10-14-1988

A Visual Programming Language for Data Flow Systems

Timothy J. Wilson

Rochester Institute of Technology
School of Computer Science and Technology

A Visual Programming Language for Data Flow Systems

by
Timothy J. Wilson

A thesis submitted to
The Faculty of the School of Computer Science and Technology
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Dr. Peter Lutz 10-18-88

Dr. Andrew Kitchen 10-18-88

Dr. Peter Anderson 10-18-88

October 14, 1988

Abstract

The concept of visual programming languages is described and some necessary terms are defined. The value of visual languages is presented and a number of different visual languages are described. Various issues, such as user interface design, are discussed.

As an example of a visual programming language, a graphical data flow programming environment is developed for the Macintosh workstation which functions as a preprocessor to a data flow simulator developed at RIT. Examples are presented demonstrating the use of the language environment. Issues related to the development of the programming environment are described and conclusions regarding the development of visual programming languages in general are presented.

CR Categories and Subject Descriptors:

D.1.1 [Programming Techniques]: Applicative (Functional) Programming;

D.2.1 [Software Engineering]: Requirements / Specifications - Languages;

D.2.2 [Software Engineering]: Tools and Techniques - Structured Programming; User Interfaces;

D.2.6 [Software Engineering]: Programming Environments;

D.3.2 [Programming Languages]: Language Classifications - Data Flow Languages;

I.3.6 [Computer Graphics]: Methodology and Techniques - Interaction Techniques; Languages;

I.3.m [Computer Graphics]: Miscellaneous.

General Terms: Data Flow, Graphical Programming, Visual Languages, Programming Languages

Acknowledgments

I would like to acknowledge the generosity of the Eastman Kodak Company for funding my Masters Degree program and my supervisors, Mr. Jack O'Grady, Mr. John Strickland, Mr. Donald Barnes and Mr. Stephen Hinman for supporting my efforts.

This thesis is dedicated to my wife, Elizabeth. It would not have been possible without her constant support and encouragement.

Table of Contents

Table of Figures

Chapter 1

Overview

1.1 Introduction

As computer workstations have become more powerful and more widely available, as well as less expensive, there has been a proliferation of applications that allow the end user more "programmability" in the application. Rather than developing specific applications through the use of expensive resources such as professional programmers, powerful applications allow the specification of a problem in a domain understood by the user.

Conventional programming techniques involve specifying the problem and its method of solution as a collection of primitive operations applied in a sequential, linear fashion. The translation of the problem domain to the computer's execution domain has been the role of the programmer.

To provide a programmable environment that is easily understood by the user, many techniques have been developed to provide a method of presenting the problem and its method of solution to the computer system (i.e. programming). One class of methods is visual programming.

1.2 Definitions

For the purposes of this thesis, visual programming is defined as the application of a system or environment that allows that user to specify a program in a multidimensional fashion, utilizing a spatial arraignment of program primitives and constructs as an integral part of the language semantics. Conventional programming with

textual languages is a one-dimensional, linear process that mimics the classical sequential execution model of computers.

A number of other terms are related to (or confused with) visual programming. The terms graphical language and graphical programming are used to designate languages or paradigms that use graphics to a significant degree. All of the languages discussed in this thesis are examples of graphical languages.

Program visualization is the use of graphical techniques to illustrate some features of a computer program or system but not its specification. They are applied after a program has been specified or in conjunction with the programming specification. Program visualization techniques can be presented in a taxonomy that is divided into code representation versus data representation and static illustration versus dynamic illustration. [Myers 1986] Static illustration presents a "snapshot" of some aspect of the execution environment while dynamic illustration provides a real-time animation of the execution environment.

Another term used in conjunction with visual programming is Programming by Example. Some systems of this type require that the user demonstrate an algorithm through a number of examples and then the system infers a general program structure. This has been an area of Artificial Intelligence research. Other systems require the user to completely specify the approach without any inference on the part of the system, although the user supplies a specify example. This approach has been called as Programming with Examples. Programming with Examples can be described as "Do What I Did" while the inferential Programming by Example systems should "Do What I Mean". As with program visualization systems, graphical techniques are sometimes utilized in Programming by Example systems but they are not intrinsic to the specification of the program as is the case with visual programming.

A user interface management system (UIMS) is a framework for managing the various aspects of a computing environment. It involves the application of semantic rules through hardware devices. These rules may be very simple, such as typing a

keyboard entry in response to a specific prompt or they may be more complex, such as picking on object from a palette displayed in one window of an output device using a mouse and then "placing" the object in another window or area. The user interface design encompasses the application of input / output (I/O) devices such as mice, tactile response devices, light pens, graphics tablets and bit-mapped graphics screens as well as many different methods of information presentation, such as scrolling windows, menus, palettes and dialog boxes.

