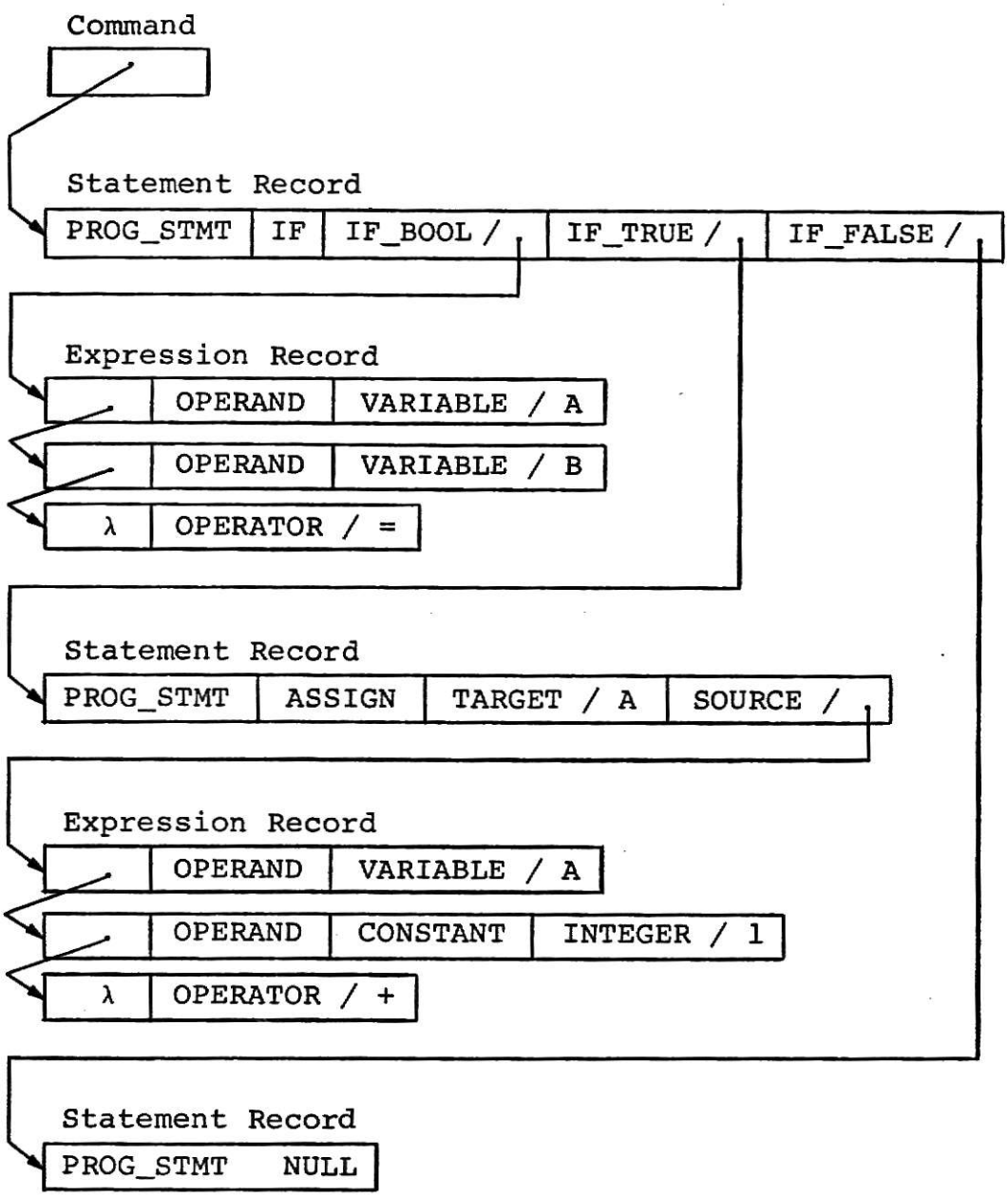


Figure 4.7: An Example of the Internal Structure of an IF_THEN_ELSE Command

required to construct an internal representation of a command, or to define a new entry in the symbol table. The space allocation method provided by SOLO is the NEW procedure. The NEW procedure locates free space in the heap of system data space for a specific type of record. Once allocated, this space may not be used to hold any other type of record. When data space is released, by a DELETE command or by dismantling the internal forms of executed commands, the freed space is stored, by record type, in the free data space structure. This structure is a record which contains a pointer for each type of record to the beginning of a linked list that holds the free areas for that record type. When space is needed to hold a record of a particular type, the free data space structure is checked. If an appropriate space exists, it is removed from the structure. If such a space does not exist, the NEW procedure is used to obtain it. Figure 4.8 shows a graphical representation of the free data space structure.

The Symbol Table

The symbol table is used primarily by the executor. However the parser inserts variables from declaration commands and picture identifiers from BUILD statements into the symbol table. This allows identifiers in the internal command structure to be defined in terms of entries in the symbol table. The parser also identifies variables as real

Free Data Space Structure

GRAPH_NODE POINTER
EXPRESSION RECORD POINTER
POINT RECORD POINTER
STATEMENT RECORD POINTER
PICTURE PART POINTER
SPECIFICATION CHAIN POINTER
PICTURE CHAIN POINTER
TEXT RECORD POINTER
STATEMENT CHAIN POINTER
KNOT CHAIN POINTER

Figure 4.8: The Organization of the Free Data Space Structure

or integer. Since declaration commands do not require any other action, they are not passed to the executor module for processing. Thus, they are not executable and cannot be included in complex commands such as the IF_THEN_ELSE command and the DO_WHILE command.

The Executor Module

Executing Process

The processing done by the executor is summarized by the following steps.

1) Initialize.
2) Execute the command.
3) Wrap up loose ends.

When the executor is invoked by the parser to run in the job process partition, it receives an argument list containing pointers to the data structures it shares with the parser. The first step of initialization is to use those pointers to map the executor's definition of the data structures onto the appropriate data space. Since the data structures are defined identically in both the parser and the executor (see appendix B), this mapping process is fairly simple. The second step involves initializing PASCAL sets of information used during the execution of some commands. The set handling capabilities of PASCAL permit the executor to check the validity of a complex set of data against those standard

sets defined during initialization.

To execute a command, the executor uses the information stored in the internal command structure by the parser. There is a procedure in the executor module to execute each command. A summary of the actions performed for each command is given later in this chapter.

Execution of a command may terminate in one of three ways: 1) the command may be executed normally to completion, 2) the command may be terminated before completion due to an execution error, or 3) the command may be terminated before completion by the execution of a HALT command in a BEGIN_END block. When execution terminates normally, control of the job process partition is returned to the parser and the argument list that was passed to the executor is passed back to the parser. The first argument is set to indicate that the command completed normally. When an execution error is encountered, execution of the command stops. The cause of the error is recorded and the first argument is set to indicate an execution error. An appropriate message is displayed at the system console identifying the error. Then control is passed back to the parser. When a HALT command is encountered, execution of the command stops. No error message is generated, and the first argument is set to show normal termination of the command. Control is returned to the parser with all the information received when the

executor was invoked. In this case the parser will terminate the entire PIGLI system when it regains control of the job process partition.

The actions performed by the executor and the parser could have been performed in a single module. That alternative would have provided advantages by removing the initialization step for all but the first command and by eliminating the overhead of parameter passing between the parser and the executor. However, by splitting the activity between two modules, the total memory space required by the running module is cut in half, increasing the memory space available for picture data storage, and the scope of each module is reduced to manageable proportions.

Executor Data Structures

To execute a command the executor makes use of three data structures, the internal command structure, the free data space structure, and the symbol table. (The PASCAL definitions of these three data structures are given in Appendix B.) The internal command structure is constructed by the parser and passed to the executor when it is invoked. This structure holds an internal form of a command, organized so that the executor can determine what command is to be executed, and how and where to retrieve the information needed to execute that command. The free data

space structure is used to economically manage the data space allocated to the PIGLI interpreter by the SOLO operating system. It contains a set of linked lists of free data space, one list for each type of record that may be freed. Both the internal command structure and the free data space structure were described in the section on parser data structures.

The Symbol Table

The symbol table is used in PIGLI to retain dynamic information defining variables, external files, and pictures. To define a symbol table entry, the executor constructs a record (GRAPH_NODE) which describes all the pertinant characteristics of the entity identified in the symbol table entry. The symbol table is implemented in PASCAL as an array of symbol table entries. The array index is the spelling index that is associated with an identifier when it is first encountered by the scanner. Each symbol table entry consists of two items, the identifier associated with the variable, file, or picture defined by the entry, and a pointer to the GRAPH_NODE record containing the definition. The definition of an entry may be one of thirteen variants of the GRAPH_NODE record, each of which contains an appropriate variant tag and a canonical definition of the entity defined by the symbol table entry. Figure 4.9 shows a graphical representation of each variant

GRAPH_NODE Variants

REAL	Value

INTEGER	Value

FILE

UNDEFINED

PICTURE	PIC PART_CHAIN POINTER

Picture Part Chain

BASE PICTURE	PIC_PART_CHAIN POINTER

LINE	START	ENDPT

CIRCLE	CENTER	RADIUS

ARC	START	ENDPT	CENTER	DIRECTION	FORM

FIGURE	KNOT CHAIN POINTER	FORM

CURVE	KNOT CHAIN POINTER	FORM

Knot Chain

KNOT	KNOT CHAIN POINTER

TRANS	DESTAX	DELTAY	BASE PICTURE

TURN	ROTATION POINT	ROTATION ANGLE	BASE PICTURE

SCALE	SCALE POINT	SCALE FACTOR	BASE PICTURE

Figure 4.9: The Variations of the GRAPH_NODE Data Structure

of the GRAPH_NODE record.

Variables

Numeric variables may be either real or integer depending on how they were initially declared. When the executor defines a variable, one of the variant tags REAL or INTEGER, and the current value of the variable are used. If the programmer attempts to use a declared variable before it has been defined, the variable is arbitrarily assumed to have a value of negative one (-1 or -1.0) to prevent an execution error.

External Files

External files, used to name log files, exec files, and display device drivers, are defined by the variant tag FILE and no other information. Picture identifiers may also be defined by the tag UNDEFINED with no other information. A picture identifier is undefined from the time it is first encountered by the parser until it is defined by the executor.

Pictures

Picture entries in the symbol table which are composed of more than one picture primitive, picture transformation, or previously defined picture, are defined by the PICTURE variant. This variant includes a pointer to a picture chain. Each element of the chain consists of a pointer to the entry in the symbol table that describes one component

of the picture, and a pointer to the next element of the chain. Pictures that are composed of only one instance of a picture primitive or a picture transformation, or an individual element of a picture, are defined by the appropriate primitive or transformation variant. Since it is not necessary for all elements of a compound picture to have identifiers associated with them, some elements are given symbol table entries with dummy identifiers. This allows them to be treated the same way as named picture components, but does not permit subsequent references to those individual elements by the user.

Primitives

The LINE variant describes an instance of a straight line primitive using the values of the x and y coordinates of the start point and the end point of the line. The CIRCLE variant includes the x and y coordinates of the center point of the circle and the value of its radius. The ARC variant is described by the x and y coordinates of the start point, the end point, and the center point, the direction of the arc (counter clockwise or clockwise), and the form of the arc (major or minor). The FIGURE and CURVE variants are similar. They include a pointer to a chain of the points used to define the figure or curve and a keyword value to indicate whether the FORM of the figure or curve is OPEN or CLOSED. Each element of the point chain contains

the values of the x and y coordinates of the point and a pointer to the next element of the chain.

Transformations

There are three transformation definition variants. The TRANS variant includes values for the change in the x and y directions, due to the translation, and a pointer to the entry in the symbol table of the picture to be translated. The TURN variant consists of the values of the x and y coordinates of the point about which the picture is to be rotated, the value of the anble through which it is to be rotated, and a pointer to the entry in the symbol table of the picture to be rotated. The SCALE variant is composed of the values of the x and y coordinates of the point about which the picture is to be scaled, the value of the factor by which it is to be scaled, and a pointer to the entry in the symbol table of the picture to be scaled.

Information is entered in the symbol table by the parser during the declaration of real and integer variables, and by the executor during the execution of assignment commands. The information stored in the symbol table is referenced during the execution of several commands. The values of arithmetic variables may be used in arithmetic expressions. Value attributes of pictures may be used in expressions, and point attributes of pictures may be used to define other

points. Picture definitions are referenced by the DRAW command, the ERASE command, and the LIST command. The command execution algorithm section includes a description of how each command uses the symbol table.

Command Execution Algorithms

Evaluating Expressions and Points

Expressions and points are two structures that are found in the internal representations of several commands. An expression is stored in the internal command structure as a linked list of operands and operators in postfix order (see Figure 4.4). When an expression is evaluated, the executor steps through the list, processing each operator and operand as it is encountered.

The executor processes the values of operands by pushing them on a temporary expression evaluation stack. Constant values are immediately available in the internal expression record. Values of variables are retrieved from the symbol table using the symbol table index for the appropriate entry that is included in the expression record. For values that the user has neglected to define prior to using them in an expression, the executor substitutes arbitrary values (-1 or -1.0). System reference functions must be evaluated before they can be stacked. For every attribute that may be used to describe a picture primitive, there is an executor

procedure that will calculate that attribute using the canonical definition of the picture primitive. For example, either the length of a line or the mid point of a line may be calculated using the end points that make up the canonical definition of that line. The picture primitive which is being referenced is located using its symbol table index, and the value attribute which is desired for that primitive, indicated by a keyword, is checked for validity against the type of the referenced primitive. If the desired value is valid, it is calculated. Values that are used in the canonical definition of a primitive are simply extracted from the definition (see Figure 4.9). For other values the executor calculates the proper value.

