# XYZZY: A Visual Program Environment

**Michael Nahas**

## 1.0 Abstract

*XYZZY is a visual programming environment designed to increase programming efficiency by preventing errors, detecting errors which could not be prevented quickly, and finding information quickly. The language is simple, allowing only integer and boolean types. Functions follow a copy-in-copy-out paradigm which allows multiple values to be returned.*

## 2.0 Introduction

The origins of the computer terminal lie with the typewriter. The typewriter paradigm, the current programming paradigm, sees programs as a set of documents constructed of lines of text. This is analogous to sets of instructions written on many pages by a typewriter.

The goal for programming is to create the most useful program with the lease amount of resources. The goal for the programming environment is to allow the programmer to write the program as fast as possible by preventing errors, quickly detecting errors which cannot be prevented, providing needed information as fast as possible, coordinating multiple programmers and providing conceptualization.

There exist many hindrances within the typewriter paradigm to achieving this goal. The first of which is that the only one document, or file, can be changed at once. This means only a piece of the program is being changed rather than the entire program. This results in errors because only one small section is available to be referenced at once. If the editor had automatic error detection, it means errors can only be found using information in this piece of the program. Although the division into files permits simple reuse of code, all the code should be available all at once to allow real-time error checking.

The second result of the division of the program into many files is that repetition and dispersion of information is necessary in most cases. Information repetition means information is repeated in many locations, which causes problems if all copies are not the same. Information dispersion means related information is not all in the same location which results in longer searches for needed information and problems with consistency.

Another consequence of the typewriter paradigm is the linear nature of the files. Programs are organized into serial lines of text. Programs contain loops and branches which the programs try to differentiate by text formatting. The serial constructs do not represent the non-linear nature of a program well.

The typewriter paradigm is oriented around a single programmer. Multiple programmers can work on different files, but never on the same file at the same time. In addition, problems with consistency arise from old information and multiple versions of files. Information passing between programmers is usually done informally by talking rather than within the programming environment.

Lastly, the typewriter paradigm is based on a typewriter. This is reflected in the fact that most editors know nothing about the files they edit. The files are seen as text without meaning rather than as programs, just as a typewriter knows nothing about the document the author is writing. This condition means editors cannot do real-time error checking, communicate with other programmers, or help the programmer find the information he needs.

These problems prevent the typewriter paradigm of programming from achieving the goal satisfactorily. A new paradigm is needed which is interactive and specialized to programming. This new paradigm must perform active error checking and features to coordinate multiple programmers.

Although this new paradigm may be done in text or hypertext, this project explores creating the new paradigm in a visual programming language. A visual programming language is one in which every construct of the language is represented by a visual object, such as a button, menu, or fill-in pallet. This is different from Visual BASIC or Visual C++ where only the interface is constructed visually, and the code for the computational part of the program is done in plain text.

The reasons for choosing a visual programming environment over one of text or hypertext are few. This interface allows a better view of loops and branches than either of the other formats. It also prevents all syntax errors due to typing, because very little typing is done except to name variables and functions. The last reason a visual programming environment was chosen is that it is a break from the norm. This project, even though it does not contain all the functionality of a programming environment, can provide useful information on programming with a visual language which might be applicable to other areas of programming.

This project is the writing and testing of an environment for a visual programming language. The goals as far as the programming environment will be to increase programming speed by preventing errors, detecting mistakes as soon as possible, and by providing the needed information as quickly as possible. The goals for the testing will be to contrast the visual programming environment with a plain-text based one for the same language.

## 3.0 Previous Research

The research by others in visual programming, or programming environment that was applicable to the project was devided into three areas: displaying the program, finding information in the program, and entering the program.

## 3.1 Information Display

A comprehensive work in program information display is *Interacting Visual Abstractions of Programs[3]* by Ford and Tallis of the University of Exeter. In the paper are listed ten different ways of representing a program, both in a static code form and in a dynamic run-time form. The goal of their project was to create appropriate abstraction mechanisms in order to form a concrete language by which programs can be understood visually. By interconnecting these different views of the program, the user is supposed to comprehend the program faster and gain a fuller understanding of the entire program.

One important piece that is mentioned in Ford and Tallis's work is the use of filters. Filters are methods to remove information that would either cause confusion and misunderstanding or information which would not be useful and takes up valuable screen real estate. By removing the uninteresting data or code, such as unused variables or error checking code, the user can gain more quickly an understanding of the program.

