A HIGH-LEVEL GRAPHICS PROGRAMMING LANGUAGE SUPPORTING THE INQUIRY OF GRAPHICAL OBJECTS

by

ROBERT VAUGHAN ROSS

B.A.Sc., The University of British Columbia, 1980

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Electrical Engineering)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

June 1982

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

Department of Electrical Engineering

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date July 22, 1982

ABSTRACT

High-level graphical programming languages provide simply expressed constructs for the definition, manipulation, and external representation of graphical data. Such languages can be used to create effective and readable application programs. This thesis investigates the value of allowing the inquiry of graphical data in a graphical language. Major design goals of an implementation of a language with these capabilities are presented. A data base model and implementation are discussed. The classification of graphical primitives as abstract data types is presented. Examples are given of several areas in which a language including graphical inquiry may be applied. It is concluded that inquiry permits the definition and manipulation of arbitrary models of graphical objects, so enabling the implementation of sophisticated graphical algorithms.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to Dr. Günther Schrack for his invaluable help and guidance. I am grateful to my parents for their support and encouragement. This research would not have been possible were it not for the financial support of the National Science and Engineering Research Council.

Chapter 1

INTRODUCTION

There has always been considerable interest in the presentation of information in pictorial form. As computers are applied in an increasing number of areas and more information is stored in machine-readable form, the demand for the automatic generation of graphical images also increases. The field of computer graphics is concerned with the generation of such images.

Most programmers' initial experience with computer graphics involves graphical subroutine systems. The major task of these systems is to act as an interface between application programs and graphical input and output devices. The subroutines generally do not provide modelling functions, this is left as a task of the application program.

Modelling functions are provided by high-level graphics programming languages. These languages consider graphical information to be values of an abstract data type. They provide constructs which define and manipulate data of the type GRAPHICAL. High-level graphics languages possess several advantages over subroutine systems: high portability of programs and programmers, ease of learning the language, and improved readability of a program. This thesis discusses some aspects of the design and implementation of such a language. The name of

this language is LIG6 (Language for Interactive Graphics Version 6). It is the most recent version of a family of languages [Schr76,Mann80], with new features and characteristics.

During the past years, many high-level graphics languages have been described or proposed, notably in papers by Kulsrud [Kuls68], Newman [Newm71], Smith [Smit71], and, more recently, Magnenat-Thalmann et al. [Magn81], and Barth et al. [Bart81]. McLean [McLe78] discusses 37 languages in a survey. The languages differ widely in both syntax and semantics, as well as in the approaches taken for their implementation.

The modelling constructs of these languages allow the explicit definition and manipulation of models of graphical objects. This permits application programs to perform passive or interactive graphics. It is difficult for programs in these languages to manipulate arbitrary models of graphical objects because of the diverse nature of these models. This thesis investigates areas in which a graphics programming language can be applied when a formal model structure and inquiry into such structures are provided.

LIG6 is implemented as an extension to a host language, FORTRAN. A preprocessor, written in PASCAL, converts LIG6 programs into standard FORTRAN programs with extension elements translated into calls to subroutines in a run-time library. These subroutines are coded in FORTRAN. When a LIG6 program is to be executed, the object deck produced by compiling the preprocessor output is run in conjunction with the run-time library. Only those details of the language and of its implementation which are pertinent to the points discussed in

this thesis are outlined; more complete information is available in the LIG6 User's Manual [Ross82].

Chapter 2 presents the major design goals of the language. The overall style of the language with respect to the host language and the importance of separate preprocessing and compilation of modules is considered. Aspects of inquiry and the partitioning of language constructs into a hierarchy are addressed.

The concept of a data base system is presented in Chapter 3. The benefits of programmer knowledge of system implementation are discussed. A model of the data base and a summary of its implementation are given.

Chapter 4 introduces the idea of regarding graphical output primitives as true abstract data types. The method by which programmers can define such primitives and the differences between primitives and graphical functions are outlined.

Examples of areas in which LIG6 can be applied are presented in Chapter 5. Tools for the construction of models of graphical objects are discussed. The systematic manipulation of graphical objects is defined. The concept of interactively editing graphical objects is presented. Employing graphical data to represent abstract ideas is introduced.

A sample of features provided in LIG6 is given in Chapter 6. An effective transformation operator is introduced, the support for the data type VECTOR is outlined, stroke precision text is described, and the ability to save and restore models of graphical objects is discussed.

Chapter 2

LANGUAGE DESIGN GOALS

2.1 Inquiry

Usually, algorithms are implemented where the flow of control depends on the results of previous operations. To make control decisions, a program must be able to inquire about values of variables and expressions and determine what operations have occurred. For interesting graphical algorithms to be implemented, the results of graphical operations must be accessible to control structures. Thus, one of the design goals of LIG6 was to support information recovery.

Information recovery is the ability to access the results of actions and thereby determine what operation occurred. It is supported by providing methods of obtaining such access for all graphical operations. Two forms of access are used in LIG6. If a natural, consistent syntactical construct is possible, it is used. Otherwise, system subprograms are provided. A call to such a subprogram results in the desired information being returned in the routine's parameters.

In LIG6, logical expressions have been extended to include the comparison of models of graphical objects. In this way, the

results of graphical assignment statements can control a program's execution. Similarly, it can be determined whether a variable's value is a primitive and, if so, which type. The structure and content of primitive records are accessible, so enabling control decisions and automated modelling. The results of transformations and attribute settings in assignment statements are also accessible. A dualism between action and inquiry has been established: for every operation there is a method to determine the nature and result of its action.

2.2 Levels of Usage

As is true for most occupations, computer programmers tend to possess different levels of sophistication related to their experience and ability. A programming language is most useful if it is attractive to programmers of all levels. This general appeal may be obtained by dividing the language into levels with each level a superset of that directly below it. The lower level features are arranged to be independent of those at higher levels. This partitioning is one of the design goals of LIG6. Another benefit of level partition is the relative ease with which the language can be learned. Programmers are able to quickly obtain results using low level constructs and then advance naturally, as their needs increase.

Programming languages in general, and LIG6 in particular, can be visualized as having three levels of usage. Each of these levels is identified by the styles of tasks performed. The first level is a subset of the second which is, in turn, a subset of

the last.

The first level is characterized by explicit commands and little interactive activity. In a conventional programming language, programs which generate tables or solve well-defined mathematical problems such as numerical integration are of this level. The corresponding level in LIG6 is represented by programs which explicitly model a well-defined graphical object and display an external representation of that model, i.e. an image, on a device.

There are four independent constructs which support this level in LIG6. The first construct consists of the various methods of declaring graphical variables. Modelling is performed with the simple assignment statement and with the standard graphical primitives. Modelling transformations and attribute settings are performed with the modification constructs. The last construct is the display statement which produces images of the results of graphical modelling. At this level, the programmer is the artist, explicitly creating the desired image.

The second level of use is characterized by a high level of interactive activity and the explicit modification of earlier processes and results. A typical program in a conventional language of this type would be one in which results are obtained in an iterative fashion with new starting values and process parameters supplied interactively by the user. At this level, LIG6 programs explicitly modify models and images of graphical objects.

In LIG6 programs, models of graphical objects constructed by assignment statements are modified using the deletion

statement. Images produced by the display statement may be modified using two types of erasure statements. Interactive activities are supported by the identification constructs. The programmer is removed one step from the use of the program at this level; he creates application programs which are then run by users to create the images they desire.

The third and highest level is characterized by programs using knowledge of their own data bases and which use inquiry of the results of previous operations to determine the flow of control. A typical problem in conventional programming languages at this level might be a searching algorithm for the roots of equations. The corresponding level in LIG6 is typified by programs which implicitly modify arbitrary models of graphical objects or create new models based on the results of previous modelling.

This level is supported in LIG6 by modelling and inquiry constructs. A data base definition model is provided. A complete set of modelling operators is provided giving a programmer the ability to precisely define and modify models. Access is provided to all parts of a model for information retrieval. Programmers at this level are often removed two steps from the use of the program. They create the program tools which application programmers then incorporate in application programs with which users finally create images.

Without this final level, it is difficult to implement programs with the ability to make intelligent decisions. Sophisticated graphics programs would be forced to maintain separate data bases, defeating one of the primary purposes of a

high-level graphics language. An idea of the importance of this level can be obtained by considering the IF statement. Most computer programs contain such conditional statements implying that they are at the third level. No inquiry constructs are available in the first two levels. A graphical language with the ability to perform inquiry on the results of graphical operations clearly has a distinct advantage over one which does not.

2.3 Fortran Consistency

Virtually all graphics programming languages are extensions to existing high-level computer languages. A graphics application program will generally require non-graphics constructs for support. When these constructs are identical to those of a language already known to a programmer, the time and effort required to become fluent with the graphics language is reduced. Any gains made with this approach, however, will be lessened if the language extension is not consistent with the host language.

Each individual construct of a given computer language has rules governing identifier names, reserved words, and format which it shares with the other constructs. In addition, operations possible with different data types are arranged to overlap as much as is feasible. If these general rules are ignored when a language is extended and a different style is used for new constructs, a programmer will be forced to learn and retain twice as much information regarding style; confusion

will occur over the choice of style for a particular construct. It is with this in mind that one of the design goals of the language was to make all extensions as consistent with FORTRAN as possible.

The most important features of the format of a FORTRAN program are that blanks are not delimiters, and that there are no reserved words. All syntax is determined by context, making FORTRAN one of the more difficult computer languages to parse. A classic example of this is the two statements

```
DO 10 I = 5
DO 10 I = 5,9
```

The first statement is an assignment statement where the integer expression 5 is coerced into a real expression and assigned to the variable DO10I. The second statement is a DO statement with the integer index I ranging in value from 5 to 9 and an object whose statement has the label 10. A parser cannot distinguish between the two statements until it reaches the comma.

Such input format conventions remain in LIG6. The extensions do not introduce reserved words, and blanks are still ignored. Some upwards compatible freedoms have been added to the format of both host and extension statements. Statements may now span lines without the use of a continuation card, although such cards are still permitted and recognized. Multiple statements per line are allowed, provided they are separated by semicolons. Comments enclosed in braces may appear anywhere. The length of a line has been extended from 72 to 255 characters and column positions are not important regarding statement labels and the beginning of statements. None of these extensions restrict

previously correct format; ANSI standard FORTRAN programs are acceptable LIG6 programs.

In FORTRAN, there is a variety of rules and operations which apply to data types generally. Identifiers are restricted to a length of six characters. Variables and function identifiers may be typed explicitly, or else implicitly with the aid of the IMPLICIT statement. There are no less than three statements which can determine the dimensions of an array variable: explicit type declaration statements, the DIMENSION statement, and the COMMON statement. Expressions may be passed as arguments to subprograms. All of these rules also apply to the extension data types.

The extension type GRAPHICAL is essentially a pointer type. Some of the declarations possible for non-pointer types, therefore, are not permissible with this type. Graphical variables may not be initialized, as in a DATA statement. They may not be equivalenced or placed in a common block position which is also referenced by a non-graphical variable. Finally, statement functions are not allowed for this type.

2.4 Separate Preprocessing and Compilation

In computer science, there is always the question of the effect a programming style has on the ability to easily generate correct, maintainable solutions to problems. A style known as modularization [Dijk76] has its origins in classical problem solving: splitting a problem so that when all of the subparts are solved, the overall solution has been obtained. Modular

programs are written in FORTRAN with the aid of subprograms.

Being able to create and individually test modules of a program has at least two benefits. The first is that it is much easier to isolate errors in a small, uncomplicated module than it is in a complete program. The second is that once a module is complete and correct, it can be used in any program which requires a similar problem to be solved. In this fashion, problems need only be solved once.

To promote modular programming and the creation of libraries of useful graphical routines, the FORTRAN module or subprogram is fully supported as a design goal of LIG6. Function subprograms of type GRAPHICAL are included, and graphical variables and function identifiers may be parameters of a subprogram. Consistent with FORTRAN, variables which are not parameters are local to a subprogram and are automatically initialized upon entry to the routine. The creation and use of libraries of effective graphical routines will allow the system to evolve into a powerful design and visualization tool.

Chapter 3

DATA BASE SYSTEM

A high-level programming language has several advantages over an assembly-level language. These advantages include portability of programs and programmers, ease of learning the language, improved readability of programs, and the removal of detail irrelevent to an application. Similar advantages are evident when comparing a high-level graphics language with subprogram packages such as CORE [GSPC79] or IG [Mair81].

The lower level constructs of LIG6 preserve the concept of data abstraction with regard to graphical data. A conventional programming language allows the natural definition and manipulation of numbers and logical values without requiring the programmer to know about the internal representation or allocation of these data types. Similarly, lower level LIG6 constructs permit the treatment of graphical data in an abstract fashion via the simple definition and presentation of models of graphical objects. Use of the advanced level constructs, especially those involving inquiry, require some knowledge of the implementation of the language and of its data base.

To some extent, the use of lower level constructs of both conventional programming languages and LIG6 also requires that some knowledge of implementation be acquired. In FORTRAN, for example, a programmer must be aware of the quantity of memory

allocated for various data types when using COMMON blocks and of the storage mechanism for multi-dimensioned arrays. In LIG6, the data type GRAPHICAL must be recognized as being a pointer type so that recursive definitions are not made. The ability to produce sophisticated programs increases with the amount of knowledge of implementation.

Knowledge of how character data is packed in various FORTRAN data types enables the user to make string comparisons using integer arithmetic. Exploitation of the variant part of structured records of PASCAL allows coercion between data types. One of the reasons assembly languages are used is the freedom a programmer has for implementations. Being the designer of an implementation which can be tailored to fit the application, the programmer is the one best qualified to use it.

To enable the use of the advanced level LIG6 constructs by a programmer, certain information about the implementation must be provided for him. This information includes a model of the data base and the effects of graphical modelling on the data base. Advanced level constructs are explained in terms of their data base effects as well as their graphical effects. These constructs are complete in the sense that all possible data base values may be created with their use. Their availability ensures that a programmer may precisely model and manipulate graphical objects within the bounds of the capability of the data base implementation. It is to the advantage of programmers using the advanced level constructs to view LIG6 as a data base management system with graphical side-effects.

There is no degrading of the system from its high-level

graphics programming language status when this view is taken. Lower level constructs do not require knowledge of the data base model and advanced level users are not required to manage the data base. New elements are allocated automatically, system interfaces are taken care of, the graphical interpretation of the data base is built in, and discarded elements are garbage-collected by the system. By providing the implementation information to programmers, the most effective use of the system becomes possible.

3.1 Data Base Model

Graphical data is inherently hierarchical in nature. This is reflected in the implementation of the data base. For the purpose of using the advanced level constructs of LIG6, the graphical data base can be visualized as a binary tree. This is a dynamically alterable n-level hierarchical data base [FOLE82]. Leaves of the tree are graphical output primitives. Modelling transformations and attribute definitions are stored in the nodes of the tree. This model of the data base system is illustrated by the following synonym assignment statements where ":-" means assignment and "+" means superposition. Their resulting tree structure is displayed in Figure 1.

```
A :- LINE FROM (X1,Y1,Z1) TO (X2,Y2,Z2) TO (X3,Y3,Z3)
B :- POLY FROM (P1,Q1,R1) TO (P2,Q2,R2) TO (P3,Q3,R3)
C :- 'HI THERE' < MOD 0 >
D :- A < MOD 1 > + B < MOD 2 >
E :- D < MOD 3 > + C < MOD 4 > + LINE FROM (T1,U1,V1)
                                  TO (T2,U2,V2)
```

In the representation of the binary tree in Figure 1, the pointer to the right of each node is called the value pointer.

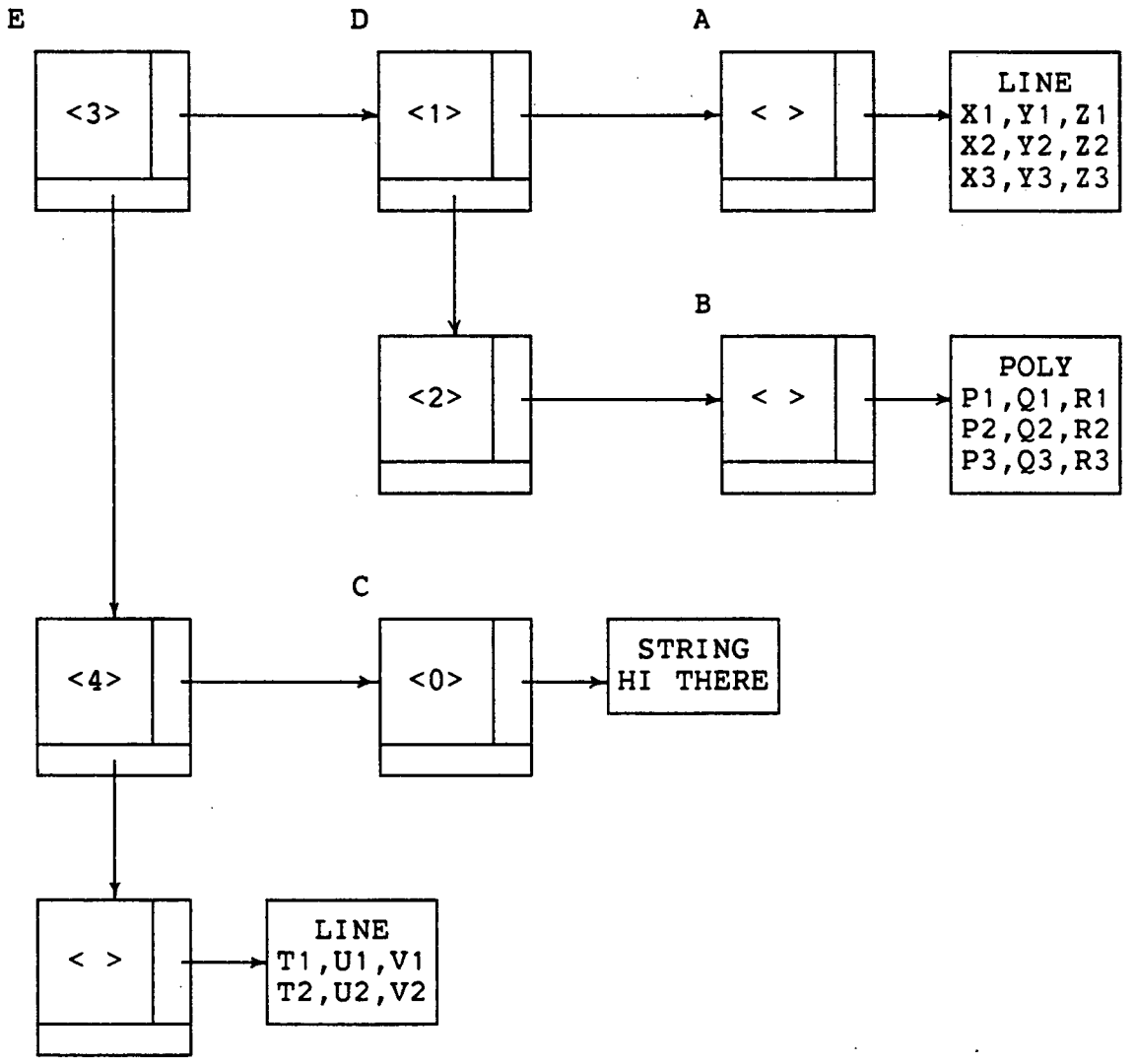


Figure 1 Data Base Model

The value of a node is the object model which its value pointer refers to, modified by the transformations stored in the node. A value pointer can point to another node or to a leaf of the tree. Leaves are graphical output primitives. The downward pointer from each node is called the super pointer. Whenever a graphical object is superimposed on another, the modelling effects are the following: create a node, direct the value pointer of the new node to the superimposing object, store any

modifications specifying the instance of that object in the new node, and direct the super pointer of the original node to the new node.

For access to subobjects, two graphical system functions are provided - VALUE and SUPER. They each take one argument of type graphical (which is a pointer type) and return the appropriate pointer of that argument's node. As for all functions in FORTRAN, these system functions must have their type declared, either implicitly or explicitly, before use. An example of their use is

```
GRAPHICAL A,B,C,D,E,F,VALUE,SUPER
A :- B + C
D :- E + A
F :- VALUE(SUPER(VALUE(SUPER(D))))                    .
```

The last statement is equivalent to

```
F :- C                                                .
```

The corresponding tree representation is presented in Figure 2.