Whenever operators are encountered during evaluation of an expression, one or two operands are popped from the expression evaluation stack, depending on whether it is a unary or a binary operator and the operation is performed on them. Unary operations are the type conversion functions, the boolean negation, and arithmetic negation (CONV, TRUNC, NOT, UMINUS). Binary operators are all logical comparison operators (<, >, =, <=, >=, <>), boolean operators for conjunction and disjunction (AND and OR), and arithmetic operators for addition, subtraction, multiplication, and division (+, -, *, /, DIV, and MOD).

Evaluation of operators is simplified by the fact that

the parser checks for type compatibility in expressions. For binary arithmetic operators, the second value popped from the stack is operated on by the first value, that is, the second value is divided by the first value for the '/', 'DIV', and 'MOD' operators, and the first value is subtracted from the second value for the '-' operator. For logical comparison operations and binary boolean operations, the same ordering is used. When two numeric values are compared using logical operators, the result is a boolean value, which is represented by integer zero for false and integer one for true. After the specified operation is performed, the result is pushed back onto the stack. When there are no more operators or operands in the expression record, the result is popped off the stack and command execution continues using that value.

Point records are found in the internal command structure of a BUILD command (see Figure 4.5). A point is defined by two expressions for its x and y coordinates or by a system point reference function. System point reference functions are evaluated by the method used to evaluate system value reference functions in expression records. The picture primitive which is being referenced is located using its symbol table index, and the point attribute which is desired for that primitive, indicated by a keyword, is checked for validity against the type of the referenced

primitive. If the desired point attribute is valid, its x and y coordinates are calculated. Points that are used in the canonical definition of a primitive are simply extracted from the definition (see Figure 4.9). For other points the executor calculates the proper value.

To execute a command, the executor first determines from the internal command structure which command is to be executed. For each command a separate procedure is used to traverse the command structure and perform the necessary actions. Figure 4.10 is a summary of the steps followed to execute each command. Figures 4.6 and 4.7 are helpful in understanding the following descriptions of the process of evaluating each command.

BUILD Command

Executing a BUILD command may require 1) building a new picture composed of one, unnamed picture primitive or transformation, 2) building a new picture composed of more than one picture component, or 3) rebuilding a previously defined picture.

The first type of BUILD command constructs a definition of a picture primitive or a transformation that is pointed to directly by the symbol table entry for the target of the command. There are four steps for building this primitive or transformation. First, all the specifications are collected and evaluated. The values of the specifications

BUILD Command

Algorithm for type one: Constructing a new picture composed of one, unnamed picture primitive or transformation component.

1. Collect and evaluate all the specifications of the component definition.
2. Analyze the specifications for completeness and correctness.
3. Calculate the canonical definition, if necessary.
4. Construct the definition record and link it to the symbol table entry being defined.

Algorithm for type two: Constructing a new picture composed of more than one component. Note: Components may not be redefinitions of previously defined pictures.

For each component:

1. If the component is a new primitive or transformation:
   - 1.1. For unnamed components, construct a dummy symbol table entry.
   - 1.2. Construct the component definition as if it were a type one BUILD command.
   - 1.3. Construct a picture chain element pointing to the symbol table entry just defined.
2. If the component is a previously defined picture:
   - 2.1. Construct a picture chain element point to the symbol table entry of that picture.

Algorithm for type three: Rebuilding a previously defined picture. Note: Previously defined pictures and components may be redefined only as targets of a build statement.

1. Move the original definition to a dummy symbol table entry.
2. Set a flag to show that no reference to the original definition has been made.

Figure 4.10: A Summary of the Execution Algorithms of all PIGLI Commands

3. Build a picture definition using algorithm one or two with the following addition:
   3.1. If reference is made to the original picture definition, then reset the flag and refer to the dummy symbol table entry.
4. If the flag shows no references were made to the old definition, then delete it.

DELETE Command

1. Return each record of the symbol table entry definition to the appropriate list in the free data space structure.

DRAW and ERASE Commands

1. Send DRAW or ERASE code to the output device driver.
2. For each component of the picture specified:
   2.1. If the component is itself a picture, then recursively draw or erase the picture.
   2.2. If the component is a transformation of a picture:
      2.2.1. Stack the definition of the transformation.
      2.2.2. Recursively draw or erase the picture which is the base of the transformation.
      2.2.3. Unstack the transformation definition.
   2.3. If the component is a picture primitive:
      2.3.1. Evaluate the transformation stack.
      2.3.2. Calculate line segments to represent the primitive from its canonical definition.
      2.3.3. Adjust the endpoints of the line segments according to the transformation stack and the current SETSCREEN definition.

(Figure 4.10, Continued)

2.3.4. Translate the adjusted line segments to MOVE and VECTOR command codes and send them to the output device driver.

HTEXT and VTEXT Commands

1. Compress all text items into one line of text.
2. Send HTEXT or VTEXT code to the output device driver.
3. Send a command to the output device driver to MOVE to the orientation point.
4. Send the text line to the output device driver.

SETSCREEN Command

1. Update the definition of the orientation point and the clipping bounds.

BEGIN Command

1. For each element of the BEGIN block statement chain, recursively execute the command.

IF-THEN-ELSE Command

1. Evaluate the boolean expression.
2. If the expression is true, execute the THEN branch of the command.
3. If the expression is false, execute the ELSE branch of the command.

WHILE-DO Command

1. Evaluate the boolean expression.
2. If the expression is true, recursively execute the DO command and return to step one of this algorithm.
3. If the expression is false, stop executing the WHILE command.

(Figure 4.10, Continued)

Assignment Command

1. Evaluate the expression that is the source of the assignment.
2. Put the RESULT in the target of the assignment.

HALT Command

1. Reset the status indicator to show that HALT was commanded.

NULL Command

1. No execution is required.

LOGOFF Command

1. If a log file is open, then close the log file.
2. Reset the status indicator to show that logging of input commands is no longer desired.

LOGON Command

1. Open the specified file.
2. Reset the status indicator to show that logging of input commands is desired.

LOAD Command

1. Terminate the output device driver that is currently running.
2. Initialize the output device driver specified by the LOAD command.

(Figure 4.10, Continued)

EXECUTE Command

1. If no exec file is open, then open the file specified in the EXECUTE command.
2. Reset the status indicator to show that input commands will be found in the exec file until that file is exhausted.

LIST Command

Algorithm for type one: Listing the identifiers of all composed pictures, or of the named instances of a specified type.

1. Search the symbol table for named entries of the specified type and print their names and type.

Algorithm for type two: Listing the definition of a specific symbol table entry.

1. List the name and type of the desired entry.
2. If the type of the entry is not PICTURE, then list its canonical definition.
3. If the type of the entry is PICTURE, then traverse the definition chain.
   - 3.1. If the element is named, then list the name and its type.
   - 3.2. If the element is unnamed, then list its type and its canonical definition.

(Figure 4.10, Continued)

may be points, expressions, keywords, or picture references. Then the collection of specifications is analyzed for completeness and correctness to be sure that the definition is adequate and unambiguous. Next the definition is adjusted to canonical form if it is not already in that form. Finally the appropriate form of a GRAPH_NODE record is constructed using the canonical definition and the record is linked to the symbol table entry being defined.

The second type of BUILD command constructs a picture by defining several components and linking them together in a chain that defines the BUILD target. An element of the picture chain is constructed for each component of the picture as it is encountered in the internal representation of the BUILD command. If the new component is unnamed, the executor constructs a dummy symbol table entry for it; if the component is named, there is already an entry for it. When the executor has located an entry for a primitive picture component, the definition is constructed by the method that was described for the first type of BUILD command. Then a picture chain element pointing to the symbol table entry is constructed and added to the chain. If the component is a previously defined picture, a picture chain element for that component, containing a pointer to the symbol table entry which defines the picture, is added to the chain. Finally, the chain is linked to the symbol

table entry for the target.

The third type of BUILD command builds a new definition for a previously defined picture. In some cases the new picture may have the old picture as one of its components. The first step in executing this type of build command is to save the original picture in an accessable location so that it is not lost. To do this the executor constructs a dummy symbol table entry and sets a flag that will indicate whether the original picture has been referenced by the new picture. Then the executor constructs a definition of the new picture. If a component is a reference to, or a transformation of, the original picture, it is defined as a reference to the dummy symbol table entry where the original was saved, and the flag is reset to show that the original picture was referenced. If the original is never referenced, the dummy symbol table entry is deleted after the new picture is completely defined. All pictures that were based on the original definition are thereafter based on the new definition.

DELETE Command

To execute a DELETE command, the executor returns each record in the definition of the symbol table entry of the specified identifier to the appropriate list in the free data space structure. The symbol table entry itself is not deleted.

DRAW and ERASE Commands

The DRAW command translates the definition of a picture stored in the symbol table into a sequence of commands to the output device driver. There are two output commands for picture display, MOVE(X,Y) and VECTOR(X,Y). The first may be interpreted as a command to move (the display cursor) to the start point of a line segment and the second as a command to draw a vector from the start point to the end point of a line segment. The picture to be drawn is approximated by a set of straight line segments and the sequence of output commands is composed of a MOVE command and a VECTOR command for each line segment. In practice, when the start point of a line segment is the same as the end point of the preceding segment, the intervening MOVE output command may be omitted.

The ERASE command makes similar use of the information stored in the symbol table. The executor produces an output command which indicates that a picture is to be erased rather than displayed. It then retraces the picture and issues the same sequence of MOVE and VECTOR commands as for drawing.

The same algorithm is used to execute both DRAW and ERASE commands. The first step is to send a code to the output device driver indicating whether the picture specified is to be drawn or erased. Then each component

that is encountered as the executor traverses the structure defining the picture is translated to MOVE and VECTOR commands for the output device driver. A component may be a reference to another picture, a transformation of another picture, or a picture primitive. If the component that is encountered is itself a picture, it is recursively drawn or erased. If the component is a transformation of a picture, three steps are necessary. First the definition of the transformation is pushed onto a transformation stack. Next the picture is drawn or erased recursively. Finally the transformation definition is popped off the transformation stack. If the component is a picture primitive, four steps are performed. All the transformation definitions on the transformation stack are evaluated to produce a composite transformation. Next a set of line segments which represent the primitive is calculated from its canonical definition. Then the end points of the line segments are adjusted according to the composite transformation and clipped to the bounds specified by the most recent SETSCREEN command. Finally the adjusted line segments are translated to a sequence of MOVE and VECTOR commands and sent to the output device driver. When all the components have been processed the command is complete.

HTEXT and VTEXT Commands

The HTEXT and VTEXT commands are both executed by one

procedure. The first step is to compact all the text items to be displayed into one text line. Text strings are placed in the text line character by character. Values to be displayed are converted to a character string of appropriate digits then placed in the text line. The decimal precision of real values is arbitrarily limited to four digits. After the text line has been constructed, the executor sends to the output device driver 1) the code for HTEXT or VTEXT, 2) a command to move to the orientation point, and 3) the text line.

The output commands for drawing and erasing pictures and for text display are buffered and sent to the output device driver. The output device driver runs concurrently with the executor, but in the output process partition. It is responsible for receiving and translating the device independent output commands from the executor and for producing the corresponding device dependent commands for driving a specific device.

SETSCREEN Command

To execute the SETSCREEN command, the executor evaluates the expressions defining the orientation point and the clipping bounds and stores their values in a special entry of the symbol table for that purpose.

BEGIN Blocks

The BEGIN block is represented internally by a statement

chain where each element of the chain points to a single statement record. To execute the BEGIN block, the executor steps through the statement chain and recursively executes the statement record pointed to by each element of the chain as it is encountered.

IF_THEN_ELSE Command

There are two steps in the execution of an IF_THEN_ELSE command. First the executor evaluates the boolean expression pointed to by IF_BOOL. Then the appropriate branch of the command is executed. If the boolean value is true the statement record pointed to by IF_TRUE is recursively executed. Otherwise, if the boolean expression is false, the statement record pointed to by IF_FALSE is executed.

WHILE_DO Command

Execution of the WHILE_DO command involves repetition of two steps. The first step is to evaluate the boolean expression pointed to by WHILE_BOOL. If that value is true, the executor recursively executes the command pointed to by WHILE_TRUE and then repeats the evaluation of the boolean expression. These two steps are repeated until the value of the boolean expression becomes false.

Assignment Command

The assignment command is executed in two steps. First the expression pointed to by SOURCE is evaluated. Then the

result of the evaluation is added to the symbol table entry for the target of the assignment.

HALT Command

When the HALT command is executed, the executor sets the status indicator to show that a HALT has occurred. If the HALT command was embedded in a BEGIN block, the remaining commands are ignored. Control of the job process partition is returned immediately to the parser which is responsible for terminating the PIGLI system.

NULL Command

A NULL command is represented by a keyword in the internal command structure. When it is encountered, usually in one of the branches of an IF_THEN_ELSE command, no execution is performed.

EXECUTE, LOGOFF, and LOGON Commands

The EXECUTE, LOGOFF, and LOGON commands require similar processing. If an EXECUTE or LOGON command is executed, the executor opens and initializes the file specified by the identifier in the command structure. These files, either log files or exec files, exist under the SOLO operating system. If a LOGOFF command is executed, the executor closes the log file that is open. Only one log file or one exec file may be open at a time so there is no ambiguity as to which file is meant to be closed. After the appropriate file has been opened or closed, the executor sets the status

indicator to reflect the desired changes in command source or output destination.