Coulmann of the Technical University of Darmstadt makes good use of filters in his program *NATURAL Visualizer[2]*. Branches of code can be collapsed to save space on the screen for more code. *NATURAL Visualizer* also uses icons as a visual aid to identify different sections of code. Comments are marker by stars, branches by diamonds, and input and output by parallelograms. The icons are not part of a flowchart, but are the first thing on a row of code. The icons provide a good method for quick identification of section as well as providing clues to what each line does.

A good example of usage of a visual programming environment's non-linear aspect is *MEANDER*, a program described by Guido Wirtz of the Universitat-GHS-Siegen in *A Visual Approach for Developing, Understanding and Analyzing Parallel Programs[8]*. *MEANDER* has blocks of code written in text, with the connections between these block displayed in a visual environment. *MEANDER* represents thread creation by branches and message passing delays by horizontal lines between the blocks of code. This representation give an excellent overview of the interaction between threads in the program and reasons for the timing of execution. *MEANDER* excels in all aspects of a programming environment by allowing good conceptualization, providing useful information, and allowing errors to be detected easily.

## 3.2 Information Finding

Haberland, Poswig and Moraga of the University of Dortmund[5] have displayed an interesting method for finding the correct functions within an environment. This is an important task when one considers the number of functions present within the all the libraries available on most workstations today. Their display consists of a set of functions presented as a cone-tree, and queries on that set seen as blocks which contain the results on a cone-tree below it. Inter-set relations are seen as lines crossing between sets. This method of query is very compact and well suited for database queries.

## 3.3 Program Entry

As mentioned before, Dr. Wayne Citrin's *Requirements for Graphical Front Ends for Visual Languages[1]* is the only paper containing concrete data. The experiment was done to determine if a gesture based interface was better than a palette based interface. A gesture based interface changes the type of input based on a motion performed by the pointer device, while a palette based interface has a window of the available types of input and the user selection on by clicking on it.

Dr. Citrin noticed that users of the palette based interface spent a great deal of time shuttling the point from the work area to the palette to select a tool. Dr. Citrin hypothesized that accelerator keys would help speed things up. Gesture based interfaces saved time from this shuttling, but often encountered errors when the interface did not correctly recognize their gesture

# 4.0 The User's View of XYZZY

XYZZY has two different types of top level windows: the manager window and the function-editor window. The manager window contains all the program wide function buttons, and a button for each function window so it can be brought on top of all the windows quickly. The function window contains all the things needed for writing a function, such as variables, coding space, and a prototype for the function.

After the description of how the pieces of XYZZY work, a step-by-step example will be done showing how to construct a function which returns the maximum of two arguments.

## 4.1 Manager Window


**figure 4.1: sample manager window**

The manager window at start-up contains 4 buttons: **quit**, **export**, **run**, and **function editor**. **Quit** will close all the windows and exit the program. **Export** saves the file into a text form. **Run** executes the code with an interpreter. **Function editor** creates a new function

editor window. When the window is created, a new button with the name of the function is added to the manager window. When this button is pressed, the function editor window become the top window on the screen. In figure 4.1, max is a function which has a function editor open.