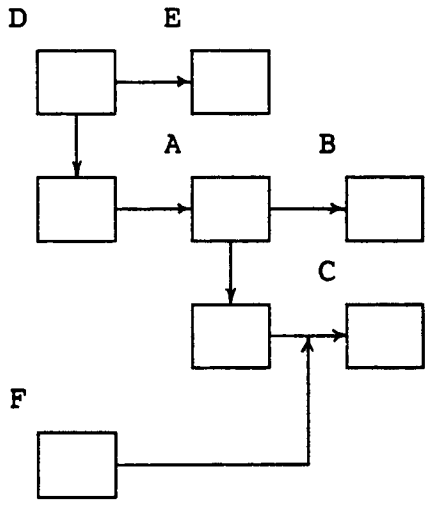


Figure 2 VALUE and SUPER Functions

Use of the VALUE and SUPER functions requires that the programmer knows the tree structure of a particular model. The structure, however, is known because it is determined by the assignment statements he has used to specify the object. In light of this, nested graphical expressions take on a new meaning.

For simple use of the language, nested graphical expressions merely provide an easy method for applying a transformation to more than one graphical object. An experienced LIG6 programmer can use nested expressions, however, to precisely specify the structure of his model as well as its external representation. The two groups of statements

```
A1 :- B + C + D + E
DISPLAY A1

A2 :- (B + (C + D) + ((E)))
DISPLAY A2
```

will cause the same image to appear on the screen but the structure of the models will be different. The structure of a model can convey information to a program as effectively as the contents of the structure's nodes. The two structures of the above assignment statements are shown in Figure 3.

To enable a programmer to exert maximum control over his models, four types of graphical assignment statements have been provided. They are the synonym assignment, copy assignment, value assignment, and the super assignment statements.

Synonym assignment has already been introduced. This is the only graphical assignment statement needed by a programmer using simple level language constructs. Such a programmer does not need to know the effect of this assignment on tree structures or

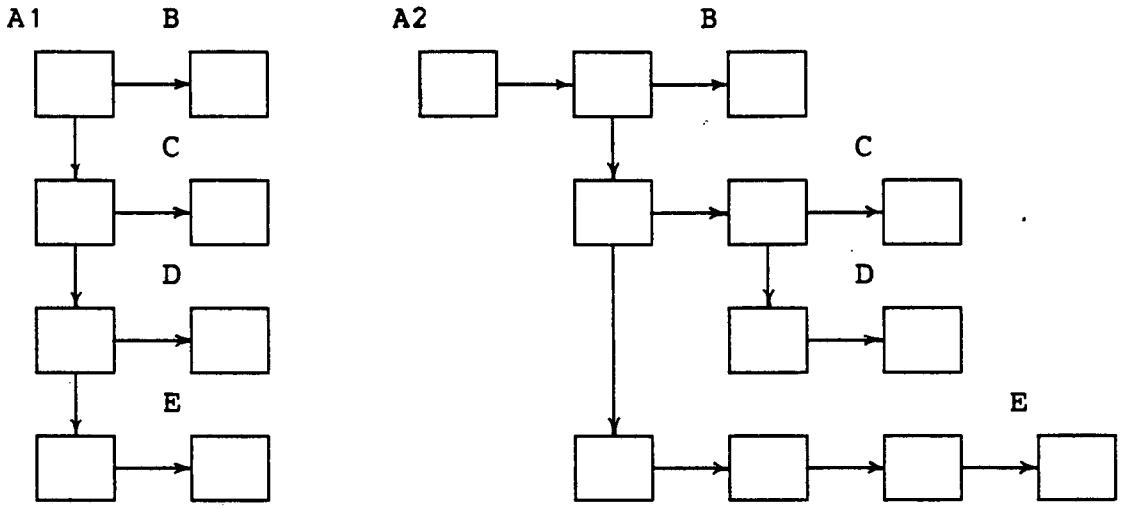


Figure 3 Nested Graphical Expressions

even that the data base implementation can be modelled as a tree structure. The assignment operator for this type is the symbol ":-". It redefines the value and super pointers and the contents of the node and automatically creates new nodes for superposition at the top level.

The synonym assignment statement allows the most general form of expression to the right of its operator of all of the assignments. The production for this expression can be called <graphexpress>. The Backus-Naur form definition for this production is

```
<graphexpress> ::= <graphterm>
                 | <graphexpress> + <graphterm>

<graphterm> ::= <graphfactor>
              | <graphfactor> <modificationlist>

<graphfactor> ::= <graphprimitive>
                | <graphprimary>
                | ( <graphexpress> )

<graphprimary> ::= <graphvariable>
                 | <graphfunction>
```

The copy assignment statement also redefines all aspects of a node. Its operator is ":=". The effect of such an assignment is simply to copy the pointers and transformations stored in the node specified on the right hand side into the node specified on the left hand side. Subsequent display of either node would yield identical visual results. The expression on the right must be a <graphprimary>, that is, a graphical variable or function with no modifications or superpositions. Consider the following statements.

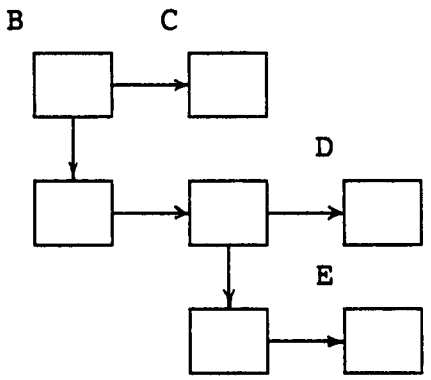
```
 i)   B :- C + (D + E)
ii)   A :- B
iii)  A := B
iv)   A := VALUE(SUPER(A))
```

If each statement were executed in order, the corresponding structures in Figure 4 would be generated.

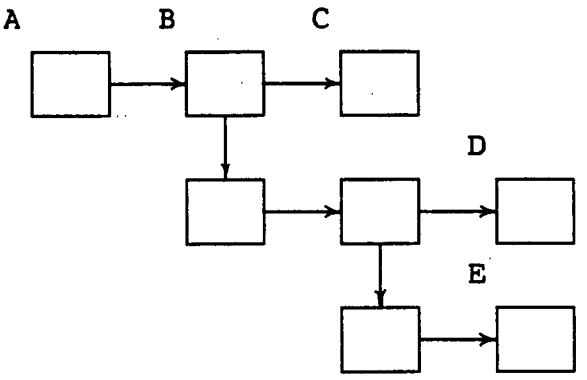
The value assignment statement affects the value pointer of a node and the transformations stored in it. The super pointer is not affected. The value assignment operator is ":>". Its effect is to change the initial value definition of a node, but not any objects which have been superimposed upon it. The expression on the right must be a <graphterm>, that is, the same as a <graphexpress> except that no superposition is allowed. The following statements generate the tree structures of Figure 5.

```
 i)   A :- B + C
ii)   A :> (D + E)<COLOUR 120>
```

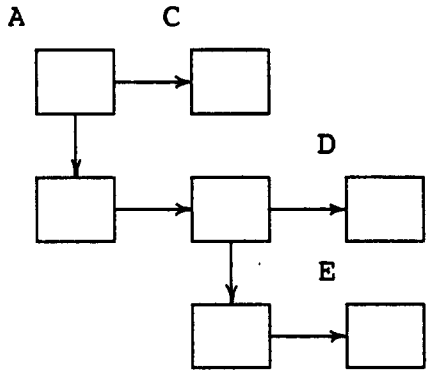
The super assignment statement redefines the super pointer of a node. The value pointer and the transformations stored in the node are not affected. Its operator is ":<". Its effect is to replace all objects that have been superimposed on an initially defined object by a new object. The expression on the



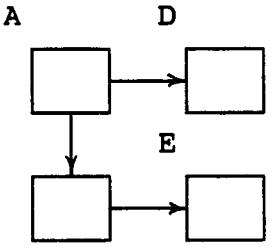
(i)



(ii)

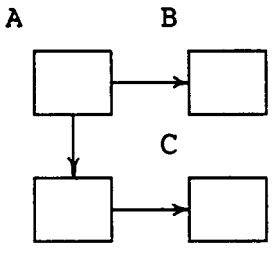


(iii)

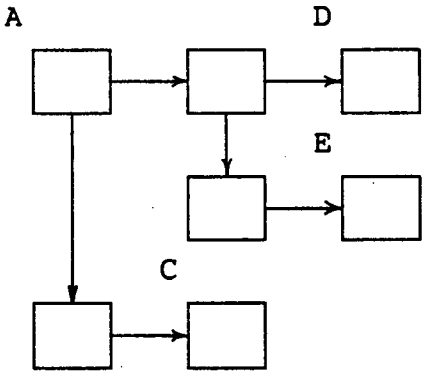


(iv)

Figure 4 Copy Assignment



(i)



(iv)

Figure 5 Value Assignment

right must be a <graphfactor>, that is, the same as a <graphterm> but no modifications are allowed. The following statements yield the structures depicted in Figure 6.

```
  i)   A :- B + C
 ii)   D :- E + B
iii)   A :< SUPER(D)
 iv)   D :< (B<COLOUR 120> + (C))
```

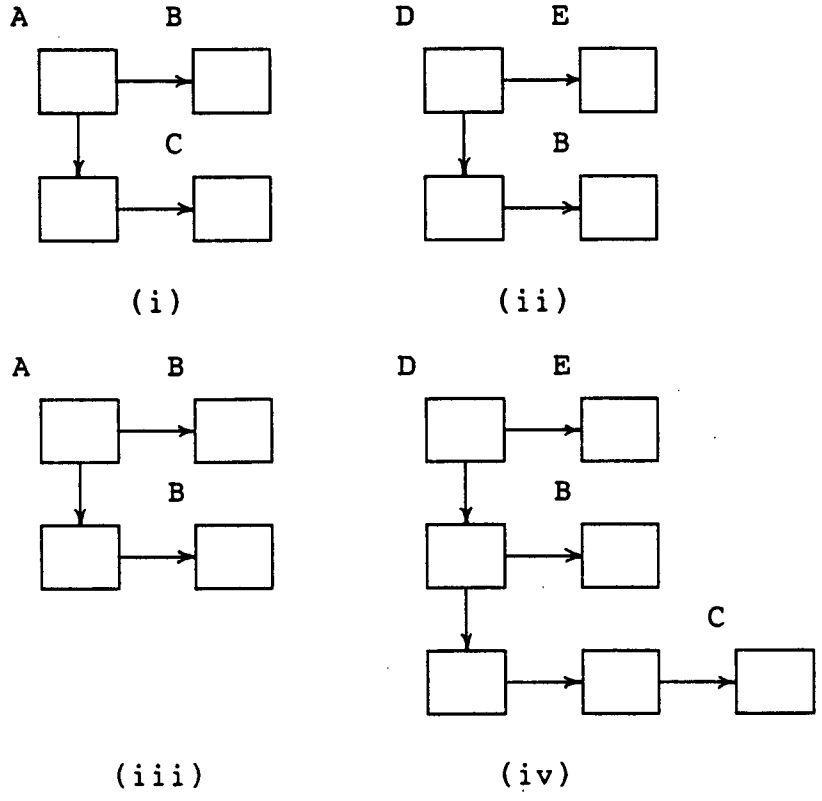


Figure 6 Super Assignment

The preceding four assignment statements form a complete set of operators. They enable programmers to create any binary tree structure they desire. Additionally, constructs are provided which enable operations on the leaves of the tree, the output primitives.

3.2 Data Base Implementation

The previous section discussed the model of the data base of LIG6. The use of the word model is important; it emphasizes that information about the fine details of the data base implementation are not needed by users of advanced level constructs. To reinforce this point, a summary of the implementation will now be given.

The nodes depicted in Figure 1 could be stored using a variety of methods: as arrays in common blocks, as lines or groups of lines in disk files, or as records in dynamically acquired virtual memory. The first approach fixes the size of the data base. Small programs would pay the price of high memory charges for the unused portions required for larger programs. The second method has high overhead in disk charges and execution time. The last method uses only as much memory as a program requires but does not have the overhead that secondary storage involves. It is the approach chosen for the implementation of the LIG6 data base.

Each node requires 24 four byte words of contiguous memory. The fields of each node are of three different types: fullword REAL, fullword INTEGER, and halfword INTEGER. Because the nodes are stored in virtual memory, access to them involves pointers and a system subroutine call. Access to the fields of different types is accomplished by passing the pointer three times with a different type declaration for each. The following statements illustrate this.

```
.....
EXTERNAL ACCESS
CALL CALLER(ACCESS,POINTR,POINTR,POINTR)
.....

SUBROUTINE ACCESS(REALA,INTA,INT2A)
REAL REALA(24)
INTEGER INTA(24),INT2A*2(48)
.....
```

The fields of a node store the modelling transformations and attribute settings and the instance definitions specified by graphical assignment statements. The structure of a node is given in Figure 7.

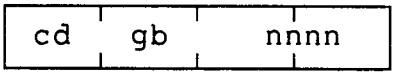
Position		Name	Type	Use
Halfword	Fullword			
	1-12	TRMAT	REAL	transformation matrix
	13	COLOUR	REAL	interior hue attribute
	14	LITNES	REAL	interior lightness attr.
	15	SATUTN	REAL	interior saturation attr.
	16	PATERN	INTEGER	interior pattern attribute
	17	BCOLOR	REAL	border hue attribute
	18	BLITNS	REAL	border lightness attribute
	19	BSATUN	REAL	border saturation attr.
	20	BPATRN	INTEGER	border pattern attribute
	21	STYLE	INTEGER	line style attribute
43		WIDTH	INT*2	line width attribute
44		FONT	INT*2	textstring font number
45		VALUE	INT*2	value pointer
46		SUPER	INT*2	superposition pointer
47		INSTAN	INT*2	instance pointer
48		GARBGE	INT*2	garbage collector storage

Figure 7 Graphical Node Structure

Every graphical variable is an integer halfword pointer to a node. These pointers are not the actual virtual memory addresses of the nodes (which would require a fullword for pointer storage), but indices into an array of nodes. This array is organized as blocks of 42 nodes. Each block is approximately one page (4096 bytes) of virtual memory. As more nodes are

required, the array is dynamically expanded one block at a time. Each block is referenced by an element in an array of pointers. Due to pointer precision, the maximum number of blocks is 780. This represents three megabytes of storage. As the maximum size of the array of pointers to blocks is thus 780, the array is dynamically kept in virtual memory as well.

The storage for primitive records is organized in a different fashion. Each record is a contiguous section of a large one-dimensioned array. When a graphical pointer is negative, it represents the negative index of the start of the primitive record in the array. This array is also dynamic; it is organized as blocks of 1024 words which are acquired when necessary. A record consists of a header, the primitive information, and a trailer. The header consists of one word; it contains what type of primitive follows, garbage collection storage, and the length of the record. The word organization is depicted in Figure 8.



cd - Primitive style indicator
01 = POLYLINE
02 = POLYGON
03 = TEXTSTRING
04 = USER DEFINED PRIMITIVE

gb - Garbage collector storage

nnnn - Length of record

Figure 8 Primitive Record Header

The primitive information depends on the type of primitive. For polylines and polygons, it consists of groups of x,y, and z

coordinates. For textstrings, it consists of bytes of character data. For user-defined primitives, it consists of an external representation procedure pointer and a list of parameters. The trailer is either a continuation or an end command. If it is an end command, that is the end of the primitive. If it is a continuation command, it contains a pointer to the part of the array where the primitive is continued. This allows for records larger than 1024 words and for concatenation assignment. The following LIG6 program would create the primitive storage presented in Figure 9 which was produced by the system debug dump, LIGDPH.

```
GRAPHICAL A,B,C
A :- POLY FROM (1,2,3) TO (4,5,6) DELTA (3,3,3)
B :- 'Line at'
C :- A + LINE FROM (-1,2) TO (3,-4)
B :+ ADDSTRING IVALUE(9+7,5)
CALL LIGDPH
STOP
END
```

Free lists are kept for both nodes and primitive record areas. Whenever statements are executed which require the acquisition of storage, the storage is acquired from the appropriate free list. If the free list is empty, the garbage collector is invoked. If the amount of storage recovered by the collector is less than a set amount (42 nodes or 256 words of primitive storage), another page of virtual memory is acquired and initialized.

The garbage collector operates as follows. Whenever a subprogram is entered, all local graphical variables used are allocated nodes and a list of those nodes is kept in a protect list in virtual memory. The execution of a RETURN statement

```
Heap space dump - Page  1
Address  Contents  Explanation
      1  02000003  Polygon          3 vertices
      2  41100000  x-component      0.1000000E+01
      3  41200000  y-component      0.2000000E+01
      4  41300000  z-component      0.3000000E+01
      5  41400000  x-component      0.4000000E+01
      6  41500000  y-component      0.5000000E+01
      7  41600000  z-component      0.6000000E+01
      8  41700000  x-component      0.7000000E+01
      9  41800000  y-component      0.8000000E+01
     10  41900000  z-component      0.9000000E+01
     11  00000000  End
     12  03000008  String           8 characters
     13  D3899585  Text - Line
     14  4081A340  Text -  at
     15  05000018  Continuation located at    24
     16  01000002  Line             2 vertices
     17  C1100000  x-component     -0.1000000E+01
     18  41200000  y-component      0.2000000E+01
     19  00000000  z-component      0.0
     20  41300000  x-component      0.3000000E+01
     21  C1400000  y-component     -0.4000000E+01
     22  00000000  z-component      0.0
     23  00000000  End
     24  03000004  String           4 characters
     25  4040F1F6  Text -   16
     26  00000000  End
     27  07E50101  Free space    229 words, continued at    257
    257  07FF0201  Free space    255 words, continued at    513
    513  07FF0301  Free space    255 words, continued at    769
    769  07FF0000  Free space    255 words, continued at      0
```

Figure 9 Primitive Record Storage

causes those nodes to be removed from the protect list. The garbage collector starts from the protect list and recursively marks the nodes which are defined by the list. When this mark phase is completed, all of the node and primitive storage is swept through with unmarked areas being added to the appropriate free list.

The details of the implementation of the data base and the operations which manage it are hidden from a LIG6 programmer. The model of the data base provides enough information to effectively model and manipulate graphical objects using the

advanced level constructs.

Chapter 4

GRAPHICAL OUTPUT PRIMITIVES

Four graphical output primitives are supplied by LIG6: BLANK, POLYLINE, POLYGON, and TEXT. Theoretically, any image which can be displayed on an output device of finite precision can be created using these primitives, thus they form a complete set of atomic graphical objects. As a convenience, however, LIG6 also permits programmer-defined primitives. This can simplify modelling of well-defined graphical objects.

4.1 Primitives as Abstract Data Types

In earlier versions of LIG [Schr76], primitives were viewed as constants of the data type GRAPHICAL. Six two-dimensional primitives were defined: BLANK, LINE, TRIANGLE, SQUARE, CIRCLE, and SCIRCLE (semi-circle). Only the first two were required; the rest could be constructed from the primitive LINE. The LINE primitive was defined as a line segment from the point (0.0,0.5) to the point (1.0,0.5). All other line segments were created by applying transformations to this constant. Polylines were created by superimposing transformed line segments.

The view taken by the current version, LIG6, is that primitives are data types in their own right. For any data type supported by a language, rules exist which define how constants

of that type are expressed, also, operators which form legitimate expressions of that type are specified, and procedures are supplied which produce external representations of values of that type.

Consider the data type COMPLEX. Its standard implementation is a record consisting of a real and an imaginary field each of type REAL. A FORTRAN complex constant consists of an opening parenthesis followed by a real constant representing the real field, a comma, a real constant representing the imaginary field, and a closing parenthesis. When operations are performed on this type, the contents of the individual fields are used to determine the result. Specifically for output, the real part and the imaginary part are printed side by side.

The primitive POLYLINE of the language LIG6, for example, has all of the features of an abstract data type. It is implemented by a record with a variable number of fields of type VECTOR. Each field represents a vertex in a line and, on output, is accessed to generate the graphical commands given to the output device. While there are no direct variables or operators for this type, the type GRAPHICAL (which supports variables and operators) can be considered to contain a field which can reference polylines.

The external representation procedures of conventional data types generally produce symbols on printers or terminals which have the same representation as constants of that type. A recursive PASCAL procedure for generating external representations of unsigned integers is presented in Figure 10; it has the above characteristic. This practice results from the

existence of a standard representation which can be used both in the specification of a program and in the results generated by executing that program. There is no rule that both representations must be the same; integers are often expressed in other bases or even as Roman numerals. This is the case with the type POLYLINE; such constants are expressed in programs with alphanumeric characters using definite syntactic rules while external representations are lines generated on graphical devices. The latter representation would probably be used for constants if it were available.

```
PROCEDURE ext_rep_integer( n : INTEGER ) ;

BEGIN
  IF n < 10 THEN write( CHR( n + ORD('0') ) )
  ELSE
    BEGIN
      ext_rep_integer( n DIV 10 );
      write( CHR( n MOD 10 + ORD('0') ) )
    END
END;
```

Figure 10 Integer External Representation

The records of the data types POLYLINE, POLYGON, and TEXT can have a variable number of fields; such data types are known as dynamic. Dynamic data types are not common as most programming languages do not support them. Both SNOBOL and BASIC, however, have dynamic string data types.