LOAD Command

The LOAD command is used to replace the device driver that resides in the output process partition. The first step in executing this command is to terminate the output device driver that is currently running. Then the executor loads and initializes the new output device driver that is specified in the LOAD command.

LIST Command

There are two different methods for executing a LIST command, depending on whether the identifiers of a specific type of symbol table entry are requested, or whether the definition of a particular identifier is desired. To obtain a list of composed pictures or of the named instances of a specific type of symbol table entry, the executor steps through the symbol table one entry at a time. If the entry is of the requested type (PICTURE, LINE, CIRCLE, FIGURE, CURVE, ARC, TRANS, TURN, or SCALE), and it is a named entry, the name is displayed on the system console. To list the definition of a particular entry, the executor must first locate that entry in the symbol table. If the entry is a type other than PICTURE, the executor displays its type and its canonical definition on the system console, labeled to be easily understood. If the type of the identifier is

PICTURE, the executor must traverse the chain of picture definition components. If an element of the chain is a named component, the executor lists the name of the component and its type. If the component is unnamed, the executor displays the type and the canonical definition of the element. The display produced by a LIST command may be larger than can be displayed at one time on the system console device. To prevent the loss of necessary data the executor keeps track of the number of lines being displayed and does not fill more than a standard display screen (20 lines). To retrieve a subsequent section of displayed information, the user enters a continue prompt and carriage return.

To illustrate the process of executing a command, consider the example given in the parser section of this chapter. The command tag indicates an IF_THEN_ELSE statement is to be executed (see Figure 4.7).

Execution begins by evaluating the expression pointed at by the IF_BOOL pointer. As operands of the expression are encountered, their values are pushed onto a temporary evaluation stack. As operators of the expression are encountered, two values are popped from the stack, the indicated operation is performed, and the result is pushed back onto the stack. When there are no more operators or

operands to be evaluated, the result is popped off the stack. In this case the result is a boolean value to be used to determine the next course of action.

If the value of the boolean expression is true, the executor executes the statement pointed to by IF_TRUE, in this case an assignment statement. For the assignment statement, the expression pointed to as the source is evaluated in the same manner as for IF_BOOL and the result is inserted in the symbol table for the entry indicated as the target. If the value of the boolean expression is false, the IF_FALSE statement is executed. In this example, the NULL statement pointed to by IF_FALSE requires no execution.

Output Device Drivers

As discussed in chapter three, PIGLI is implemented under the SOLO operating system. The parsing/executing functions of PIGLI run in SOLO's job process partition and the output device driver runs in the output process partition. Separating the execution and output functions made it possible to take advantage of the SOLO system's capability of changing output drivers. The PIGLI command LOAD, discussed in chapter two, allows the programmer to change output devices dynamically.

When a PIGLI command is issued that produces graphical display output, it is translated into a sequence of input commands to the device driver. The driver then issues a sequence of output commands to the device. The input commands are uniform for all devices and will be discussed in the first section of this chapter. The output commands are device dependent and will be discussed in the second section.

Device Driver Input

Input for a graphical output device is produced during the execution of four PIGLI commands, DRAW, ERASE, HTEXT, and VTEXT. The DRAW command is translated into a sequence of input commands, each of which can be a MOVE command or a VECTOR command. Both MOVE and VECTOR are absolute commands;

that is, they indicate movement of the display device cursor from its current position to a new absolute position, rather than to a position obtained by adding an increment to the former cursor position. The parameters passed to MOVE and VECTOR are two real values representing the x-coordinate and y-coordinate of a point in two-space. The point is interpreted as a position on the output device, relative to an origin at the lower left corner and measured in inches. MOVE(x,y) specifies that the cursor should be positioned at the point (x,y) on the display device. VECTOR(x,y) specifies that a straight line should be drawn from the current position of the cursor to the point (x,y). A sequence of MOVE and VECTOR commands causes a picture to appear on the screen.

On a display device that allows selective erasure, removing a portion of a display is accomplished by retracing the drawn picture while the display device is in the erase mode. To change the mode of the device, status commands are used; WRITE for drawing a picture and ERASE for erasing a picture. When it is necessary to erase an entire display, it is more efficient to clear the device than to retrace all the displayed pictures. The CLEAR command is used to erase the entire display surface.

Text can be displayed both horizontally, left to right, and vertically, top to bottom, using the PIGLI commands

HTEXT and VTEXT. These same commands are used as input commands to the display device driver. The input commands HTEXT and VTEXT have as a parameter the string of text characters to be displayed. A general form of the device driver input from each PIGLI display command is given in figure 5.1.

PIGLI / Device Driver Interface

The device driver input commands are buffered into pages of information before being sent to the output process. Each page used to communicate with the output process is a block of integers 512 bytes long (256 integers). Every command is assigned a unique integer value as follows:

ERASE status	0
WRITE status	1
MOVE	2
VECTOR	3
HTEXT	4
VTEXT	5
CLEAR screen	6
End Of Transmission	7

The two real-valued coordinates of points for MOVE and VECTOR commands are each packed into four integer blocks (8

PIGLI Command	Device Driver Input Commands
DRAW	WRITE; MOVE(x1,y1); VECTOR(x2,y2); ... MOVE(0,0); EOT.
ERASE	ERASE; MOVE(x1,y1); VECTOR(x2,y2); ... MOVE(0,0); EOT.
HTEXT	WRITE; MOVE(x1,y1); HTEXT(string); MOVE(0,0); EOT.
VTEXT	WRITE; MOVE(x1,y1); VTEXT(string); MOVE(0,0); EOT.
ERASE SCREEN	CLEAR; MOVE(0,0); EOT.

Figure 5.1: Input Commands to the Output Device Driver Generated by each Display Producing PIGLI Command

bytes). The character string parameter of HTEXT and VTEXT, normally 132 characters long, is packed into 66 integer blocks (two characters per integer block). A page is sent to the output process as soon as it is full, and processing begins on a new page. The SOLO system maintains synchronization between the job process and the output process so that no data is lost during page transfers. When all the device driver input commands have been generated for the PIGLI command requiring display, the End Of Transmission command is added to the current page and the page is transmitted to the output device driver.

The output process receives its input page of commands as an array of 256 integers. It then processes the commands, breaking out command parameters as necessary. Commands ERASE status, WRITE status, CLEAR screen and End Of Transmission (integers 0,1,6, and 7 respectively) do not have accompanying parameters. Commands MOVE, VECTOR, HTEXT, and VTEXT are followed in the array by parameters as described above. The following section discusses the device drivers for specific output devices.

Computek 300 GT Device Driver

PIGLI was originally implemented with graphical output limited to a COMPUTEK 300 GT CRT. The Computek is a storage tube, thus eliminating the need for a refresh cycle in the

output driver. It has three display modes, alphanumeric mode for character display, four byte absolute mode for absolute vector generation, and one byte incremental mode for incremental vector generation. The Computek terminal allows selective erasure or total screen erasure. Selective erasure is accomplished by putting the terminal in erase status and retracing the lines to be erased.

Normally the terminal is in character mode. When a printable ASCII character is sent to the terminal while it is in character mode, the character is displayed at the current cursor position. When certain unprintable ASCII characters are sent to the terminal while it is in character mode, they are interpreted as control characters and produce special actions. Table 5.1 shows the decimal and binary equivalents of each of the ASCII control characters and the functions they specify.

The Computek terminal must be in character mode to execute the input commands WRITE, ERASE, CLEAR, HTEXT, and VTEXT. To execute the commands WRITE, ERASE, or CLEAR, the appropriate ASCII character, as shown in Table 5.1, is transmitted to the terminal. To execute HTEXT or VTEXT, the characters of the text string are transmitted. In the case of VTEXT, between each text character the ASCII characters for LINEFEED and BACKSPACE are transmitted.

The terminal must be in four byte absolute mode to

Decimal	Binary	Task Specified
8	00001000	BACKSPACE
10	00001010	LINEFEED
12	00001100	HOME / ERASE
14	00001110	ERASE STATUS
15	00001111	WRITE STATUS
28	00011100	FOUR BYTE ABSOLUTE MODE

Table 5.1: The Decimal and Binary Equivalents of the ASCII Characters Used to Control the Computek 300/GT

execute the input commands MOVE and VECTOR. To execute either MOVE or VECTOR, four ASCII characters are transmitted. The decimal equivalents of these four characters are computed by the output device driver using the coordinate values of the point parameter of the command, the type of the command (MOVE or VECTOR), and other details that are unique to the Computek terminal.

When it is necessary to change the terminal mode from character to four byte absolute, the ASCII character represented by decimal value 28 is transmitted. When the change is from four byte absolute to character, four ASCII characters, corresponding to decimal 64, decimal 0, decimal 0, decimal 0, are transmitted.

The Computek driver uses the SOLO IOTRANSFER facility to transmit data to the Computek terminal. For this procedure, the ASCII characters produced by executing the display device driver input commands are buffered into a page array (512 characters). The transfer takes place when the page is completely full or when the End Of Transmission command is encountered.

Conclusion

This report presents the design of an interactive graphics language and aspects of its implementation as a portable interpreper. The strong points of the PIGLI system include interactive capabilities, general purpose programming capabilities, hierarchical generation of pictures from primitives, built-in referencing functions, debugging facilities, exec files, output device independence, and portability. There are three final areas to be discussed in the chapter; the elements of the described language that were not included in the initial implementation, the desirable additions to the initial design, and the relationship of the initial design to the recently published standards for a core graphics system.

The system, as initially implemented, does not contain all the elements that were discussed in this paper. Specifically, those unimplemented items are

1) the building and drawing of curve and arc primitives,
2) the LIST debugging facilities,
3) the SETSCREEN command and clipping facilities, and
4) part of the built-in reference functions.

These features were not implemented at the time this report was completed but will hopefully be added in the near

future.

Several aspects of the PIGLI language and its implementation are sufficient for a preliminary design but should be reworked in the future in the interests of elegance and efficiency.

Currently, the exec file facilities offer the user access to predefined files of PIGLI commands. Any variable information needed within the file to accomplish the desired task must be assigned to the variable used in the exec file before the exec file is invoked. This requires a deeper knowledge of the contents of the exec file than should be needed. A more reasonable method to permit the use of variable information in an exec file is to provide a parameter list for passing information when invoking an exec file. Exec files are also restricted in that they cannot invoke another exec file, although situations are easily imagined where this ability would be an advantage.

PIGLI includes only two-dimensional line drawing capabilities and transformations for scaling, translating, and rotating pictures. The language should be expanded to include three-dimensional picture descriptions, as well. Three dimensions require changes in the transformations that already exist and the addition of perspective viewing transformations and hidden line removal. Further work might also be desirable to add wire network surface

representations.

In designing this language, no emphasis was placed on supplying input facilities other than keyboard command input, primarily because such devices were not available. The commands themselves are often verbose and the parser requires the syntax to be perfect before accepting a command. The parser should be revised to allow abbreviations of keywords. Incomplete commands should result in a prompt to supply the missing information. Error handling should allow the user to build on what was correct rather than repeat the entire command. Command punctuation is currently ignored after the command is parsed. Future revision should make punctuation optional. These additional features would help both the novice user who is prone to errors and the experienced user who wants shortcuts.

A complication arises in the design of hierarchic picture construction facilities when a picture is redefined in terms of itself. An arithmetic analogy would be the evaluation of the assignment statement A := A + 1. In the assignment, the value of the variable A is redefined in terms of its previous value. For pictures, the analogy holds unless there is another picture defined in terms of the original definition of picture A. The decision had to be made regarding the effects of the redefinition of a component of a picture in PIGLI. It was arbitrarily decided

that changes in a component would also change previously defined pictures based on that component. It would be advantageous to allow the user to specify that a picture definition is static and cannot be changed if the definition of a component of that picture changes. Other aspects of assignment of picture data types are discussed by Schrack (1976).

One of the strong points of PIGLI is the ease with which the system may be adapted to new output devices. As graphics devices are acquired, drivers should be added to the system.

In the past few years there has been increasing discussion of the need for graphic standards as a guide to graphic language designers. The design of the PIGLI language was completed prior to the final report of the committee engaged in establishing the proposed graphic standards (Status, 1977). Their report distinguishes between two types of graphics systems, a graphic viewing system and a graphic modelling system. The graphic viewing system is primarily concerned with the mechanics of displaying a graphic object on some hardware device, that is, picture generation. The graphic modelling system is concerned with the necessary functions for constructing and manipulating objects to be displayed. The PIGLI language focuses on the modelling aspects of picture construction and

manipulation, and contains only a small portion of the display generation facilities described in the standards report. Consequently, the graphic standards presented apply only to the area of PIGLI where the execution of commands requiring graphical output interfaces with the output device driver.

REFERENCES

Bassman, M.J., "APLBAGS - An APL Basic Graphics Subroutine Package for Tektronix 4013 Storage Tube Terminal", Computer Graphics, Vol. 7, No. 4, 1973.

Boullier, P., J. Gros, P. Jancene, A. Lemaire, F. Prusker, E. Saltel, "METAVISU - A General Purpose Graphic System", Graphic Languages, Proc. of IFIP Working Conf. on Graphic Languages, North-Holland Press, 1972.

Brinch Hansen, P., "The SOLO Operating System", Software - Practice and Experience, Vol. 6, No. 2, April - June, 1976.

Caruthers, L.C., J. van der Bos, A. van Dam, "A Device Independent General Purpose Graphic System for Stand-Alone and Satellite Graphics", Proc. of SIGGRAPH '77, Computer Graphics, Vol. 11, No. 2, 1977.

Dewar, R.B.K., SPITBOL, Version 2.0, Chicago, Illinois, Illinois Institute of Tecxhnology, 1971.