## 4.2 Function Editor Window

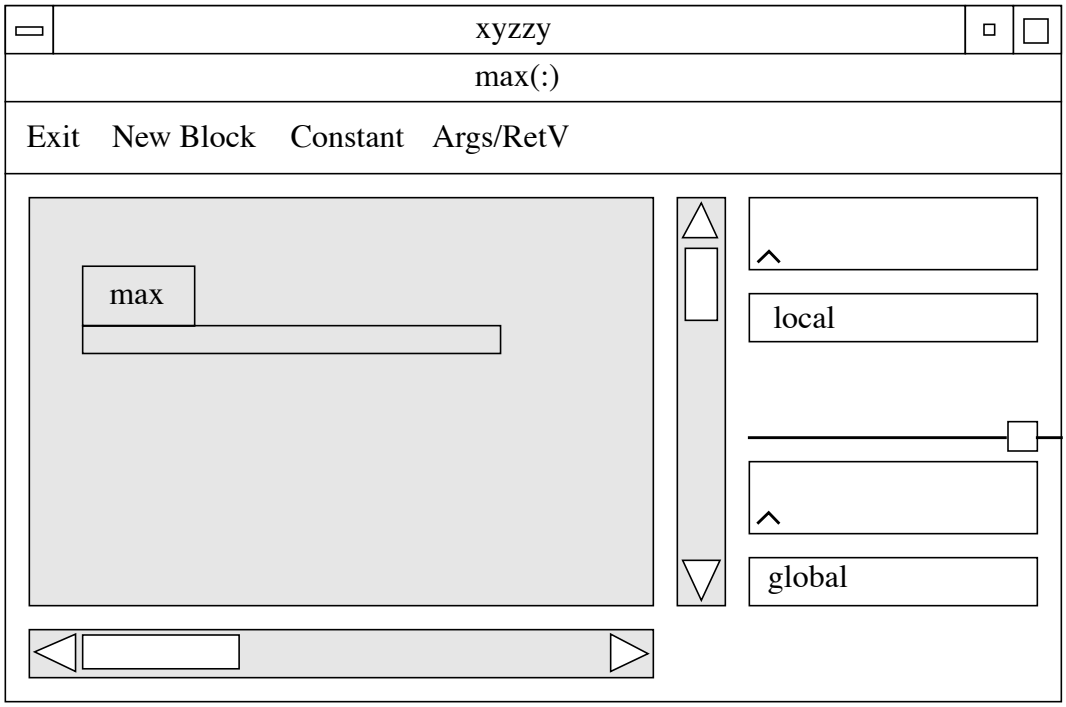



**figure 4.2: Function Editor**

The function editor is where the code for each function is constructed. At the top of the window is the prototype for the function being edited, in this case “max”. It currently has no arguments and no return values. Just below the prototype is the menubar. The scrolled window on the left half of the window is the drawing area where the code is displayed; On the right half is the variables display area.

There are four simple options on the menubar. **Exit** quits just this function editor, and removes calls to the function that are in code anywhere. **New Block** creates a new block in the drawing area for code to go into. A block is either a branch of an if statement or the body of a while loop. **Constant** creates a constant item for the selected argument of a function. **Constant** determine the type of the selected argument and prompts the user for the correct type of constant expression. **Args/RetV** is a pulldown menu that has options to create and delete both arguments and return values of the function. It is not a very good method to handle the requests, but it works.

### 4.2.1 Variable Display Area

The variable display area shows all the local variables of the function and all the global variables of the program. The two kinds of variables are divided by a sash that can be moved up and down. At the top of both the global and local variables is a text-edit window. When the return key is hit in this window, a new variable is created with the name gotten from the text-edit window. If the variable is global, the variable will appear in all the function editors simultaneously.