4.2 Programmer Defined Primitives

The definition of a primitive is equivalent to the creation of an abstract data type. The treatment of primitives by LIG6

requires that a data type definition includes constant specification rules, internal representation information, and external representation procedures. The construct with which a programmer creates a LIG6 primitive is a primitive definition unit.

A primitive definition unit has a prescribed structure. First a symbolic name is specified for the primitive. A definition rule must follow in the form of a template which constants of that type will match. The template consists of type declarations and a grammar production using a notation similar to Wirth's [Wirt77]. The remainder of the subprogram consists of statements which define a procedure for producing an external representation of values of that primitive. The internal representation is derived automatically from the constant definition rule.

There are a variety of methods used to define grammars; Backus-Naur forms and syntax directed diagrams are two examples. Wirth's notation provides a general description mechanism which allows iteration, alternation, option, and recursion constructs to be expressed. The grammar production in a LIG6 primitive definition unit template uses a subset of this notation since only alternation and option constructs are permitted. The allowable grammar productions can themselves be represented by a grammar; this is presented in Figure 11 using Wirth's notation.

In a LIG6 primitive template, an alternation construct is a list of templates separated by vertical bars and enclosed by parentheses. An option construct is a template enclosed by brackets optionally followed by the explicit setting construct

```
production = primitive_identifier "::=" subtemplatelist ";"

subtemplatelist = subtemplate { subtemplate }

subtemplate = ( terminal          | alternation          |
                option            | explicit_setting     |
                nonterminal       )

terminal = ( literal_string | field_identifier )

literal_string = "'" { ( character | "''" ) } "'"

alternation = "(" subtemplatelist { "|" subtemplatelist } ")"

option = "[" subtemplatelist "]" [ explicit_setting ]

explicit_setting = "<" setting { "," setting } ">"

setting = field_identifier "=" expression

nonterminal = primitive_identifier "(" { character } ")"
```

Figure 11 Allowable Primitive Template Grammar Productions

which defines the default. The nonterminal construct allows the insertion of the template of a previously defined primitive. The characters in the list following the identifier are appended to all of the field identifiers of the previously defined primitive when it is inserted. This allows multiple use of the primitive in a template. A template which demonstrates alternation and option constructs and constants which match it are presented in Figure 12. A template which demonstrates the use of nonterminals and constants which match it are shown in Figure 13.

The body of the primitive definition unit defines the external representation procedure for that data type. It consists of statements which, based upon the values of the fields of the data type, produce images on output devices. The body may contain any type of statement except those which deal with graphical modelling or display. These statements are excluded because they deal with information at a higher level.

```
PRIMITIVE SQUARE
LOGICAL FILL
REAL SIDE
VECTOR CENTRE
SQUARE ::= 'SQUARE' [ 'FILLED' <FILL=.TRUE.> ]<FILL=.FALSE.>
           [ ( 'AT' CENTRE [ ',SIDE' SIDE ]<SIDE=1.0> |
               'SIDE' SIDE [ ',AT' CENTRE ]<CENTRE=(0,0,0)> ) ]
           <CENTRE=(0,0,0),SIDE=1.0> ;
.....

SQUARE
SQUARE FILLED
SQUARE AT (5.1,5,0)
SQUARE FILLED SIDE 3.0
SQUARE AT (3,2.7,6), SIDE 0.5
SQUARE FILLED SIDE 3*PI, AT (V1#V2)
```

Figure 12 Square Primitive

```
PRIMITIVE ONEARG
INTEGER OPT
VECTOR V
ONEARG ::= ( <OPT=1> 'NW' | <OPT=2> 'NE' | <OPT=3> 'SE' |
             <OPT=4> 'SW' | <OPT=5> 'N'  | <OPT=6> 'E'  |
             <OPT=7> 'S'  | <OPT=8> 'W'  | <OPT=9> 'C'  )
           '=' V ;
.....

PRIMITIVE PARALL
INTEGER OPT1, OPT2, OPT3
VECTOR V1, V2, V3
PARALL ::= 'PARALLELOGRAM'
           ONEARG(1) ',' ONEARG(2) ',' ONEARG(3) ;
.....

PARALLELOGRAM N=(2,3,4),C=(5,2,2),SE=(X,Y,Z)
PARALLELOGRAM NW=(1,1,1),NE=(5,2,2),SW=(X,Y,Z)
```

Figure 13 Parallelogram Primitive

Two statements are provided which cause output to occur, the DRAW and DRAW WITH statements.

The form of the draw statement is the keyword DRAW followed by a previously defined primitive, either one of the four basic primitives or one which was programmer-defined. A complete primitive definition unit for a square primitive follows.

```
PRIMITIVE SQUARE
REAL SIDE
SQUARE ::= 'SQUARE' SIDE ;
DRAW LINE FROM (0,0)       TO (0,SIDE)
           TO (SIDE,SIDE) TO (SIDE,0)
           TO (0,0)
RETURN
END
```

Whenever a model whose specification includes a SQUARE primitive is displayed, the external representation procedure is invoked. All of the modelling transformations and attribute settings of the model affect the procedure. In addition to primitives, appropriate concatenation expressions may follow the DRAW keyword. An example of this usage is

```
      PRIMITIVE CIRCLE
      CIRCLE ::= 'CIRCLE' ;
      REAL PI/3.141593/
      DRAW LINE FROM (1,0)
      DO 20 I = 1,100
        THETA = I*PI/50
        DRAW ADDLINE (COS(THETA),SIN(THETA))
20    CONTINUE
      RETURN
      END
```

Concatenation expressions may only be executed if the last DRAW statement executed was a concatentation or primitive of the same type.

Attributes of the draw statement may be changed with the DRAW WITH statement. Its form is the keyword DRAW WITH followed by a list of attribute settings enclosed in angle brackets. Any attribute which has not been specified in the modelling may be set by such a statement.

Programmer-defined primitives permit efficient and concise modelling of regular or parameterized graphical objects. An example of a primitive which models spheres constructed with

arbitrary resolution and illuminated by an arbitrary light source is presented in Figure 14.

4.3 Graphical Functions Versus Graphical Primitives

Strictly speaking, the capability of programmer definition of graphical output primitives is not necessary. Any graphical effect which such a primitive can produce can also be produced by a function of type GRAPHICAL. The differences between primitives and graphical functions are subtle. Graphical functions are executed as soon as they are invoked, returning a model of a graphical object. Such models require memory space for storage and have structures and values which can be subsequently manipulated.

An invocation of a primitive, on the other hand, does not result in execution of code. The parameters of the primitive are stored and it is not until the value of that primitive is to be displayed that the external representation procedure is invoked. This results in savings in both memory and execution time.

The decision whether to use a primitive or a graphical function, therefore, should be made using the following guidelines. If the resulting object is always treated as a unit or if it contains curves or subobjects which can be easily parameterized, then a primitive implementation should be considered. If an object has a definite structure and hierarchy and subparts of it will be accessed and possibly modified, a graphical function implementation should be considered.

```
      PRIMITIVE SPHERE
      INTEGER RES,PATERN
      VECTOR LIGHT,LSORCE,P1,P2,P3,P4,TX,TY,TZ,NORM
      SPHERE ::= 'SPHERE' RES ',' LIGHT ;
      PI = ATAN(1.)*4
      ARC = PI/RES
      LSORCE = LIGHT/|LIGHT|
      TX = (1,0,0)
      TY = (0,COS(ARC),SIN(ARC))
      TZ = (0,-SIN(ARC),COS(ARC))
      DO 10 I = 1,RES
        P1 = (1,0,0)
        P3 = P1
        DO 10 J = 1,RES
          P2 = (COS(J*ARC),SIN(J*ARC)*COS(I*ARC),
                           SIN(J*ARC)*SIN(I*ARC))
          P4 = ((P2.TX),(P2.TY),(P2.TZ))
          NORM = P1 + P2 + P3 + P4
          COSANG = (NORM.LSORCE)/|NORM|
          PATERN = 0
          IF(COSANG.GT.0) PATERN = 24.*COSANG + 1.5
          DRAW WITH <PATTERN PATERN>
          DRAW POLY FROM (P1) TO (P2) TO (P4) TO (P3)
          P1 = P2
          P3 = P4
10    CONTINUE
      RETURN
      END
```

Figure 14a Sphere Primitive Definition

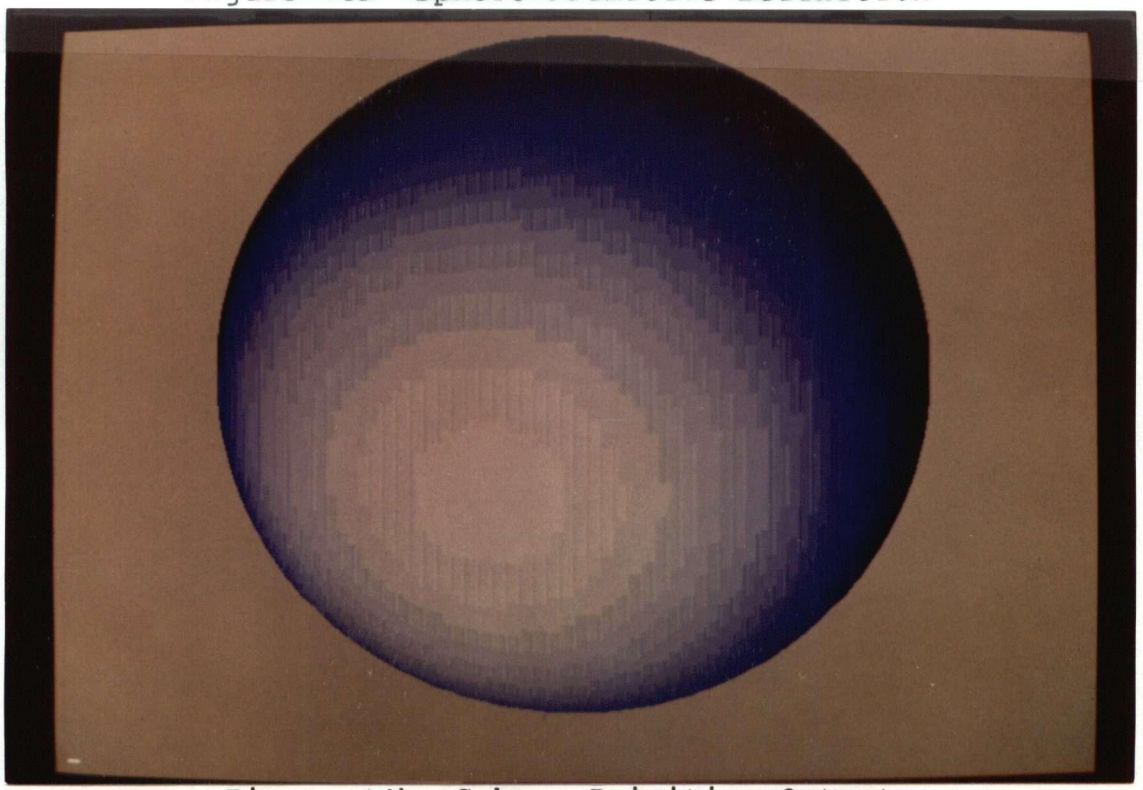


Figure 14b Sphere Primitive Output

Programmer-defined graphical output primitives have been used in other languages but with different terminology and emphasis. Earlier versions of LIG had a construct called a graphical function [Schr78]. Its effect was to store the arguments and a pointer to the function in the data base. It was only when a model which contained a reference to the function was displayed that the function was executed. This differs markedly from the standard concept of a function which is supported by LIG6. LIG6 graphical functions are executed immediately upon their invocation and return a model of a graphical object. The early version functions were in fact an implementation of programmer-defined primitives although they lacked the generality of the current implementation. They had a maximum of six parameters, were invoked by a fixed structure, and could only draw lines, not all previously defined primitives. The actual syntax definition provided by LIG6 was not permitted.

The language MIRA allows the definition of graphical types [Thal79,Magn81]. The language implementation of these types, however, is more in line with graphical functions. The type declaration includes modelling operations, not display commands. The type is invoked via a procedural construct, not by assignment statements containing constants of the type. The modelling process occurs immediately upon invocation requiring storage and execution time; it is not delayed until an instance is actually displayed. In addition, the pattern matching facility of LIG6 is not available.

Chapter 5

GRAPHICAL DOMAINS

There are many domains in which graphical programming languages may be applied, but use of a particular graphical programming language will be limited if the domains in which it can be applied are restricted. High-level graphical languages provide simply-expressed constructs for the definition and external representation of graphical data. These constructs allow such languages to be applied to domains at the drafting systems level. More sophisticated applications, however, require the analysis of arbitrary graphical data. The LIG6 constructs which provide inquiry and manipulation of graphical data permit the language to be applied in a variety of interesting domains; four of these will be discussed using a LIG6 example for each.

5.1 Construction Tools

Due to the large quantity of data involved in most graphics applications, the explicit definition of graphical objects is often tedious and time consuming. Graphical application programs must capture as much regularity of the input data as possible to allow economy of gesture in the modelling process. This can be achieved with the aid of construction tools in the form of procedures which create models of complex graphical objects with

minimal input. An example of two construction tools which would be useful in a ductwork application is given in Figure 15. Both could find application in other contexts where regular surface generation is required.

The procedure EXTRUD takes as input an arbitrary cross-section and a direction and length specified as a vector. The cross-section is extruded as specified by the vector, generating a column. The procedure REVOLV takes as input an arbitrary cross-section, an axis specified by two points, an angle, and a step parameter. The cross-section is revolved around the axis by the angle specified in the given number of steps, generating a 3-D figure. Three system procedures are invoked. PRILEN returns the number of vertices of a polyline or polygon primitive, LINPNT returns the specified vertex of a line primitive, and APLYMD applies the transformations stored in a graphical node to a point, returning the transformed value.

Other construction tools which would complement the above two would be procedures to join two dissimilar cross-sections or to compute solid intersections. A complete ductwork application program might include a systematic manipulation procedure which would analyze any models generated and produce patterns for sheet metal construction of the piping. Figures 16 through 19 show two views each of two graphical objects constructed using the EXTRUD and REVOLV procedures.

```
      GRAPHICAL FUNCTION EXTRUD(XSECT,DIREC)
        GRAPHICAL XSECT
        VECTOR DIREC,OLD,NEW
        INTEGER ORDER,PRILEN
        EXTRUD :- BLANK
        ORDER = PRILEN(XSECT)
        CALL LINPNT(XSECT,1,OLD)
        DO 10 I=2,ORDER
           CALL LINPNT(XSECT,I,NEW)
           EXTRUD :- EXTRUD + POLY FROM (OLD) TO (OLD+DIREC)
                                   TO (NEW+DIREC) TO (NEW)
           OLD = NEW
10      CONTINUE
        RETURN
      END

      GRAPHICAL FUNCTION REVOLV(XSECT,AXIS1,AXIS2,DEGRES,STEPS)
        GRAPHICAL XSECT,MODSTR,ONEARC,MDSTR1
        VECTOR AXIS1,AXIS2,OLD,NEW,OLD1,NEW1
        REAL DEGRES
        INTEGER STEPS,ORDER,PRILEN
        MODSTR :- BLANK < MAP (AXIS1),(AXIS2)
                           TO (0,0,0),(0,0,1) ,
                          ROTZ DEGRES/STEPS 'DEG' ,
                          MAP (0,0,0),(0,0,1)
                           TO (AXIS1),(AXIS2)       >
        ONEARC :- BLANK
        ORDER = PRILEN(XSECT)
        CALL LINPNT(XSECT,1,OLD)
        CALL APLYMD(MODSTR,OLD,OLD1)
        DO 10 I = 2,ORDER
           CALL LINPNT(XSECT,I,NEW)
           CALL APLYMD(MODSTR,NEW,NEW1)
           ONEARC :- ONEARC + POLY FROM (OLD) TO (NEW)
                                   TO (NEW1) TO (OLD1)
           OLD = NEW
           OLD1 = NEW1
10      CONTINUE
        REVOLV :- BLANK
        MDSTR1 :- BLANK
        DO 20 I = 1,STEPS
           REVOLV :- REVOLV + ONEARC < MODIFICATION(MDSTR1) >
           MDSTR1 :- MDSTR1 < MODIFICATION(MODSTR) >
20      CONTINUE
        RETURN
      END
```

Figure 15 Graphical Ductwork Construction Tools

5.2 Systematic Manipulation

Systematic manipulation of graphical objects is performed by procedures which process a graphical model of arbitrary