DISSPLA Beginners/Intermediate Manual, Integrated Software Systems Corp., San Diego, California, 1970.

Eastman, C., M. Henrion, "GLIDE - A Language for Design Information Systems",Proc. of SIGGRAPH '77, Computer Graphics, Vol. 11, No. 2, 1977.

Giloi, W.K., "On High-Level Programming Systems for Structured Display Programming", Proc. of SIGGRAPH '75, Computer Graphics, Vol. 9, No. 1, 1975.

Giloi, W.K., J. Encarnacao, "APLG - An APL Based System for Interactive Computer Graphics", Proc. of AFIPS NCC, 1977.

GINO-F, The General Purpose Graphics Package Reference Manual, Computer Aided Design Centre, Cambridge, England, 1975.

Gries, D., Compiler Construction for Digital Computers, Johm Wiley and Sons, Inc., New York, 1971.

Hartman, A.C., A Concurrent PASCAL Compiler for Minicomputers, Thesis, California Institute of Technology, Pasadena, California, 1976.

Hurwitz, A., J.P. Citron, J.B. Yeaton, "GRAF - GRaphic

Addition to FORTRAN", Proc. AFIPS SJCC, Thompson Books, Washington, D.C., 1967.

Knowlton, K.C., "EXPLOR - A Generator of Images from Explicit Patterns, Local Operations, and Randomness", Proc. of UAIDE Annual Meeting, Stromberg Datagraphix, 1970.

Kulsrud, H.E., "A General Purpose Graphic Language", CACM, Vol. 11, No. 4, 1968.

Neal, D., G. Anderson, J. Ratliff, V. Wallentine, KSU Implementation of Concurrent PASCAL - A Reference Manual, Tech. Rep. 76-16, Kansas State University Dept. of Computer Science, 1977.

Newman, W.M., "Display Procedures", CACM, Vol. 14, No. 10, 1971.

O'Brien, C.D., H.G. Bown, "IMAGE - A Language for the Interactive Manipulation of a Graphic Environment", Proc. of SIGGRAPH '75, Computer graphics, Vol. 9, No. 1, 1975.

Robbins, F.E., W.G. Green, "WAVE - Interactive Color Graphics for Waveform Analysis", Proc. of SIGGRAPH '75, Computer Graphics, Vol. 9, No. 1, 1975.

Schrack, G.F., "On the Semantics of the Assignment Statement of High Level Graphical Languages", Proc. of SIGGRAPH '76, Computer Graphics, Vol. 10, No. 2, 1976.

Shapiro, L.G., "ESP3 - A High-Level Graphics Language", Proc. of SIGGRAPH '75, Computer Graphics, Vol. 9, No. 1, 1975.

Shapiro, L.G., R.J. Baron, "ESP3 - A Language for Pattern Description and a System for Pattern Recognition", IEEE Transactions on Software Engineering, Vol. SE-3, No. 2, 1977.

"Status Report of the Graphic Standards Planning Committee of ACM/SIGGRAPH", Computer Graphics, Vol. 11, No. 3, 1977.

Streit, E., "VIP - A Conversational System for Computer Aided Graphics", Pertinent Concepts in Computer Graphics, Proc. of Second University of Illinois Conf. on Computer Graphics, University of Illinois Press, 1969.

Sutherland, I.E., SKETCHPAD - A Man-Machine Graphical Communication System, MIT Lincoln Laboratory, Tech. Rep. 296, 1965.

Van Dam, A., D. Evans, "A Compact Data Structure for Storing, Retrieving, and Manipulating Line Drawings", Proc. AFIPS SJCC, Thompson Books, Washington, D.C., 1967.

Wallace, V.L., GRASP - A PL/1 Oriented Machine Independent Graphics Structure Handler, Tech. Rep., University of North Carolina Dept. of Computer Science, Chapel Hill, N.C., 1974.

Wexeblat, R.L., H.A. Freedman, "The MULTILANG On-Line Programming System", Proc. of AFIPS SJCC, Thompson Books, Washington, D.C., 1967.

Williams, R., "A General Purpose Graphical Language", Graphic Languages, Proc. of IFIP Working Conf. of Graphic Languages, North-Holland Press, 1972.

Syntax and Semantics

This appendix discusses the precise syntax and semantics of PIGLI commands. The syntax diagrams use an extended Backus-Naur form as discussed by Gries (1971). In each syntactic construct the terminals are keywords and special symbols. (PIGLI keywords and special symbols are shown in Table A.1.) Non-terminals are enclosed in angular brackets, < and >. Constructs that may be repeated an indefinite number of times are enclosed in metabrackets, { and }. Superscript and subscript numbers on the right metabracket indicate the upper and lower bounds allowed for repeated constructs. Where the upper bound is theoretically infinitely large, it may in fact be constrained by the limits of a particular hardware configuration. When a non-terminal may produce one of several possible constructs, the alternatives are separated by ¦.

Identifiers, strings, and real and integer numbers are elements of several constructs. Identifiers may name arithmetic variables, pictures, picture components, or external files (for EXEC files, saved picture display files, and alternate display device drivers). Identifiers may be one to twelve characters long, beginning with an alphabetic character and containing only alphabetic and numeric characters. Keywords are reserved identifiers, which may be used only for their special purposes.

KEYWORD TOKEN CODES

BEGIN	SAVE	CIRCLE	DEG
REAL	LOAD	ARC	ANGLE
INTEGER	NOT	FIGURE	FRDEG
IF	AND	CURVE	TODEG
WHILE	OR	TRANS	FORM
EXECUTE	DIV	TURN	OPEN
ENTER	MOD	SCALE	CLOSED
EXIT	CONV	START	DIR
LOGON	TRUNC	MID	CW
LOGOFF	VALU	ENDPT	CCW
NULL	XVAL	CENTER	POINT
HALT	YVAL	TOP	PNT
BUILD	THEN	BOT	LTOP
CHANGE	ELSE	LEFT	RTOP
DELETE	DO	RIGHT	LBOT
SETSCREEN	END	PDIS	RBOT
DRAW	XLEN	PDEG	TRIGHT
ERASE	YLEN	ABOUT	BRIGHT
LIST	DEF	LENGTH	TLEFT
HTEXT	SCREEN	RADIUS	BLEFT
VTEXT	LINE	FACTOR	

SYMBOL TOKEN CODES

>	-	;	)
<	+	EOM	.
>=	/	;	=>
< =	*	(	:
=	< >	:=	&

Table A.1: PIGLI Keywords and Special Symbols

```
<identifier> ::= <letter> {<letter_or_digit>}
<letter_or_digit> ::= <letter> | <digit>
<letter> ::= A | B | C | D | E | F | G | H | I | J
     | K | L | M | N | O | P | Q | R | S | T | U |
     V | W | X | Y | Z | _
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A string is a sequence of eighty or fewer characters enclosed in single quotes. In the current implementation of PIGLI, any ASCII character may be in a string. If the string is to contain a single quote, it must be written as two single quotes, ' '' '.

```
<string> ::= ' <character> {<character>} '
```

Numbers are unsigned integer or real values.

```
<integer> ::= <digit> {<digit>}
<real> ::= <integer> . <integer>
```

A comment is a sequence of characters enclosed in double quotes. They may be inserted between any two keywords, identifiers, numbers, strings, or special symbols. Comments cannot contain double quotes as part of the body of the comment. No action is associated with a comment.

```
<comment> ::= " <character> {<character>} "
```

A PIGLI program is a sequence of PIGLI commands issued interactively at a programming console. Each command is terminated by a period. The commands fall into four general categories; programming commands, picture construction

commands, picture display commands, and special utility commands.

```
<program> ::= <command> . {<command> .}
<command> ::= <programming_commands> ¦
     <picture_construction_commands> ¦
     <picture_display_commands> ¦
     <special_utility_commands>
```

Programming Commands

Programming commands are for type declaration and assignment of arithmetic variables, and for program control.

```
<programming_commands> ::= <declaration> ¦
     <assignment> ¦ <IF_THEN_ELSE> ¦
     <WHILE_DO> ¦ <BEGIN_END>
```

Identifiers to be used as variables in arithmetic assignment commands must be declared to be of type real or type integer before they are used. Once a variable has been declared, its type is fixed so it may not be used in any other capacity. Identifiers to be used to name pictures are declared by the context in which they appear.

```
<declaration> ::= REAL <variable_list> ¦
     INTEGER <variable_list>
<variable_list> ::= <identifier> {, <identifier>}
```

The assignment command is used to replace the current value of a variable by a new value represented as an expression. The target variable and all variables and constants in the expression must agree in type.

Expressions are sequences of operands (variables and constants), operators, and functions. Each operator or function indicates a specific action on one or more operands which produces a value. Rules of precedence determine the order in which operators and functions are evaluated. The highest precedence is given to parenthetical expressions, conversion functions, and value reference functions. Expressions in parentheses are evaluated recursively according to all precedence rules. The conversion functions CONV and TRUNC are for converting integer values to real and real values to integer, respectively. Value reference functions are built-in functions used to retrieve the numeric values of the attributes of pictures. The value reference functions VALU, XVAL, and YVAL will be more completely discussed below with the picture construction commands.

The second level of precedence includes multiplication and division operators. Real and integer multiplication are indicated by the '*' operator. Real division is performed by the '/' operator. Integer division is done by two operators; DIV produces the integer quotient of integer division and MOD produces the remainder of integer division.

The lowest level of precedence includes addition and subtraction operators. The '+' operator indicates addition for both real and integer values. The '-' operator

indicates subtraction when used with two operands and as a unary minus when used with one operand. When several operations have the same precedence level they are evaluated from left to right.

```
<assignment> ::= <identifier> := <expression>
<expression> ::= - <term> {<aop> <term>} ¦
     <term> {<aop> <term>}
<aop> ::= + ¦ -
<term> ::= <factor> {<mop> <factor>}
<mop> ::= * ¦ / ¦ DIV ¦ MOD
<factor> ::= <conversion_function> ¦
     <value_reference_function> ¦
     ( <expression> ) ¦ <identifier> ¦
     <real> ¦ <integer>
<conversion_function> ::= TRUNC ( <expression> ) ¦
     CONV ( <expression> )
<value_reference_function> ::=
     VALU ( <identifier> , <value_reference> ) ¦
     XVAL ( <identifier> , <point_reference> ) ¦
     YVAL ( <identifier> , <point_reference> )
```

Boolean expressions are used to determine execution alternatives in program control commands. A boolean expression is made of logical expressions and the operators AND, OR, and NOT. Logical expressions are parenthetical boolean expressions or two arithmetic expressions separated by one of the logical operators <, >, =, <=, >=, <>. The arithmetic expressions must be of the same type real or integer. Logical operators have higher precedence than boolean operators. Of the boolean operators, AND and OR have higher precedence than NOT.

```
<boolean_expression> ::=
     NOT <logical_expression>
     {<bop> <logical_expression>} ¦
     <logical_expression>
     {<bop> <logical_expression>}
<bop> ::= AND ¦ OR
<logical_expression> ::= (<boolean_expression>) ¦
     <expression> <lop> <expression>
<lop> ::= < ¦ > ¦ = ¦ <= ¦ >= ¦ <>
```

PIGLI has two programming control commands, the conditional IF_THEN_ELSE command and the repetitive WHILE_DO command. The IF_THEN_ELSE command contains a boolean expression and two alternate command clauses. If the evaluation of the boolean expression is true, the THEN clause of the command is executed. If it is false, the ELSE clause is executed. Both command clauses must be present to avoid a syntax error.

```
<IF_THEN_ELSE> ::= IF <boolean_expression> THEN
     <command> ELSE <command>
```

The repetitive WHILE_DO command contains a boolean expression and a command clause. If the evaluation of the boolean expression is true the command clause is executed until the boolean expression becomes false. It is possible that the command clause is never executed or that it is executed infinitely within the limits of the computer on which PIGLI is implemented.

```
<WHILE_DO> ::=
     WHILE <boolean_expression> DO <command>
```

The command clauses of the IF_THEN_ELSE command and the WHILE_DO command may not contain a declaration command.

The BEGIN_END command is used to compound several commands to be used as a single PIGLI command. Each PIGLI command within the compound command is terminated by a semicolon, ';'.

```
<BEGIN_END> ::=
     BEGIN <command> ; {<command> ; } END
```

Picture Construction Commands

Picture construction commands are used to define two-dimensional line drawings and to delete them when they are no longer wanted.

```
<picture_construction_commands> ::=
     <BUILD> | <DELETE>
```

A picture may be made of several picture elements. To BUILD the definition of a picture, the description of elements of the picture is associated with a picture identifier. The '&' operator indicates the composition of two or more picture elements.

```
<BUILD> ::= BUILD <identifier> :=
     <picture_element> { & <picture_element> }
```

There are three general types of picture elements. Each

element may be a picture that is already defined, a transformation of a picture that is already defined, or a picture primitive, that is, a generalized building block of PIGLI pictures. To use a previously defined picture as an element of a new picture, the identifier to which the first picture was assigned is specified in the definition of the new picture. Transformations of pictures and picture primitives are specified by precise definitions. The unit of measure used to describe transformations and picture primitives is inches. Transformations and picture primitives may optionally have individual names apart from the name of the whole picture. These names permit the individual elements of the picture to be referenced. Names are assigned to picture elements by preceding the definition of the element with the desired identifier and the immediate assignment operator ':'.