In the design of any programming language, it is the application of a specific grammar to a set of symbols that defines the language and its problem domain.

A visual language has its grammar defined by the spatial orientation of the graphic elements that form the language symbols. A user interface defines the set of operations on the symbols for their manipulation and management. In the development of a visual language, the language implementor is now faced with the the definition of all three elements (grammar, symbols, and interface), rather than just the grammar and symbols of a conventional textual language.

1.3 The Value of Visual Languages

This thesis reviews the issues in the development and implementation of visual programming languages. The basic issues of text versus graphics are discussed in the context of programming languages and the power of visual language systems are shown. Important considerations in visual languages are shown to be the user interface, the representation of the problem domain, and the operational features and characteristics of the programming system.

Programming paradigms are also described with respect to visual programming languages. These paradigms include classical control flow programming and data flow programming. The advantages of the data flow paradigm for visual programming languages is discussed.

For a background in the development of visual programming languages, a number of different systems are described. They are presented in an order that roughly represents the important developments of visual programming languages and systems. Some relevant issues are discussed with respect to each language. It is shown that visual programming languages started as a method to visualize control flow programming and then evolved into systems that move to specific problem domains that take advantage of the users familiarity with the abstract objects of the problem domain.

The visual programming language developed for this thesis, FlowGraph, is then presented through the development of a simple example program. The specific issues that were encountered in the development of FlowGraph are then described.

Based on the background discussion and the issues encountered in the development of FlowGraph, a number of conclusions about visual programming languages are presented.

Chapter 2

Issues in Visual Programming

2.1 Introduction

With the advent of fast, inexpensive bit-mapped graphics workstations and the increase in so-called "non-technical" computer users, research is proceeding in the development of systems that present a computer system paradigm which is easier for the end user to understand and control. Much of this work is in the field of User Interface Management Systems, which are usually used to enhance the operating system software. These systems use graphical representations of objects to present the machine state information or provide tools for applications to generate consistent, easy-to-use user interfaces. As the user interface research matures and the workstations become more available, the research is moving toward applications that provide powerful methods for the user to utilize the computer to solve problems in a fashion more closely related to the problem domain.

This section describes approaches used to develop computer programming environments using graphical objects to represent the abstract entities that are manipulated in a computer program. The range of approaches to this problem is fairly broad, with the utilization of graphics being used in different degrees and for different reasons.

[Halbert 1984] has shown that non-programmers can create fairly complex programs with visual languages. Visual languages provide the tools to express a problem in terms well understood by the user. It becomes the burden of the visual programming system to manage the translation of the problem from its natural domain to the domain of the execution environment rather than placing that burden on the user.

2.2 The Application of Graphics

Historically, visual programming languages can be traced back to two early developments: SKETCHPAD [Sutherland 1963], an early interactive graphics workstation developed at MIT and a project on drawing flowcharts using a computer [Knuth 1963] to assist in program development. Both of these types of systems represent the two early developments that have paved the way for modern graphical programming languages, with SKETCHPAD providing the interactive graphics and the Knuth system using the computer to develop some graphical output for providing program visualization.

2.2.1 The Use of Graphics vs. Text

The stress on graphics versus text is derived from the manner in which humans assimilate information. It has been shown that information represented in pictures can be more rapidly and completely understood. The human eye rapidly scans an entire scene before it focuses on the area of interest. As the brain attempts to pick out those areas of the scene which are important, the whole scene is repeatedly rescanned and the key points of interest are registered. When this process is applied to text, the brain tends to scan for illustrations, headings and highlights before it begins the process of rigorously following the linear flow of the text. By allowing the eye / brain system to function on graphical data, the brains preferred pattern of scene scanning can be used to gather and assimilate the information quite rapidly.

2.2.2 Naming and Qualities

Another attribute of graphical information is its ability to portray information and properties without naming. That is, objects in textual programs must be identified by names. These names may be misleading. Graphical objects can be presented in a fashion that presents pertinent information in a less ambiguous fashion.

2.2.3 Animation

Graphical programs and representations can also be used to present an execution model through the use of animation. Animation techniques can be used to show much of the information about the program's execution state(s).

2.3 User Interface Considerations

One of the major concerns in software development is programmer productivity. Many people are studying the methods by which user and programmer productivity can be increased. The design and development of user interface systems has the goal of providing a more complete picture of the current state of a computing system to the user, beyond that which is provided by many current operating systems, such as the classic

%

from the UNIX c-shell operating system.