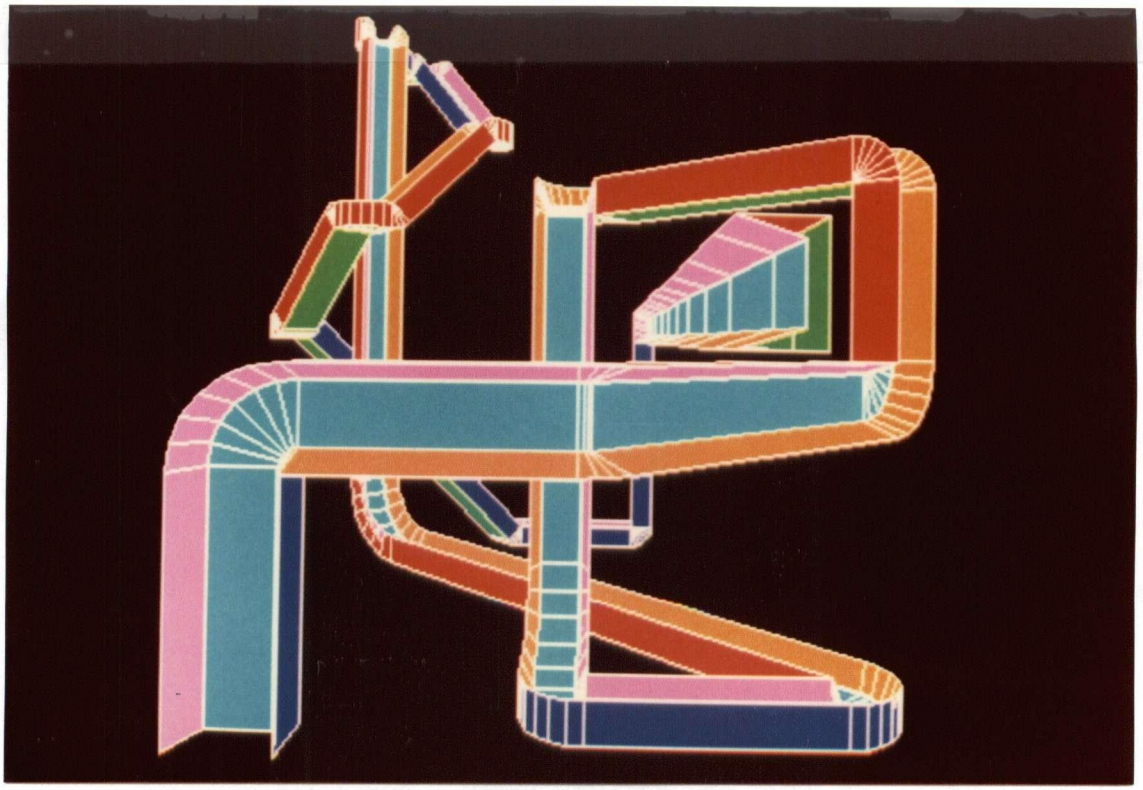


Figure 16 Arbitrary Pipe Model

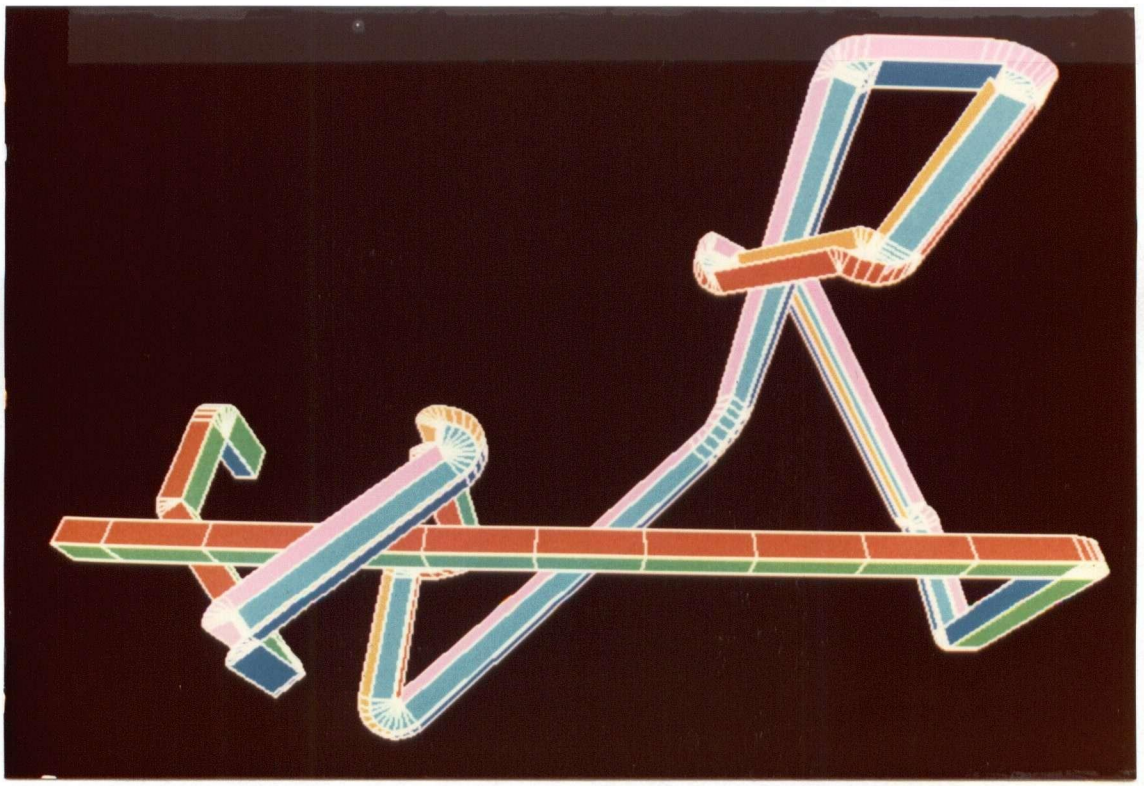


Figure 17 Second View of Pipe

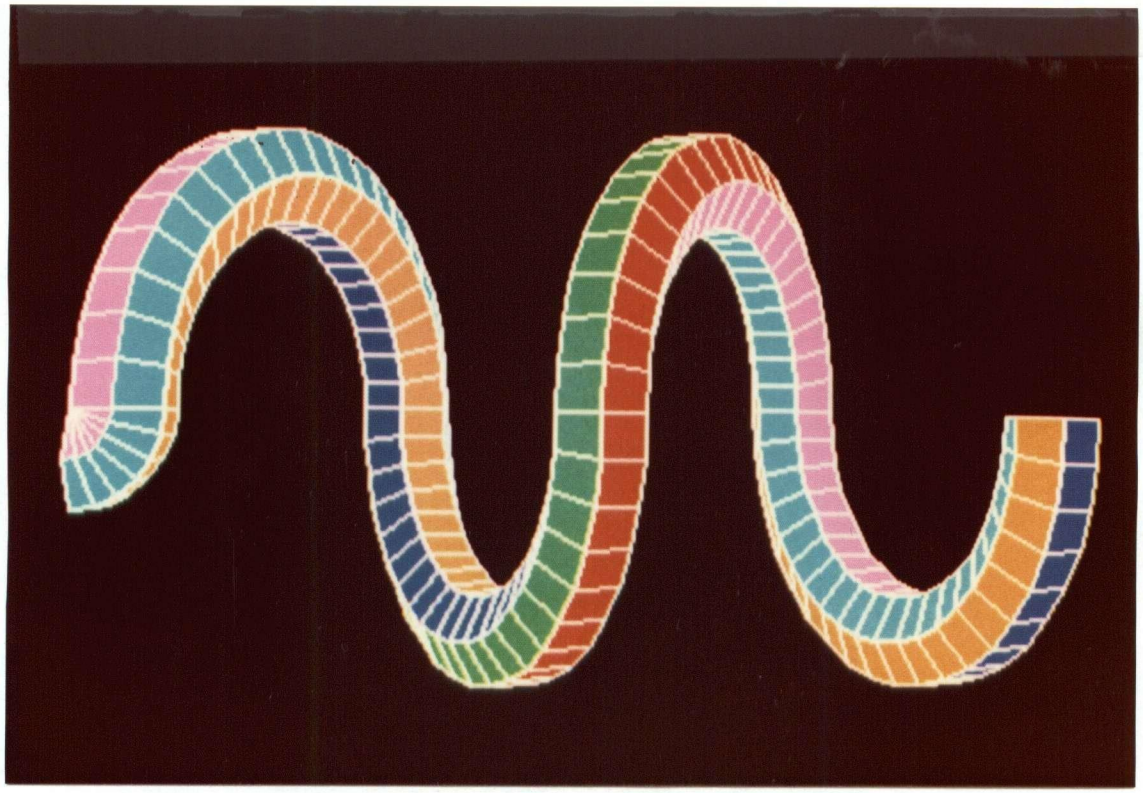


Figure 18 Helix Constructed With REVOLV

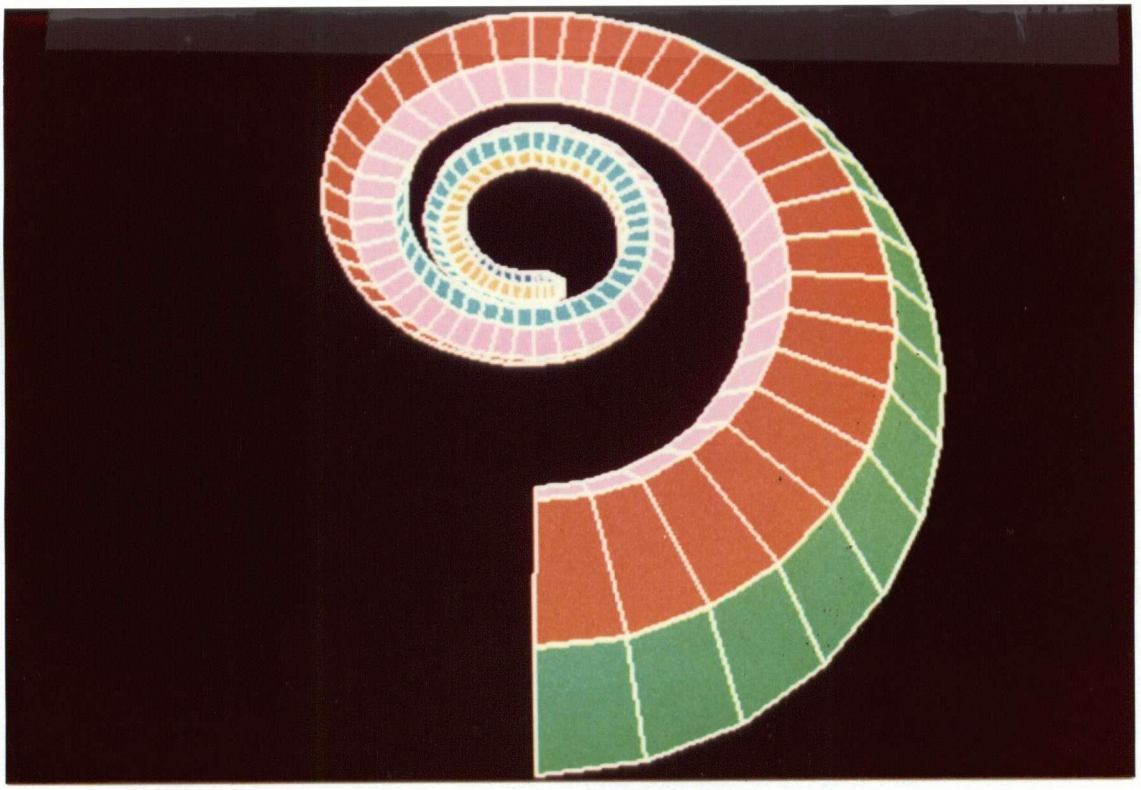


Figure 19 Second View of Helix

structure and content. Such procedures permit application programs to perform modelling at various levels of detail. An example of systematic manipulation is the substitution of one subobject for another in an arbitrary graphical object. To demonstrate LIG6's ability to implement systematic manipulation procedures, a deep replace algorithm is presented in Figure 20.

The problem is divided into two parts. A general double recursion subroutine RECURS performs recursive manipulation of models of graphical objects. The parameter TREE is the model which is to be manipulated. The parameter FIRST specifies a graphical function which controls the first recursion. The parameter WORK specifies a subroutine which performs the desired manipulation whenever FIRST returns a stopping condition. The parameter SECOND specifies a graphical function which controls the second recursion.

The second part comprises four routines which set up RECURS to perform deep replacement. The graphical functions DEEP1 and DEEP2 correspond to FIRST and SECOND, respectively. The subroutine SWAP corresponds to WORK. The subroutine DREPLC invokes RECURS specifying the above parameter assignments. The result of a call to DREPLC is shown in Figure 21. In a graphical object representing a shift register, the graphical symbols for RS flip flops were replaced by their logic gate representations.

Due to the hierarchical nature of graphical data, recursive manipulation of models is desirable. Because LIG6 is an extension to FORTRAN, however, recursive routines are not permitted. This can be overcome easily with routines such as RECURS. Other examples of uses of RECURS are copying trees,

```
      SUBROUTINE RECURS(TREE,WORK,FIRST,SECOND)
      GRAPHICAL TREE,FIRST,SECOND,STACK(25),VALUE,SUPER
      INTEGER POINT
      POINT = 1
      STACK(1) :- TREE
10    POINT = POINT + 1
      STACK(POINT) :- FIRST(STACK(POINT-1))
      IF ( .NOT. PRIMITIVE(VALUE(STACK(POINT))) .AND.
     .       POINT .NE. 25 ) GOTO 10
      POINT = POINT - 1
      CALL WORK(STACK,POINT)
20    STACK(POINT) :- SECOND(STACK(POINT))
      IF ( .NOT. PRIMITIVE(VALUE(STACK(POINT))) ) GOTO 10
      POINT = POINT - 1
      IF ( POINT .NE. 0 ) GOTO 20
      RETURN
      END

      GRAPHICAL FUNCTION DEEP1(NODE)
      GRAPHICAL NODE,REPLAC,WITH,VALUE,SUPER
      COMMON /$REPL$/ REPLAC,WITH
      DEEP1 = VALUE(VALUE(NODE))
      IF ( VALUE(NODE) .NE. VALUE(REPLAC) ) RETURN
      DEEP1 = BLANK
      RETURN
      END

      GRAPHICAL FUNCTION DEEP2(NODE)
      GRAPHICAL NODE,REPLAC,WITH,VALUE,SUPER
      COMMON /$REPL$/ REPLAC,WITH
      DEEP2 = SUPER(VALUE(NODE))
      IF ( VALUE(NODE) .NE. VALUE(REPLAC) ) RETURN
      DEEP2 = BLANK
      RETURN
      END

      SUBROUTINE SWAP(STACK,POINT)
      GRAPHICAL STACK(25),REPLAC,WITH,VALUE,SUPER
      INTEGER POINT
      COMMON /$REPL$/ REPLAC,WITH
      IF ( POINT .EQ. 1 ) RETURN
      IF ( VALUE(STACK(POINT)) .NE. VALUE(REPLAC) ) RETURN
      VALUE(STACK(POINT-1)) :> VALUE(WITH)
      RETURN
      END

      SUBROUTINE DREPLC(TREE,OUT,IN)
      GRAPHICAL TREE,OUT,IN,REPLAC,WITH,DEEP1,DEEP2
      COMMON /$REPL$/ REPLAC,WITH
      EXTERNAL SWAP,DEEP1,DEEP2
      REPLAC :- OUT
      WITH   :- IN
      CALL RECURS(TREE,SWAP,DEEP1,DEEP2)
      RETURN
      END
```

Figure 20 Deep Replace Algorithm

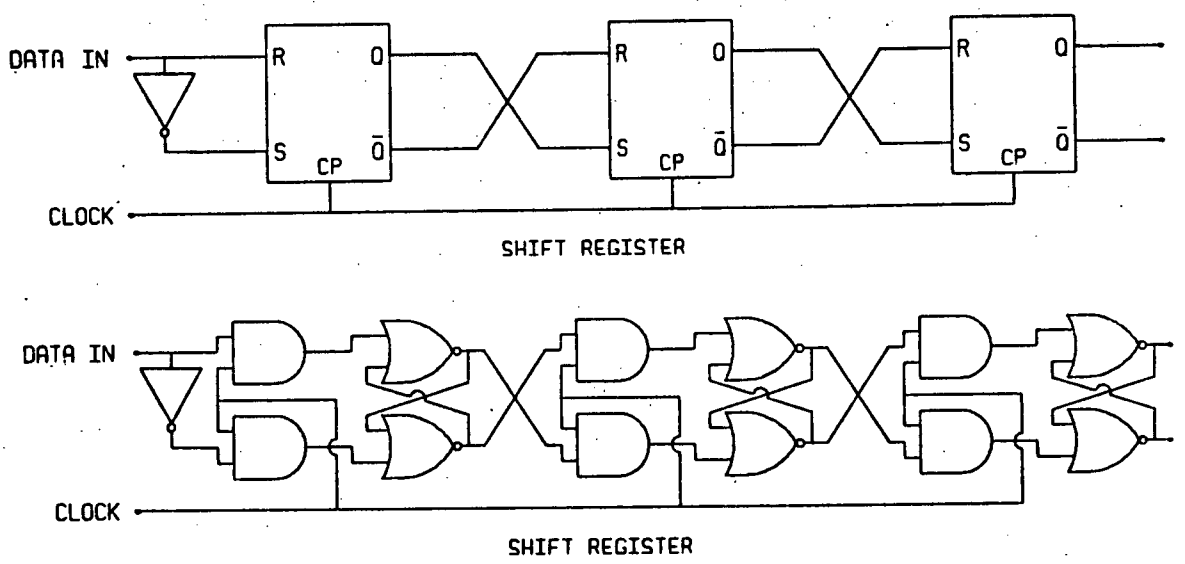


Figure 21 Graphical Substitution

flattening trees, and reversing the order of superposition at any level.

5.3 Graphical Editing

Line file editors are used to manipulate programs, data, and text. The structure and content of the data in a file is modified by editor commands which delete, alter, or insert information. An analogous form of manipulation can be performed on graphical data.

An interactive graphical editor has been implemented using the ability of LIG6 to analyze and modify models of graphical objects. The program generates a visual representation of the structure and content of the model being generated. The representation is analogous to a listing of a file. With the aid

of the representation, the model can be manipulated using editor commands.

The visual representation produced is a drawing of the tree structure of the model. The branches of the tree correspond to the pointers of the model. Within each node of the tree an image of its graphical value is displayed. Using the locator input of the terminal, the model is manipulated by moving pointers, specifying modelling transformations at nodes, inserting or deleting nodes, and other editing functions. As the model is manipulated, corresponding changes in the visual representation occur. The audit trail of a sample editing session is given in Figures 22 through 30.

Figure 22 is the image of the original model and Figure 23 is its tree structure representation. The result of adding to part of the structure is shown in Figure 24. At any time, any portion of the model can be displayed in full size. The graphical value of the previous addition is displayed in Figure 25.

The tree representation is shown only to a limited depth and breadth. Any portion of the model can be displayed, however, by moving a leaf of the representation to the root position. This is demonstrated in Figure 26, enabling part of the model to be manipulated, the structure of which was previously not visible. Figure 27 shows the structure of the subobject after it has been modified. After editing at this level is complete, the structure including the original root is redrawn, resulting in Figure 28.

Altering the structure of the model is demonstrated by Figure 29. Two top level superimposed objects are grouped together so that modelling transformations can be applied in parallel. The final result of the manipulation by editing is shown in Figure 30.

The editing process is independent of the graphical object being manipulated. Electrical diagrams, architectural drawings, or artwork can be edited by the same program in the same manner that programs in various languages or data files can be manipulated by a line file editor.