### 4.2.2 Drawing Area

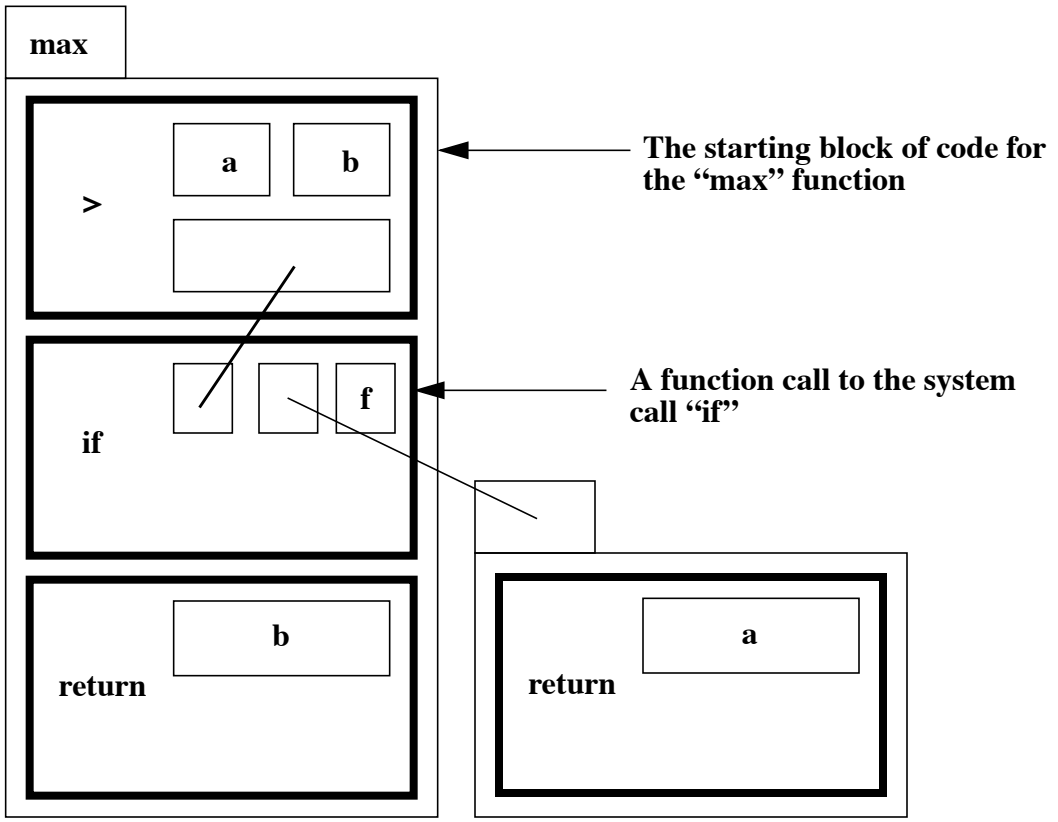



**figure 4.3: Example contents of the drawing area**

As mentioned before, the drawing area is where the code is displayed and edited. When the function editor is created the only thing in the drawing area is an empty block with the name of the function. This is the point at which execution begins when the function is called. The function ends when it encounters a return function.

To add a function to a block, the user must first select the block. This is done by clicking the left mouse button in the block, which should cause the block to be highlighted. When the user presses the right mouse button in the drawing area, a list of functions appears. Which ever function the user selects, a prototype of that function will appear in the block.

When the block is selected, the function will be added to the top of the block, but when a function is selected, the new function will be added after the function selected.

On the left side of a function call is the name of the function being called. On the top right is originally a grayed list of the arguments of the function. On the lower right is a grayed list of return values. The graying indicates that the argument or return value has not been filled yet with the actual variable that will be referenced. The names of the arguments and return values are displayed so the programmer understands what is supposed to go in that spot.

There are many ways to fill an argument or return value. The simplest for an argument is selecting the argument and pressing the constant button on the menubar. Depending on the type of the argument, the programmer will be prompted to enter a string, a number, or either “true” or “false”. The argument will be filled in with a non-grayed constant.

Another way to fill a reference is by selecting the reference and pressing one of the variable buttons in the variable display. The reference will be filled in with a non-grayed reference to the variable. This can be seen in figure 4.3 where “a” and “b” are referenced multiple times.

A third method to filling a variable reference is with a temporary variable. To create a temporary variable, the left mouse button is depressed in the return value that is the source of the value for the variable. The mouse is then released in the argument that will receive the temporary value. The temporary variable will be represented by a line drawn between the return value and the argument. Temporary variable are extremely useful when doing math or logic, such as in the max function where the value of the greater than is used in the if statement.

The final method for setting an argument is used only with if and while statements. The branches of an if statement and the body of a while statement are considered arguments in XYZZY. To create this reference to a block, the programmer must depress the left mouse button over the argument and release it over the tag of a block. Blocks may only be referenced once, and a named block cannot be referenced. A reference of this type is drawn as a line very similar to the temporary variable.

There is also one more method for modifying arguments and return values. If a filled reference is selected and the delete key is pressed, the reference will become unfilled and return to its original grayed state.

## 4.3 Example of Function Construction

1. Press the **function editor** button on the manager window to launch a new function editor for the function, since every function needs its own function editor.

- A dialog box will appear asking for the name of the function.

2. Type the name of the function, “max”, and either hit return or press the **Ok** button.

- A function editor window will appear with “max(:)” in the prototype area. This shows the name of the function, “max”, and no arguments before the colon, and no return values after the colon.
- A button with the name “max” will appear in the manager window. Pressing this button will bring the function editor for max to the front of the screen.

3. Select **Add Argument** on the **Args/RetV** menu.

- A dialog box will appear asking for the name of the argument.

4. Type “a” and either hit return or press the **Ok** button.

- The text in the prototype area will change to “max(a:)”, which shows the function max has an argument a, assumed to be integer.
- A button named “a” will appear in the local variable window. This argument can now be used as a variable in the function.

5. Repeat steps 3-4 using the name “b”

- The prototype now reads “max(b, a:)”.
- Another button named “b” has appeared in the local variables.

6. Select **Add Return Value** from the **Args/RetV** menu

- A dialog box will appear prompting for the name of the return value

7. Type “z” into the dialog box and hit return.

- The prototype now contains “max(b, a:z)” showing two arguments before the colon, and one return value after the colon.
- Note that no variable appears in the local variables. The return values are not variables, but do modify the return function, which will be shown later.

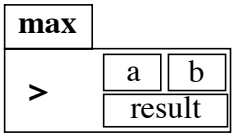
If the wrong variable name was typed in for either an argument or a return value, it can be removed by typing the incorrect name into the dialog box brought up by selecting **Remove Argument** or **Remove Return Value** from the **Args/RetV** menu.

