AN ABSTRACT OF THE THESIS OF

Juanita J. Ewing for the degree of Master of Science in

Computer Science presented on June 7, 1984

Title: Specification of Graphical Figures with Attribute Grammars

Redacted for Privacy

Abstract approved: Fred M. Tonge

The need for graphical presentation arises in many disciplines. This thesis describes a prototype software system, the graph creation system, which gives unspecialized users the ability to create and display a variety of graphical figures. Templates for types of graphical figures, and the user interactions for creating instances of a figure, are specified by a knowledgable designer.

Templates specify the figure to be drawn in a context-dependent hierarchical manner. Each template represents an attribute grammar encoded in a tree form. Templates contain spaces for selected end user input and rules for computing the entire figure given those inputs. The end user creates a figure by invoking a program and supplying the necessary inputs such as colors, fill patterns, and locations.

Specification of Graphical Figures with Attribute Grammars

by

Juanita J. Ewing

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed June 7, 1984

Commencement June 1985

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Chairperson, Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented June 7, 1984

TABLE OF CONTENTS

Introduction 1

Design Considerations 4

Technical Details 23

Designer Guide 31

Historical Perspective 63

Conclusion 72

Bibliography 74

LIST OF FIGURES

FIGURE 1. The selection and confirmation process 8

FIGURE 2. A wedge in tree form 14

FIGURE 3. An attribute grammar representing a barchart 19

FIGURE 4. A template tree representing a barchart 20

FIGURE 5. An instance of a template tree representing a barchart 21

FIGURE 6. A barchart graph 22

FIGURE 7. A template tree for the house example 34

FIGURE 8. A prompt file for a house graph 47

FIGURE 9. An alternate prompt file for a house graph 48

SPECIFICATION OF GRAPHICAL FIGURES WITH ATTRIBUTE GRAMMARS

INTRODUCTION

Overview

This paper presents a prototype software tool for designing and creating a variety of graphical figures quickly and easily. Software of this type could also be a component of a document preparation system , a statistical package or other system requiring the inclusion of graphical figures. This prototype, called the graph creation system (GCS), is composed of two distinct parts -- template creation and instance creation. Template creation defines a graph template which can then be used to build a particular instance of a graphical figure. Each graph template is represented by an attribute grammar. Instance creation uses a graph template to produce an instance of the template, known as a graph. The attribute grammar defines and controls this production process.

Terminology

We shall refer to the person who creates a graph template as the template designer, and to the person who uses a template to produce a graph as the user. The names also imply that the designer is more knowledgeable than the user. The interface to the user is more simplistic and requires no specialized knowledge of how the templates are created and maintained.

When the user invokes GCS the graphics terminal screen is divided into 2 viewports. The picture viewport is used for displaying graphical figures. The prompt viewport is used for displaying menus, prompts and instructions.

From the Designer's Point of View

The template designer creates a template which describes the target graphical figure. User interactions during the instantiation process are also specified by the designer. It is up to the designer to include instructions for the user's response, and friendliness in his interactions. Tools for interactions include facilities for instructions, prompts, and menus. The designer can also create operations for the modification of the graph. A subset of the C language and a set of built-in routines

for manipulating the instantiated template are provided for these operations.

From the User's Point of View

When a user invokes the system, he has a choice of four actions. 1) The user can create an instance of a graphical figure using a template defined beforehand. 2) The user can recall a previously created graph. 3) The user can save (store) a graph for future recall. 4) The user can quit (exit from) the graphical system. When either create or recall is choosen, the user may invoke operations to modify the graph. These are known as customized operations, since they are specified with respect to a specific template.

CHAPTER 2

DESIGN CONSIDERATIONS

Graph Creation

The design of a general purpose graph creation system takes into account several factors. Primitive graphic segment types are selected to make possible the creation of many types of graphical figures. The graph creation system allows a user to instantiate each graph template with ease. The instantiation section provides for selection of attribute values, visualization of attribute values subject to interpretation, such as color, and revision of selections.

The primitive graphic segment types are

- o point (marker)
- o line
- o circular arc
- o rectangle
- o text.

Combinations of these primitives are used to build a variety of complex segments. For example, two lines and a circular arc can make a complex segment called a wedge in

a pie chart. Similarly, a complex segment, triangle, can be formed with three lines. An axis for a scatter plot can be created out of a combination of a line and plus characters for tick marks. Complex segments can also be built out of other complex segments, or combinations of primitive and complex segments. A complex segment constructed using the three primitives marker, line and arc can be filled with patterns or solid colors, as can the primitive segment rectangle. For example, a wedge can be filled.

There is a distinction between complex segments and random groups of segments. The parts of a complex segment can be manipulated as a single unit. For example, during a translation operation, it is more convenient to specify a move for a wedge than it is to specify a move for two lines and an arc. The usefulness of this distinction becomes more apparent when operations on graphs are discussed.

Levels of complex segments can define a hierarchy. The advantage of defining such a hierarchy is that a graphical figure can be represented to the user at a higher level of abstraction. To return to the wedge example, the user might wish to specify one color for all parts of the wedge instead of specifying the same color three times (once for each line and once for the arc) to

achieve the same result. Any level of the hierarchy is composed of one or more segments, each of which may be primitive or complex.

Instantiation of a template depends on the features of the template. The template designer designs the picture portion of the graph and designates which portions require user interaction to complete. The type and details of the interaction are also specified. Several forms of interaction are provided; one form is the menu.

The designer may make use of preformed menus or create his own menus. The preformed menus are designed to help the user visualize attribute values. Some of these preformed menus are for color, text size and fill patterns. In creating his own menus, the designer may specify color and size of the displayed menu text as well as location. Not all displayed text need be a menu item - text is also used for instructions or prompts.

While creating an instance of a template, if the template designer has so designated, the user is prompted to select an item from the displayed menu. Menu selection is accomplished via graphic input in which the user moves an arrow cursor to point at the desired selection. Once a menu selection is made, the selection is highlighted and the user is asked to verify the selection. If the user

verifies the selection, the instantiation process continues to the next attribute. If the user does not confirm the selection, he is prompted to reselect and the verification process is repeated. The process is illustrated in Figure 1.

CREATE
SAVE
RECALL
QUIT

CREATE
SAVE
RECALL
QUIT

CREATE
SAVE
RECALL
QUIT

CREATE
SAVE
RECALL
QUIT

Create graph ?

Create graph ?
no

Create graph ?
no

Figure 1. The selection and confirmation process

Other forms of interaction with the user are possible. If the designer wishes he may also ask for information in a question and answer style. Answers may be integers, real numbers, or strings. Another form, graphic input may be used to reference locations in the picture portion of the screen.

A set of change operations allows the template designer to control which attributes the user may modifies after an entire graph has been created. These change operations allow the user to revise attribute values. An example of such a revision is a change in the color of a line from red to blue. A set of change operations is provided which modify the attributes of primitive segments. The template designer may combine these provided operations with a simple interpreted language to create complex operations known as customized operations. An example of a customized operation is reordering the wedges in a pie chart. This operation is complex because the locations of individual wedges are relative to their adjacent wedges. In addition, the relocation operation associated with the reordering must be applied to each wedge in a pie chart. Repetition requires some form of control structure, here provided by the interactive language. Each template may have a separate set of complex operations, customized to that template.

Template Creation

Template creation is distinct from the instance creation portion of the system. Here, the emphasis is on facilitating the creation of varied templates. Template instantiation requires an unambiguous and precise description of the resulting graph. To meet this requirement, the template designer actually defines a grammar.

For our purpose a grammar is a formal algebraic system describing the form of an acceptable instance of a graph. A grammar consists of a set of symbols and a set of production rules. It is the grammar which imparts a precise structure to the graph template.

A grammar to describe a graph template must be able to reference properties of the primitive segments such as color, position and size. The grammar's production rules should be able to refer to properties of primitive and complex segments defined in other production rules. Attribute grammars fulfill these requirements.

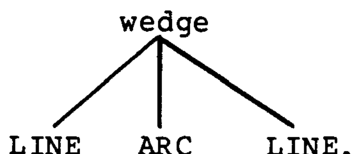
Each graph template is actually a different attribute grammar. The set of terminal symbols is restricted to correspond to the limited set of primitive segments and is the same for all templates. The set of rules and the set of nonterminals varies from template to template. Creation of these template grammars is accomplished using a

package which eases the tediousness of specifying the details of syntax and of attributes.

After specification, the grammar is restructured into a tree representing a graph template. The hierarchy inherent in the grammar's production rules is inherent to this tree structure also. The grammar rule

<wedge> ::= LINE ARC LINE

is represented by a tree form:



Explicitly, the child pointers taken together represent the grammar symbol '::=' (read, consists of). The next chapter formally discusses attribute grammars and the form they take. Symbols consisting of capital letters represent primitive segments. Symbols of small letters represent complex segments. For now, informal examples suffice.