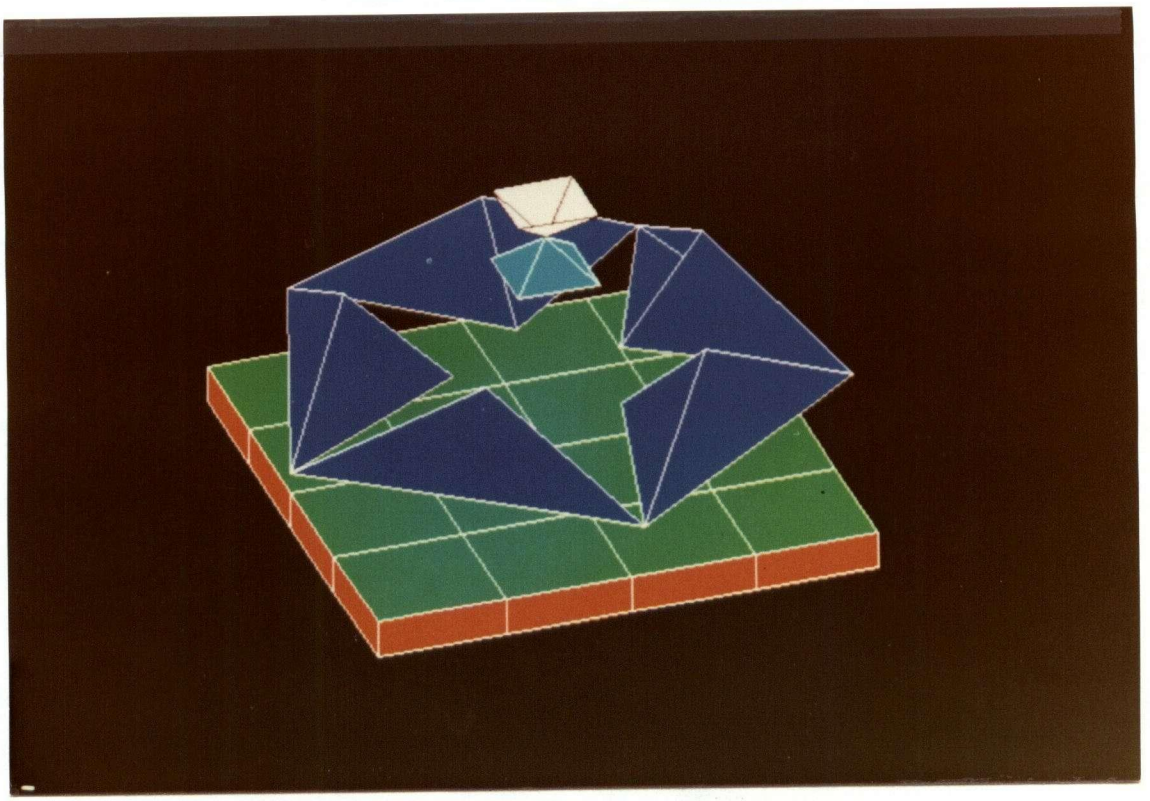


Figure 22 Original Model Image

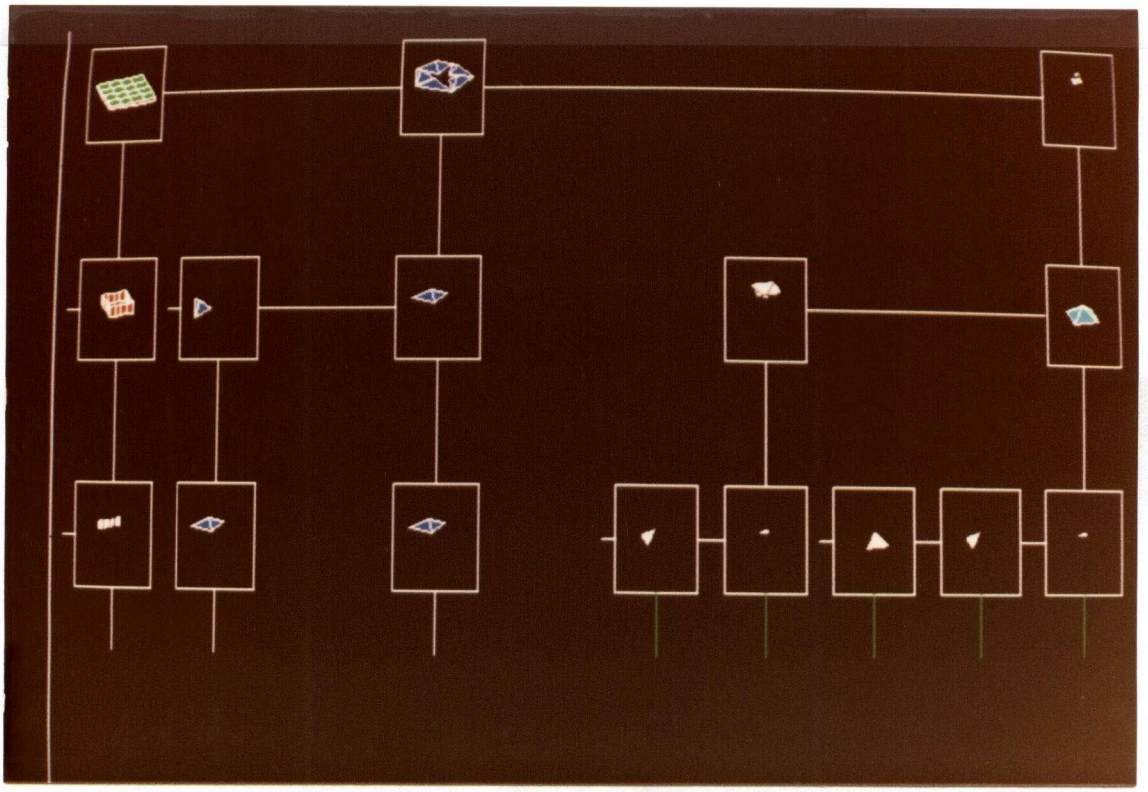


Figure 23 Original Tree Structure

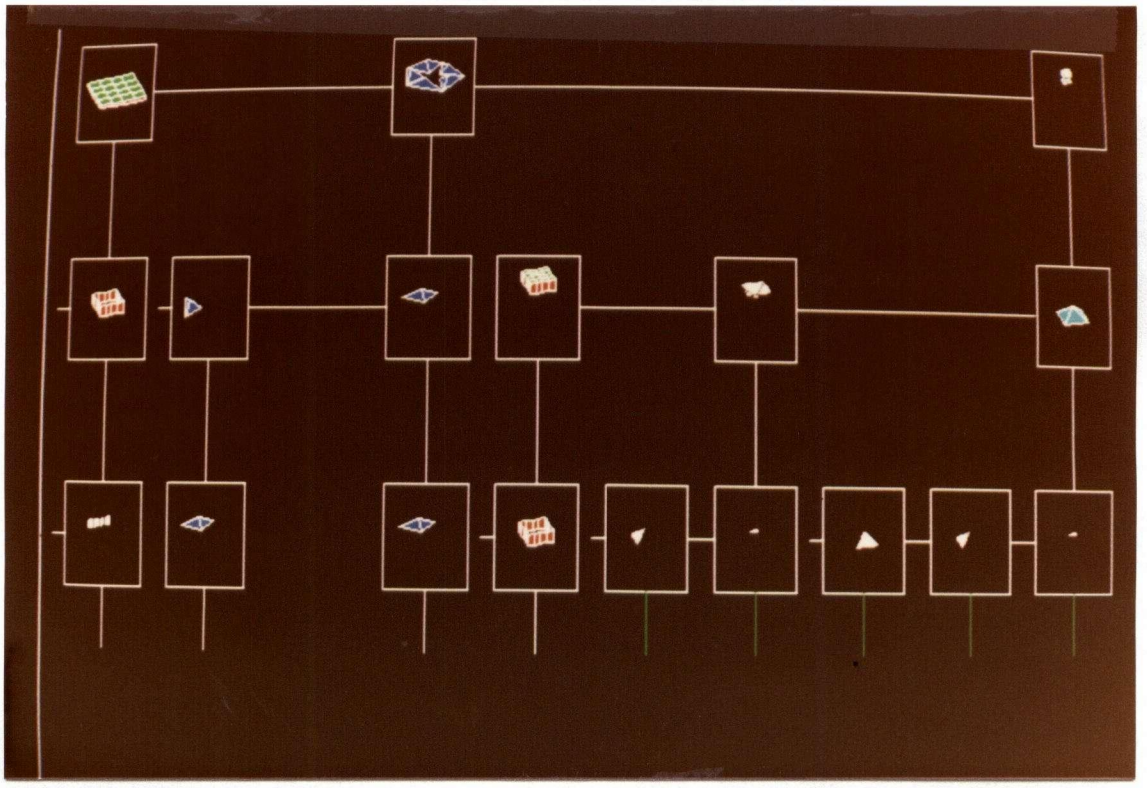


Figure 24 Graphical Editing Addition

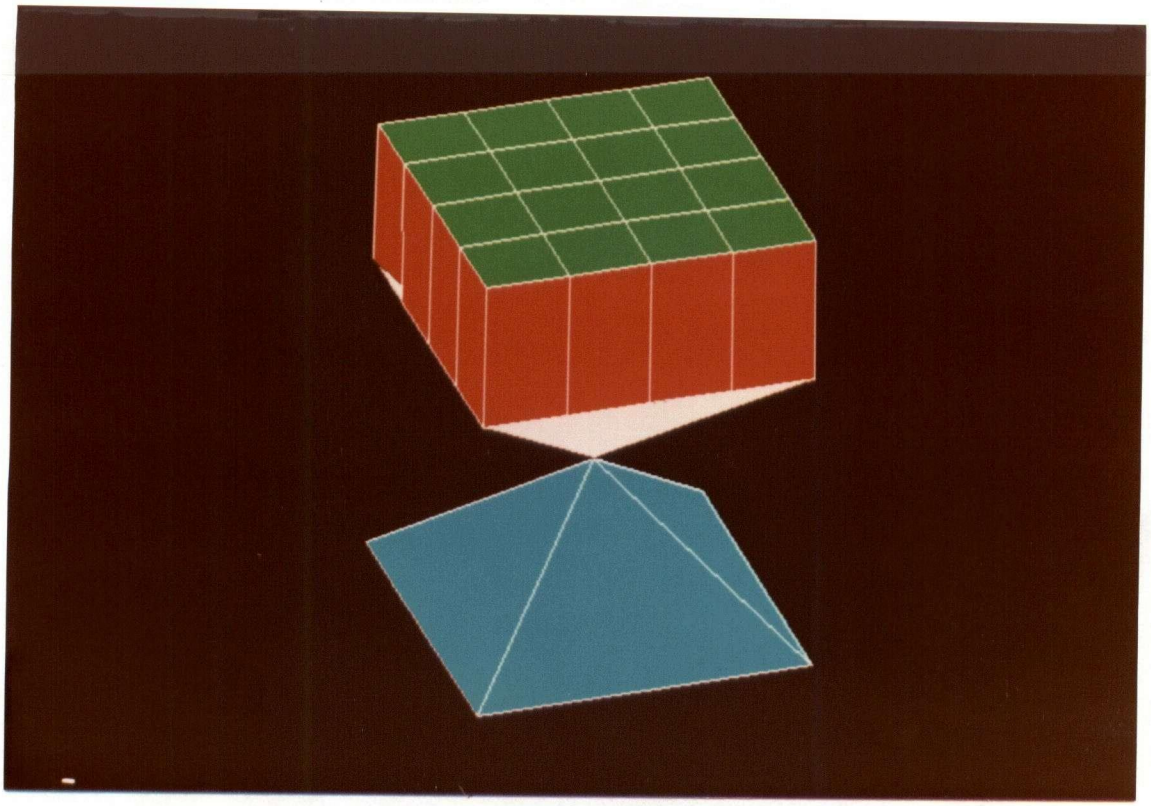


Figure 25 Viewing Editing Result

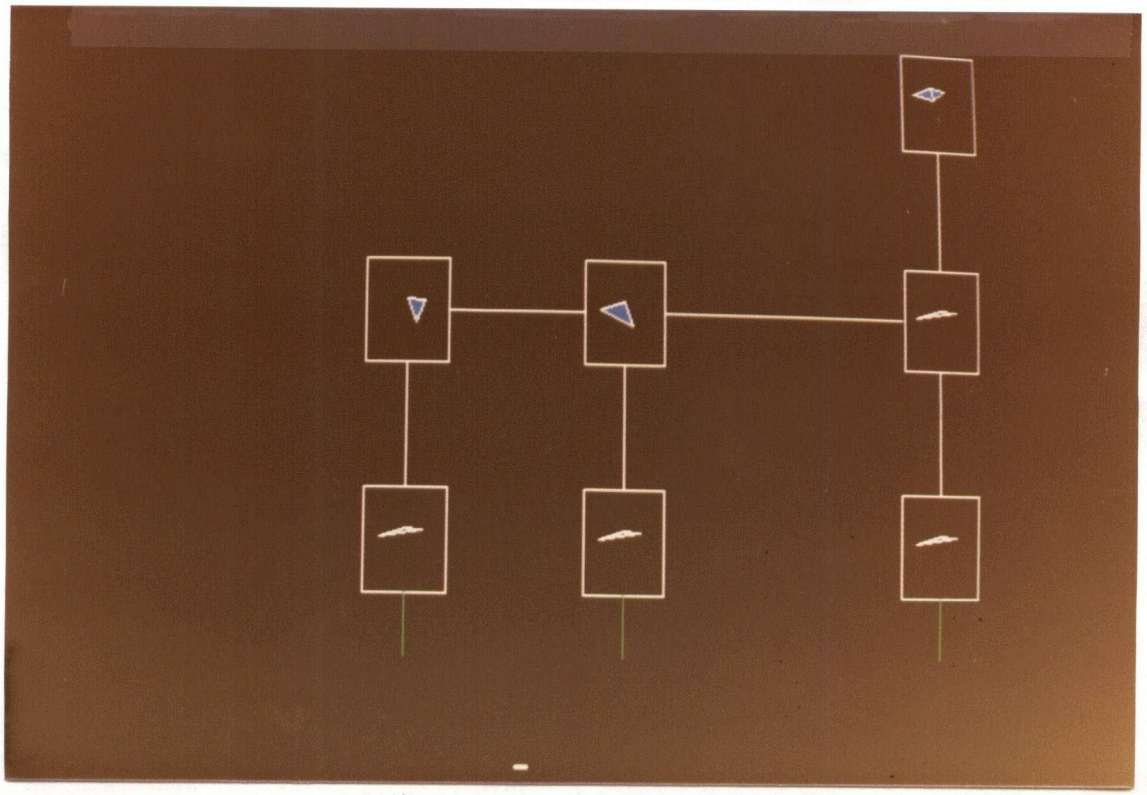


Figure 26 Editing Level Change

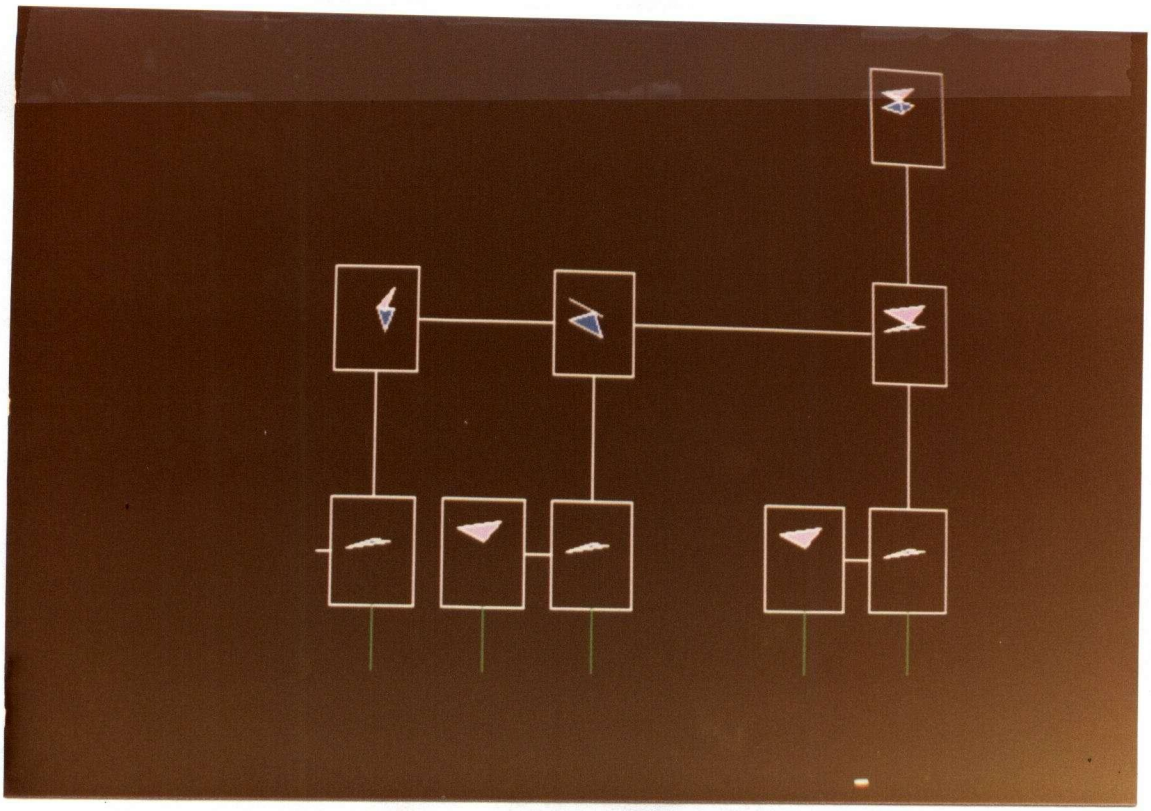


Figure 27 Subobject Modification

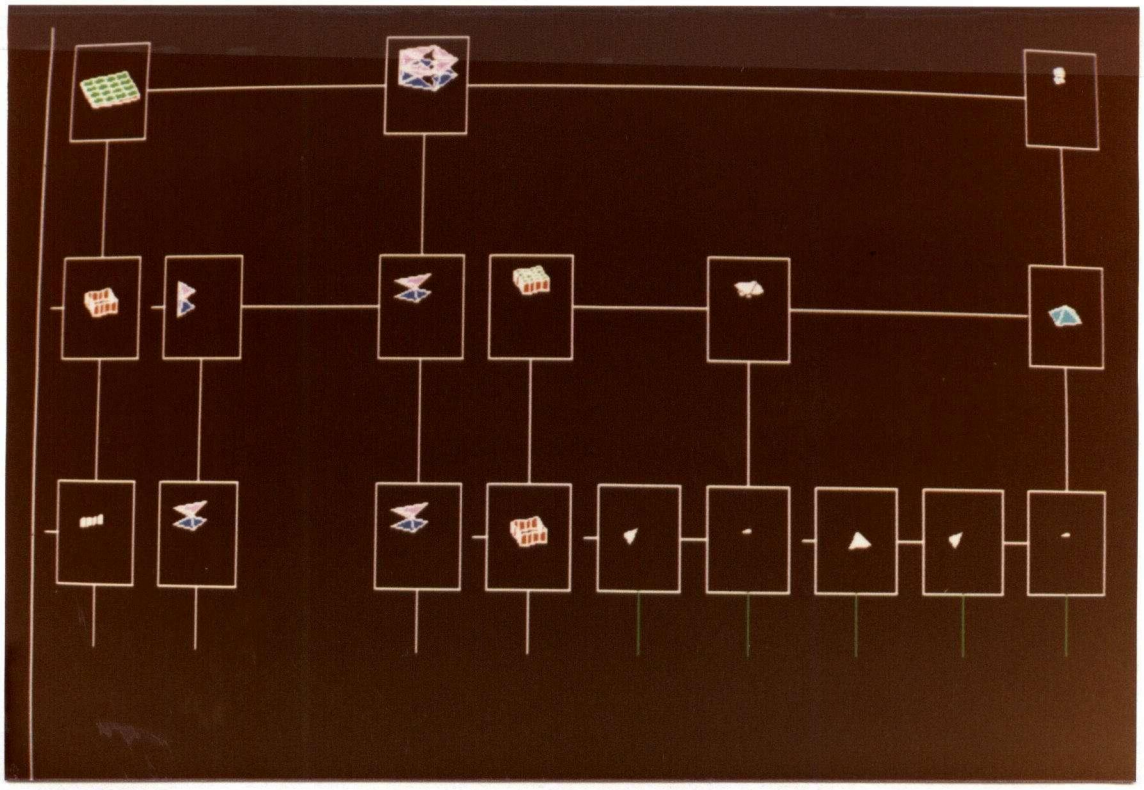


Figure 28 Original Root Redrawn

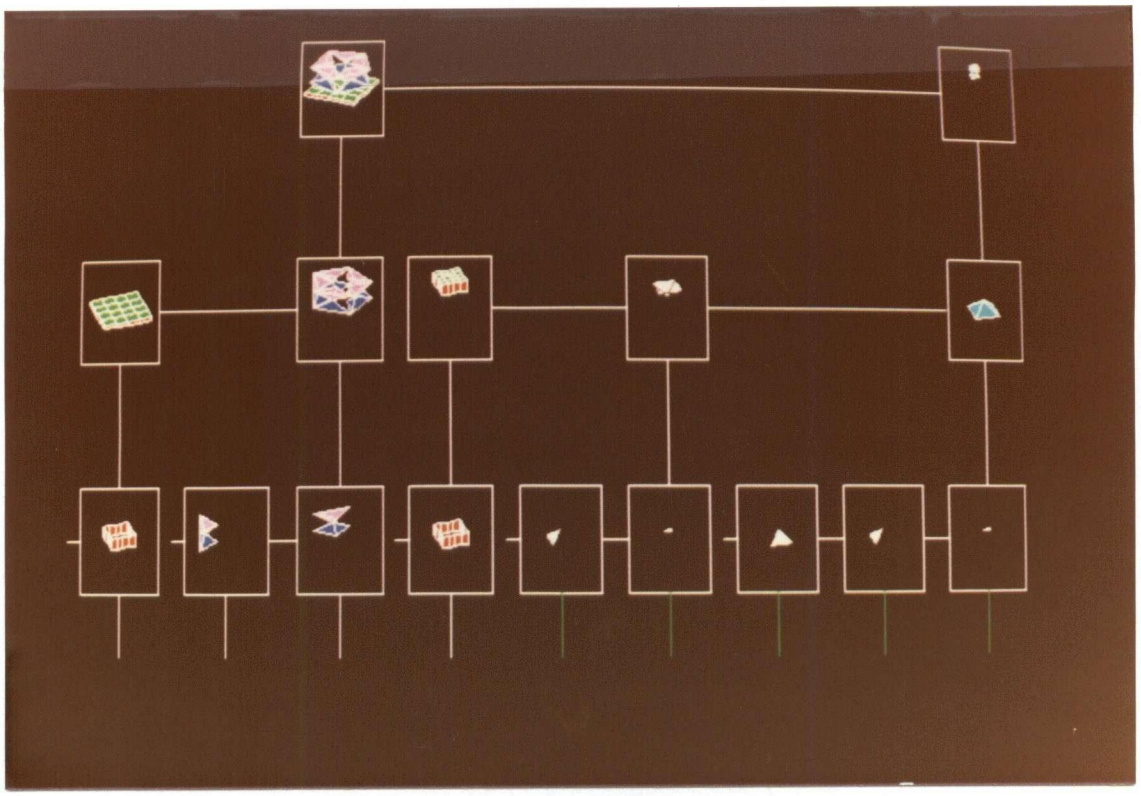


Figure 29 Structure Alteration

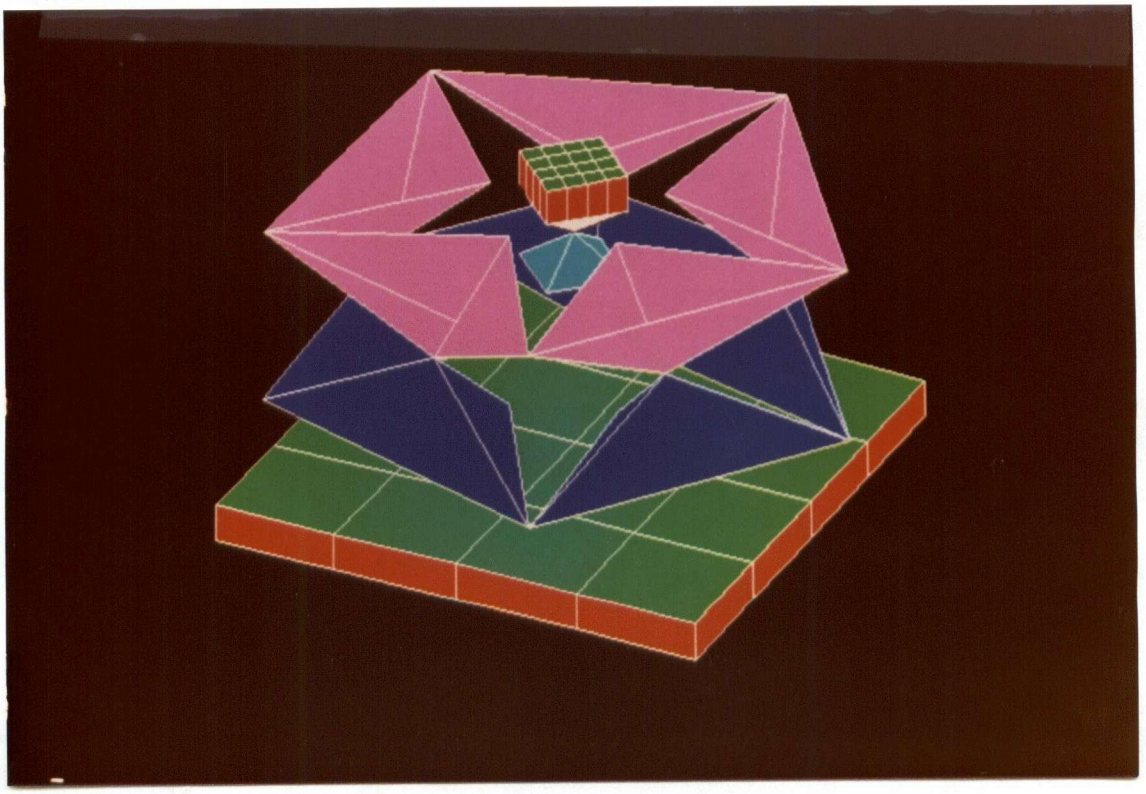


Figure 30 Graphical Editing Result

5.4 Data Structures

PASCAL allows programmers to explicitly define their own data types and structures. This promotes good programming practice because an understanding is required of the data involved in a problem. FORTRAN does not permit data type definitions. LIG6 permits explicit definition of data types belonging to the class graphical output primitive, but arbitrary data types cannot be explicitly defined. The tree structure of the graphical data base, however, allows implicit definition of data types and structures in a similar fashion as the language LISP.