```
<picture_element> ::= <picture_identifier> ¦
     <transformation> ¦
     <identifier> := <transformation> ¦
     <primitive> ¦ <identifier> := <primitive>
<picture_identifier> ::= <identifier>
```

A transformation is based on a picture that was previously defined. A keyword indicates which transformation is desired (translation, rotation, or scaling), followed by the definition of that transformation enclosed in parentheses. The definition includes the

picture to be transformed, specified by its picture identifier, followed by specific parameters for each type of transformation. For translation (TRANS), the parameters are two points, separated by the operator '=>', which define the translation. The first point indicates the point in the base picture that will be mapped to the second point after translation. Rotation (TURN) has three parameters: ABOUT (a point in the base picture about which to rotate the picture), ANGLE or DEG (the amount of rotation in degrees to rotate the picture), and DIR (the direction to rotate the picture, either clockwise or counterclockwise (CW or CCW)). Scaling has two parameters: ABOUT (a point in the base picture about which to scale the picture) and FACTOR (the factor by which to scale the picture). Scaling factors may be greater than one to increase picture size or less than one to decrease size. A scaling factor equal to one does not affect the size of the picture.

For rotation and scaling, the parameters are keyword parameters. They do not need to be in any fixed order; they are identified by the keyword, not by position in the parameter list. Keyword parameters are separated from each other by commas.

```
<transform> ::= TRANS ( <picture_identifier> ,
     <point> => <point> ) ¦
     TURN ( <picture_identifier>
     { , <TURN_parameters> } ) ¦
     SCALE ( <picture_identifier>
     { , <SCALE_parameters> } )
<TURN_parameters> ::= ABOUT = <point> ¦
      ANGLE  = <expression> ¦
      DEG
             CW
     DIR =  CCW
<SCALE_parameters> ::= ABOUT = <point> ¦
     FACTOR = <expression>
```

The values that are assigned to transformation parameters may be points, expressions or keywords. When a parameter requires a point as its assigned value it may be defined explicitly using the keyword PNT and the x and y coordinates of the point represented by expressions. Or it may be defined by a point reference function. A point reference function is a built-in function used to retrieve the coordinates of point attributes of picture primitives. (The point reference function POINT will be discussed later.) Expressions used as values of transformation parameters follow the same rules as expressions used in arithmetic assignment commands. When the value of a parameter is a keyword the possible choices are those given in the syntax diagrams.

```
<point> ::= PNT ( <expression> , <expression> ) ¦
     <point_reference_function>
<point_reference_function> ::=
     POINT ( <identifier> , <point_reference> )
```

Picture primitives are the building blocks of the pictures constructed using PIGLI. They are descriptions of simple two-dimensional line drawings from which more complex two-dimensional drawings may be composed. There are five picture primitives in PIGLI; a straight LINE, a CIRCLE, an ARC of a circle, a FIGURE made of points connected by straight lines, and a CURVE made of points connected by curved lines. An instance of a picture primitive is defined by the primitive keyword followed by a parameter list enclosed in parentheses.

```
<primitive> ::= LINE ( <LINE_parameters> ) |
     CIRCLE ( <CIRCLE_parameters> ) |
     ARC ( <ARC_parameters> ) |
      FIGURE  ( <FIGURE-CURVE_parameters> )
       CURVE
```

The parameter list of a PIGLI picture primitive defines the size and position of the primitive. There is a finite set of keyword parameters that may be specified for each type of picture primitive (See Figure A.1). These keyword parameters do not need to be in any fixed order within the parameter list. Each parameter defines an attribute of the primitive. LINEs, CIRCLEs, and ARCs have several attributes. Different subsets of these attributes may be used to define a particular instance of a LINE, a CIRCLE, or an ARC. To avoid specification redundancy, only the attributes of one accepted subset may be used to define an

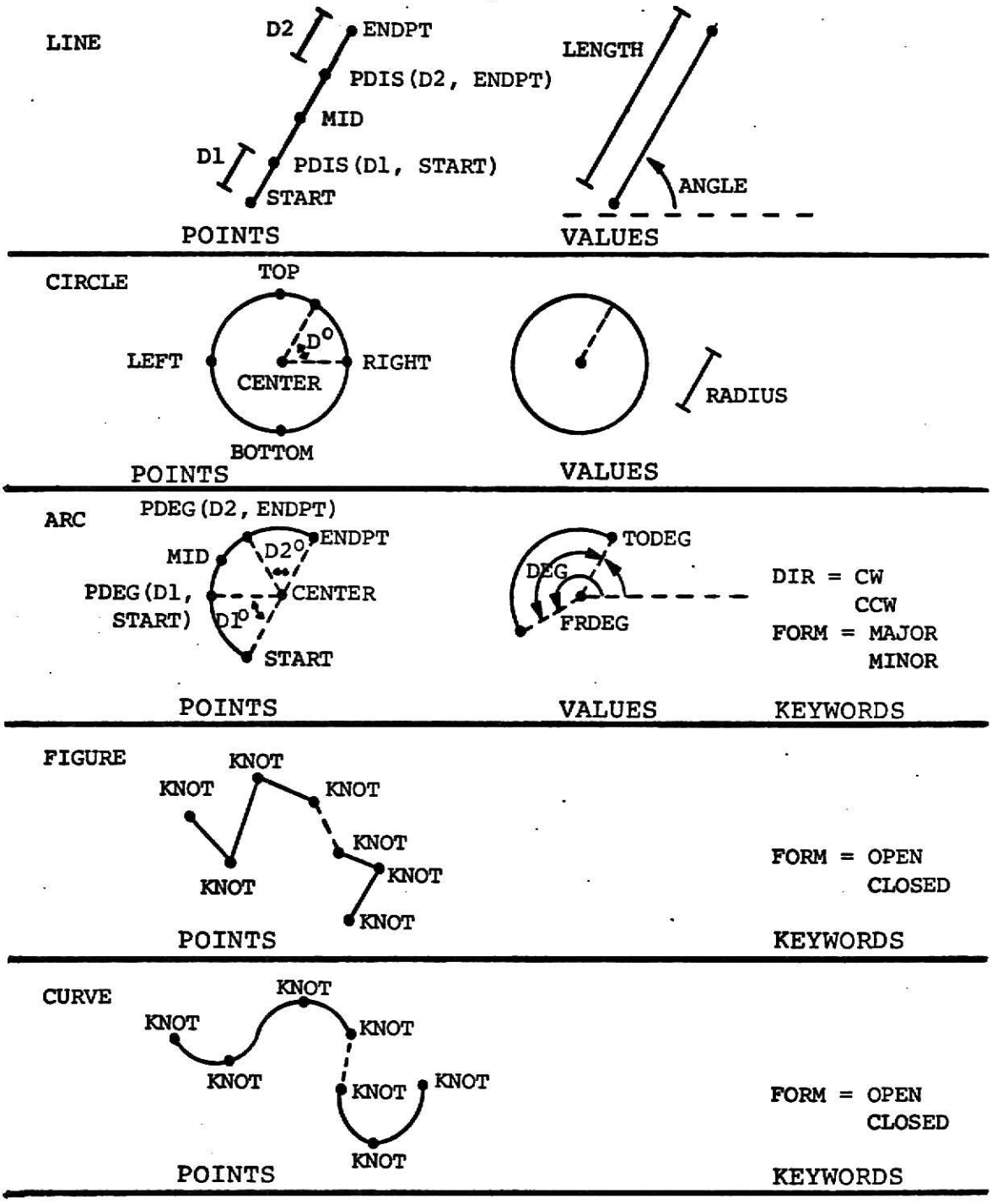


Figure A.1: The Specification Attributes for Picture Primitives

instance of a picture primitive and all of the attributes in that subset must be used. The accepted attribute subsets that may be used to define LINEs, CIRCLEs, and ARCs are given in Table A.2.

The values that are assigned to the attributes of picture primitives will be points, arithmetic expressions, or keywords. These values are specified for picture primitives the same way that they are specified for transformation parameters.

The values of LINE attributes are points and expressions. The START point, the MID point, and the ENDPT point correspond to physical points on a LINE. The LENGTH of a LINE is the distance between the START point and the ENDPT point. The ANGLE or DEG of a LINE is the number of degrees in the counterclockwise direction from a horizontal line through the START point of the LINE. PDIS is a point on the LINE defined by giving the length from either the START point or the ENDPT point.

```
<LINE_parameters> ::= <LINE_keyword_parameter>
     { , <LINE_keyword_parameter> }
<LINE_keyword_parameter> ::=
      START
       MID    = <point> ¦
      ENDPT                    START
     PDIS ( <expression> ,     ENDPT   ) = <point> ¦
      ANGLE  = <expression> ¦
       DEG
     LENGTH = <expression>
```

The values assigned to CIRCLE attributes are also points

LINEs are defined by the following attribute combinations:

1) START, END
2) START, MID
3) END, MID
4) START, PDIS(D, END)
5) END, PDIS(D, START)
6) MID, PDIS(D, END)
7) MID, PDIS(D, START)
8) START, ANGLE, LENGTH
9) MID, ANGLE, LENGTH
10) END, ANGLE, LENGTH
11) PDIS(D, START), ANGLE, LENGTH
12) PDIS(D, END), ANGLE, LENGTH

A CIRCLE point is one of the following six attributes:
TOP, BOT, LEFT, RIGHT, CENTER, PDEG(D)

A CIRCLE is defined by
any CIRCLE point, RADIUS

An ARC point is one of the following:
START, MID, END, CENTER, PDEG(D, START), PDEG(D, END)

Arcs are defined by one of the following attribute combinations:

1) any ARC point, TODEG, FRDEG, RADIUS, DIR
2) any ARC point, TODEG, DEG, RADIUS, DIR
3) any ARC point, FRDEG, DEG, RADIUS, DIR
4) START, END, RADIUS, DIR, FORM
5) START, MID, RADIUS, DIR, FORM
6) END, MID, RADIUS, DIR, FORM
7) END, PDEG(D, START), RADIUS, DIR, FORM
8) START, PDEG(D, END), RADIUS, DIR, FORM
9) MID, PDEG(D, START), RADIUS, DIR, FORM
10) MID, PDEG(D, END), RADIUS, DIR, FORM
11) PDEG(D1, START), PDEG(D2, END), RADIUS, DIR, FORM

FIGUREs and CURVEs are defined by a set of points called KNOTS and an attribute called FORM that may be either OPEN or CLOSED:

1) KNOTS, FORM

Table A.2: Legal Definitions of all PIGLI Picture Primitives

and expressions. CENTER, TOP, BOT, LEFT, and RIGHT correspond to physical points on or in a CIRCLE. RADIUS is the distance between the CENTER point and any point on the CIRCLE. PDEG is a point on the CIRCLE defined by giving the distance in degrees in the counterclockwise direction from the RIGHT point.

```
<CIRCLE_parameters> ::=
     <CIRCLE_keyword_parameters>
     { , <CIRCLE_keyword_parameter> }
<CIRCLE_keyword_parameter> ::=
      CENTER
       TOP
       BOT     = <point> |
       LEFT
       RIGHT
     PDEG ( <expression> ) = <point> |
     RADIUS = <expression>
```

The values of ARC attributes are points, expressions, and keywords. The START point, the MID point, and the ENDPT point correspond to physical points on the ARC. The CENTER point is the center of the circle of which the ARC is a part. PDEG is a point on the ARC defined by giving the distance in degrees from either the START point or the ENDPT point of the curve. The values FRDEG, TODEG, and DEG are angles measured in degrees in the counterclockwise direction. FRDEG measures the distance of the START point from the horizontal line extending from the CENTER point of the ARC to the right. TODEG measures the distance of the ENDPT point from the same horizontal line. DEG measures the

distance from the TODEG point to the FRDEG point, that is the length of the ARC in degrees. The RADIUS value is the distance from the CENTER point of the ARC to a point on the ARC. The DIR is the direction from the START point to the ENDPT point of the ARC. The keyword values of DIR may be CW for clockwise or CCW for counterclockwise.

```
<ARC_parameters> ::= <ARC_keyword_parameter>
     { , <ARC_keyword_parameter> }
<ARC_keyword_parameter> ::=
      START
       MID     = <point> ¦
      ENDPT
      CENTER                  START
     PDEG ( <expression> ,    ENDPT   ) = <point> ¦
      TODEG
      FRDEG   = <expression> ¦
       DEG
     RADIUS = <expression> ¦
     DIR =  CW
            CCW
```

The values of FIGURE and CURVE attributes are points, and keywords. KNOTS are the nodes of the FIGURE or CURVE listed in the order they are encountered when tracing the FIGURE or CURVE. The FORM of a FIGURE or CURVE may be either OPEN or CLOSED. When the FORM is OPEN, there is no connecting line between the last KNOT and the first KNOT of the FIGURE or CURVE.

```
<FIGURE-CURVE_parameters> ::=
     <FIGURE-CURVE_keyword_parameter>
     { , <FIGURE-CURVE_keyword_parameter> }
<FIGURE-CURVE_keyword_parameter> ::=
     FORM =  OPEN     |
             CLOSED
     KNOTS ( <point> { , <point> } )
```

Reference functions are built-in functions for retrieving some of the attributes of picture primitives that have already been defined. In any production where a point or an expression may be used, PIGLI allows point reference functions and value reference functions, respectively, to be used. Since points may be specified by the values of their x and y coordinates, represented by expressions, value reference functions may be used in the definition of points also. Reference functions have as parameters the identifier of the picture primitive and the attribute of that picture primitive whose value is to be returned. If the picture primitive is a component of a complex picture, it must be a named component in order to have attributes retrieved for it.

The POINT reference function is used to return the values of both the x coordinate and the y coordinate of points from the stored description of a picture primitive. The value of any legal, point-valued attribute of aa picture primitive may be retrieved by the POINT reference function rather than just the attributes that were actually used. For example, if a circle has been defined by the position of

its TOP point and the value of its RADIUS, any of the points TOP, BOT, LEFT, RIGHT, and CENTER may be retrieved using the POINT reference function. XVAL and YVAL are value reference functions which retrieve the x coordinate and the y coordinate, respectively, of points from the stored description of a picture primitive.

The VALU reference function returns the values of picture primitive attributes that were defined by expressions. Again, the attributes to be retrieved do not need to be the same as the attributes used to describe the picture primitive, but they do need to be legal attributes of the definition of the desired picture primitive. For example, it is not correct to retrieve the value of RADIUS from a picture primitive of type LINE.