Attributes of the grammar symbols are encoded as properties of nodes in the tree. In the following, the above example grammar rule is expanded to include the appropriate attributes. The value of attributes are specified by

attribution rules.

```
<wedge> ::= LINE ARC LINE

wedge.color = INPUT
wedge.fill_flag = TRUE
wedge.radius = INPUT

LINE(1).color = wedge.color
LINE(2).color = wedge.color
ARC.color = wedge.color

LINE(1).start_pos.x = INPUT
LINE(1).start_pos.y = INPUT
LINE(1).angle = 0
LINE(1).length = wedge.radius
ARC.angle = INPUT
ARC.center.x = LINE(1).start_pos.x
ARC.center.y = LINE(1).start_pos.y
ARC.radius = wedge.radius
ARC.start_pos.x = LINE(1).end_pos.x
ARC.start_pos.y = LINE(1).end_pos.y
LINE(2).start_pos.x = ARC.center.x
LINE(2).start_pos.y = ARC.center.y
LINE(2).angle = LINE(1).angle + ARC.angle
LINE(2).length = wedge.radius
```

Arcs can be specified in several different ways. This example uses a specification consisting of the center position of the virtual circle defining the arc, the radius of this circle, the angle of the arc and the starting position of the arc. The arc specification uses an attribution rule referring to the ending position of LINE(1), which is calculated by the system.

If a grammar symbol appears in a grammar rule more than once, the different instances of the symbol are distinguished by subscripts or postscripts. The numbering is

based on the order of appearances of the symbol.

In tree form this example template appears in Figure 2.

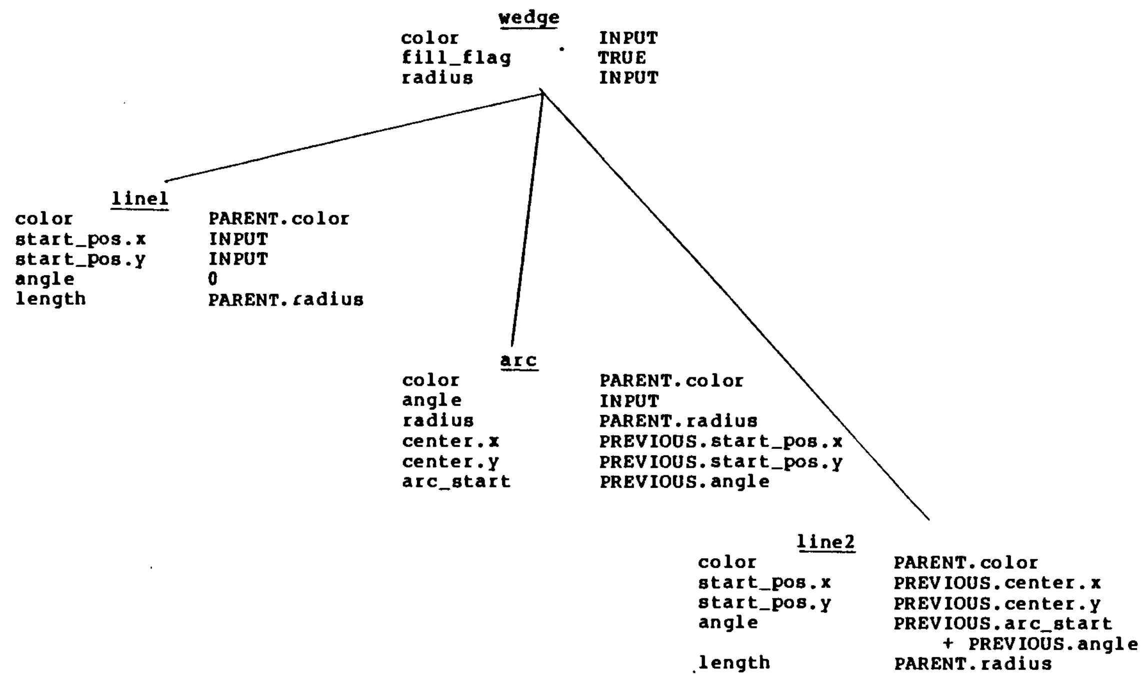


Figure 2. A wedge in tree form

References to attributes in the tree can be made to ancestors and to previous siblings (siblings to the left). This is because the tree is formed and processed in a preorder fashion, starting at the root and traversing depth first. Grammar rules which reference descendants or sibling to the right must be rewritten (possibly introducing additional attributes).

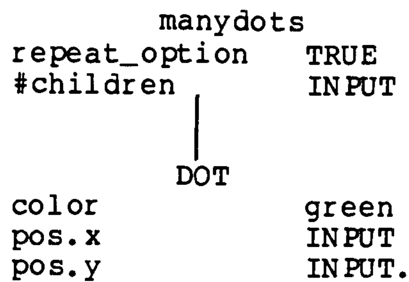
Attributes of nodes representing primitive segments are fixed in number and meaning, while nodes representing complex segments may generate instances with different numbers of attributes. Some of these attributes can be defined by the template designer, such as a wedge's radius. These attributes are known as designer defined attributes. The meaning of these attributes is conferred by the designer. Complex segment nodes also have predefined attributes such as color, and a flag to indicate filling.

Many grammars employ the shorthand notation {} to represent zero or more instances of a symbol. This notion is represented in the tree by a REPEAT attribute.

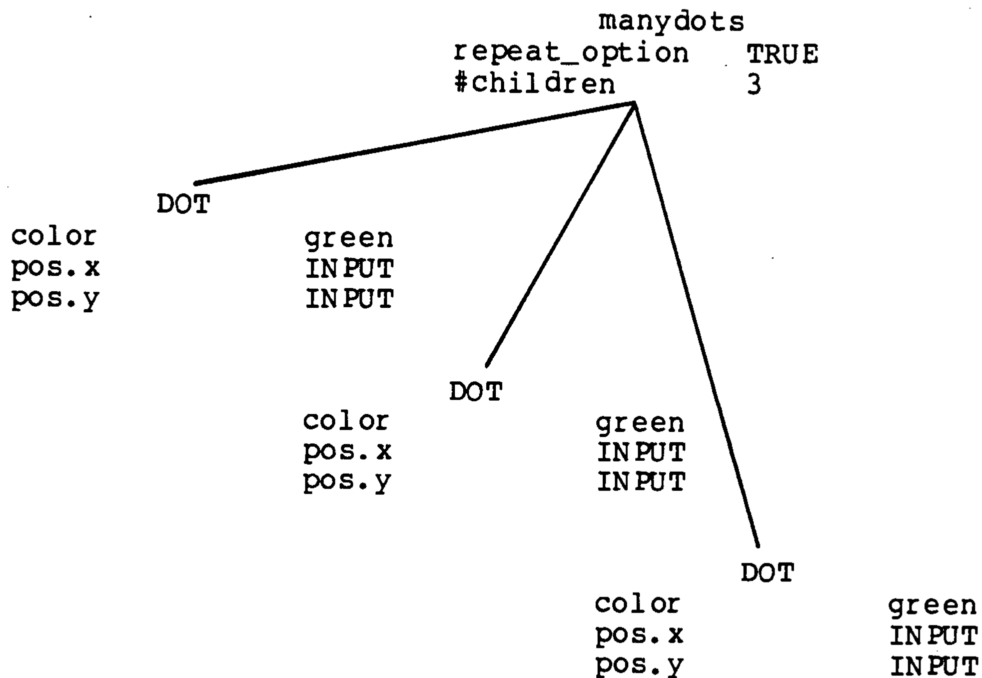
The grammar rule

<manydots> ::= { DOT }

is represented by the graph template tree



Note that in the instance of the graph the node 'manydots' will have as many children as the user desires. If the user specifies three children, the instance of this tree will appear as follows:



As a simplifying assumption, repeating nodes are restricted to one type of child . The following grammar rule

must be modified to accommodate the single child restriction.

```
<wedges> ::= { LINE ARC LINE }
```

The result of modification is two rules:

```
<wedges> ::= { <wedge> }
<wedge>  ::= LINE ARC LINE
```

These two rules are then made into a tree structure in a like manner.

Details of Instantiation

An instance of a graph is created by passing each attribution rule through a parser. This interpretive parser evaluates the expression specifying the attribute's value and returns this value. In order to simplify the parser, only two types of values are allowed, numbers and strings. Attributes which are instinctively thought of as other kinds of values, such as colors, are encoded as numbers.

As the template is processed, the instance tree is built to conform to the structure of the template tree, except for the use of the repeat option. The repeat option, as discussed above, allows the designer to specify a variable number of child nodes of the same description. This feature, in combination with a user specified number of children, is useful in barcharts and piecharts. These graphical figures naturally vary in the number of bars and wedges for each instance of the corresponding graph template.