Data types and structures can be implicitly defined in LISP by interpreting the structure of a list as its data type. Different data types can be specified by the length of a list of that type, which elements of the list are atoms or lists, or what data type a list element is interpreted to be. The tree structures of LIG6 can be used in the same way as the list structures of LISP to define data types implicitly. An example which uses the graphical data base of LIG6 to represent the abstract data of a grammar will now be presented.

The goal of the program is to take a grammar specified in Backus-Naur form and produce an equivalent grammar specification in the form of syntax directed diagrams. This is achieved by storing the grammar, manipulating its representation, and generating an external representation in the form of a diagram.

Grammar driven compiler writing systems [McKe70,Leca74] often do not allow iteration, alternation, or option constructs

in the grammar specification. Such constructs must be implemented by the top level alternation construct and by recursive definition, allowing the grammar to consist of only two types of symbols: terminals and non-terminals. The same restriction is enforced by the program. The input grammar, however, is analyzed and manipulated so that the resulting diagrams have iteration, low-level alternation, and option structures.

Three data types are required to allow storage of the grammar: terminals, non-terminals, and structural elements. The graphical data base structures which are interpreted as representing these types are shown in Figure 31. In addition, the two variables TRMLST and NTRLST are used to maintain a record of all of the terminals and non-terminals of the grammar, respectively. The structures of their values are also given in Figure 31.

As the grammar is read in, an internal representation using TRMLST, NTRLST, and the three data types is constructed. If the input grammar is the simple definition

```
<subprogramlist> ::= <subprogram>
                   | <subprogramlist> <subprogram>          ,
```

then the internal representation given in Figure 32 would be constructed. This structure does not represent the simplest equivalent grammar because the definition is using recursion to implement iteration. The initial internal representation of any grammar will have to be manipulated to produce a structure which utilizes iteration, alternation, and option constructs. The internal representation after manipulation of the above grammar

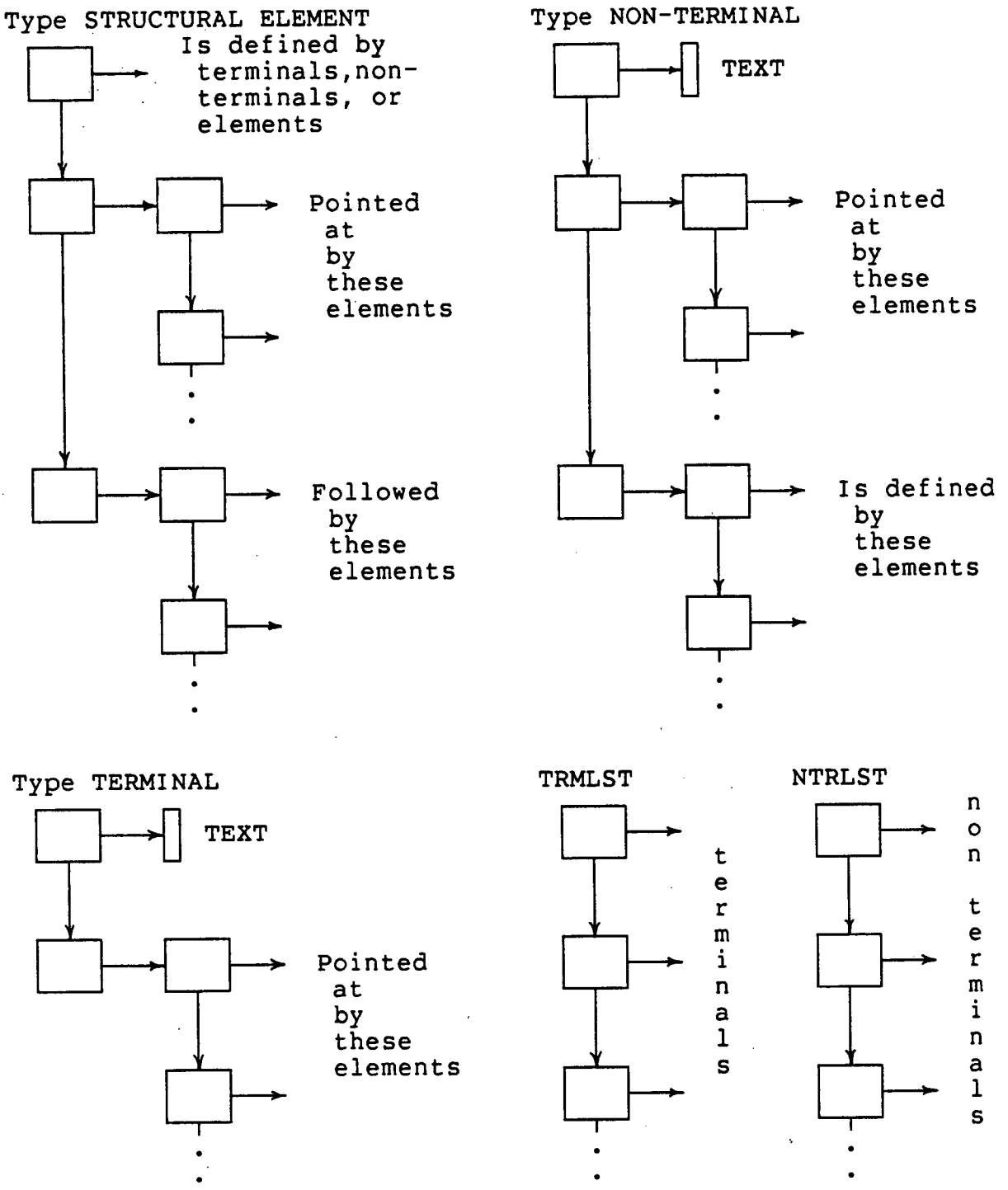


Figure 31 Grammar Storage Data Types and Variables

is given in Figure 33.

Once the desired grammar representation has been created, it is further manipulated so that it becomes a model of the graphical object, the syntax directed diagram. The literals

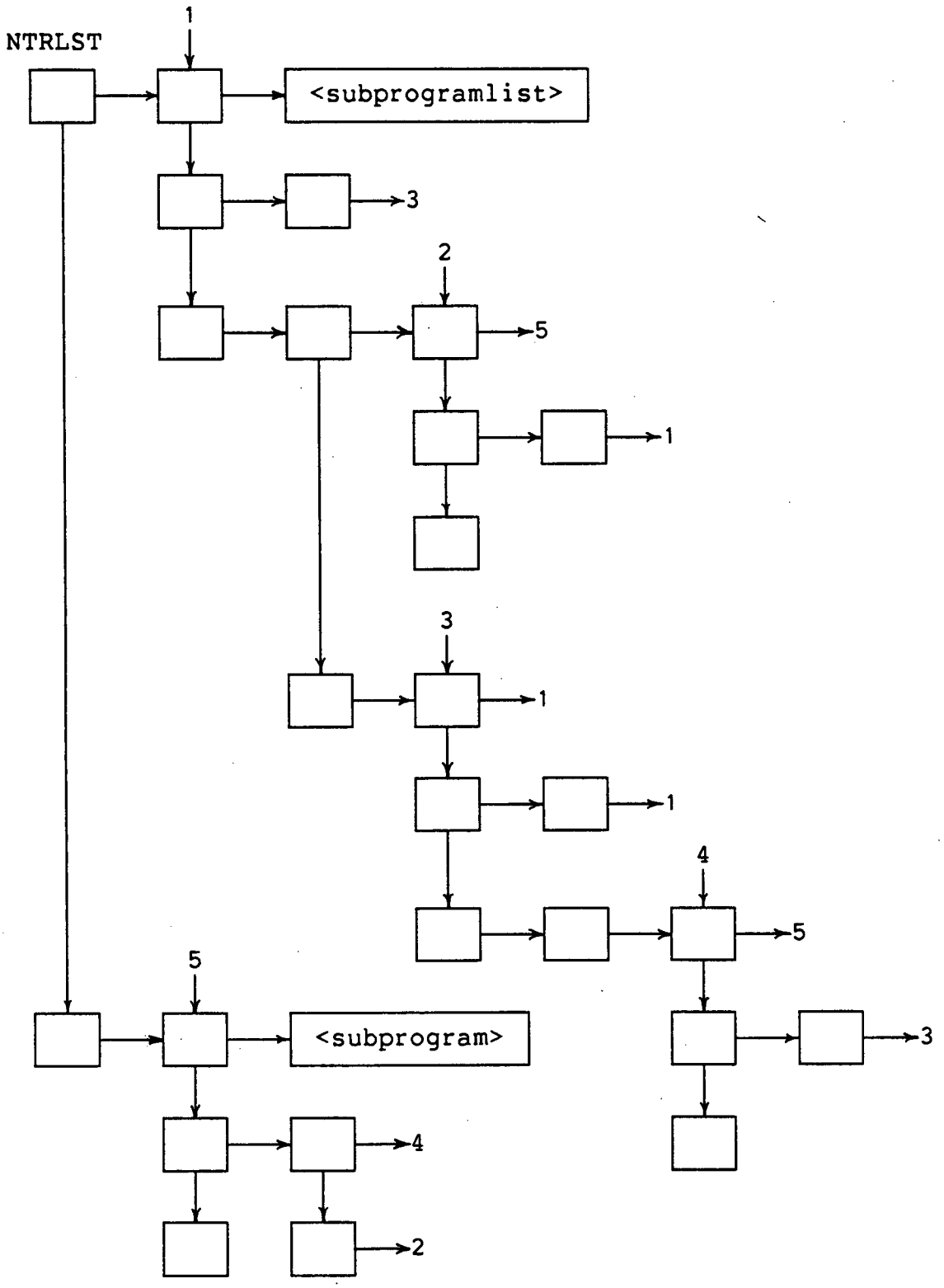


Figure 32 Original Grammar Internal Representation

representing the grammar symbols are positioned and connected with lines and arrows as specified by the definition pointers. A

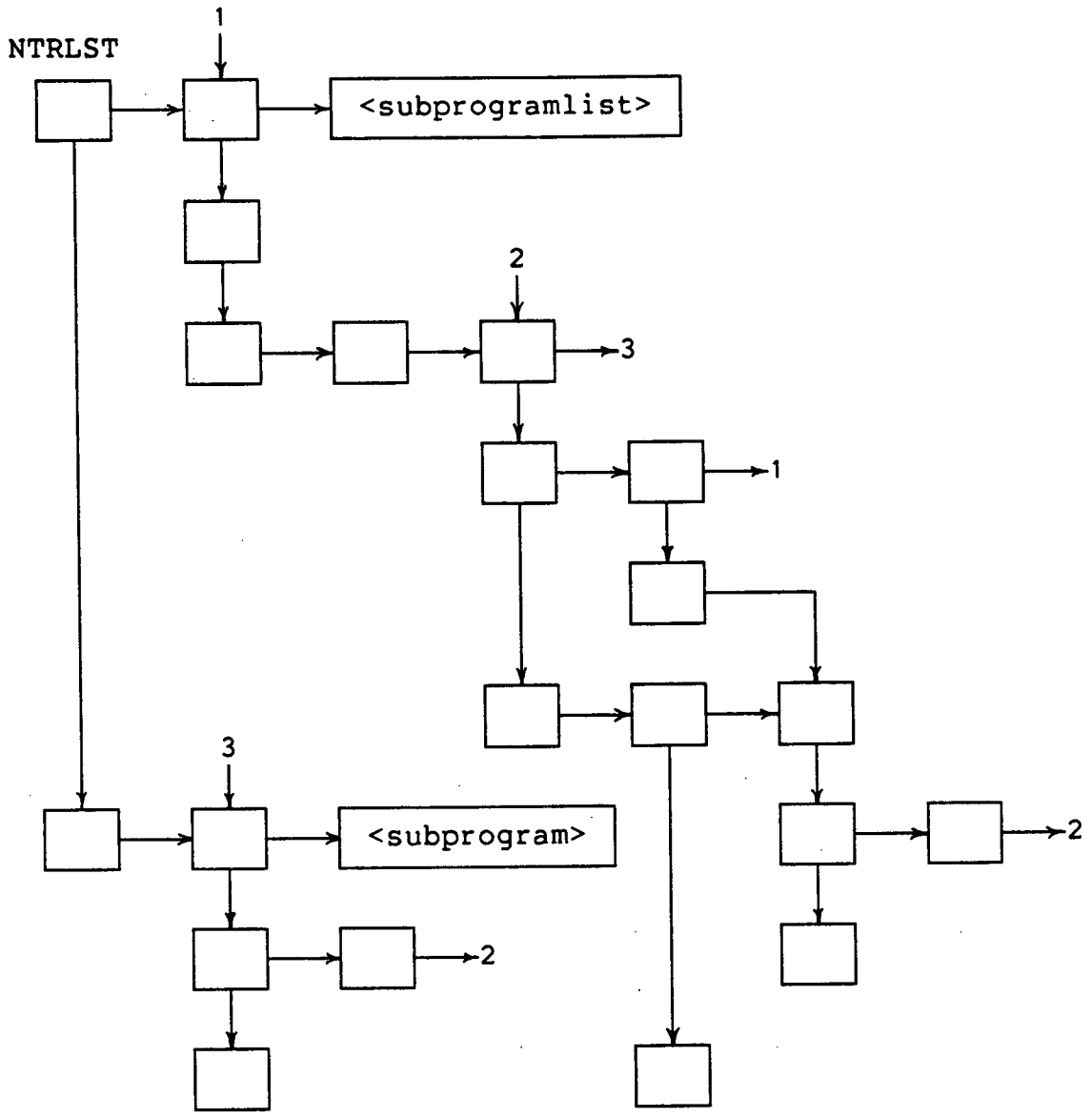


Figure 33 Manipulated Grammar Internal Representation

more complete grammar and its resulting diagram is shown in Figures 34 and 35, respectively.

The graphical data base of LIG6 lends itself to problems where dynamic data structures are involved. Another possible use would be the implementation of a minimal LISP interpreter in LIG6 which could produce visual representations of any s-expressions in the standard cons cell format of LISP.

```
<program> ::= <subprogramlist>
            | <subprogramlist> <main>
            | <subprogramlist> <main> <subprogramlist>
            | <main>
            | <main> <subprogramlist>

<subprogramlist> ::= <subprogram>
                   | <subprogramlist> <subprogram>

<subprogram> ::= <subroutine>
               | <function>
               | <primitivedef>
               | <blockdata>

<subroutine> ::= <subhead> <block>
               | <subhead> <parameterlist> <block>

<function> ::= <funchead> <parameterlist> <block>

<subhead> ::= SUBROUTINE <identifier>

<funchead> ::= FUNCTION <identifier>
             | <type> FUNCTION <identifier>

<type> ::= INTEGER
         | REAL
         | COMPLEX
         | LOGICAL
         | GRAPHICAL
         | VECTOR
```

Figure 34 Sample Input Grammar

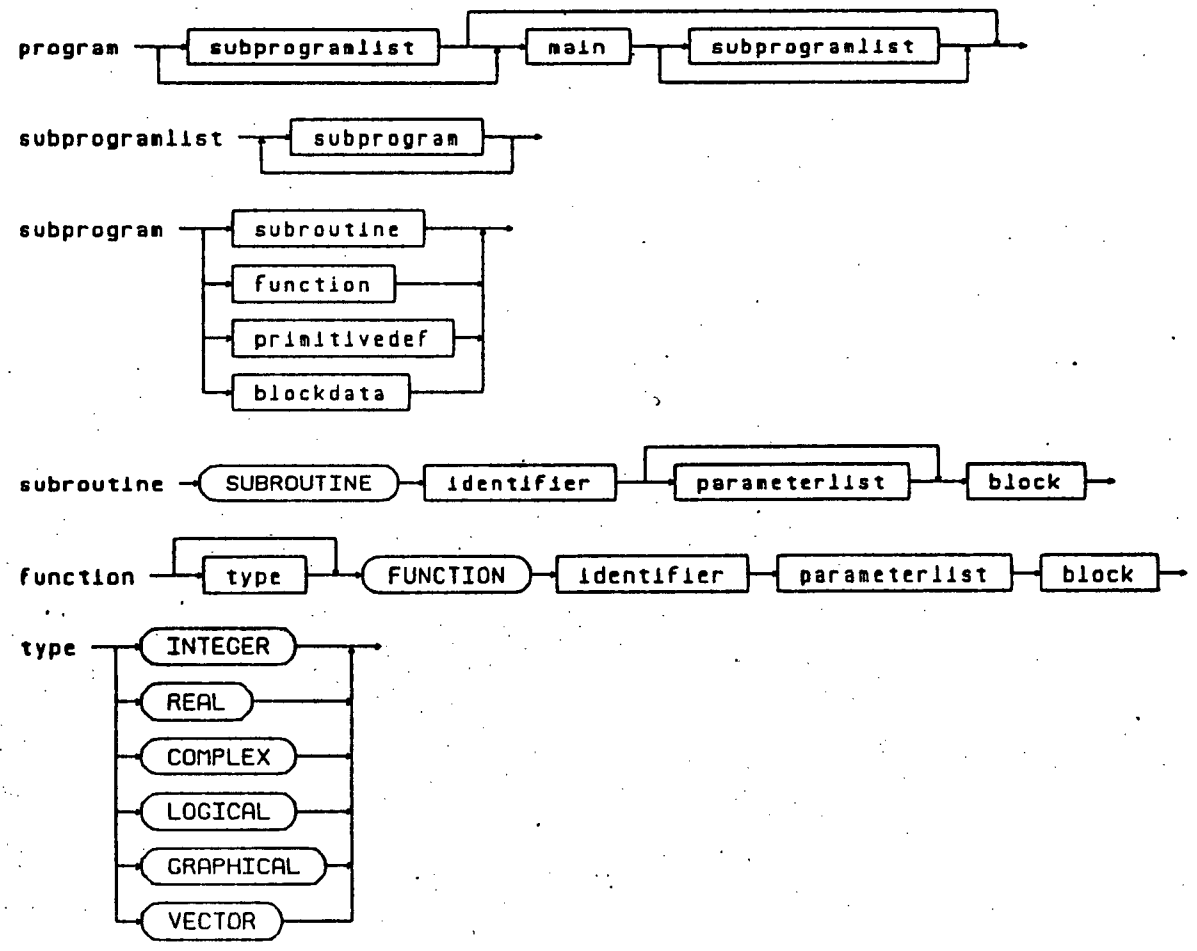


Figure 35 Sample Syntax Directed Diagram

Chapter 6

NEW LANGUAGE FEATURES

Apart from the shift of emphasis from a system where the data base is completely hidden from users to a data base model system and the addition of programmer-defined primitives, there have been a number of other new features included in the language LIG6. These features have varying degrees of originality; some were not available in earlier LIG versions but were available elsewhere to some extent, while others have not been published or implemented previously. The new features include graphical operators, interpretation decisions, and language constructs.

6.1 Vector Data Type

To provide economy of expression when dealing with three dimensional data, the data type VECTOR was included in LIG6. Simple variables, single and multi-dimensioned array variables, function subprograms, and statement functions of type VECTOR are permitted. Vector variables may be typed explicitly, or else implicitly with the aid of the IMPLICIT statement. Vector expressions may be passed as arguments to subprograms. Vector variables may be equivalenced and placed in COMMON blocks. They may not be initialized in a type declaration or in a DATA

statement nor may they be output or input as a unit with WRITE or READ statements. Vector constants are of the form (X,Y,Z) where X, Y, and Z represent arithmetic expressions and where the parentheses are mandatory. The Z expression is optional; if missing, it defaults to 0.0.

Vector variables may be assigned values which are vector expressions. The operators which are defined for the type VECTOR are summarized in Figure 36.

OPERATOR	EXAMPLE	OPERATION	RESULT TYPE
+	V1 + V2	vector addition	VECTOR
-	V1 - V2	vector subtraction	VECTOR
#	V1 # V2	vector cross product	VECTOR
.	V1 . V2	vector dot product	REAL
| |	|V|	vector magnitude	REAL
*	A * V	multiplication by scalar	VECTOR
/	V / A	division by scalar	VECTOR
.EQ.	V1 .EQ. V2	vector comparison	LOGICAL
.NE.	V1 .NE. V2	vector comparison	LOGICAL

Figure 36 Vector Operators

The program fragment presented in Figure 37 illustrates the use of the VECTOR data type. While the inclusion of operators and nested expressions allow concise implementation of vector arithmetic algorithms, the major use of this data type will be in graphical primitives.