There has been a large number of studies on the appropriate design of user interfaces for computing. These range from the ergonomics of computer hardware design to the appearance and interactive approaches of system software. The major consideration is that user interaction techniques must be designed to support three types of basic human processes: cognition, perception and motor activity. A comprehensive study in this area is provided by [Foley, Wallace and Chan 1984] and [Foley and Wallace 1974].

These considerations are also important in the area of programmer support. That is the fundamental reason behind graphical programming and visual language systems. Some of the systems described above were developed solely to increase programmer productivity through a better understanding of the algorithms and program flow in a

program (see especially the PECAN system [Reiss 1984b]). Others attempt to provide a better fundamental model for programming, using a model with an interactive visual representation (i.e. PROGRAPH [Matwin and Pietrzkowski 1985], PiP [Raeder 1985 and Raeder 1984], Programming by Rehearsal [Finzer and Gould 1984]), ThinkerToy [Gutfreund 1987] and ARK [Smith 1986]. The goal of providing a visual, graphical environment that better represents the problem domain or model to the user is a common goal of these systems and of the system that was developed for this thesis. (Additional references to user interface design considerations are included in the Bibliography and the Further Reading section).

2.4 The Problem Domain and the Completeness of Domain State Presentation

Visual languages provide a method of presenting a large amount of information at one time. By keying graphics to the users problem domain, information is more rapidly communicated to the user. Tying the graphics to the problem domain is both a benefit and a limitation in visual languages. The benefit is the amount of information that can be presented to the user. The user is expected to understand this information because of the users knowledge of the properties of the objects that the graphics represent. The limitation is that if the user does not have a knowledge of the objects represented in the program, the information presented is not as useful. The concentration on the problem domain may also limit the applicability of the language. A tradeoff in visual languages is the level of complexity of the properties that the objects in the language represent. As the problem domain of the language is narrowed, the amount of property information that an object can convey can increase.

2.5 Operational Considerations

Most of the research presented here on visual programming systems has some common features:

• The program editor (graphic or text) is syntax-directed to eliminate syntax errors in the code while it is being entered;

• The user manipulates the graphics while the program is being created or edited and then the system animates the graphics during execution. This should lead to a better understanding by the programmer of his algorithm;

• In many cases the programs are incrementally compiled or interpreted so that they could be executed at any time in the development [Reiss 1984a, Schwartz, Delisle and Begwani 1984]. Many systems allowed the execution of unfinished or incomplete programs, so that debugging the algorithm could begin before the program was complete.

All of these points are aimed at the goal of making the system responsive to the user and giving the user the maximum flexibility in the approaches to solving his problems.

2.6 Some Useful Programming Paradigms

It is important to briefly discuss some of the different programming paradigms that are being studied. Control flow programming and data flow programming are discussed below. It is important to note which programming paradigms are more applicable to visual languages systems.

2.6.1 Control Flow Programming

Control flow systems represent the conventional computer processors of today. Control flow programs are stored and executed in a sequential manner. No instruction can execute until all previous instructions have executed. Regardless of the algorithm, the execution process follows the sequential ordering of the instructions.

The principles behind todays computers generally reflect those ideas developed by von Neuman. Von Neuman systems are sequential in nature, with separate central processing units and memory. These components are connected by a single channel, in which only one word or address may pass at one time. This system provides a serious limit to the expansion of computing power [Backus 1978].

Programming languages have been developed around these machine architecture concepts and so provide little help in the advancement of computing capability.

2.6.2 Data Flow Programming

In response to this dilemma, a number of different computer architectures have been proposed. One area of research, pioneered by Jack Dennis at MIT [Dennis 1975], is the development of data flow systems. These data flow (or data driven) systems provide an architecture that supports parallel processing quite naturally. It also eliminates sequential processing, by allowing any operations to occur whenever their required operands are available. These operands are passed as tokens, rather than being maintained in static memory locations, so the processor / memory "bottleneck" is eliminated [Backus 1978].

As an example of the difference between data flow and control flow programs, consider the following sequence of instructions, as represented in a control flow system:

```
  P = X + Y
  Q = P / Y
  R = X * P
  S = R - Q
  T = R * P
RESULT = S/T
```

In parallel processing data flow systems, these instructions can be processed more efficiently using the data dependencies rather that the instruction sequence. For this example, the instruction sequence is as follows:

```
          P = X + Y
Q = P / Y   &   R = X * P
S = R - Q   &   T = R * P
      RESULT = S/T
```

The parallel nature of the algorithm is utilized by the data flow system.