Figures 3, 4, 5 and 6 show , for barchart as an example, an attribute grammar, a graph template tree, an instance of the corresponding graph tree, and the actual graph, respectively. This barchart example makes use of several numeric constants. The constants 15 designate the origin (lower left hand corner) of the barchart. 15 was chosen because it allows room for the barchart label. The constants 90 designate the outside (top and right) of the barchart. They were chosen to leave some aesthetically pleasing margins. Most attributes have been removed from Figure 5 in order to simplify the graph tree.

```
<bar chart> ::= <set of bars> <axis>

        barchart.xorigin = 15
        barchart.yorigin = 15

        set of bars.xorigin = barchart.xorigin
        set of bars.yorigin = barchart.yorigin
        set of bars.current x = barchart.xorigin
        set of bars.current y = barchart.yorigin
        set of bars.num_childs = INPUT
        set of bars.width = (90 - bar chart.xorigin)/
                set of bars.num_childs
        set of bars.rect width = set of bars.width * 0.7

        axis.xorigin = barchart.xorigin
        axis.yorigin = barchart.yorigin


<set of bars> ::= <bar> <set of bars> | [empty]

        bar.xorigin = set of bars(1).xorigin
        bar.yorigin = set of bars(1).yorigin
        bar.xstart = set of bars(1).currentx
        set of bars(2).currentx = set of bars(1).currentx  +
                set of bars.rect width
        bar.xend = set of bars(2).currentx
        bar.ystart = set of bars(1).yorigin
        bar.height = INPUT * 0.70 + set of bars(1).yorigin


<bar> ::= <rectangle> <bar label>

        rectangle.segment type = RECTANGLE
        rectangle.fill = TRUE
        rectangle.corner_pos.x = bar.xstart
        rectangle.corner_pos.y = bar.origin
        rectangle.oppcorner_pos.x = bar.xend
        rectangle.oppcorner_pos.y = bar.height

        bar label.segment type = TEXT
        bar label.text_chars = INPUT
        bar label.position_on = CENTER
        bar label.text_size = SMALL
        bar label.rotation = 30.0
        bar label.xcoord = (rectangle.corner_pos.x +
                                rectangle.oppcorner_pos.x)/2
        bar label.ycoord = rectangle.oppcorner_pos.y + 3


<axis> ::= <x-axis> <y-axis> <graph label>

        x-axis.segment type = LINE
        x-axis.start_pos.x = axis.xorigin
        x-axis.start_pos.y = axis.yorigin
        x-axis.end_pos.x = 90
        x-axis.end_pos.y = axis.yorigin

        y-axis.segment type = LINE
        y-axis.start_pos.x = axis.xorigin
        y-axis.start_pos.y = axis.yorigin
        y-axis.end_pos.x = axis.xorigin
        y-axis.end_pos.y = 90

        graph label.segment type = text
        graph label.rotation = 0.0
        graph label.text_size = LARGE
        graph label.position_on = CENTER
        graph label.pos.x = 50
        graph label.pos.y = 7.5
        graph label.text_chars = INPUT
```

Figure 3. An attribute grammar representing a barchart

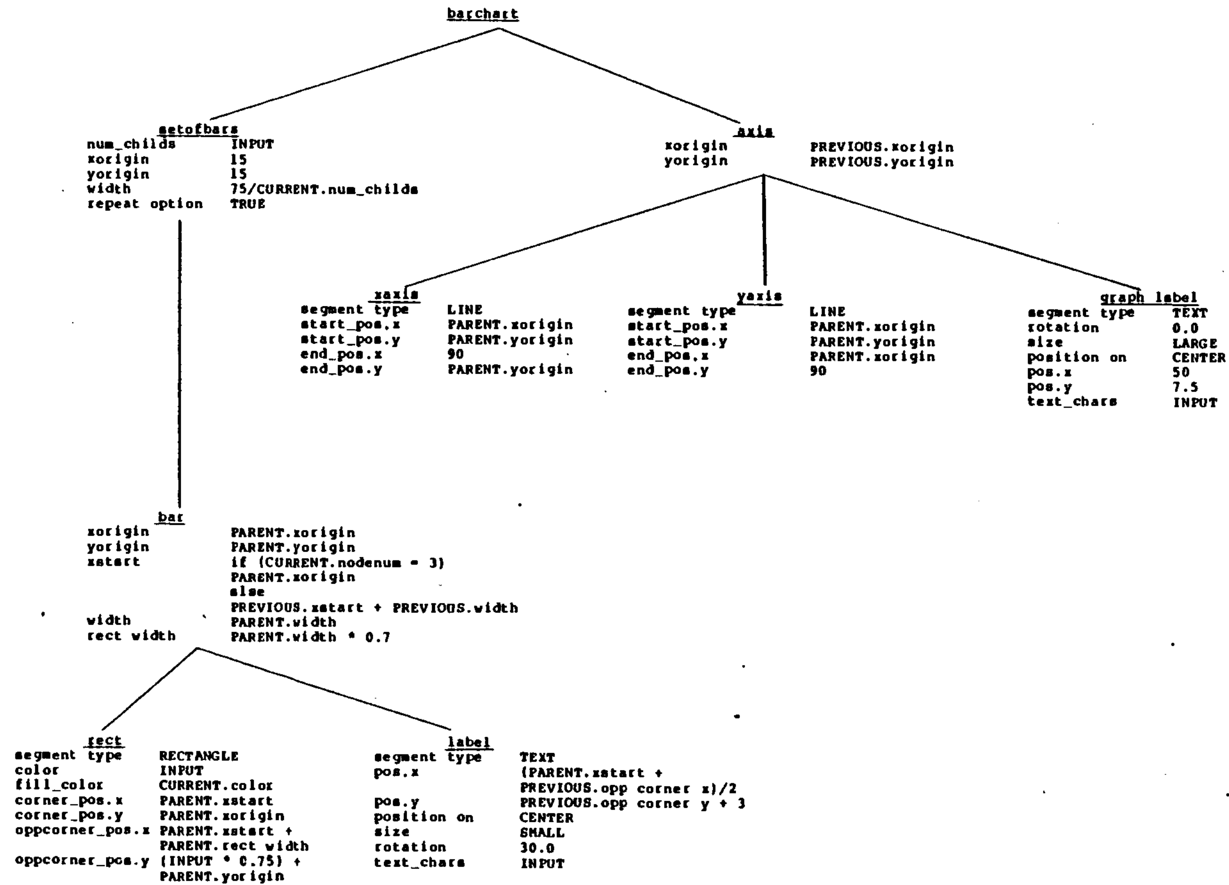


Figure 4. A template tree representing a barchart

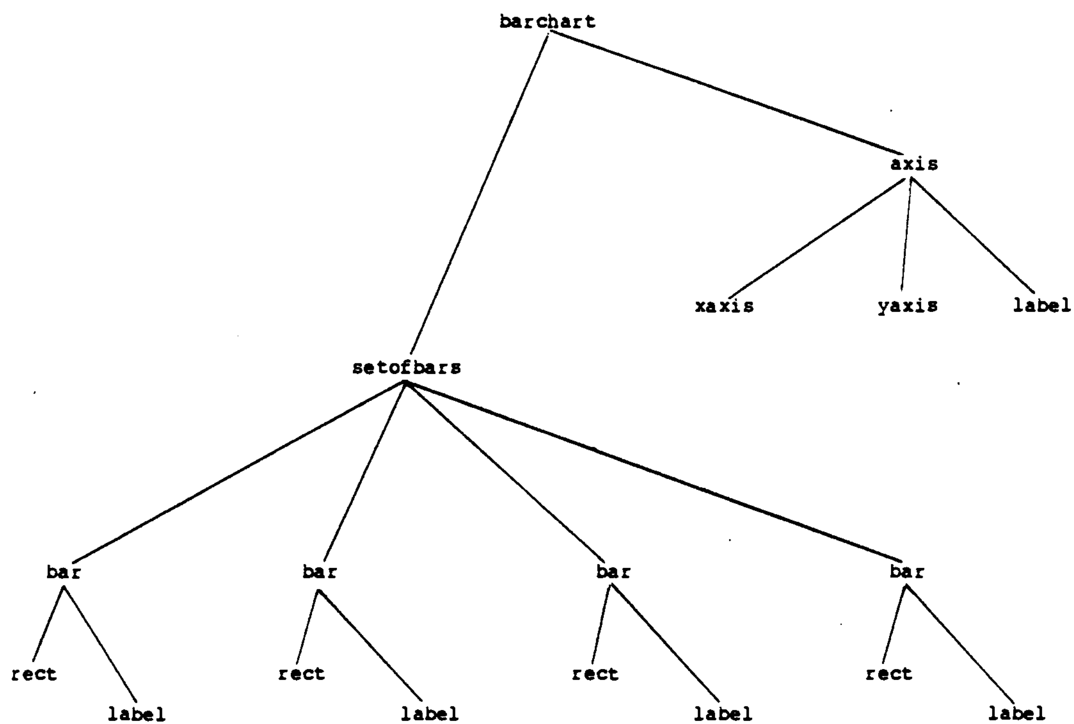


Figure 5. An instance of a template tree representing a barchart

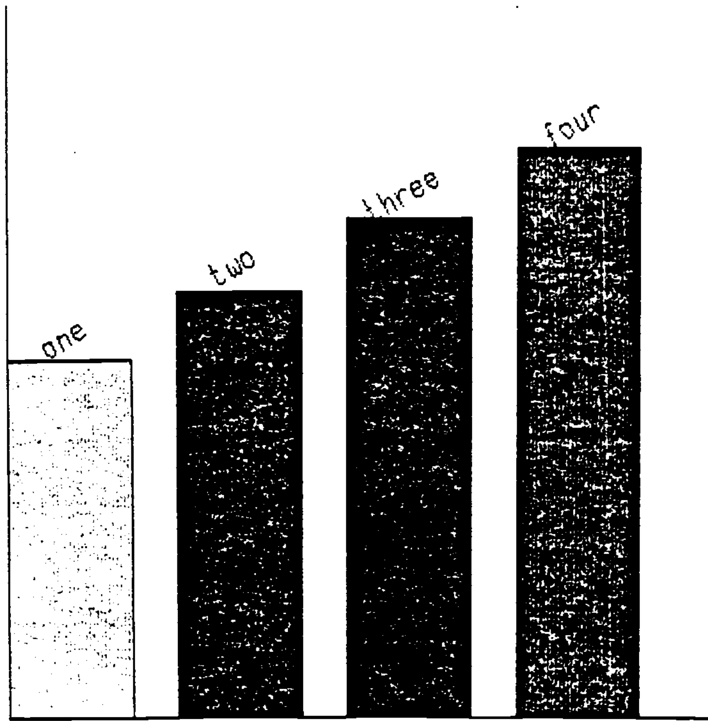


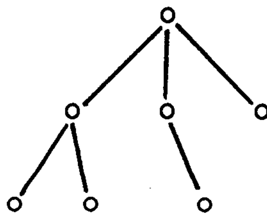
Figure 6. A barchart graph

CHAPTER 3

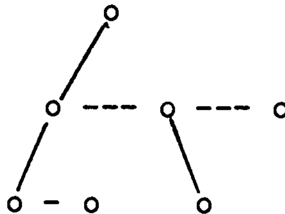
TECHNICAL DETAILS

Although the template designer builds a tree to describe the proposed graphical figure, the system never actually creates a template tree. Just the instantiated template tree, or the graph tree is built upon a CREATE or RECALL selection. The graph tree is a generalized binary tree, a convenient way to represent a tree with a variable number of children.