```
<point_reference_function> ::=
     POINT ( <identifier> , <point_reference> )
<point_reference> ::= START ¦ MID ¦ ENDPT ¦
     TOP ¦ BOT ¦ LEFT ¦ RIGHT ¦ CENTER

<value_reference_function> ::=
     XVAL ( <identifier> , <point_reference> ) ¦
     YVAL ( <identifier> , <point_reference> ) ¦
     VALU ( <identifier> , <value_reference> )
<value_reference> ::= LENGTH ¦ ANGLE ¦ DEG ¦
     RADIUS ¦ FRDEG ¦ TODEG
```

The DELETE command is used to remove a description of a picture referenced by its picture identifier, from the data structure where it was stored. There is no checking done to insure that other picture descriptions do not depend on the

picture to be deleted.

<DELETE> ::= DELETE <picture_identifier>

Picture Display Commands

Picture display commands are for drawing and erasing pictures, for displaying text, and for adjusting window and clipping boundaries.

```
<picture_display_commands> ::=
     <DRAW> ¦ <ERASE> ¦
     <HTEXT> ¦ <VTEXT> ¦
     <SETSCREEN>
```

To draw or erase a picture, the name of the picture is specified in a DRAW or ERASE command. Picture elements with individual names may be drawn or erased also. An accummulated picture display may be obtained by issuing several DRAW commands in succession.

The picture identifier of a picture is entered in the SCREEN file when it is drawn and removed from the file when it is erased. ERASE SCREEN clears the display device and empties the SCREEN file. DRAW SCREEN is designed to allow the programmer to reproduce a set of drawn pictures. For example, pictures may be interactively constructed on a graphics CRT, then redrawn using the DRAW SCREEN command after changing the output device with the LOAD command.

<DRAW> ::= DRAW <identifier> ¦ DRAW SCREEN

```
<ERASE> ::= ERASE <identifier> ¦ ERASE SCREEN
```

The HTEXT and VTEXT commands are for displaying text material either horizontally or vertically. The text commands are followed by an orientation point for the beginning of the text to be displayed and a list of text items. Text items are strings or arithmetic expressions. Precision of real values is arbitrarily limited to four decimal digits.

```
<HTEXT> ::= HTEXT <point> , <text_list>
<VTEXT> ::= VTEXT <point> , <text_list>
<text_list> ::= <text_item> { , <text_item> }
<text_item> ::= <string> ¦ <expression>
```

The SETSCREEN command is used to specify the window and clipping boundaries for displaying pictures. A SETSCREEN command will also clear the picture display screen without resetting the SCREEN file. SETSCREEN has three parameters which may be given in any order. It is not necessary to specify all three parameters. Initially the parameters default to PNT(0,0), XLEN=8, and YLEN=7. After a SETSCREEN command has been issued those parameters specified are used as defaults. An orientation point indicates which picture point is to be mapped to the lower left hand corner of the display screen. Two keyword parameters give the values needed to calculate the clipping boundaries based on the orientation point. XLEN is the width of the area within the

clipping bounds and YLEN is the height of that area. Figure A.2 illustrates the three parameters.

```
<SETSCREEN> ::= SETSCREEN <point> ,
     XLEN = <expression> , YLEN = <expression>
```

Special Utility Commands

Utility commands perform several varied functions.

```
<special_utility_commands> ::= <HALT> ¦
     <NULL> ¦ <LOAD> ¦ <LIST> ¦
     <LOGON> ¦ <LOGOFF> ¦ <EXECUTE>
```

The HALT command terminates execution of the PIGLI interpreter. The NULL command performs no operation. It is intended to be used in branches of the IF_THEN_ELSE command where no operation is desired.

```
<HALT> ::= HALT
<NULL> ::= NULL
```

The LOAD command is used to dynamically change the display output device driver during a programming session. The LOAD command has one parameter, the identifier by which the SOLO operating system knows the device driver. (Specific information concerning device drivers is given in chapter five.)

```
<LOAD> ::= LOAD <identifier>
```

The LIST command is used to retrieve information about

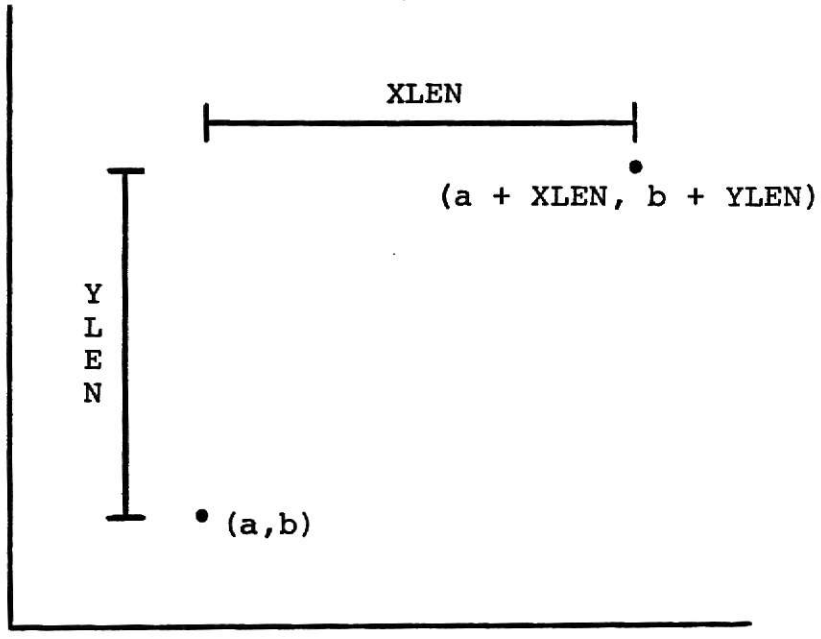


Figure A.2: An Illustration of SETSCREEN Parameters

Pictures processed in the programming session. There are four variations of LIST for reviewing different information. One form (LIST) may be used to list the identifiers of pictures. A second form (LIST <primitive_or_transform>) may be used for listing the identifiers of picture elements according to the type of the element. These two forms do not show any of the relationships between pictures and picture elements. The second form does not list any unnamed picture elements.

The third form (LIST DEF <identifier>) has one parameter which is the identifier of a picture or a picture element. The LIST DEF command returns the definition of the picture or element identified by the parameter. For components of pictures that are pictures or named transforms and picture primitives, LIST DEF lists the identifier naming that element. For unnamed transforms and picture primitives, the canonical definition of the picture element is listed. The fourth form (LIST SCREEN) returns the identifiers of all the pictures currently being displayed.

```
<LIST> ::= LIST ¦ LIST <primitive_or_transform> ¦
     LIST DEF <identifier> ¦ LIST SCREEN
<primitive_or_transform> ::= LINE ¦ CIRCLE ¦ ARC ¦
     FIGURE ¦ CURVE ¦ TRANS ¦ TURN ¦ SCALE
```

The LOGON and LOGOFF commands are used to make a record of PIGLI programming sessions as they are in progress. The LOGON command has one parameter to specify the name of the

external disk file where the commands that are issued are to be recorded. The LOGOFF command closes the log file. Two LOGON commands may not be issued in a programming session without an intervening LOGOFF command.

```
<LOGON> ::= LOGON <identifier>
<LOGOFF> ::= LOGOFF
```

The EXECUTE command is used to direct the interpreter to an exec file or a log file to execute commands from the file. The EXECUTE command has one parameter, the identifier by which the SOLO operating system knows the file to be executed. The interpreter executes commands from the exec file until the file is exhausted at which time it will begin to accept commands from the console device again. The exec file may not contain an EXECUTE command.

```
<EXECUTE> ::= EXECUTE <identifier>
```

```
"###########
#  PREFIX  #
###########"


CONST NL = '(:10:)';   FF = '(:12:)';   CR = '(:13:)';   EM
= '(:25:)';

CONST PAGELENGTH = 512;
TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF CHAR;

CONST LINELENGTH = 132;
TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;

CONST IDLENGTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;

TYPE FILE = 1..2;

TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);

TYPE FILEATTR = RECORD
                  KIND: FILEKIND;
                  ADDR: INTEGER;
                  PROTECTED: BOOLEAN;
                  NOTUSED: ARRAY (.1..5.) OF INTEGER
                END;

TYPE IODEVICE =
  (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE,
CARDDEVICE);

TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);

TYPE IOARG = (WRITEEOF, REWIND, UPSPACE, BACKSPACE);

TYPE IORESULT =
  (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
   ENDFILE, ENDMEDIUM, STARTMEDIUM);

TYPE IOPARAM = RECORD
                 OPERATION: IOOPERATION;
                 STATUS: IORESULT;
                 ARG: IOARG
               END;

TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);

TYPE ARGTAG =
  (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE, STAPTR,
```

```
   SCRPTR, SHVPTR, FDSPTR, SYMPTR, REALTYPE);

TYPE POINTER = @BOOLEAN;

TYPE STMT_PTR = @ STMT_RECORD;

TYPE SCREEN_PTR = @ SCREEN_FILE;

TYPE SYMBOL_PTR = @ SYMBOL_TABLE;

TYPE FREE_PTR = @ FREE_SPACE;

TYPE SHARED_PTR = @ SHARED_DATA;

TYPE ARGTYPE = RECORD
                  CASE TAG: ARGTAG OF
                    NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
                    INTTYPE: (INT: INTEGER);
                    IDTYPE: (ID: IDENTIFIER);
                    PTRTYPE: (PTR:POINTER);
                    STAPTR: (STMT: STMT_PTR);
                    SCRPTR: (DISPLAYS: SCREEN_PTR);
                    SHVPTR: (SHARE: SHARED_PTR);
                    FDSPTR: (FREE: FREE_PTR);
                    SYMPTR: (SYMB: SYMBOL_PTR);
                    REALTYPE: (RL: REAL)
                END;

CONST MAXARG = 10;
TYPE ARGLIST = ARRAY (.1..MAXARG.) OF ARGTYPE;

TYPE ARGSEQ = (INP, OUT);

TYPE PROGRESULT =
  (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR,
VARIANTERROR,
   HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);


CONST

"SYMBOLS"

BEGIN2=0;          KREAL2=1;          KINTEGER2=2;       IF2=3;
WHILE2=4;          ID2=5;             EXECUTE2=6;
ENTER2=7;
EXIT2=8;           LOGON2=9;          LOGOFF2=10;
NULL2=11;
HALT2=12;          BUILD2=13;         CHANGE2=14;
DELETE2=15;
```

```
SETSCREEN2=16;      DRAW2=17;          ERASE2=18;
LIST2=19;
HTEXT2=20;          VTEXT2=21;         SAVE2=22;
PLOT2=23;
NOT2=24;            AND2=25;           OR2=26;
GT2=27;
LT2=28;             GE2=29;            LE2=30;
EQ2=31;
NE2=32;             MINUS2=33;         PLUS2=34;
SLASH2=35;
STAR2=36;           DIV2=37;           MOD2=38;
UMINUS2=39;
CONV2=40;           TRUNC2=41;         VALU2=42;
XVAL2=43;
YVAL2=44;           REAL2=45;          INTEGER2=46;
NEW_ID2=47;
STRING2=48;         SEMICOLON2=49;     THEN2=50;
ELSE2=51;
DO2=52;             END2=53;           EOM2=54;
COMMA2=55;
CLOSE2=56;          AT2=57;            OF2=58;
BECOMES2=59;
OPEN2=60;           PERIOD2=61;        MAPSTO2=62;
IS2=63;
WITH2=64;           XDIS2=65;          YDIS2=66;
ADD2=67;
DEF2=68;            SCREEN2=69;        KNOTS2=70;
LINE2=71;
CIRCLE2=72;         ARC2=73;           FIGURE2=74;
CURVE2=75;
TRANS2=76;          TURN2=77;          SCALE2=78;
START2=79;
MID2=80;            ENDPT2=81;         CENTER2=82;
TOP2=83;
BOT2=84;            LEFT2=85;          RIGHT2=86;
PDIS2=87;
PDEG2=88;           ABOUT2=89;         LENGTH2=90;
RADIUS2=91;
FACTOR2=92;         DEG2=93;           ANGLE2=94;
FRDEG2=95;
TODEG2=96;          FORM2=97;          OPENED2=98;
CLOSED2=99;
DIR2=100;           CCW2=101;          CW2=102;
POINT2=103;
PNT2=104;           LTOP2=105;         RTOP2=106;
LBOT2=107;
RBOT2=108;          TRIGHT2=109;       BRIGHT2=110;
TLEFT2=111;
BLEFT2=112;
```

```
ID_PIECE_LENGTH = 11 "TWELVE CHARS PER PIECE";
NULL=32767;
SPAN=26 "NUMBER OF DISTINCT ID CHARS";

MIN_ORD=0;          MAX_ORD=127;        MAX_INTEGER=32767;
MAX_EXPONENT=38;
MAX_STRING_LENGTH = 80;
MAX_SYMBOLS=501;
TEXT_LENGTH = 12;
INFILE = 1;         OUTFILE = 2;

TYPE

  TEXT_TYPE = ARRAY (.1..TEXT_LENGTH.) OF CHAR;