8. Move the mouse pointer over the drawing area.

- The max block should be in the highlighted color, if it is not click on the tag of the block (the part with the word “max” in it.)

9. Press the right mouse button to bring up a pop-up menu and select “>”.

- This adds the function call to the function “>” to the max block.
- The “>” function call is selected.
- The argument and variable boxes of the function call are grayed-out.



10.Click on the variable reference box in the function call labeled “a”.

- The box is now highlighted.

11.Press the variable button “a” in the local variable window.

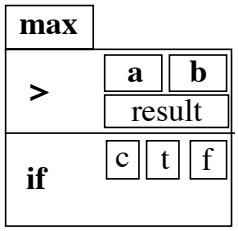
- variable reference box now contains the label “a” and is no longer grayed-out.
- The variable reference box named “b” is now highlighted.

12.Press the variable button “b” in the local variable window.

- variable reference box now contains the label “b” and is no longer grayed-out.
- The variable reference box named “result” is now highlighted.

13.Select **if** on the pulldown menu of functions.

- The function call “if” appears below the function call “>”.
- The function call “if” is highlighted.




14.Press down the mouse button while the mouse is in the “result” box of “>”

15.Release the mouse button when the mouse is in the “test” box of “if”.

- Both the “result” and “test” block are no longer grayed-out, and there is a line drawn between the two boxes. This represents the temporary variable being set by “>” and being used by the “if” function call.

16.Select **return** on the pulldown menu of functions.

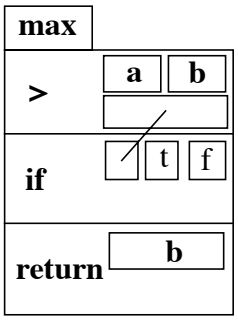
- The function “return” appears below the “if” function call. The arguments to “return” are the same as the return values of the function being edited, in this case just one named “z”.

17.Click on the variable reference box “z”.

- “z” is now highlighted.

18.Press the variable button “b” in the local variable window.

- The variable box now contains the label “b” and is no longer grayed-out.




19. Press the **New Block** button on the menubar.

- A new block with no name (an empty tag) has appeared.

20. Press down the mouse button while the mouse is in the “true” box of “if”.

21. Release the mouse button when the mouse is in the tag of the unnamed block.

- The variable reference box is no longer grayed out, and a line connects it to the block. This shows the block is now the “true” branch of the if statement.

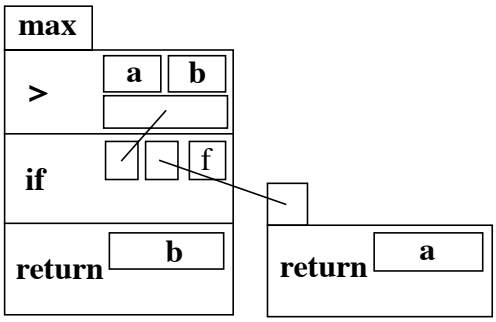
22. If the block is not highlighted, click in the tag to select it.

23. Select **return** on the pulldown menu of functions.

- A return function call has appeared in the new block.

24. Select the variable reference block “z” in the “return” function call.

25. Press the variable button “a” in the local variables window.




The function is now finished. It translates to if a is greater than b, return a. If the if does not exit the function, return b. It could also be done using another block for the false branch of the if statement which returned b.

# 5.0 How XYZZY Works

There may be many interesting features of the Motif windowing system used by XYZZY but these are incidental. This section will concentrate on the data structures used to store the program in memory.

The most important data structure is the one to store the code. This used objects that combined both the logical and visual aspects of the code displayed. It also had to implement most of the features of a windowing system because the drawing area widget provided by Motif did very little processing to events.

The windows that were used to display the list of variables are a very interesting piece of XYZZY. The global variables had to be displayed in every function editor and have update occur to all the windows at once. The variables themselves were complicated because they could be arguments and have references to them, as well as have windows to them in every function editor.

The next piece in this section will describe the way functions are store in memory, and how they are accessed.