Regular Tree:



Equivalent Binary Tree:



Instead of a connection to each child node, there is a connection to the first child and successive children are connected. The two types of pointers in a generalized binary tree are first child pointers, and sibling pointers.

Attribute Grammars

An attribute grammar is a context free grammar augmented with attributes. The production rules of this grammar are augmented with attribution rules. Each attribution rule determines the value of an attribute. Attributes are synthesized, meaning information can be passed up the tree, or attributes are inherited. Inherited attributes pass information down the tree.

Most users of attribute grammars state that the grammars must be well formed and non-circular. Circularities could exist in attribution rules since these rules can reference other attributes. Attribute grammars are well formed when the terminal symbols have no synthesized attributes, the root symbol has no inherited attributes, and each attribute is accounted for by an attribution rule of the appropriate type, either synthesized or inherited . The restriction on attribute grammars goes one step further in this prototype of a graph creation system. Not

only are our attribute grammars non-circular and well formed, but the left-side attribute (the attribute being defined) can only have inherited attributes. The right-side symbols may still have synthesized attributes for horizontal information passing. This means information can only be passed down the tree and horizontally along the tree, in a left to right manner. The order corresponds to the order of grammar symbols in the production rules. Information passing also corresponds to the order of creation of nodes, a pre-order creation. That is to say, attribution rules cannot refer to attributes that do not exist at the time of evaluation, even though they may exist at some later time.

The form of attribution rules can be stated as: Every attribute grammar can be put in normal form, in which every semantic equation defines a value for an inherited attribute of a right-side symbol, in terms of zero or more inherited attributes of the left-side nonterminal and synthesized attributes of the right-side symbols. If the attribute being defined is an inherited attribute of a right-side symbol, it can only be defined in terms of inherited attributes of the left-side nonterminal, and synthesized attributes of the right-side symbols which are to the left of the symbol whose attribute is being defined. We consider only attribute grammars in normal

form.

Instantiated Tree

The entire graph creation system is written in C. Each node of the instantiated tree is represented by the C struct construct. Information in the node struct is divided into three categories. One category is the information contained by both types of nodes -- primitive and complex. The second category is information contained only by the nodes representing complex segments. Nodes containing complex segment information are internal nodes in the tree. The third category is information contained only by nodes representing primitive segments. Nodes containing primitive segment information are always leaf nodes in the instantiated tree.

The instantiated tree represents an attribute grammar as the template tree did. However, the instantiated tree's attribute values are constants, the result of evaluating the template tree's attribution rules. The dependencies of attributes on constants or other attributes is not expressed in the instantiated tree.

YACC

Output from the Unix tool YACC is used on two separate occasions in this system. The first occasion is when a graph is created. The instantiated tree is made by evaluating the template's attribution rules. When YACC is given the simple grammar of attribute expressions, it produces a parser called the expression parser. Though YACC is intended to be a compiler tool, it is modified to be an interpreter tool, so the expression parser is really an expression interpreter. Expressions which describe attribute values are parsed by the expression interpreter. YACC parses LALR(1) grammars. LALR(1) stands for a grammer requiring one lookahead token parsed left to right. The expression interpreter evaluates expressions involving the control structure "if" and additive, multiplicative and relational operators.

The second occasion which requires the use of the YACC's output is the customized operations. When given the LALR(1) grammar for customized operations, YACC produces a parser called the operation parser. When the customized operation is selected, the corresponding customized operation code is read by the operation parser, and a parse tree is constructed. Once the parse tree is completed, it is evaluated to perform the customized operation. No compiled code is ever produced, but compiled C procedures can

be called in the customized operation language.

Structure Overview

The C program which comprises the graph creation system, has five mainly disjoint program flows. All program flows start from the main menu selection routine. Selection of the Create option initiates the first program flow. The user is prompted for a template name and the corresponding files are opened. The instantiated tree is constructed in a pre-order fashion by successive calls to ggparse Each call to ggparse evaluates an attribution rule. Segments are drawn in the picture viewport as the tree nodes representing them are created. Prompts and menus for user interaction are displayed in the prompt viewport as indicated by the evaluation of attribution rules. After the instantiated tree has been built, control returns to the main menu selection routine, which now also has available the customized operations. The user can select either customized operations or options from the original main menu. The customized operations will continue to be available as long as this particular graphical figure is displayed. If another graph is Created or Recalled, a new corresponding set of customized operations will become available.

Another program flow results if the Save option is selected. The user is prompted for a file name to contain the graph. The instantiated tree, which also contains the TEK code (code intended for the Tektronix graphics terminal) for drawing the graph, is written to that file. Control then returns to the main menu selection routine.

The program flow resulting from the selection of the Recall option reverses the Save operation. The instantiated tree is recreated from the file. After the instantiated tree is built, all the TEK code to recreate the graph is sent to the terminal and the graph drawn. The customized operations for this graph are readied and control returns to the main menu selection routine.

The program flow for the Quit option is quite short. If the graph in its current state has not been saved, the user is questioned. If the user chooses to save the graph, flow is diverted to the Save option. After Saving (if it is selected), the program exits by closing files and returning to the normal mode of operation.

Selection of customized operations have a more complicated program flow since the code to perform the operation must be parsed and executed. One call to yyparse , a YACC output routine, creates the whole parse tree and one recursive function call evaluates and performs the opera-

tion. Necessary user interaction is displayed in the prompt viewport as indicated by the code for the customized operation. After the operation control returns to the main menu selection routine.

CHAPTER 4

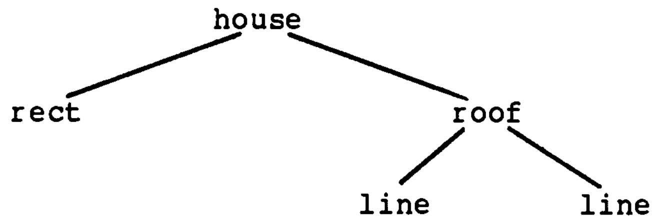
DESIGNER GUIDE

The process of making a template set is discussed in a step by step manner. A template set consists of up to three parts, a graph template, a set of user interactions, and a set of customized operations. The graph template describes the graphical figure. It is stored in a template file. The set of user interactions describe the means of getting information to and from the user. These interactions are stored in a prompt file. The code for customized operations is stored in the customized operations file. The minimum requirement for a template set is the graph template. The other parts are added as needed.

Making a Template

With the proposed graphical figure in mind, divide the figure into its components. Divide the components into their components, etc., until the components can be divided no further, that is, until the components are primitive segments. The process of division should have produced a tree-like structure. The root of the tree represents the proposed graphical figure. If one were to create a template for a picture of a house, the tree

representing the template might appear as follows.



Fill in the attributes values of the nodes by composing expressions to describe complicated attribute values. Use the attribute descriptions given below with a list of operators and control structures to compose these expressions. In the house example, some simple attribute descriptions are given for some of the tree nodes in Figure 7.

The house node has three designer defined attributes, center_x, height, and width. These three attributes are passed to the child node roof. The roof node also sets up filling attributes so that the roof will be filled with a pattern. Note that the user not only chooses the width and height of the house, but also the color of the roof. The notation for attribute expressions used here is as follows. The symbol CURRENT stands for the current node,

PARENT stands for the parent node and PREVIOUS for the previous sibling node. The actual template expressions use a shorthand notation for these symbols. The symbol PARENT would be replaced by &, CURRENT by $, and PREVIOUS by %. Designer defined attributes of complex segments are referred to by number.