Access to the individual components of vectors is provided by the three LIG6 system functions, COORDX, COORDY, and COORDZ. The individual components may be assigned using the above identifiers using an assignment statement construct. Component access is demonstrated in the program fragment of Figure 38.

The VECTOR data type was not supported in earlier FORTRAN

```
VECTOR FUNCTION CROSS(X,Y,N)
IMPLICIT VECTOR(V,X-Z)
VECTOR A,B(5),T(2,3),Y(N),FUNC
GRAPHICAL D
DIMENSION A(3)
COMMON V1,V2(3,4)
EQUIVALENCE (B(3),A(1))
VFUNC(XV,YV,ZV) = XV.YV#ZV
A(1) = (FUNC(3,Q) + X)/(5.+Q)
CALL TEST(3*A(2)#V2)
IF(X#Y(1).EQ.Y(2)#Z/4) CROSS = (COS(R),SIN(R))
D :- LINE FROM (1.5,3.2) TO (V2(2,I)#(Y(1)+Y(3)))
.....
```

Figure 37 Vector Data Type Usage

```
VECTOR V,V1
RVAL = COORDX(V) + 3*COORDZ(V)
COORDY(V) = 3.2 + COORDX(V1#(P,Q,R) + V)
```

Figure 38 Vector Component Access

versions of LIG. It is possible to declare a PASCAL vector type in the LIG/P implementation, but operators and natural inclusion in graphical primitives are not possible. The language MIRA supports the data type vector, but the only operators permitted are addition and dot product [Magn81].

6.2 Map Operator

In addition to the standard graphical transformations scale, rotate, and translate, LIG6 provides a map operator. This transformation operator can be used to perform any combination of scaling, rotation, translation, and shearing. Its syntax is the keyword MAP followed by a list of 1, 2, 3, or 4 points, the keyword TO, and another list of points. The two lists of points must each have the same length. Each point may be a vector

constant or a vector expression which is enclosed in parentheses.

The implementation of the map operator is as straightforward as its invocation: the operator creates a linear transformation which maps the points in the first list into those in the second. The points in the lists need not have any relation to coordinates of the model to which the transformation is applied, although this is one form of the operator's use.

Careful choice of the points in the lists will create any of the standard graphical transformations. The map operator also provides a concise method for expressing shearing transformations. Combining the map operator with the standard transformations facilitates the construction of interesting transformations. Examples of the map operator which demonstrate these points are given in Figure 39.

```
 i)  <MAP (0,0),(1,0) TO (0,0),(COS(THETA),SIN(THETA))>
 ii) <MAP (V1),(V2) TO (V1),(V1 + S*(V2-V1))>
iii) <MAP (1,0),(0,0),(0,1) TO (1,0),(0,0),(-COS(T),SIN(T))>
 iv) < MAP (V1),(V2) TO (0,0),(|V1-V2|,0) ,
       ROTX THETA ,
       MAP (0,0),(|V1-V2|,0) TO (V1),(V2) >
```

Figure 39 Map Operator Examples

In Figure 39, example (i) produces a transformation equivalent to a rotation about the z-axis by an amount THETA. Example (ii) performs scaling with respect to the point V1 in the direction V2-V1 by a quantity S. Example (iii) will cause shearing by an angle T about the z-axis. Example (iv) illustrates compounding transformation operators. Its effect is

to rotate objects by an amount THETA about the axis specified by the line passing through the points V1 and V2.

All LIG6 transformations use matrices to produce the desired effect. The coordinate triples of an object which is to be transformed are converted into homogeneous coordinates [Roge76] and then multiplied by a matrix; the resulting coordinates represent the transformed object. Compounded transformations are created by multiplying matrices. A matrix which rotates objects about the z-axis is presented in Figure 40.

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} P \\ Q \\ R \\ 1 \end{bmatrix}$$

Figure 40 Z-Rotation Matrix

The linear transformation which implements the map operator is the solution of a matrix equation. The matrix A of Figure 41 is the transformation which implements the map operator of the same figure.

The matrix A is found by solving an equation of the type

$$\begin{bmatrix} A \end{bmatrix} \begin{bmatrix} X \end{bmatrix} = \begin{bmatrix} B \end{bmatrix} \qquad (1)$$

One solution is

$$\begin{bmatrix} A \end{bmatrix} = \begin{bmatrix} B \end{bmatrix} \begin{bmatrix} X \end{bmatrix}^{-1} \qquad (2)$$

however, this involves finding the inverse of a matrix and then

```
<MAP (a,b,c),(d,e,f),(g,h,i),(j,k,l) TO
     (m,n,o),(p,q,r),(s,t,u),(v,w,x)  >
```

$$\begin{bmatrix} \\ A \\ \\ \\ \end{bmatrix} \begin{bmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & l \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} m & p & s & v \\ n & q & t & w \\ o & r & u & x \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 41 Map Operator Implementation

multiplying. Another solution is to take the transpose of both sides of equation 1 yielding equation 3. The transpose of A can then be solved for by using a LU decomposition which has been shown to involve fewer operations [Fors67].

$$\left[X \right]^T \left[A \right]^T = \left[B \right]^T \qquad (3)$$

There is a unique linear transformation which maps a given set of four points in three-dimensional space to another set. When the other variants of the map operator are used, however, there is not a unique solution. All solutions achieve the mapping objective, but they differ in their effect on points which are not in the plane specified by the three member variant, points not on the line specified by the two member variant, or points other than the one specified by the one member variant. The solution chosen for these variants is the one that minimizes the distortion of graphical objects.

For the one member variant map operator, the choice is simple: the equivalent translation transformation is used. The two and three member variant map operators are implemented by choosing additional appropriate points and solving as for the

four member variant.

The three points in a list of a three member variant define an origin and two vectors. The fourth point chosen is the cross product of the two vectors. This generates a third vector perpendicular to the plane defined by the points in the list. Its length is determined by the lengths of the two original vectors. When the same process of fourth point specification is applied to both lists, any scaling or shearing in the plane is applied naturally to points off the plane.

The two points in a list of a two member variant define an origin and a vector. A third point is arbitrarily chosen which generates a vector which is of the same magnitude as and is perpendicular to the original vector. The same procedure as for the three member variant is then followed.

6.3 Stroke Precision Text

There are three levels of text appearance precision: string, character, and stroke precision text [GSPC79]. In string precision text, only the position of the first character of a string may be specified; the size and orientation of the string is hardware dependent. In character precision text, the position of every character of the string is affected by transformations but the size and orientation of the individual characters are still hardware dependent. Stroke precision text treats strings as if each character were constructed from short lines; all transformations apply to such strings. The differences between the precisions are summarized in Figure 42.

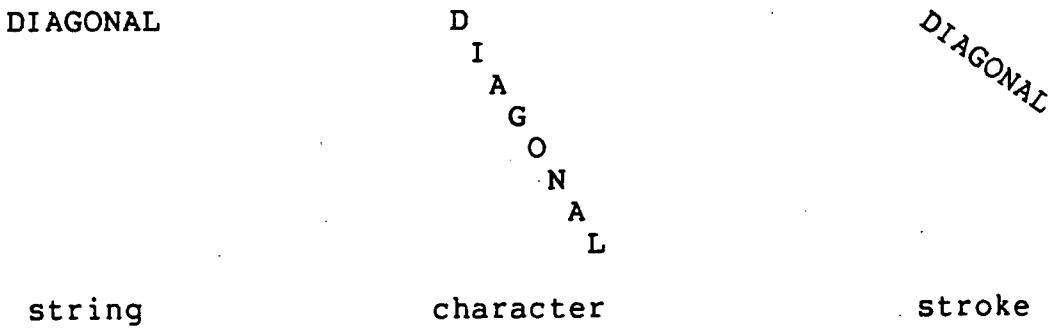


Figure 42 Text Precision

The use of string or character precision text in images of models of three-dimensional graphical objects yields poor results. The precise positioning and orientation of text strings requires stroke precision capability; LIG6 has this capability. All images of strings are generated using software because most hardware generators are capable of only string precision text. Figure 43 gives an example of some of the possibilities of stroke precision text.

6.4 Structured Statements

As software costs increase relative to hardware costs, more effort is made to ensure that programs are correct, readable, and maintainable. Programs with these qualities are most easily produced when a rigorous consistent programming style is used. A style which has developed a considerable following in recent years is called Structured Programming [Dahl72].

The constructs of a language have an effect on the style in which programmers create programs in that language. FORTRAN is one of the original high-level programming languages; it lacks the structured control constructs available in more modern languages such as PASCAL [Jens76]. Structured programming in

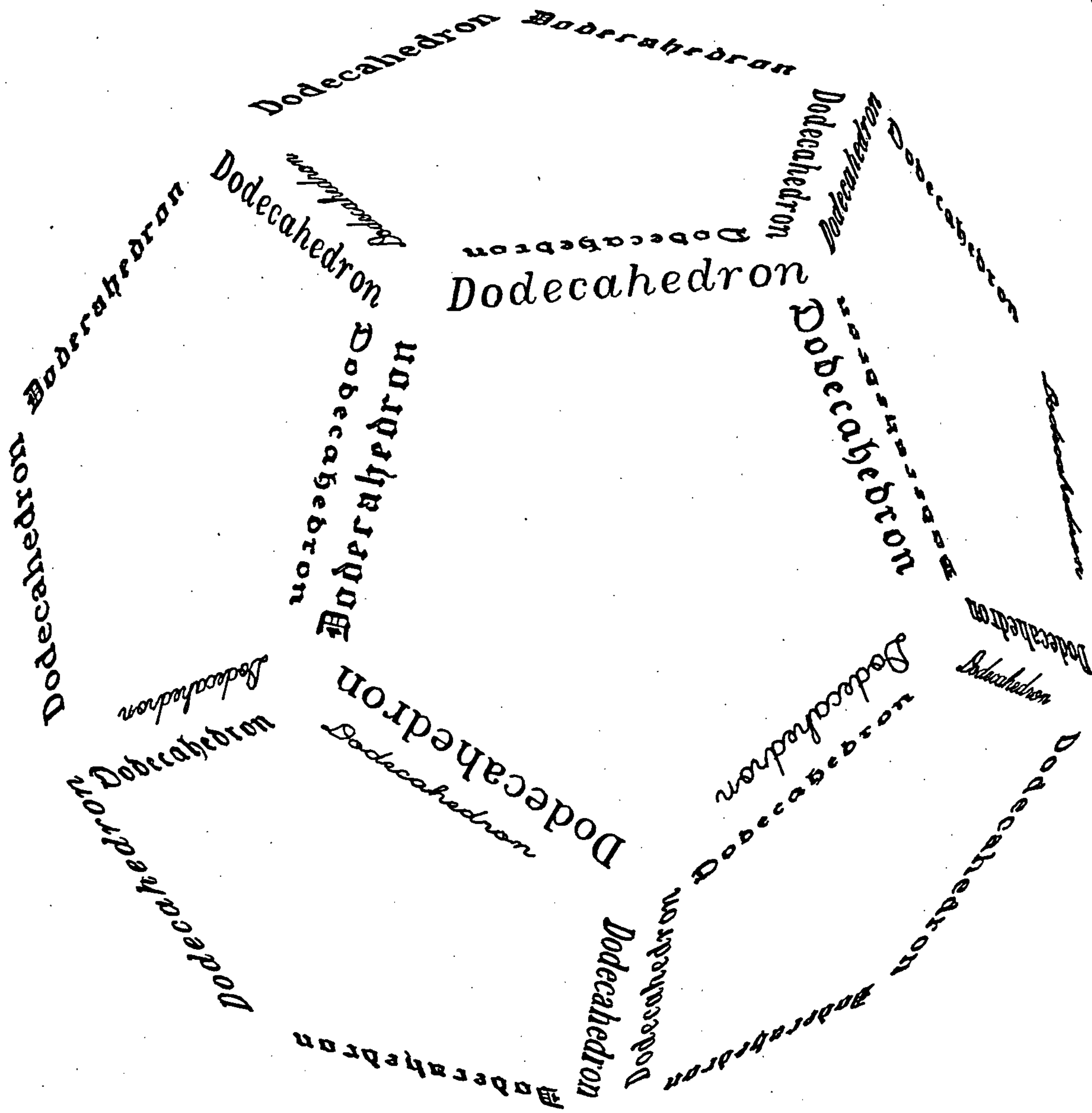


Figure 43 Stroke Precision Text

FORTRAN has been accomplished by interpreting groups of statements as control structures and restricting the usage of statement labels and the GOTO statement. In an effort to promote a structured, modular programming style, more modern implementations of FORTRAN such as FORTRAN'77 [Meis77] or WATFIV/S [Frie82] have included structured control constructs.

LIG6 extends the control constructs of FORTRAN with the addition of four structured constructs. These constructs are an

IF-THEN-ELSE structure, a REPEAT structure, a WHILE structure, and a CASE structure. The syntactic rules of these constructs are formally defined by the grammar presented in Figure 44. Use of these constructs is demonstrated in the program fragments in Figure 45.

```
<statementblock> ::= BEGIN <statementlist> END
<statementlist> ::= <statement> <statseparator>
                  | <statementlist> <statement> <statseparator>

<statseparator> ::= <eol>
                  | ;

<structuredif> ::= <truepart>
                 | <truepart> <falsepart>

<truepart> ::= IF ( <logicalexpr> ) THEN <statementblock>
<falsepart> ::= ELSE <statementblock>

<repeat> ::= REPEAT <statementlist> UNTIL ( <logicalexpr> )

<while> ::= WHILE ( <logicalexpr> ) DO <statementblock>

<case> ::= CASE <expression> : <type> OF <caseexpr>

<caseexpr> ::= <truelist>
             | <truelist> <falselist>
             | <falselist>

<truelist> ::= <success>
             | <truelist> <success>

<falselist> ::= '<>' : <statementblock>

<success> ::= <exprlist> : <statementblock>

<exprlist> ::= <expression>
             | <exprlist> <expression>
```

Figure 44 Structured Statements Grammar

All structured constructs may be nested to any depth. The requirement of statement lists being bracketed by BEGIN and END ensures that all syntactic structures are completely

```
IF(X.LT.Y) THEN BEGIN T=X; X=Y; Y=T; END
IF(3**J.GT.2**K) THEN
  BEGIN
    J=0; K=1
  END
ELSE
  BEGIN
    K=0; J=1
  END
REPEAT
  READ(5,10) I
  CALL DUM(I*3)
UNTIL(I.GT.32)
WHILE(J.LT.10) DO
  BEGIN
    .....
  END
CASE R*T/(5.+Q) : INTEGER OF
  2,3 : BEGIN
          .....
        END
  <>  : BEGIN
          CASE 5*Q : REAL OF
            R/Q : BEGIN
                    .....
                  END
            END { REAL CASE }
        END
  END { INTEGER CASE }
```

Figure 45 Structured Statements Examples

unambiguous.

6.5 Archivation

Graphical application programs are often executed interactively in order to construct models of graphical objects. This process is usually time-consuming and it is difficult for a user to repeat model specifications exactly. To create menus or to continue modelling begun in previous executions of a program, it is necessary that arbitrary models can be stored and retrieved. This is implemented in LIG6 by archivation.

Archivation is the saving on and restoring from secondary

storage models of graphical objects. It can be used to pass models between different programs or to save generated models for a subsequent run of the same program. There are three statements which are involved with archivation: the STORE statement, the POSITION statement, and the LOAD statement.

There are two forms of the store statement. Some examples of this type of statement are

```
STORE ON UNIT 7,BIRD
STORE ON UNIT N+2,FOWL
STORE ON UNIT 3, VALUE(LAST(A))
STORE ON UNIT 3, IDENTIFICATION 7.5, A
STORE ON UNIT 3, IDENTIFICATION 3*R, A
```

The integer valued expression following the keyword UNIT is the logical I/O unit on which the model is stored. It must be assigned to a disk file on the run command or with a FORTRAN I/O subprogram. The real valued expression following the keyword IDENTIFICATION is a number which is placed in the header of the stored model in the file so that it can be identified later. The default value for the identification is 0.0. The last item in the list is a graphical variable or function invocation whose value is a node. The effect of the statement is to store at the end of the file a header and codes which represent the model which is the value of the last item.

The archivation file is in the form of a sequential tape. Each additional object is placed at the end of the tape. When they are later loaded, the loading will occur in the same order as the order in which they were stored. A certain degree of random access can be obtained, however, with the position statement.

The position statement is used to position an archivation

file at the desired model storage. Examples of possible forms of this statement follow.

```
POSITION UNIT 2
POSITION UNIT N+3,5.5
POSITION UNIT 3,4.*Q
POSITION UNIT 3,7.2,2
POSITION UNIT 3,,2
```

The first arithmetic expression in the list following the keyword is the unit expression; it must be integer valued and has the same meaning as in the store statement. The second expression is the identification expression; it must be real valued. The third expression is the version expression; it must be integer valued. The default value for the identification expression is 0.0. The default value for the version expression is 0.

The effect of the statement is to position the archivation file attached to the unit at the specified version of the identification value. The fourth statement in the examples above will position the file at the model storage of the second instance of a model being stored with identification 7.2. If the version number is 0, the last model stored with the specified identification is the point at which the file is positioned. The default identification value of 0.0 will match all identification values, thus the last statement will position the file to the second model stored, while the first statement will position the file to the last model stored.

The load statement is used to restore a model. The model stored in the archivation file at its present position is loaded into the graphical variable specified. An example is

```
LOAD FROM UNIT N+2, BIRD
```

Chapter 7

CONCLUSIONS

Earlier versions of LIG preprocessors did not analyze the host language statements of a program. LIG6, the research topic of this thesis, analyzes both the FORTRAN and language extension statements. The primary reason for this is that the language constructs of LIG6 cannot be easily split up into those which are strictly FORTRAN and those which are strictly extensions. There is considerable overlap between the constructs, as is evident by the number of FORTRAN statements which may contain extension elements.

Several benefits are realizable when all statements are analyzed. There is no need to demarcate extension statements by the use of a special character in a designated column or by other means. It is possible to mix host and extension statements on a single line. Any syntactic errors in the host language statements which would not normally be detected until the preprocessor output is compiled are trapped. The preprocessor is moved one step closer to being a compiler; object code generation is only possible when complete parsing is performed.

Another difference between LIG6 and previous LIG versions is the implementation of the preprocessor. The LIG6 preprocessor does not make use of a Compiler Writing System (CWS); it is written completely in PASCAL. There are several factors

supporting this choice. Currently available compiler writing systems [McKe70,Leca74] do not support the free form input conventions of FORTRAN; delimiters and reserved words are required. With a CWS, it is not possible to carry out language extensions which are defined by programmers. Such extensions require a dynamically alterable parser; CWS generated parsers are determined solely by the original grammar specifications. LIG6 programmer-defined primitives include the definition of syntactic constructs which affect the parser. The LIG6 preprocessor itself can thus be thought of as a load and go CWS. Because a CWS must be able to handle general grammars, it cannot generate preprocessors which exploit characterstics of a specific grammar. A preprocessor created without the use of grammar driven aids is not limited in this way; it can utilize ad hoc techniques which provide more efficient execution.

Current research in computer graphics appears to be concentrated in two distinct areas. A great deal of effort has been expended regarding standards for graphical subroutine systems [GSPC79,Fole76]. Such systems are used to create portable graphics application programs but they do not provide modelling features. The other area of intense interest is the creation of realistic images. This area contains hidden line and surface removal, shadowing, shading, and texturing algorithms. The research in this area is concerned with the process of analyzing scenes, not with the models on which the analysis is based.

High-level graphical programming languages are an extension to graphical subroutine systems. They provide a data base and

the capability to naturally model graphical objects. It is only with the addition of inquiry, however, that such a language can form a bridge between the two areas of current research. A high-level language can facilitate the modelling of graphical objects, but constructs must be available which allow the analysis of those models if complex graphical algorithms are to be implemented.

The data base approach of LIG6 permits the analysis of arbitrary models of graphical objects. Use of LIG6 to perform experiments with graphical algorithms allows the scenes required to be modelled naturally. Any algorithms so devised and implemented are then available for use in all LIG6 application programs.

A language can be considered to have been expanded when a facility is provided which was previously unavailable. An example of such a facility for FORTRAN is the system function MAX which returns the maximum value of its variable number of parameters. These facilities are usually implemented by a systems programmer because the language constructs do not permit their direct implementation. This is exemplified by MAX: FORTRAN has no method of specifying a variable number of formal parameters. It is possible for non-systems programmers to expand a language, however, if the language's constructs form a kernel which is complete in its ability to manipulate the language's data structures.