## 5.1 Visual Objects, Code Storage, and Code Manipulation.

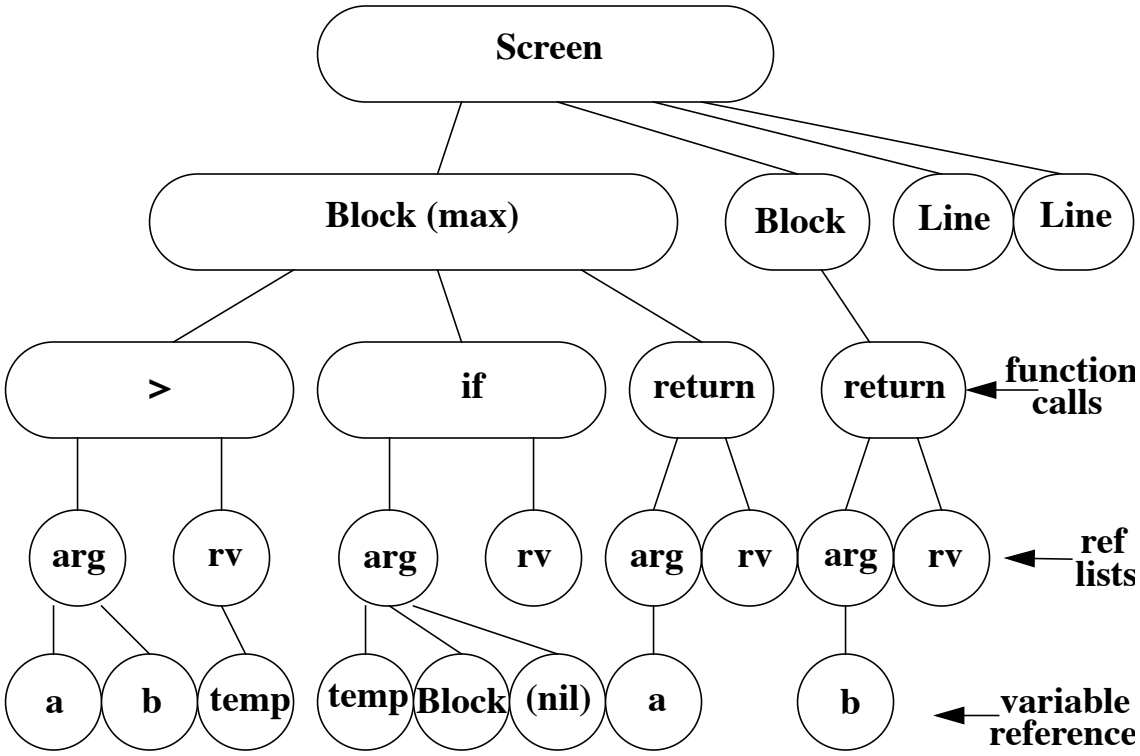



**figure 5.1: Parse Tree of Visual Object in Max Function**

The code of XYZZY is stored in memory in the same objects that handle the graphical representation. All references to the visual objects are done through the root visual object, called *screen*. There is one exception to this rule; the function editor does have a pointer to the name block of the function in the drawing area.

The *screen* object is the root of the visual object tree. Underneath it are the *lines* which are used to represent the temporary variables and references to block. *Blocks* are also children of the *screen* object. *Function calls* are children of the *blocks* and contain two *variable reference lists*, one for arguments and one for return values. Lastly, *variables references* are children of the *lists*.

The *screen* object is very different from the other visual objects. When it is redraw, it draws the background color. When a refresh of an area occurs, the *screen* draws the background color to erase anything that was there, and the rest of the objects redraw themselves in that area.

Ideally, *lines* would be children of the variables references they represent, but this cannot be done because of the refresh mechanism. Because only a part of the screen may be need to be refreshed, the refresh mechanism will only refresh that part exposed. The way this is achieved is by traversing the tree and stopping the decent if an object is not in the refreshed area. In order for this to work, every child must be contained within the area of its parent. The only way lines can be refreshed then is for them to be children of the *screen* object.

The *block* object is the most complex object because it is the only one that resizes itself. When a function call is added to a block, it must resize itself, move all the function calls after the one to be inserted, and then insert the function call into the correct position. The block also inherits the virtual object called value, which allows it to be referenced in if and while statements.

The *function call* is interesting for the fact that it barely contains anything at all. It has a pointer to an object called a function prototype, and to the two *variable reference lists*. The function prototype object contains the information on a function, its name and its list of arguments.

The *variable reference list* may be one of the visual objects, but it not drawn on the screen. It is used solely to manage the *variable reference* objects in the *function call*.

The *variable reference* contains a pointer to two things, its prototype argument and to the reference, if it is filled. The prototype argument gives the reference its type and its name is used for unfilled references to tell the programmer what it is used for. The actual reference is a pointer to a value object. The value object is a virtual object which is inherited by variables, constants, temporary variables and block.