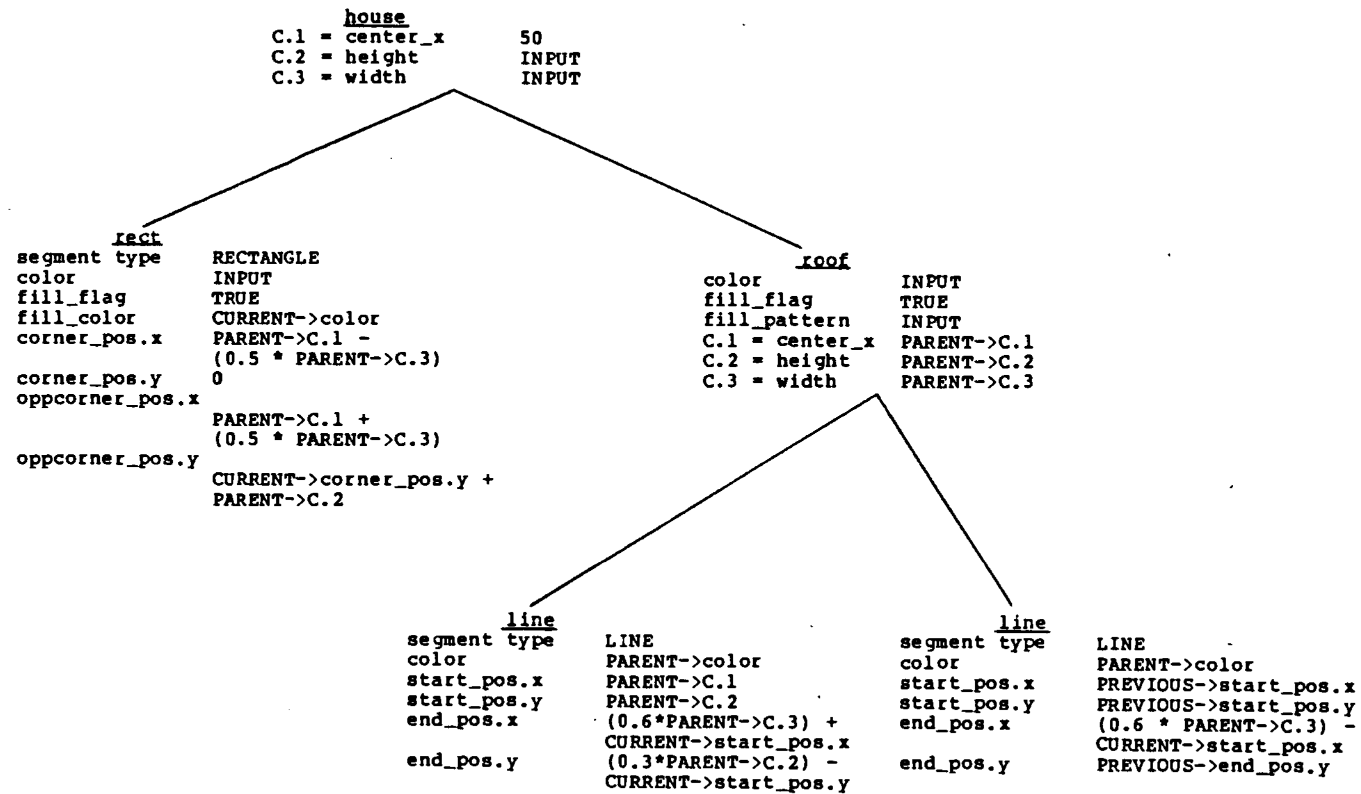


Figure 7. A template tree for the house example

Attribute Descriptions

All nodes have the attributes:

node_num

A preorder numbering scheme for the instantiated tree. Note that this numbering may be different from a similar numbering of the template tree if the repeat option is exercised.

color

If the node represents a complex segment, specification of color indicates this same value is used by the descendants. However, this color may be overridden by a differing specification in a descendant. Possible colors are transparent, blue, green, cyan, red, magenta, yellow, white, orange, olive, sea-green, dark blue, purple, burgandy, dark grey, light grey. (Colors available in an actual situation are dependent on the graphics device being used.)

Nodes representing complex segments also have the attributes:

fill_flag

This attribute indicates whether the segment is filled. It has a yes or no value.

fill_color

This attribute indicates which color is used for solid color filling. Possible colors are the same as for the color attribute.

fill_pattern

This attribute indicates which pattern is used for patterned filling. There are 15 possible patterns. Only one of the filling types, fill_color , or fill_pattern should be specified. If both are specified, patterned filling is ignored.

integral_flag

This attribute indicates if the children of this node are manipulated as a unit. It has a value of yes or no.

num_childs

This attribute indicates how many children this segment has.

num_consts

The number of designer defined attributes is indicated by this attribute.

C.number-representing-attribute

These attributes are defined by the template designer. The graph creation system assigns no semantic meaning to them. Designer defined attributes are referred to by number.

Nodes representing the primitive MARKER also have the attributes:

style

Markers are drawn with a dot, o or x mark.

pos.x

pos.y

Position of the marker in user coordinates: (0

- 100, 0 - 100)

Nodes representing the primitive LINE also have the attributes:

style

Lines are drawn with a solid, broken or dotted style. More styles are available with an interactive line style menu.

length

The length of the line in user coordinates is specified by this attribute.

angle

This attribute specifies the angle of the line from the horizontal in degrees rotating counterclockwise.

A line can be specified by a combination of starting position, length, and angle.

start_pos.x

start_pos.y

end_pos.x

end_pos.y

A line may also be specified by the starting and ending positions. The position attributes expect user coordinates, zero to one hundred in both dimensions.

Nodes representing the primitive TEXT also have the attributes:

position_on

The part of the text which is used as a location point - center, upper left, upper right, lower left lower right, center left and center right.

text_size

Text can be tiny, small, medium, or large.

rotation

Rotation of a text string is specified in degrees from the horizontal, rotating CCW.

pos.x

pos.y

This attribute specifies the location of the text as directed by the position on attribute. For example, if position_on = CENTER and pos.x = 50 and pos.y = 50, the center of the text string would be positioned at (50, 50).

Nodes representing the primitive RECTANGLE also have the attributes:

fill_flag

This attribute indicates whether the segment is filled.

fill_color

This attribute indicates which color is used for solid color filling. Possible colors are the same as for the color attribute.

fill_pattern

This attribute indicates which pattern is used for patterned filling. There are 15 possible patterns. Only one of the filling types, fill_color , or fill_pattern should be specified. If both are specified, patterned filling is ignored.

width

The width of the rectangle is given by this attribute.

height

The height of the rectangle is given by this attribute.

rotation

The rotation of the rectangle in degrees off the horizontal. A rotation of zero means the side of length width is horizontal. A rectangle may be specified by width, height, rotation, and corner point.

corner_pos.x

corner_pos.y

oppcorner_pos.x

oppcorner_pos.y

A rectangle can also be specified by positions of a corner and opposite corner, and rotation. Since the corners are fixed in this specification, differing rotations vary the dimensions of the rectangle.

adjcorner1_pos.x

adjcorner1_pos.y

A rectangle can also be specified by three points, corner, opposite corner and adjacent corner. The rotation of the rectangle in this

case is determined by the positions of the corners, and not by the rotation attribute. points.

adjcorner2_pos.x

adjcorner2_pos.y

All of the corner positions are assigned values during instantiation, and may be referred to, even though these attributes may have not been explicitly defined in the template. Starting from the lower left hand corner (before rotation) and going clockwise around the rectangle, the names of the corresponding corner attributes are corner_pos, adjcorner1_pos, oppcorner_pos, and adjcorner2_pos. Rotations do not change the relative positions of the corner attributes.

Nodes representing the primitive ARC also have the attributes:

radius

This attribute represents the radius of curvature of the specified arc. It can also be thought of as the radius of the virtual circle defining the arc.

angle

This attribute represents the angle the arc encompasses.

start_angle

This attribute represents the angle at which the arc begins. It is specified in degrees CCW from the horizontal.

start_pos.x

start_pos.y

mid_pos.x

mid_pos.y

end_pos.x

end_pos.y

Three points can specify an arc. These three points are at the start of the arc, the end of the arc, and any point in between the start and ending positions. To specify a circle using three points, make the starting and ending points the same position.

center_pos.x

center_pos.y

This attribute specifies the center of the virtual circle defining the arc. Arcs can also be specified by the center position, start position and angle, or by the center position, radius of the circle, angle and start_angle.

All of the arc positions are assigned values during instantiation, and may be referred to, even though these attributes may have not been explicitly specified in the template.

Operators and Control Structure

References to attributes in expressions specifying attribute values have the following syntax:

node indicator->attribute name.

The following node indicators are defined:

$	CURRENT
%	PREVIOUS
&	PARENT
# number	LITERAL
_	ROOT (equivalent to #1)

The following operators can be used in expressions specifying attribute values.

+	
-	
*	
/	
<	
<=	
>	
>=	
=	equality
!=	inequality
()	change in precedence

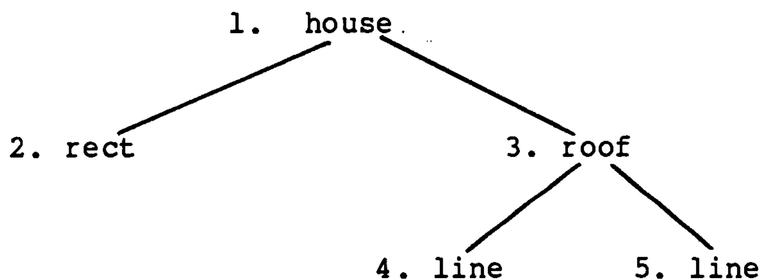
The control structure IF is available in the following form:

if (boolean) then expression else expression

The precedence of operators is the same as standard algebraic precedence.