The language constructs of LIG6 provide such a kernel. All possible data base values can be created using the various assignment statements and all modelling results can be accessed

and modified. The surface generation construction tools and the editing utility described in Chapter 5 form language extensions which use only LIG6 modules.

The ability of non-systems programmers to expand LIG6 implies that the system will be able to evolve much faster. Libraries of routines which are useful in general applications can be augmented by any person fluent in the language. Heavily used facilities can still be coded by a systems programmer to improve efficiency; such a task will be simplified by the existing algorithm implementation in the high-level constructs.

The abilities of LIG6 compare favorably with those of other high-level graphics languages. The modelling and display functions of LIG6 have benefitted from the experience gained from the use of previous versions of the LIG family. They provide a natural and human-oriented method of displaying pictorial information. The advanced level constructs of LIG6 permit the analysis of pictorial information; this ability is not available with any other graphics language using graphical constructs.

An evaluation of LIG6 can be derived from the experiences of a summer student. In the summer of 1982, a second-year electrical engineering undergraduate was hired to create a graphics interface to a circuit analysis program. This student's previous programming experience consisted of a first-year computer science course dealing with FORTRAN and ASSEMBLER and job-related experience using a microcomputer and BASIC. He had no previous graphics experience. After a short period in which the student became familiar with graphical terms and concepts,

he was easily able to produce graphical output using LIG6.

The task for which the student was hired, however, entails more than the generation of images. It involves the creation of a maintainable, expandable program using the graphical data base to generate both an image of an arbitrary circuit and a specification of that circuit which can be understood by the analysis program. Although encouraging results were obtained, this work proceeded more slowly.

Two conclusions can be drawn from this experience: only minimal programming experience is needed to use LIG6 to produce graphical output, but more programming skills are required to use the advanced level constructs of LIG6 to implement sophisticated algorithms. Experience with the design and manipulation of dynamic data structures, pointers, and linked lists is especially useful. Such experience is common in programmers familiar with the languages PASCAL and LISP.

As is the case with most projects, further work would be beneficial. The implementation of LIG6 has proceeded to the point where it is a useful graphical system. The preprocessor and run-time library are complete and have been tested. Further work to improve the system involves increasing the number and capabilities of the device drivers. The language LIG6 assumes that all devices have the same, high quality capabilities. This is far from the case. It is the task of the device drivers to approximate or simulate those features expected by LIG6 but which are lacking in the device addressed. At the present time, the drivers support only a subset of the features expected by the system.

The only colour terminal presently connected to the system is a Tektronix 4027. This terminal is capable of displaying eight of a possible sixty-four colours at any one time. The eight colours currently chosen are white, blue, magenta, red, yellow, green, cyan, and black. An approach yielding more realistic results would be for the driver to choose the eight colours which best matched the intended image.

There are two distinct problems involved in using a varying palette of colours. The first is to devise a metric so that a numerical value can be associated with the difference between two colours. This requires research into the way the eye perceives colour. A possible metric is to use the red, blue, green (RBG) colour model and assume the colour axes are orthogonal and of equal length. Once the palette of colours has been determined, the metric is used to choose which colour of the palette best approximates the colour an application program wishes to display.

The second problem is the choice of palette members after a screen refresh. At that time, all of the colours which will appear on the screen are known. The optimum palette choice is the one where the sum of the errors is minimized. The error involved is the distance between the colour desired and the closest colour of the palette as defined by the metric. As there are sixty-four choose eight (roughly $4.4*10^9$) possible palettes, an exhaustive search is not practical. A possible heuristic is to consider each palette colour as a data concentrator, each desired colour as a terminal, the metric value between colours

as the terminal-concentrator link cost and to use the Add or Drop algorithms from communications theory [Schw77] to determine concentrator location / palette colour.

There are many monochromatic devices. Two such devices interfaced to LIG6 are Tektronix 4010 series terminals and a plotter. The current implementation displays all colours in the foreground colour of these devices. A possible improvement would be the approximation of colour on these devices by the use of hatching. Several options could be implemented. The hatching could be performed in image space with the result that all images of surfaces with the same colour would have the same orientation and spacing of hatch lines. Secondly, the orientation of the surfaces of the model could be used to determine the orientation of the hatching of the image. The hatching could also be performed in object space resulting in both the orientation and spacing of the hatch lines being affected. The last two implementations would provide the viewer with additional information regarding depth and the modelling process.

The portrayal of three-dimensional information is usually more meaningful if hidden lines and surfaces are removed. As LIG6 is concerned with the modelling and display of three-dimensional objects, the addition of a hidden line and surface removal capability would be an important improvement. It would be possible to place these algorithms in the driver part of the system because such algorithms are usually dependent on the output device. For example, a priority buffer algorithm [Fole82] can be used with raster terminals but not with direct view

storage tubes or plotters.

Most terminal locators provide only two-dimensional information. Interacting with an image of a three-dimensional object is difficult because there is no method to indicate depth. It is possible to implement a three-dimensional locator for a vector refresh terminal using three valuator inputs. Three cross-hairs would be drawn parallel to the axes of the modelling coordinate system through a point determined by the valuator inputs. Experiments could be carried out to determine if finite length cross-hairs or cross-hairs which include measuring tick marks are required to assist a user's depth perception.

BIBLIOGRAPHY

[Bart81] Barth, W., J. Dirnberger, and W. Purgathofer, The high-level graphics programming language PASCAL/GRAPH, Proc. Eurographics 81, Amsterdam, North-Holland, 1981, 151-164.

[Dahl72] Dahl O-J., Dijkstra, E. W., Hoare, C., Structured Programming, Academic Press, New York, 1972.

[Dijk76] Dijkstra, E. W., A Discipline of Programming, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

[Fole76] Foley, J. D., Picture Naming and Modification: an Overview, Computer Graphics 10, Spring 1976, pp. 49-53.

[Fole82] Foley, J. D., Van Dam, A., Fundamentals of Interactive Computer Graphics, Addison-Wesley, Reading, Massachusetts, 1982.

[Fors67] Forsythe, G. E., Moler, C. B., Computer Solutions of Linear Algebraic Systems, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1967.

[Frie82] Friedman, F. L., Koffman, E. B., Problem Solving and Structured Programming in WATFIV, Addison-Wesley, Reading, Massachusetts, 1982.

[GSPC79] Heilman, R., Herzog, B. (Eds.), Status Report of the Graphics Standards Planning Committee, Computer Graphics 13, No. 3, August 1979.

[Jens76] Jensen, K., Wirth, N., PASCAL: User Manual and Report, Springer-Verlag, New York, 1976.

[Kuls68] Kulsrud, H.E., A general purpose graphic language, Comm. ACM, 11 (1968), 247-254.

[Leca74] Lecarme, O., and G.V. Bochmann, A (truly) usable portable compiler writing system, Information Processing 74, Amsterdam, North-Holland, 2(1974), 218-221.

[Magn81] Magnenat-Thalmann, N., and D. Thalmann, A graphical Pascal extension based on graphical types, Software -- Practice and Experience, 11 (1981), 53-62.

[Mair81] Mair, S. G. (Ed.), UBC IG: The Integrated Graphics System, Computing Centre, The University of British Columbia, 1982.

[Mann80] Mannhardt, Ch., and G.F. Schrack, Modelling and display concepts in a high-level graphics programming language, C.E. Vandoni, Ed., Eurographics 80, Amsterdam, North-Holland 1980, 225-236.

[McKe70] McKeeman, W.M., J.J. Horning, and D.B. Wortman, A Compiler Generator, Englewood Cliffs, Prentice-Hall, 1970.

[McLe78] McLean, M.J., A survey of interactive graphics software, Austral. Comput. J., 10 (1978), 11-22.

[Meis77] Meissner, L. P., FORTRAN 77, SIGPLAN Not. (USA), vol. 12, no. 1 (Jan. 1977), pp. 93-94.

[Newm71] Newman, W.M., Display procedures, Comm. ACM, 14 (1971), 651-660.

[Roge76] Rogers, D. F., Adams, J. A., Mathematical Elements for Computer Graphics, McGraw-Hill, New York, 1976.

[Ross82] Ross, R., LIG6: Language for Interactive Graphics, User's Manual, Department of Electrical Engineering, The University of British Columbia, 1982, 55 pp.

[Schr76] Schrack, G.F., Design, implementation and experiences with a high-level graphics language for interactive computer-aided design purposes, Computer Graphics, 10 (Spring 1976) and SIGPLAN Notices, 11(June 1976), 10-17 (joint issue).

[Schr78] Schrack, G.F., LIG User's Manual, Departments of Electrical Engineering and Computer Science, The University of British Columbia, 1978, 50 pp.

[Schw77] Schwartz, M., Computer-Communication Network Design and Analysis, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[Smit71] Smith, D.N., GPL/I -- A PL/I extension for computer graphics, AFIPS Conf. Proc. 38 (1971: SJCC), 511-528.

[Thal79] Thalmann, D., Magnenat-Thalmann, N., Design and Implementation of Abstract Graphical Data Types, Proc. COMPSAC'79, Chicago, IEEE Press, 1979, pp. 519-524.

[Wirt77] Wirth, N., What can we do about the unnecessary diversity of notation for syntactic definitions?, Comm. ACM, 20 (1977), 822-823.

APPENDIX A

A Sample Program

The LIG6 program shown on the following pages demonstrates the ability of the language to implement interesting graphical algorithms. The graphical function HIDDEN is an example of a priority buffer hidden surface removal algorithm implementation. It returns a model which is equivalent to its argument but whose structure has been modified so that when it is displayed on a raster refresh terminal, the hidden lines and surfaces will not appear.

```
GRAPHICAL FUNCTION HIDDEN(TREE)
GRAPHICAL TREE,ROOT,STACK(25),TRAVRS,VALUE,SUPER
INTEGER POINT
VECTOR VUPNT,AIMPNT,VIEWUP
REAL VUANGL
COMMON /HID$/ VUPNT
CALL CAMPRM(VUPNT,AIMPNT,VIEWUP,VUANGL)
ROOT :- BLANK <COLOUR -1E35>
POINT = 1
STACK(1) :- TREE
REPEAT
  IF(VALUE(STACK(POINT)).EQ.BLANK) THEN
    BEGIN POINT = POINT - 1; END
  ELSE
    BEGIN
      IF(PRIMITIVE(VALUE(STACK(POINT)))) THEN
        BEGIN
          CALL INSERT(ROOT,STACK(POINT))
          POINT = POINT - 1
        END
      ELSE
        BEGIN
          IF(POINT.EQ.25) THEN
            BEGIN
              STACK(POINT) :- SUPER(VALUE(STACK(POINT)))
                              <MODIFICATION(STACK(POINT))>
            END
          ELSE
            BEGIN
              STACK(POINT+1) :- VALUE(VALUE(STACK(POINT)))
                        <MODIFICATION(VALUE(STACK(POINT))),
                         MODIFICATION(STACK(POINT))>
              STACK(POINT) :- SUPER(VALUE(STACK(POINT)))
                              <MODIFICATION(STACK(POINT))>
              POINT = POINT + 1
            END
        END
    END
UNTIL (POINT.EQ.0)
HIDDEN = TRAVRS(ROOT)
RETURN
END

SUBROUTINE INSERT(ROOT,NODE)
GRAPHICAL ROOT,NODE,POINTR,FATHER,COPY,VALUE,SUPER
REAL AVERGE,DEPTH
POINTR :- ROOT
FATHER :- BLANK
DEPTH = AVERGE(NODE)
WHILE(DEPTH .LT. COLOUR(VALUE(POINTR))) DO
  BEGIN
    FATHER :- VALUE(POINTR)
    POINTR :- SUPER(VALUE(POINTR))
  END
IF(VALUE(FATHER) .EQ. BLANK) THEN
```

```
      BEGIN
        ROOT :< COPY(ROOT)
        ROOT :> COPY(NODE)<COLOUR DEPTH>
      END
    ELSE
      BEGIN
        VALUE(FATHER) :< (COPY(NODE) <COLOUR DEPTH>)
        SUPER(VALUE(FATHER)) :< VALUE(POINTR)
      END
    RETURN
    END

    GRAPHICAL FUNCTION COPY(NODE)
    GRAPHICAL NODE
    COPY := NODE
    RETURN
    END

    REAL FUNCTION AVERGE(NODE)
    GRAPHICAL NODE,VALUE
    INTEGER ORDER,PRILEN
    VECTOR VUPNT,POINT,POINT1
    COMMON /HID$/ VUPNT
    AVERGE = 0.0
    IF(VALUE(NODE).EQ.BLANK) RETURN
    IF(POLYLINE(VALUE(NODE))) THEN
      BEGIN
        ORDER = PRILEN(NODE)
        DO 10 I = 1,ORDER
          CALL LINPNT(NODE,I,POINT)
          CALL APLYMD(NODE,POINT,POINT1)
          AVERGE = AVERGE + |VUPNT-POINT1|
10      CONTINUE
        AVERGE = AVERGE/ORDER
      END
    ELSE
      BEGIN
        IF(POLYGON(VALUE(NODE))) THEN
          BEGIN
            ORDER = PRILEN(NODE)
            DO 20 I = 1,ORDER
              CALL POLPNT(NODE,I,POINT)
              CALL APLYMD(NODE,POINT,POINT1)
              AVERGE = AVERGE + |VUPNT-POINT1|
20          CONTINUE
            AVERGE = AVERGE/ORDER
          END
        ELSE
          BEGIN
            POINT = (0,0)
            CALL APLYMD(NODE,POINT,POINT1)
            AVERGE = |VUPNT-POINT1|
          END
      END
    RETURN
```

```
END

GRAPHICAL FUNCTION TRAVRS(ROOT)
GRAPHICAL ROOT,POINTR,VALUE,SUPER
TRAVRS = VALUE(ROOT)
IF(TRAVRS .EQ. BLANK) RETURN
POINTR :- ROOT
WHILE(VALUE(VALUE(POINTR)) .NE. BLANK) DO
  BEGIN
    VALUE(VALUE(POINTR)) :< VALUE(SUPER(VALUE(POINTR)))
    POINTR :- SUPER(VALUE(POINTR))
  END
RETURN
END
```

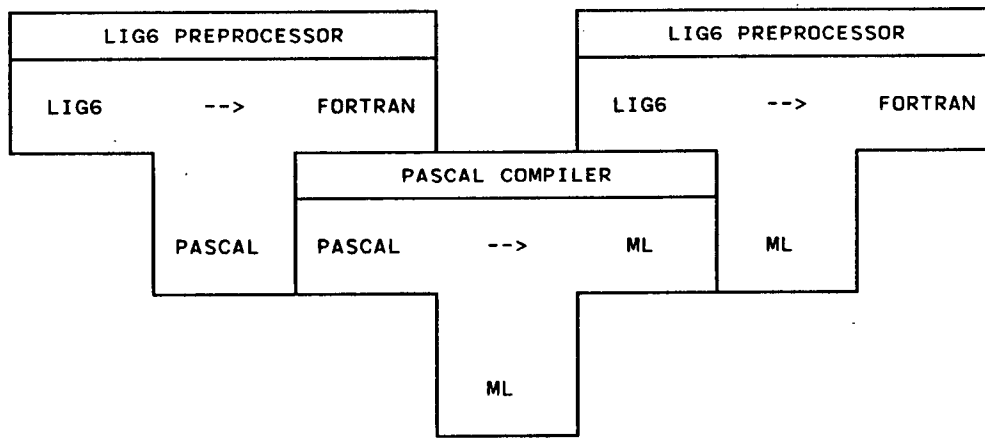
APPENDIX B

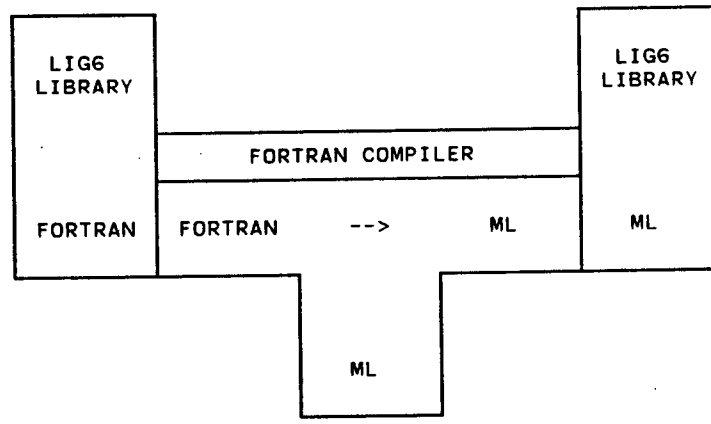
Implementation Notes

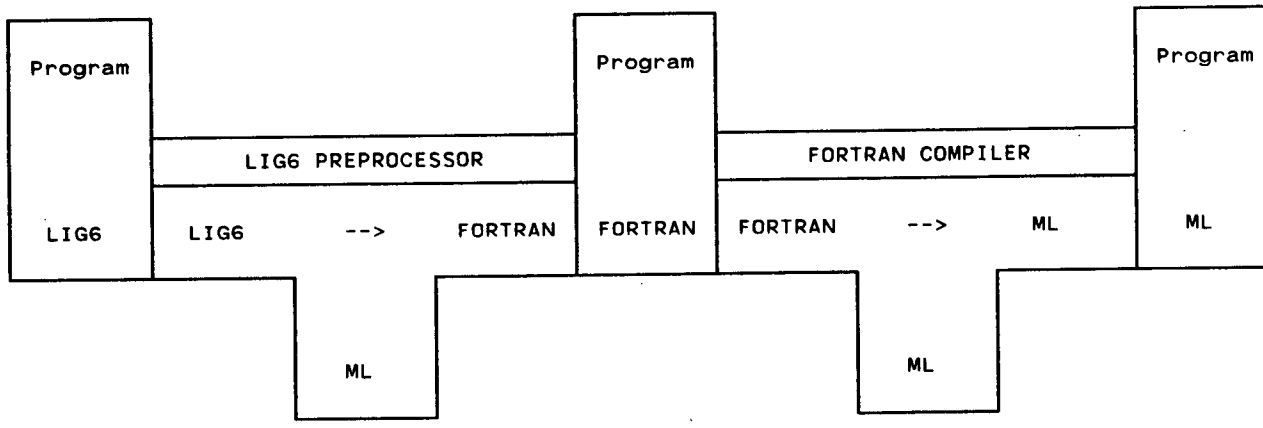
The language LIG6 is implemented on the University of British Columbia Computing Centre's Amdahl 470 V/8 computer under the Michigan Terminal System (MTS) operating system. The following MTS commands will generate the preprocessor and run-time library and execute a LIG6 program contained in the file PROG.S.

```
$RUN *PASCAL SCARDS=LIG6.P SPUNCH=LIG6.O
$RUN *FTN SCARDS=LIG6.LIB.F SPUNCH=LIG6.LIB.O
$RUN LIG6.O SCARDS=PROG.S SPRINT=PROG.L SPUNCH=PROG.F
$RUN *FTN SCARDS=PROG.F SPUNCH=PROG.O
$RUN PROG.O+LIG6.LIB.O
```

The following "T" diagrams, using the notation of McKeeman et al. [McKe70], represent the same process.







Several statistics regarding the implementation of LIG6 have been obtained. The preprocessor was written in PASCAL; it consists of 328 procedures totalling 18,100 lines of source code. This represents a listing of over 300 pages and occupies a disk file containing 168 pages (each disk page contains 4096 bytes). The object code resulting from the compilation of the preprocessor source requires a disk file of 84 pages. It takes 0.413 seconds of CPU time to load. A 650 line LIG6 test program containing examples of all syntactic constructs took 3.449 seconds of CPU time and cost $1.08 for the preprocessor to analyze the program, create an equivalent 1660 line FORTRAN

program, and produce a listing. In comparison, the cost of listing the same program was $0.50.

The translation of a LIG6 program into an equivalent FORTRAN program involves the replacement of extension constructs with calls to subroutines in a run-time library. The concise graphical information in a LIG6 statement generally requires more than one subroutine call to be equivalently expressed. The actual expansion that occurs depends upon the program being preprocessed. In the test program mentioned above, the section dealing with purely graphical extensions was 134 lines long. The equivalent FORTRAN code was 530 lines, an expansion factor of 3.96. A LIG6 program which was used to automatically generate the data structure diagrams of this thesis is 197 lines long; its equivalent FORTRAN code is 364 lines yielding an expansion factor of 1.85. The hidden surface removal algorithm of Appendix A is 145 lines long; its equivalent FORTRAN code is 428 lines yielding an expansion factor of 2.95.

The run-time library was written in FORTRAN; it consists of 339 procedures totalling 10,232 lines of source code. This represents a listing of 166 pages and occupies a disk file of 95 pages. The resulting object code requires a 70 page disk file and takes 0.402 seconds of CPU time to load.