  PIECE=ARRAY(.0..ID_PIECE_LENGTH.) OF CHAR;

  STRING_TYPE = ARRAY(.1..MAX_STRING_LENGTH.) OF CHAR;


  SYMBOL_INDEX=0..MAX_SYMBOLS;

  GRAPH_TYPE=(REALP,INTEGERP,LINEP,PICTUREP,CIRCLEP,
              ARCP,FIGUREP,CURVEP,TRANSP,TURNP,SCALEP,
              FILEP,UNDEFP);

  PART_PTR = @ PICTURE_PART;
  PICTURE_PART = RECORD
     THIS_PART : SYMBOL_INDEX;
     NEXT_PART : PART_PTR
  END; "PICTURE PART"

  KNOT_CHAIN_PTR = @ KNOT;
  KNOT = RECORD
     KNOTX,KNOTY : REAL;
     NEXT_KNOT : KNOT_CHAIN_PTR
  END; "KNOT"

  GRAPH_PTR=@ GRAPH_NODE;
  GRAPH_NODE = RECORD
     CASE GRAPH_TAG:GRAPH_TYPE OF
        REALP : (RVAL:REAL);
        INTEGERP : (IVAL : INTEGER);
        LINEP : (STARTX,STARTY,ENDX,ENDY : REAL);
        PICTUREP : (FIRST_PART : PART_PTR);
        CIRCLEP : (CIRCENX,CIRCENY,RADIUS : REAL);
        ARCP : (ARCSTARTX,ARCSTARTY,ARCENDX,ARCENDY,
                ARCCENX,ARCCENY : REAL;
                ARCDIR,ARCFORM : INTEGER);
        FIGUREP,CURVEP : (KNOT_CHAIN : KNOT_CHAIN_PTR;
```

```
                        F_C_FORM : INTEGER);
      TRANSP : (DELTAX,DELTAY : REAL;
                TRANSBASE : SYMBOL_INDEX);
      TURNP,SCALEP : (POINTX,POINTY,FACTOR : REAL;
                      PICBASE : SYMBOL_INDEX);
      FILEP,UNDEFP : (DUM1 : INTEGER)
   END; "GRAPH NODE"

SYMBOL_ENTRY = RECORD
   NAME : PIECE;
   DEF : GRAPH_PTR
END; "SYMBOL ENTRY"

EXP_PTR = @ EXP_RECORD;
EXP_VALU_TYPE = (OPERATOR,OPERAND);
OPTYPE = NOT2..YVAL2;
CONST_TYPE = (REALC,INTEGERC);
OPAND_TYPE = (VARIABLE,CONSTANT,SYS_VALUE_REF);
EXP_RECORD = RECORD
   NEXT_EXP : EXP_PTR;
   CASE EXP_VALU_TAG : EXP_VALU_TYPE OF
      OPERATOR : (OPTOR : OPTYPE);
      OPERAND : (
         CASE OPAND : OPAND_TYPE OF
            VARIABLE : (VAR_INDEX : SYMBOL_INDEX);
            CONSTANT : (
                CASE CONST_TAG : CONST_TYPE OF
                    REALC:(RL : REAL);
                    INTEGERC:(INT : INTEGER));
            SYS_VALUE_REF : (VALUE_REF : INTEGER))
END "EXP_RECORD";

DISP_TEXT_TYPE = (STRING_TEXT,VALUE_TEXT);
DEF_TYPE = (SPEC_POINT,SPEC_PARM,SPEC_PIC,SPEC_VALUE);
POINT_TYPE = (POINT,SYS_POINT_REF);

POINT_PTR = @ POINT_RECORD;
POINT_RECORD = RECORD
   CASE POINT_TAG : POINT_TYPE OF
      POINT : (XCOORD,YCOORD : EXP_PTR);
      SYS_POINT_REF : (POINT_REF : INTEGER)
END; "POINT RECORD"

SPEC_CHAIN_PTR = @ SPEC_CHAIN;
SPEC_CHAIN = RECORD
   SPEC_TAG : INTEGER;
   NEXT_SPEC : SPEC_CHAIN_PTR;
   CASE DEF_TAG:DEF_TYPE OF
      SPEC_POINT : (POINT_DEF:POINT_PTR);
      SPEC_PARM  : (PARM_DEF:INTEGER);
```

```
        SPEC_PIC    : (PIC_DEF:SYMBOL_INDEX);
        SPEC_VALUE  : (VALUE_DEF:EXP_PTR)
  END; "SPEC CHAIN"

  PIC_DEF_TYPE = (SCREEN,OLD_PIC,PRIMITIVE);

  PIC_CHAIN_PTR = @ PICTURE_CHAIN;
  PICTURE_CHAIN = RECORD
     COMPONENT_ID : SYMBOL_INDEX;
     NEXT_PIC : PIC_CHAIN_PTR;
     CASE PIC_DEF_TAG:PIC_DEF_TYPE OF
        SCREEN,OLD_PIC : (OLD_PIC_ID : SYMBOL_INDEX);
        PRIMITIVE : (PRIM_TAG : INTEGER;
           PRIM_DEF : SPEC_CHAIN_PTR)
  END; "PICTURE CHAIN"

  BREAK_TYPE = (PIC_STMT,PROG_STMT);
  PROG_TYPE = (BEGINS,IFS,INTEGERS,REALS,
                  WHILES,ASSIGNS,HALTS,NULLS);
  PIC_TYPE = (BUILDS,DRAWS,ERASES,HTEXTS,VTEXTS);

  TEXT_CHAIN_PTR = @ TEXT_RECORD;
  TEXT_RECORD = RECORD
     NEXT_TEXT : TEXT_CHAIN_PTR;
     CASE TEXT_TAG : DISP_TEXT_TYPE OF
        STRING_TEXT : (STRING : STRING_TYPE;
                       STRING_LENGTH : INTEGER);
        VALUE_TEXT : (VALUEXP : EXP_PTR)
  END; "TEXT RECORD"

  STMT_CHAIN_PTR = @ STMT_CHAIN;
  STMT_CHAIN= RECORD
     STMT : STMT_PTR;
     STMT_CH : STMT_CHAIN_PTR
  END; "STMT_CHAIN"

  STMT_RECORD = RECORD
     CASE BREAK_TAG : BREAK_TYPE OF
        PIC_STMT : (
           CASE PIC_TAG:PIC_TYPE OF
              BUILDS : (PIC_ID : SYMBOL_INDEX;
                  PIC_CHAIN : PIC_CHAIN_PTR);
              ERASES,DRAWS : (DRAW_ID : SYMBOL_INDEX);
              HTEXTS,VTEXTS : (HVPNT : POINT_PTR;
                  HVTEXT_CHAIN : TEXT_CHAIN_PTR));

        PROG_STMT : (
           CASE PROG_TAG : PROG_TYPE OF
              BEGINS : (B_CHAIN : STMT_CHAIN_PTR);
              IFS : (IF_BOOL : EXP_PTR;
```

```
                    IF_TRUE : STMT_PTR;
                    IF_FALSE : STMT_PTR);
                WHILES : (WHILE_BOOL : EXP_PTR;
                    WHILE_TRUE : STMT_PTR);
                ASSIGNS : (TARGET : SYMBOL_INDEX;
                    SOURCE : EXP_PTR);
                HALTS : (DUM2 : INTEGER);
                NULLS : (DUM3 : INTEGER))
  END; "STMT RECORD"

  FREE_SPACE = RECORD
     FREE_GRAPH_NODE : GRAPH_PTR;
     FREE_EXP : EXP_PTR;
     FREE_POINT : POINT_PTR;
     FREE_STMT : STMT_PTR;
     FREE_PART : PART_PTR;
     FREE_SPEC : SPEC_CHAIN_PTR;
     FREE_PIC_CHAIN : PIC_CHAIN_PTR;
     FREE_TEXT : TEXT_CHAIN_PTR;
     FREE_STMT_CH : STMT_CHAIN_PTR;
     FREE_KNOT : KNOT_CHAIN_PTR;
     FREE_SCREEN : SCREEN_PTR
  END; "FREE SPACE"

  SHARED_DATA = RECORD
     OLD_IND : INTEGER;
     MAX_SYMB, MIN_DUMMY : SYMBOL_INDEX
  END; "SHARED DATA"

  SCREEN_FILE = RECORD
     NEXT_SCREEN : SCREEN_PTR;
     THIS_SCREEN : SYMBOL_INDEX
   END; "SCREEN FILE"

  SYMBOL_TABLE = ARRAY[SYMBOL_INDEX] OF SYMBOL_ENTRY;


PROCEDURE READ(VAR C: CHAR);
PROCEDURE WRITE(C: CHAR);

PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
PROCEDURE CLOSE(F: FILE);
PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
FUNCTION LENGTH(F: FILE): INTEGER;

PROCEDURE MARK(VAR TOP: INTEGER);
PROCEDURE RELEASE(TOP: INTEGER);

PROCEDURE IDENTIFY(HEADER: LINE);
```

```
PROCEDURE ACCEPT(VAR C: CHAR);
PROCEDURE DISPLAY(C: CHAR);

PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);
PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);
PROCEDURE READLINE(VAR TEXT: UNIV LINE);
PROCEDURE WRITELINE(TEXT: UNIV LINE);
PROCEDURE READARG(S: ARGSEQ; VAR ARG: ARGTYPE);
PROCEDURE WRITEARG(S: ARGSEQ; ARG: ARGTYPE);

PROCEDURE LOOKUP(ID: IDENTIFIER; VAR ATTR: FILEATTR; VAR
FOUND: BOOLEAN);

PROCEDURE IOTRANSFER
  (DEVICE: IODEVICE; VAR PARAM: IOPARAM; VAR BLOCK: UNIV
PAGE);

PROCEDURE IOMOVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);

FUNCTION TASK: TASKKIND;

PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;
              VAR LINE: INTEGER; VAR RESULT: PROGRESULT);

PROGRAM P(VAR PARAM: ARGLIST);

CONST
```

```
"############
#  PREFIX   #
############"


CONST NL = '(:10:)';   FF = '(:12:)';   CR = '(:13:)';   EM
= '(:25:)';

CONST PAGELENGTH = 512;
TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF CHAR;

CONST LINELENGTH = 132;
TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;

CONST IDLENGTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;

TYPE FILE = 1..2;

TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);

TYPE FILEATTR = RECORD
                  KIND: FILEKIND;
                  ADDR: INTEGER;
                  PROTECTED: BOOLEAN;
                  NOTUSED: ARRAY (.1..5.) OF INTEGER
                END;

TYPE IODEVICE =
  (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE,
CARDDEVICE,
   A,B,COMPUTEK);

TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);

TYPE IOARG = (WRITEEOF, REWIND, UPSPACE, BACKSPACE);

TYPE IORESULT =
  (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
   ENDFILE, ENDMEDIUM, STARTMEDIUM);

TYPE IOPARAM = RECORD
                 OPERATION: IOOPERATION;
                 STATUS: IORESULT;
                 COUNT: INTEGER
               END;

TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);

TYPE ARGTAG =
```

```
  (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);

TYPE POINTER = @BOOLEAN;

TYPE ARGTYPE = RECORD
                 CASE TAG: ARGTAG OF
                   NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
                   INTTYPE: (INT: INTEGER);
                   IDTYPE: (ID: IDENTIFIER);
                   PTRTYPE: (PTR: POINTER)
               END;

CONST MAXARG = 10;
TYPE ARGLIST = ARRAY (.1..MAXARG.) OF ARGTYPE;

TYPE ARGSEQ = (INP, OUT);

TYPE PROGRESULT =
  (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR,
VARIANTERROR,
   HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);

PROCEDURE READ(VAR C: CHAR);
PROCEDURE WRITE(C: CHAR);

PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
PROCEDURE CLOSE(F: FILE);
PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
FUNCTION LENGTH(F: FILE): INTEGER;

PROCEDURE MARK(VAR TOP: INTEGER);
PROCEDURE RELEASE(TOP: INTEGER);

PROCEDURE IDENTIFY(HEADER: LINE);
PROCEDURE ACCEPT(VAR C: CHAR);
PROCEDURE DISPLAY(C: CHAR);

PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);
PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);
PROCEDURE READLINE(VAR TEXT: UNIV LINE);
PROCEDURE WRITELINE(TEXT: UNIV LINE);
PROCEDURE READARG(S: ARGSEQ; VAR ARG: ARGTYPE);
PROCEDURE WRITEARG(S: ARGSEQ; ARG: ARGTYPE);

PROCEDURE LOOKUP(ID: IDENTIFIER; VAR ATTR: FILEATTR; VAR
FOUND: BOOLEAN);

PROCEDURE IOTRANSFER
  (DEVICE: IODEVICE; VAR PARAM: IOPARAM; VAR BLOCK: UNIV
```

```
PAGE);

PROCEDURE IOMOVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);

FUNCTION TASK: TASKKIND;

PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;
              VAR LINE: INTEGER; VAR RESULT: PROGRESULT);


PROGRAM P(VAR PARAM: ARGLIST);

CONST
  HOME_ERASE = '(:12:)';
  ERASE_STATUS = '(:14:)';
  WRITE_STATUS = '(:15:)';
  LINE_FEED = '(:10:)';
  BACKSPACE = '(:8:)';
  FOUR_BYTE_ABS = '(:28:)';
  AT_SIGN = '(:64:)';
  NULL_BYTE = '(:0:)';

"INPUT COMMANDS"

  ERASE = 0;
  WRITE = 1;
  MOVE = 2;
  VECTOR = 3;
  HTEXT = 4;
  VTEXT = 5;
  CLEAR = 6;
  EOT = 7;