Preparation for Input

After the template tree has been constructed and assigned attributes, linearize the tree. Number the nodes starting at the root and progressing to children before siblings (pre-order). This is the order of input to the template creation portion of the system. Invoke the template creation portion of the system by typing mk_tml. One of the first things asked for is a name for the template. Call the template a name representative of the figure you wish to create. The program will add a '.gsp' extension to the name to create a file containing the template tree. Input the attribute information for each node in the order of the numbering scheme. For example:



User Interaction

Make a separate list of attributes requiring user interaction, keeping the order of attributes the same as in the linear tree. The list of attributes requiring user

interaction for the house example is 1)height 2)width 3)house color 4)roof outline color 5)roof pattern. Decide what type of interaction is appropriate for each attribute and enter these interactions into a prompt file named the concatenation of the template name and the '.pmt' extension. For instance, the house might have a template tree file called house.gsp. The prompt file should have the corresponding name house.pmt.

Each specification of user interaction can have combinations of 4 components. These components are text-specifications, input-specifications, text-strings, and menus.

A text-specification directs how the text-strings should be printed. A text-specification is preceeded by a tilde character. An input specification directs how information gets from the user to the program. Input specifications are preceeded by a bar or pipe character. Menus and text-strings are a way to get information to the user from the program. Menus are preceeded by a percent character and text-string are lines beginning with any other characters than percent, bar, and tilde. Text-strings are simply displayed on the prompt viewport as directed by the most recent text-specifications. Usually they are intended for instruction or prompting. Menus are more complicated. Menus can consist of a predefined menu,

or a designer defined menu. The predefined dot style menu is indicated by two lines of specification:

```
%d
%e
```

The %d indicates the display of the dot style menu and the %e indicates the end of the menu. A designer defined menu begins with a %m and lines before the %e contain text-strings and/or menu-strings which have a corresponding value. The house example uses a designer defined menu to choose the size of the house. Before the menu begins 2 strings instruct the user to choose the height of the house. The menu-string '1 story' has an associated value (indicated by the @ sign) of 40, and the menu-string '2 story' has an associated value (indicated by the @ sign) of 70. Strings inside the menu definition with no associated values are assumed to be displayed for their information content, and not a valid selection of the menu. A designer defined menu could also contain other menus:

```
%m
transparent  @  0
%c
%e
```

or strings that have no value and are intended for display:

```
%m
Line Styles
%l
%e
```

Usually after a menu is displayed, the user is asked to make a menu selection. Menu selection is indicated by the input specification

```
|m
```

Follow the BNF below for each interaction with the user. Be sure to end each specification of interaction with a single percent (%) alone on a line. An example prompt file for the house example appears in Figure 8. The actual prompt file should contain only the specifications and not the comments shown in Figure 8.

```
Choose height
of House

%m             user constructed menu
1 story @ 40        menu contents
2 story @ 70
%e             end of menu
|m             input from the menu
%              end of this prompt specification
Enter width
of House
(0 - 100)
|i             integer input
%              end of this prompt specification
What color is
the house ?
%c             predefined color menu
%e             end of menu
|m             menu input
%
What color is
the outline
of the roof ?
%c             predefined color menu
%e             end of menu
|m             input from the menu
%              end of this prompt specification
What pattern
is the roof
filled with ?
%f             predefined fill pattern menu
%e
|m
%
```

Figure 8. A prompt file for the house example

The prompts can have many variations in size and color. A prompt file which takes advantage of these abilities appears in Figure 9.

```
~textcolor 4     Color is red
~textsize s      Size of printed text is small
Choose height
of House
%m
~textsize m      Size of text is medium
~textcolor 5     Color is magenta
1 story @ 40
~textcolor 6     Color is yellow
2 story @ 70
%e
|m
%
~textcolor 3     Color is cyan
Enter width
of House
(0 - 100)
|i
%
~textsize m     Size is medium
~textcolor 8    Color is orange
What color is
the house ?
%c
%e
|m
%
~textsize s      Size of text is small
What color
is the roof ?
%c
%e
|m
%
~textsize s
~textcolor 2     Color is green
What pattern
is the roof
filled with ?
%f
%e
|m
%
```

Figure 9. An alternate prompt file for a house graph

BNF for User Interactions

Interactions with the user are specified in a text file. The name of the file is the template name concatenated with a ".pmt" extension. The contents of this file, known as the prompt file, must conform to the following Backus-Naur specification.

```
<interaction> ::=  {<prompt> CR}

<prompt>  ::=     <menu> |
                  <text_specification> |
                  <input_specification> |
                  <text_string>


<text_specification>  ::=
                  <size_specification> |
                  <color_specification> |
                  <justification>

<size_specification> ::= TEXT_INDICATOR
                  textsize SIZE_CHAR

<color_specification> ::= textcolor COLOR_NUM

<justification> ::= center | left | right

TEXT_INDICATOR ::= ~

SIZE_CHAR   ::= s | m | l

COLOR_NUM   ::=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
                 9 | 10 | 11 | 12 | 13 | 14 | 15


<input_specification>  ::=
                  INPUT_INDICATOR (m | g | i | r | s)

INPUT_INDICATOR ::= |


<menu>  ::=   <predefined_menu> |
              <designer_defined_menu>

<predefined_menu> ::=    MENU_INDICATOR MENU_CHAR
                         MENU_INDICATOR e

<designer_defined_menu> ::=
             MENU_INDICATOR m CR
              { (<prompt> | <menu_string>) CR }
              MENU_INDICATOR e

MENU_INDICATOR ::= %

MENU_CHAR ::= c | t | l | d | p | f
```

```
<menu_string> ::= <prompt> VALUE_INDICATOR <number>

VALUE_INDICATOR  ::= @


<text_string> ::=  <any printable character or
                    space, except ~, %, |>
                    {<any printable character or space>}

<number> ::=  <any sequence of digits with an
                   optional period character to compose
                   an integer or a real number>

CR  ::= <carriage return>
```

Interpretations of Symbols

The meaning of some symbols from the above grammar are explained in this section.

Predefined menus are specified by the following symbols:

%c	color menu
%d	dot style menu
%f	fill pattern menu
%l	line style menu
%p	position_on menu
%t	textsize menu

Designer defined menus are specified by the following group of symbols. The contents of the menu can include other menus, strings or specifications.

```
%m
contents-of-menu
%e
```

Types of input are specified by the following symbols.

```
|m      menu choice
        (arrow in prompt viewport)
|g      graphic input
        (crosshairs in picture viewport)
|i      integer
|r      real number (no exponential notation)
|s      string.
```

Prompts and string both inside and outside of menus can vary in size, color, and justification. Use these specifications to vary the text.

```
~textsize  S
~textcolor  N
~left
~right
~center
```

A numbering scheme is used to represent colors. This numbering scheme is the same one that is used in template creation. The argument N in textcolor can have these integer values.

0 transparent
1 blue
2 green
3 cyan
4 red
5 magenta
6 yellow
7 white
8 orange
9 olive
10 sea-green
11 dark blue
12 purple
13 burgandy
14 dark grey
15 light grey

Default Specifications

Defaults at the beginning of each interaction are white, medium, and centered text.

Recommendations

A good deal of literature has been published about user interfaces. Appropriate ideas have been incorporated into the user interaction language of GCS, such as the use of menus for novice users. Some ideas would be relevant to a template designer. Savage et al [18] found that most user interaction problems were due to ambiguous terminology. Marcus [13] found that lower case letters were much more readable than all upper case letters. Malone [12]

showed that there is an optimal level of informational complexity. In other words, environments such as user interactions should be neither too complicated nor too simple.

Customized Operations

The addition of customized operations requires the designer to create another file which contains the code for these operations. The name of this file follows the established conventions of concatenating an extension to the template name, the extension to the file name in this case being ".cop". If a template named barchart has customized operations, they would be in a file called barchart.cop. Up to three files could exist for one template. In the barchart template example, the template would be stored in barchart.gsp, the prompt specifications in barchart.pmt and the customized operations in barchart.cop.

The customized operations are specified in a subset of the C language, with a number of built in procedures. The built in procedures are designed to facilitate the modification of attributes in the instantiated tree. Customized operations can also ask for input from the user. The input specification for customized operations is given

in the same file where the input is needed, rather than in a separate file. The input specifications in the customized operations file follow the same BNF as the input specifications in the prompt file.

BNF for Customized Operations

```
<customized operation>  ::=
        [<declarations>]  {statements} @

<declarations>  ::=
        { int | rel | nodeptr
        <identifier>  CR }

<statement>      ::=  <while_stmt> |
                      <if_stmt> |
                      <assign_stmt> |
                      <proc_call>

<while_stmt> ::=
        while (<bool>) {<statement>} endwhile

<if_stmt> ::=
        if ( <bool> ) {<statement>}  else
                      {<statement>}  endif

<assign_stmt> ::= <identifier> = <expression>

<proc_call> ::= <identifier> ( <param_list> )

<param_list>    ::= <exp> {, <exp>}

<bool>   ::= <relational_exp> |
             <not_exp> |
             <exp>

<relational_exp> ::=
        <exp> <relational_op> <exp>

<relational_op>  ::=
         >=  | >  | <=  | <  | ==  | !=

<not_exp>   ::= ! <exp>

<exp>    ::=  <binary_exp> | <minus_exp> |
              <primitive_data>

<binary_exp>  ::= <exp> <binary_op> <exp>

<binary_op>   ::= + | - | * | /

<minus_exp>   ::=  - <exp>

<primitive_data>  ::=  <numeric_constant> |
                       <identifier> |
```

```
                             <literal_string> |
                             <input_specification> |
                             <pointer> |
                             <pointer_field>

<input_specification> ::=
        input  <a user interaction following
        the BNF of user interactions>

<pointer>   ::=  ROOT | LIT

<pointer_field>  ::=
        <nodeptr> -> <attribute_name>

<nodeptr>  ::= <pointer> | <identifier>

<identifier>  ::=  <any sequence of letters
        and digits beginning with a letter>

<numeric_constant>  ::=  <any sequence of
        digits containing an optional period>

<literal_string>  ::=  <any sequence of
        letters and digits>

<attribute_name>  ::=  <any string which
        constitutes a legal attribute - see
        list of attributes>
```

Attributes are referenced via a <nodeptr> in the language of customized operations. The list of available attributes is the same as in the previous section, with two additions. All nodes have the attribute sibling_ptr. This attribute allows horizontal movement. Nodes defining complex segments have the attribute child_ptr. This attribute allows downward movement in the tree.