## 5.2 The Variable Display

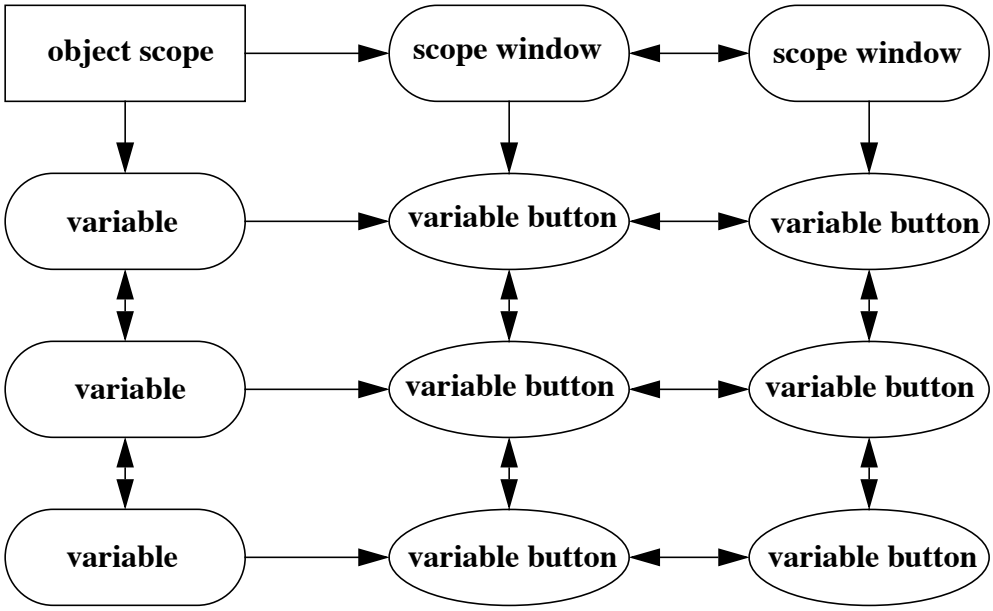



**figure 5.1: Data storage for variable display**

Variable visualization is difficult because there can be more than one window displaying a variable scope at one time. Changes made to the scope must be done to all windows displaying the scope at the same time.

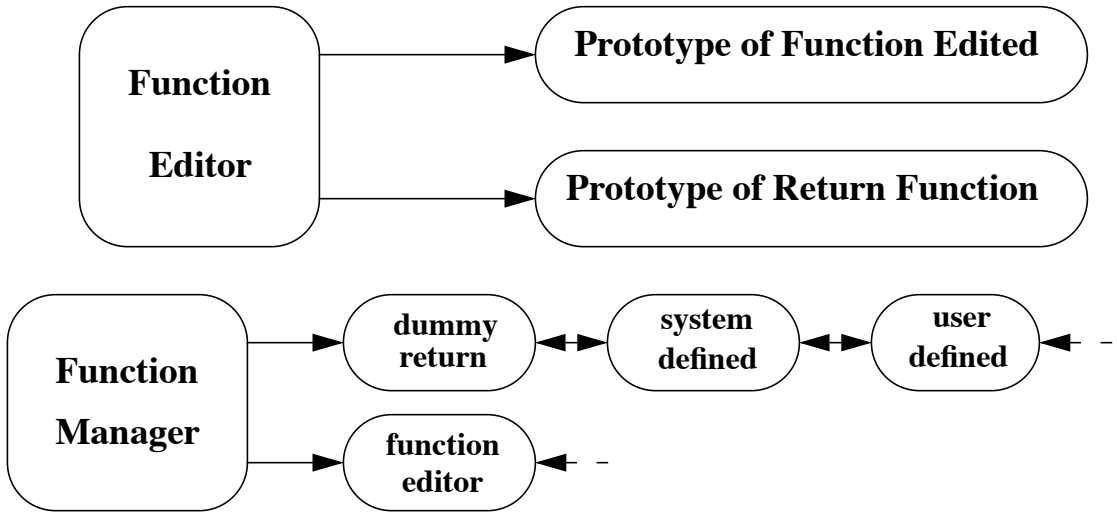
The *object scope* is the object which represents the scope. To this object new variables are added and new windows to display the scope are added. Both the *variable* objects and the *scope window* objects are kept in separate doubly-linked lists.

The *variable* is the object which represents a variable in the scope. All objects derived from the *variable* object can be inserted into this list, such as the *arg_rv* object which is used to represent arguments and return values to functions. It is derived from the *value* object so that it can be referenced by variable references. When the button representing the variable is pressed, a pointer is set to this object.