TYPE

  PACKED_LINE = ARRAY [1..66] OF INTEGER;
  PACKED_REAL = ARRAY [1..4] OF INTEGER;
  PAGE_INDEX = 0..256;
  SEND_INDEX = 0..512;
  LINE_INDEX = 0..132;

  CH_HWD = ARRAY[1..2] OF CHAR;

VAR

  HEX : ARRAY [0..15] OF CHAR;

  DISPLAY_PAGE : ARRAY [1..256] OF INTEGER;
  OUTPAGE : PAGE;
  PAGE_CNTR : PAGE_INDEX;
```

```
  SEND_CNTR : SEND_INDEX;
  TEXT_LINE : LINE;
  EOF,CHAR_MODE,VISIBLE : BOOLEAN;

PROCEDURE BUMP_PAGE_CNTR;

"INCREMENT INDEX OF INPUT PAGE
 WHEN ONE PAGE IS EXHAUSTED GET A NEW PAGE
   AND INITIALIZE THE PAGE INDEX"

  BEGIN
    IF PAGE_CNTR = 256 THEN BEGIN
      READPAGE(DISPLAY_PAGE,EOF);
      PAGE_CNTR := 0;
    END;
    PAGE_CNTR := SUCC(PAGE_CNTR);
  END;

PROCEDURE GET_LINE (VAR INTLINE : UNIV PACKED_LINE);

"UNPACK THE CHARACTER LINE PARAMETER FOR
 HTEXT AND VTEXT COMMANDS"

  VAR I:INTEGER;
  BEGIN
    FOR I := 1 TO 66 DO BEGIN
      BUMP_PAGE_CNTR;
      INTLINE[I] := DISPLAY_PAGE[PAGE_CNTR];
    END;
  END;

PROCEDURE GET_POINT(VAR X,Y: UNIV PACKED_REAL);

"UNPACK POINT PARAMETER FOR MOVE
 OR VECTOR COMMANDS"

  VAR I:INTEGER;
  BEGIN
    FOR I := 1 TO 4 DO BEGIN
      BUMP_PAGE_CNTR;
      X[I] := DISPLAY_PAGE[PAGE_CNTR];
    END;
    FOR I := 1 TO 4 DO BEGIN
      BUMP_PAGE_CNTR;
      Y[I] := DISPLAY_PAGE[PAGE_CNTR];
    END;
  END;

PROCEDURE GET_TERMINAL_POINT(VAR XINT,YINT:INTEGER);
```

```
"COMPUTE COMPUTEK POINT COORDINATE
 EQUIVALENTS OF REAL WORLD POINT COORDINATES"

  VAR X,Y:REAL;
  BEGIN
    GET_POINT(X,Y);
    XINT := TRUNC(X * 32.0) MOD 256;
    YINT := TRUNC(Y * 36.57) MOD 256;
  END;

PROCEDURE GETBYTES(VAR HIGH,LOW,HALFWD: UNIV CH_HWD);
BEGIN
  HIGH[1] := '(:0:)';
  HIGH[2] := HALFWD[1];
  LOW[1] := '(:0:)';
  LOW[2] := HALFWD[2];
END;

PROCEDURE PRINTABS(ARG:UNIV INTEGER);

"DEBUGGING PROCEDURE"

"CALCULATE AND DISPLAY HEX EQUIVALENTS OF
 ASCII CHARACTERS TO BE TRANSMITTED TO
 COMPUTEK TERMINAL"

  VAR T:ARRAY[1..4] OF CHAR;
      LOW,HIGH,REM,DIGIT,I:INTEGER;
  BEGIN
  REM := ARG;
  GETBYTES(HIGH,LOW,REM);
  T[1]:= HEX[HIGH DIV 16];
  T[2] := HEX[HIGH MOD 16];
  T[3] := HEX[LOW DIV 16];
  T[4] := HEX[LOW MOD 16];
  DISPLAY(T[1]); DISPLAY(T[2]); DISPLAY(T[3]);
DISPLAY(T[4]); DISPLAY(' ');
END;

PROCEDURE SEND;

"TRANSMIT ASCII CONTROL CHARACTER PAGE
 TO COMPUTEK TERMINAL"

  VAR ARG:IOPARAM; CH:CHAR; I : INTEGER;
  BEGIN
    FOR I := 1 TO SEND_CNTR DO BEGIN
      CH := OUTPAGE[I];
      PRINTABS(CH);
      IF I MOD 10 = 0 THEN DISPLAY(NL);
```

```
    END;
    DISPLAY(NL);
    WITH ARG DO BEGIN
      OPERATION := OUTPUT;
      COUNT := SEND_CNTR;
    END;
    IOTRANSFER(COMPUTEK,ARG,OUTPAGE);
    "IF IORESULT <> COMPLETE THEN BOMB;"
  END;

PROCEDURE BUMP_SEND_CNTR;

"INCREMENT INDEX OF THE OUTPUT PAGE.
 WHEN PAGE IS FULL TRANSMIT AND
 BEGIN NEW OUTPUT PAGE"

  BEGIN
    IF SEND_CNTR = 512 THEN BEGIN
      SEND;
      SEND_CNTR := 0;
    END;
    SEND_CNTR := SUCC(SEND_CNTR);
  END;

PROCEDURE SEND_BYTE(CH:CHAR);

"ADD A SINGLE ASCII CHARACTER TO THE OUTPUT PAGE"

  BEGIN
    BUMP_SEND_CNTR;
    OUTPAGE [SEND_CNTR] := CH;
  END;

PROCEDURE SEND_AT_SIGN;

"ADD THE ASCII CHARACTERS TO THE OUTPUT PAGE
 WHICH WILL CHANGE THE COMPUTEK TERMINAL
 FROM FOUR BYTE ABSOLUTE MODE TO
 CHARACTER MODE"

  VAR I:INTEGER;
  BEGIN
    SEND_BYTE(AT_SIGN);
    FOR I := 1 TO 3 DO SEND_BYTE(NULL_BYTE);
  END;

PROCEDURE SEND_4_BYTE(X,Y:INTEGER; VISIBLE:BOOLEAN);

"COMPUTE THE FOUR ASCII CHARACTERS THAT
 EXECUTE A MOVE OR VECTOR COMMAND.
```

```
 X AND Y ARE INTEGER COORDINATES OF
 THE TARGET POINT.  VISIBLE = TRUE FOR
VECTOR.  VISIBLE = FALSE FOR MOVE"

  VAR ACCUM:INTEGER;
  BEGIN
    ACCUM := 2;
    IF (Y MOD 2) = 1 THEN ACCUM := ACCUM + 16;
    IF ((Y DIV 2) MOD 2) = 1 THEN ACCUM := ACCUM + 32
    ELSE ACCUM := ACCUM + 64;
    IF VISIBLE THEN ACCUM := ACCUM + 1;
    SEND_BYTE(CHR(ACCUM));
    ACCUM := Y DIV 4;
    IF ((ACCUM DIV 32) MOD 2) = 0 THEN ACCUM := ACCUM + 64;
    SEND_BYTE(CHR(ACCUM));
    ACCUM := 0;
    IF (X MOD 2) = 1 THEN ACCUM := ACCUM + 16;
    IF ((X DIV 2) MOD 2) = 1 THEN ACCUM := ACCUM + 32
    ELSE ACCUM := ACCUM + 64;
    SEND_BYTE(CHR(ACCUM));
    ACCUM := X DIV 4;
    IF ((ACCUM DIV 32) MOD 2) = 0 THEN ACCUM := ACCUM + 64;
    SEND_BYTE(CHR(ACCUM));
  END;

PROCEDURE PROCESS_DRAW(KEY:INTEGER);

"PUT COMPUTEK IN FOUR BYTE ABSOLUTE MODE.
 UNPACK AND TRANSLATE TO COMPUTEK
 COORDINATES THE TARGET POINT.
 COMPUTE AND TRANSMIT A FOUR BYTE ABSOLUTE COMMAND"

  VAR X,Y:INTEGER;
  BEGIN
    IF CHAR_MODE THEN BEGIN
      SEND_BYTE(FOUR_BYTE_ABS);
      CHAR_MODE := FALSE;
    END;
    GET_TERMINAL_POINT(X,Y);
    IF KEY = VECTOR THEN VISIBLE := TRUE ELSE VISIBLE :=
FALSE;
    SEND_4_BYTE(X,Y,VISIBLE);
  END;

PROCEDURE PROCESS_TEXT(KEY:INTEGER);

"PUT COMPUTEK IN CHARACTER MODE.
 UNPACK TEXT PARAMETER.  TRANSMIT TEXT
 CHARACTERS.  FOR VERTICAL TEXT,
 TRANSMIT LINEFEED AND BACKSPACE
```

```
 BETWEEN TEXT CHARACTERS"

  VAR VERTICAL:BOOLEAN; I:LINE_INDEX;
  BEGIN
    IF NOT CHAR_MODE THEN BEGIN
      SEND_AT_SIGN;
      CHAR_MODE := TRUE;
    END;
    IF KEY = VTEXT THEN VERTICAL := TRUE ELSE VERTICAL :=
FALSE;
    GET_LINE(TEXT_LINE);
    I := 1;
    REPEAT
      SEND_BYTE(TEXT_LINE[I]);
      I := SUCC(I);
      IF VERTICAL THEN BEGIN
        SEND_BYTE(LINE_FEED);
        SEND_BYTE(BACKSPACE);
      END;
    UNTIL TEXT_LINE[I] = EM;
  END;

PROCEDURE PROCESS_PAGE;

"INITIALIZE COUNTERS AND FLAGS.
 PROCESS COMMANDS UNTIL EOT.
 LEAVE TERMINAL IN CHARACTER MODE
 FOR NEXT TRANSMISSION"

  VAR EOTRANSFER:BOOLEAN; KEY:INTEGER;
  BEGIN
    SEND_CNTR := 0;
    PAGE_CNTR := 0;
    EOTRANSFER := FALSE; CHAR_MODE := TRUE;
    REPEAT
      BUMP_PAGE_CNTR;
      KEY := DISPLAY_PAGE[PAGE_CNTR];
      CASE KEY OF
        MOVE,VECTOR: PROCESS_DRAW(KEY);
        CLEAR,WRITE,ERASE:BEGIN
          IF NOT CHAR_MODE THEN SEND_AT_SIGN;
          CASE KEY OF
            CLEAR: SEND_BYTE(HOME_ERASE);
            WRITE: SEND_BYTE(WRITE_STATUS);
            ERASE: SEND_BYTE(ERASE_STATUS)
          END; "CASE"
        END;
        HTEXT,VTEXT: PROCESS_TEXT(KEY);
        EOT: EOTRANSFER := TRUE
      END; "CASE"
```

```
    UNTIL EOTRANSFER;
    IF NOT CHAR_MODE THEN SEND_AT_SIGN;
    SEND;
  END;

BEGIN
  "INITIALIZE;"
  HEX[0] := '0';
  HEX[1] := '1';
  HEX[2] := '2';
  HEX[3] := '3';
  HEX[4] := '4';
  HEX[5] := '5';
  HEX[6] := '6';
  HEX[7] := '7';
  HEX[8] := '8';
  HEX[9] := '9';
  HEX[10]:= 'A';
  HEX[11]:= 'B';
  HEX[12]:= 'C';
  HEX[13]:= 'D';
  HEX[14]:= 'E';
  HEX[15]:= 'F';


"INITIALIZE COMPUTEK TERMINAL BY
 CLEARING SCREEN AND HOMING THE CURSOR.
 READ INPUT COMMAND PAGES AND PROCESS
 UNTIL PAGE MARKED EOF IS RECEIVED.
 SEND NORMAL TERMINATION MESSAGE TO
 INITIATING PROCESS"

  SEND_CNTR := 0;
  CHAR_MODE := TRUE;
  SEND_AT_SIGN;
  SEND_BYTE(HOME_ERASE);
  SEND;
  REPEAT
    READPAGE(DISPLAY_PAGE,EOF);
    IF NOT EOF THEN PROCESS_PAGE;
  UNTIL EOF;
  "WRAPUP;"
  PARAM[1] .BOOL := TRUE;
END.
```

Design and Implementation of a Portable Interactive Graphics Language Interpreter

by

MARY CATHERINE NEAL
B.S. Iowa State University, 1973

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University
Manhattan, Kansas

1978

Design and Implementation of a Portable Interactive Graphics Language Interpreter

ABSTRACT

PIGLI (Portable, Interactive Graphics Language Interpreter) is a new high-level graphics system. The PIGLI command language allows the programmer to construct two-dimensional pictures in terms of picture primitives, to apply transformations to existing pictures, and to display pictures and text on a variety of vector graphics devices. The language also contains general purpose programming constructs, including type declarations, assignment statements, if-then-else statements, do-while statements, and begin-end blocks. PIGLI also provides, as a debugging aid, a general purpose LIST command which gives the programmer access to 1) the definitions of pictures, primitives, and variables, 2) the relationships between pictures and their components, and 3) the identifiers associated with displayed pictures. Another useful feature is the ability of the system to access external files of PIGLI commands that perform tedious or frequently needed tasks, as well as to accept commands interactively from a console device.

PIGLI is implemented in PASCAL on an Interdata 8/32 minicomputer under SOLO, a single-user PASCAL operating system that was designed to be portable and is currently running on several different machines. PIGLI is a two-pass interpreter and is implemented with an overlay structure in order to provide more space for storage of data structures. Data structures representing pictures are translated into a sequence of device independent output commands that are executed by output device drivers. An output device driver must be written for each device to be used. The structure of SOLO and the design of PIGLI allows the programmer to dynamically activate any of the output device drivers during a programming session.