Customized operations have access to the following predefined procedures:

```
set_complexseg_filling
( nptr, fill_flag,
  fill_color, fill_pattern)
        fill_flag: integer
                0 = TRUE
                1 = FALSE
        fill_color: integer
                0...15  correspond
                to color values in
                user interaction
                specifications
        fill_pattern: integer
                1 = diagonal crosshatch
                2 = vertical zigzag
                3 = horizontal zigzag
                4 = horizontal lines
                5 = horizontal crosshatch
                6 = brick pattern
                7 = vertical lines
                8 = diagonal lines
                9 = dense dots
                10 = sparse dots
                11 = diagonal dots
                12 = vertical dots
                13 = even sparser dots
                14 = stone pattern

set_const (nptr, number, value)
        number: integer
        value: real

set_dot_color (nptr, color)
        color: integer  corresponding
                to color values in
                user interaction
                specification. All
                color parameters follow
                this convention.

set_line_color (nptr, color)

set_text_color (nptr, color)

set_rect_color (nptr, color)

set_arc_color (nptr, color)

set_dot_style (nptr, dot_style)
        dot_style: integer
                0 = dot
                1 = little + sign
```

```
                2 = normal + sign
                3 = percent sign
                4 = circle
                5 = X
                6 = square
                7 = diamond

set_line_style (nptr, line_style)
        line_style: integer
                0 = solid
                1 = dotted
                2 = dot - dash
                3 = small dashes
                4 = big dashes

set_dot_pos (nptr, pos.x, pos,y)
        pos.x, pos.y or any other
        arguments ending in pos.x
        or pos.y: real
                0.0...100.0

set_line_length_etc
(nptr, length, angle,
 start_pos.x, start_pos.y)
        length: positve real
        angle: real

set_line_pos_etc
(nptr, start_pos.x, start_pos.y,
 end_pos.x, end_pos.y)

set_text_size (nptr, text_size)
        text_size: integer
                0 = tiny
                1 = small
                2 = medium
                3 = large

set_text_chars (nptr, text_chars)
        text_chars: a string of characters

set_text_rotation (nptr, rotation)
        rotation: real

set_text_pos
        ( nptr, position_on, pos.x, pos.y)
        position_on: integer
                1 =  top right
                2 = top left
                3 = center right
                4 = center left
```

```
                 5 = bottom right
                 6 = bottom left
                 7 = center center

set_rect_filling
(nptr, fill_flag,
 fill_color, fill_pattern)
       See set_complexseg_filling.

set_rect_width_etc
(nptr, corner_pos.x, corner_pos.y,
 width, height, rotation)
       width: real
       height: real

set_rect_2pt
(nptr, corner_pos.x, corner_pos.y,
 oppcorner_pos.x, oppcorner_pos.y,
 rotation)
       rotation: real

set_rect_3pt
(nptr, corner_pos.x, corner_pos.y,
 adjcorner_pos.x, adjcorner_pos.y,
 oppcorner_pos.x, oppcorner_pos.y)

set_arc_3pt
(nptr, start_pos.x, start_pos.y,
 mid_pos.x, mid_pos.y, end_pos.x,
 end_pos.y)

set_arc_2pt
(nptr, center_pos.x, center_pos.y,
 start_pos.x, start_pos.y, angle)
       angle: real

set_arc_angles_etc
(nptr, center_pos.x, center_pos.y,
 start_angle, radius, angle)
       start_angle: real
       radius: real
       angle: real


find_node (nptr, idstr)
       idstr: a string of characters
```

The argument nptr represents a pointer to a node in the instantiated tree. The node pointer must point to the appropriate node to perform the operation. For example, set_text_size () changes the size of the displayed text. The first argument nptr must be a pointer to a tree node representing the primitive segment TEXT.

Here is an example of a simple customized operation for the house example. This operation changes the color of the house to a new color interactively specified by the user.

```
NewColor
Change House Color ?
nodeptr rect_ptr
int color

rect_ptr = ROOT->child_ptr
color = INPUT
House Color:
%c
%e
|m
%
set_rect_filling (rect_ptr, rect_ptr->fill_flag,
    color, rect_ptr->fill_pattern)
@
```

This operation has two variables, rect_ptr and color. The value of color is gotten from the user via a user interaction specification. The color of the filled rectangle representing the house is changed with the procedure set_rect_filling (). Two arguments to the procedure are the values before the procedure was called, rect_ptr-

>fill_flag and rect_ptr->fill_pattern. These values will remain unchanged. Note that the customized operation ends with an @ sign.

CHAPTER 5

HISTORICAL PERSPECTIVE

The graph creation system (GCS) borrowed ideas from research areas of attribute grammars, user interfaces and graphics. With the exception of the Unix tool YACC, the system was designed and implemented by the author.

Most work with attribute grammars has involved languages and compilers. Originally attribute grammars were used to represent the semantics of languages [11]. Uhl et al [21] used attribute grammars as a tool for the formal specification of the static semantics of Ada. Farrow [5] writes of an attribute grammar describing semantic analysis, storage location and translation to intermediate code for Pascal-86.

Attribute grammars are useful for describing situations that are context dependent, such as the semantics of a programming language. Because of their declarative nature and modular way of stating dependencies, Reps et al [16] have stated attribute grammars are a good basis for language-based editors. Another use of attribute grammars is as a representation of the instruction set of a target architecture, allowing machine dependent code generation and optimization. The use of attribute grammars in GCS to

describe graphical figures is another example of a formal specification of a context-dependent situation. The context-dependency is that specifications of one portion of a graph are dependent on another portion of the same graph.

The semantic trees produced by Reps et al [16], in the specification of a language based editor, are much like our instantiated template tree in that each attribute instance has a value. Semantic trees, however, store the attribution rules and can be dynamically updated. This allows changes in one part of the tree to be propagated through out the whole tree. This ability is not present in the instantiated template tree.

User Interfaces

GCS has a way for the template designer to specify when and what types of user interaction are performed. Borufka and Kuhlmann [4] defined a method called Dialogue Cells to define user interactions. One of the elements of this method is a language to specify dialogue-data structures and dialogue control flow. Dialogue Cells have 5 input classes which make up the dialogue-data structures. GCS has only 4 input classes and no higher level data structure. Dialogue Cells can have independent loops in

the user interaction, which GCS can not. Clearly, Dialogue Cells are a general tool much more sophisticated than the user interaction language of GCS. However, GCS's language embodies many of the same concepts such as echoing, prompting, and conversion of input data into various forms. The tree-like structure found in Dialogue Cells produces a hierarchical context dependent interaction. For example, prompts can vary depending on the task performed, in which the task becomes the context of the interaction. This is similar to the approach GCS uses in graphical figure definitions.

Blesser and Foley [3] designed a specification language to describe the human factors aspect of user interfaces. Some of the features they emphasized were too sophisticated to compare to the simple user interaction language of GCS, but are important enough to deserve mention. Their specification language emphasized consistent use of output characteristics such as color, font capitalization and highlighting. GCS allows variations in color, size and capitalization, but does not enforce any consistent use of these characteristics. Another emphasis of their specification language was completeness and alternative subdialogues. Subdialogues gave alternatives for different interaction styles. GCS does not even attempt coverage in these areas.

Kasik [10] of Boeing discusses a user interface management system that removes the burden of physical interaction handling from the system designer. Essentially, that is what the user interaction language of GCS has tried to do for the template designer. Interactions are specified with a language called TICCL. TICCL provides recursive command re-entry, implied reject processing in which the user chooses a command in the middle of another command and implicitly rejects the first command, and situation dependent commands. These features, though beyond the scope of GCS, make for a good user interface. TICCL allows a designer to specify dialogue outside the application, just as GCS does. Kasik argues that this separation makes it more natural to design the dialogue sequences before or in parallel with the algorithms. Therefore, the end user is provided with a "more powerful and consistent interaction capability" in both TICCL and GCS.

These user interaction systems are generally more extensive than GCS's user interaction language, but share some of the same ideas. A system such as Dialogue Cells may have been worth adapting to GCS, rather than building a separate specification language. It is certainly more complex than the current specification language.

Graphical Figure Descriptions

A number of groups have produced systems which describe and draw graphical figures. Features of these systems which are related to hierarchical descriptions, interactive use, generality of the system, and graph component interdependence are discussed.