The *scope window* is the object which holds the window that displays the scope. All buttons representing variables must use this object to display the button in the window.

The *variable button* object is a very interesting object, which is the heart of this data structure. Each *variable button* is in two doubly-linked lists which form a grid between the *variable* objects and the *scope window* objects in the object scope. This way the buttons of a variable can be accessed in order, as can all the buttons in a window.

## 5.3 Function Storage




The function storage mechanism had some interesting problems. First, all the functions in the program had to be listed on a menu in every function editor. Second, system defined functions would no have editors for them. Third, the return function was different for every function editor since it depended on the return values of the function.

The first step was to create a *function manager* object. This object has a doubly-linked list of *function editors* for which it is maintaining menus, and a doubly linked-list of all *function prototype* objects for all the functions in the program. Whenever the list of *function prototypes* changes, the function manager object corrects all the menus of the *function editors* in its list. The function manager is initialized with all the system defined functions and a dummy return function.

The *function editor* contains two *function prototype* objects. The first is the function prototype for the function the editor edits. The second is the function prototype for the return function, since this is different for every function editor. The function the editor edits and the function editor itself are added to the *function manager's* lists when the function editor is created, but the return function is not.

When a function is selected from a menu, the *function manager* handles the request. If the function is not the return function, a pointer to the correct *function prototype* object in the list is returned. If the function selected is the return function, the *function manager* finds the *function editor* which contained the menu that requested the return function, and returns a pointer to the return function of that *function editor*.

# 6.0 Conclusion

After using XYZZY for a good bit of time, I have seen some of its benefits and some of weaknesses. The block structure of the language provides causes many likes and dislikes.

The block structure allows XYZZY to return multiple arguments from a function, which is a big benefit in a language where pointers are unavailable. In fact it is necessary in order to write a function which will swap two numbers. The fact that it prevents more than one function call being on the same line is not that important because of the use of temporary variables.

The major problem with the block structure is that logic and arithmetic constructs become very awkward to read and can be confusing. This is especially true for while statements, where the conditions for the loop are above the while statement. So functions before the while statement need to be executed after every loop of the while block. This is a confusing way to print the while condition.

XYZZY is also slower to program in, but this might be compensated by the fact that no syntax errors can be made. A major part of the slow down is that the programmer in XYZZY must swap between the keyboard and the mouse. This could be solved by allowing the user to change the selection using the keyboard and by shortcuts for the menus.

The goal of preventing errors was achieved, but very little progress was made in detecting errors and providing information quickly. A good next step for this project would be to include the interpreter into the program and provide semantic checking. Another item would be to find a better way to provide arithmetic and logic statements, probably by magnifying the area when it is selected for editing. Lastly, a good method for handling pointers would be paramount for movement toward being a useful language.

There are many places in XYZZY that could be changed to enlarge the project. Expanding the project for multiple programmers could be done easily using mulitple X window displays with one program, but a more interesting variant would be to use multiple programming environments to manipulatint a common database which held the code.

# 7.0 Bibliography


[1] W. Citrin, “Requirements for Graphical Front Ends for Visuala Languages”, *Proceedings of the IEEE International Symposium on Visual Languages*, 5/93, 142-150.

[2] L. Coulmann, “General Requirements for a Program Visualization Tool”, *Proceedings of the IEEE International Symposium on Visual Languages, 5/93, 37-41*.

[3] L. Ford and D. Tallis, “Interacting VIsual Abstractions of Programs”, *Proceedings of the IEEE International Symposium on Visual Languages*, 5/93, 93-97.

[4] E. Golin and T. Magliery, “A Compiler Generator for Visual Languages”, *Proceedings of the IEEE International Symposium on Visual Languages*, 5/93, 314-321.

[5] B. Haberland, J. Poswig, and C. Moraga, “On the Way to Intelligent Query for Functions in Visual Languages”, *Proceedings of the IEEE International Symposium on Visual Languages*, 5/93, 365-367.

[6] C. Holt “Control Flow in a Dataflow Language”, *Proceedings of the IEEE International Symposium on Visual Languages*, 5/93, 384-385.

[7] K. Swenson, “A Visual Language to Describe Collaborative Work”, *Proceedings of the IEEE International Symposium on Visual Languages*, 5/93, 298-303.

[8] G. Wirtz, “A Visual Approach for Developing, Understanding and Analyzing Parallel Programs”, *Proceedings of the IEEE International Symposium on Visual Languages*, 5/93, 261-266.