Another facet of the graph creation system (GCS) is the ability to describe graphical figures. Shimomura [19] describes Nippon's method for generating business graphs. Attributes such as color, shading and translation can be specified by the user in Nippon's system, as they can in GCS. Another important characteristic of Nippon's system and the graph creation system is the interactive editing. Things like modifying sequences, colors, size and location of comments are possible in both systems. A nice feature included in Nippon's system is the storage of all specifications, including the interactively modified ones, in a frame. This frame allows plotting of new data automatically with the same chart definition. Nippon's system simplifies the problem of drawing graphical figures by allowing a limited set of graphical figures, all of which are business graphs. GCS on the other hand, can draw an unlimited number of graphical figures, and is not restricted to business graphs.

Wong and Reid [22] describe TRW's user interface dialogue design tool, called FLAIR. Besides tools for user interface, FLAIR includes the ability to use various input devices including voice, integration with a relational database and manipulation of graphics objects or primitives. The graphics primitives are similar to the set of primitives provided by GCS. FLAIR has an additional primitive, free drawn lines controlled by cursor movement. Graphics objects are built out of sets of primitives. Once an object is created, it too can become a primitive. Primitive creation in this manner allows a graphics object to be represented in a tree-like manner. The description of a graphics object is actually a hierarchy of primitives, some of which may be decomposed into another set of primitives, etc. GCS also describes its graphical figures in a tree form.

Graphics objects can be associated with a set of steps used to create the object. This step-wise pathway can be reproduced, so a user could see the building and modification of a barchart, for instance, drawn again step by step. Though all the primitives exist to construct many types of graphs, FLAIR doesn't have the capability to make a model or template for a graphical figure. Each graph would have to be created interactively and can only be stored as an instance. The creation of graphs in this

manner would be tedious, since no facility is available for variable numbers of graph components, such as wedges in a pie chart. Since FLAIR has no way to store a model for a graphics object, there is no way to store context dependencies the way GCS can. A GCS template can make the color of a graph component be dependent on the color of some other component. GCS can easily represent relative types of dependencies in an attributed template tree. Though FLAIR represents graphics objects in a hierarchical manner, no facilities for models or context dependencies exist. However, FLAIR is a powerful and useful tool in many other respects.

AT&T's language and system for the interactive analysis of data, called S, shares some design goals with GCS [2]. Making graphics interactive and easy to use, and making the user interface easy to use, yet flexible are goals both S and GCS share.

Interactive graphics are an important part of S. S has graphic input to indicate positions on the screen. The major graphical figure in S is the scatter plot. Extensions to scatter plots can be constructed using macros, and new kinds of plots can be built with the components lines, points, axis, labels, etc. These components are powerful because they are linked to complex algorithms which analyze the data. However, these

components are not as general purpose as the components of GCS.

Much of S is devoted to data analysis and statistics; however, S has an interface language to describe user-algorithm interactions. GCS also has a language to describe user interactions. This is consistent with the goal of making the effectiveness of the user very important in both systems. S provides metafiles for the storage of graph specifications but no models for graphical figures besides variations of the scatter plot. S has no hierarchical descriptions facilities or context dependent properties that are so important to GCS.

An intermediate language for pictures (ILP) developed by ten Hagen et al [20] has some features in common with GCS. ILP has the ability to produce a hierarchical description of graphical pictures by the inclusion of subpictures. The subpictures are independent and can include further subpictures. This subpicture inclusion creates a tree structure much like the trees produced by GCS. The trees of GCS allow dependencies between subtrees via attribution rules, but only certain kinds of dependencies are allowed in subtrees in ILP. ILP allows graph locations to be relative, so one subtree may depend on the location of the picture drawn by the previous subtree.

ILP's description method is general enough to produce many types of graphical figures, including many too sophisticated for GCS to produce, such as three dimensional figures. ILP is meant to interface to some other programming language, and so has no facilities for user interaction besides graphic input. GCS has the ability to function as a stand alone system, as well as interface to other software such as document preparation or statistical systems. The interactive use of GCS has greatly influenced some features, such as the user interaction language, which ILP ignores.

GCS, unlike all these other systems, provides a general context dependent hierarchical method of describing graphical figures. It also has the ability to interactively create and modify graphical figures. The method is general purpose and can produce many different graphical figures.

CHAPTER 6

CONCLUSION

We have presented a graph creation system in which a designer specifies both a graphical figure and the user interactions necessary to create that figure. A user creates the graph according to the designer's specifications, in an interactive manner. If the designer has written customized operations, the user can modify the graph.

Among the unique features of this system is the data structure representation of the hierarchical nature inherent in graphical figures. Another feature is the use of attribute grammars as a clean (precise) method of representing context dependencies within a graphical figure. The facilities for interactive creation and modification of graphs are another important ability. The other unique feature is the ability to design so many different graphical figures with a basic set of building blocks.

Areas for extension and improvement include modifications of the instantiated tree (customized operations). If the instantiated tree had the attribution rules encoded, operations could make use of the dependencies elucidated in the graph template. This would make

operations on complex segments with dependent children much cleaner. For instance, if the height of the bar in a barchart depended on the overall size of the graph, changing the overall size would automatically affect the individual bar's size since the dependency could be enforced with the encoding present in the tree. Another improvement would be conversion from data type to data type within the graph creation, instead of just in the user interface. This would allow a number representing, for instance the height of a bar, to be used as a textual label also. An extension to the graph creation system would be the embedding of this system in a text system to make a document preparation system. The document preparation system would combine the functions of text editor, formatter and graph creation system without exiting one system and entering another to perform different functions. This would also allow text and graphics to be displayed simultaneously on the screen.

BIBLIOGRAPHY

1. Aho, A.V., Ullmann, J. D., Principles of Compiler Design, Addison-Wesley Pub. Co., Reading, Massachusetts, 1977.

2. Becker, R. A., Chambers, J. M., Design of the S System for Data Analysis, Comm. ACM 27, 5, (May 1984), 486-495.

3. Bleser, T., Foley, J. D., Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces, 1982 Proc. Human Factors in Computer Systems, 309-314.

4. Borufka, H. G., Kuhlmann, H. W., ten Hagen, P. J. W., Dialogue Cells: A Method for Defining Interactions, IEEE Comp. Graphics and Applic. 2, 5, (Jul 1982), 25-33.

5. Farrow, R., Experience with an Attribute Grammar-Based Compiler, 1982 ACM, 95-107.

6. Foley, J. D., Van Dam, A., Fundamentals of Interactive Computer Graphics, Addison-Wesley Pub. Co., Reading, Massachusetts, 1982.

7. Ganapathi, M., Fischer, C. N., Description-Driven Code Generation using Attribute Grammars, ACM 1982 Sym. on Principles of Programming Languages, 108-119.

8. Harrington, S., Computer Graphics A Programming Approach, McGraw-Hill Book Co., New York, N. Y., 1983.

9. Jacob, R. J. K, Using Formal Specifications in the Design of a Human-Computer Interface, 1982 Proc. Human Factors in Computer Systems, 315-319.

10. Kasik, D. J., A User Interface Management System, SIGGRAPH 82 Conf. Proc., 99-105.

11. Knuth, D. E., Semantics of Context-Free Languages, Math. Systems Theory 2 (1968), 127-145.

12. Malone, T. W., Heuristics for Designing Enjoyable User Interfaces: Lessons from Computer Games, 1982

Proc. Human Factors in Computer Systems, 63-68.

13. Marcus, A. Typographic Design for Interfaces of Information Systems, 1982 Proc. Human Factors in Computer Systems, 26-30.

14. Meyrowitz, N., van Dam, A., Interactive Editing Systems: Part I, ACM Comp. Surv. 14, 3, (Sep 1983), 321-333.

15. Meyrowitz, N., van Dam, A., Interactive Editing Systems: Part II, ACM Comp. Surv. 14, 3, (Sep 1983), 353-410.

16. Reps, T., Teitelbaum, T., Demers, A., Incremental Context-Dependent Analysis for Language-Based Editors, ACM Trans. on Programming Lang. and Sys. 5, 3, (Jul 1983), 449-457.

17. Rosenthal, D., Micher, J. C., Pfaff, G., Kessner, R., Sabin, M., The Detailed Semantics of Graphics Input Devices, SIGGRAPH 82 Conf. Proc., 33-38.

18. Savage, R. E., Habinek, J. K., Barnhart, T. W., The Design, simulation, and Evaluation of a Menu Driven User Interface, 1982 Proc. Human Factors in Computer Systems, 36-40.

19. Shimomura, T., A Method for Automatically Generating Business Graphs, IEEE Comp. Graphics Applic. 3, 6, (Sep 1983) 55-59.

20. ten Hagen, P. J. W., Hagen, T., Klint, P., Noot, H., Sint, H. J., Veen, A. H. A., Intermediate Language for Pictures, Mathmatisch Centrum, Amsterdam, 1980.

21. Uhl, J., Drossopoula, S., Persch, G., Goos, G., Dausmann, M., Winterstein, G., Kirchgassner, W., An Attribute Grammar for the Semantic Analysis of Ada, Springer-Verlag, Berlin, 1982.

22. Wong, P. C. S., Reid, E. R., FLAIR - User Interface Design Tool, SIGGRAPH 82 Conf. Proc., 87-98.