The principles are being applied in hardware through a number of different multi-processor, data flow computer designs. Some of the major systems include the M.I.T. Data Flow Computer, the Manchester Data Flow Computer, the Irvine Data Flow Machine, the Texas Instruments Distributed Data Processor, the Utah Data-Driven Machine, the LAU system and the Newcastle Data-Control Flow Computer. These data flow architectures are described in more detail in [Treleaven, Brownbridge and Hopkins 1982], [Torsone 1985], [Lawson 1986] and [Benjamin 1988].

Data flow languages provide the tools to directly support these system architectures. The most natural representation is that of the directed graph. This makes data flow languages attractive to implement with visual programming systems, such as the system developed here.

2.7 A Review of Some Existing Visual Language Systems

Different visual language systems have be reviewed to identify the critical issues and insights that they provide. They represent a cross section of implementation techniques, from the utilization of some visual or graphical techniques to languages that are highly visual to support their problem domain.

2.7.1 PECAN

The PECAN system [Reiss 1984a, 1984b] essentially provides different views of a program being developed in a conventional, control flow style. These views include a syntax-directed text editor, a view based on Nassi-Schneiderman flow charts, and a

view of the program declarations. These provide views of the program syntax. Views of the symbol table being generated, the program's defined data types, expression trees and flow of control charts provide semantic views of the program. The PECAN system developers are integrating a system of execution views based on their BALSA algorithm animation system [Brown, Meyrowitz and van Dam 1983]. This program visualization system was developed at Brown University.

PECAN is utilizes the techniques described above to support conventional programming techniques. Its use of multiple views of code and data, syntax directed editing and incremental compilation are the highlights of the system. However, it is not able to utilize more advanced graphical or visual techniques because it is so heavy tied to conventional, control flow programming languages.

2.7.2 Think Pad

The Think Pad system [Rubin, Collin and Reiss 1985] uses a programming-by-example system which allows the user to create graphical representations of data structures. These graphical data structure representations can then be combined to form functions, which are then combined to form programs. Although the user is manipulating graphics, he must also provide some textual information about the graphic object. As the programmer manipulates the various data structures that were created, the function semantics are defined. As each object is created, ThinkPad creates a Prolog program. This is the final output of the Think Pad system. Current work being done includes the inverse mapping of Prolog programs to Think Pad graphics representations. This would also allow the trace function in Prolog to provide input for a form of algorithm animation in Think Pad. It was also developed at Brown University.

Although Think Pad utilizes graphics in its interface to display the data structures and their combination, its is still bound by the textual representation of the data

structures. The graphical interface merely provides a better user interface for manipulating the text that represents the data structures.

2.7.3 OMEGA

The OMEGA System at Berkeley [Powell and Linton 1984] is a system that allows users to manipulate graphical objects and structures. They use icons to represent a abstract programming concept (such as sorting a database, defining a variable, a variable itself, a control structure, a data type). The graphical icons simply represent objects whose complete specification is still given in a textual form. OMEGA allows manipulation of the defined icons, but at a lower level, it still requires text for the detailed specifications of the program which makes it useful for program visualization rather than visual programming.

OMEGA actually is iconic in nature, with abstract representations of objects that can be picked by the user in a syntax-free manner. However, OMEGA is still heavily dependant on text and essentially uses the icons to represent complex objects. It mixes text and icons in a manner such that the icons become "words" that extend the vocabulary of the language. Once again, the graphics are used to enhance the user interface of a textual system, rather than a visual system.

2.7.4 GAL

The GAL system [Albizuri - Romero 1984a, 1984b] was developed at the University of Sussex and uses graphics in a manner similar to PECAN, although not as extensively as PECAN. Basically, it provides a method to imbed textual code into Nassi-Shneiderman charts. This is really not visual programming but advanced graphical support for textual programming systems.

2.7.5 PICT

The PICT system is a graphical approach to traditional von Neuman languages. The PICT system was developed as a visual language at the University of Washington in Seattle, WA. [Glinert and Tanimoto 1984]. The PICT system uses no text in the development of programs. The program's flow of control and its data are totally represented by graphical objects. The user interacts with the system through a joystick which controls a cursor on a color graphics CRT. The user creates colored icons that represent operations, data structures and variables. Directed, colored paths represent control structures which direct the program flow. The user's actions are managed by a syntax-directed editor, which eliminates syntactic inconsistencies during program development and editing. At run time, the program's graphics are animated to show its operation, which gives it the ability to support program visualization. Thus the program is represented both in development and execution by a consistent graphical structure to the user.

The PICT system is useful for doing some simple computer programming. However, it has some drawbacks. Each module (program or subprogram) has available only four variables (represented by four colors: red, green, blue and orange) which are restricted in type to being six-digit, non-negative integers. The system also has only 16 primitive operations, of which two are for system control, five are boolean operations and the remaining nine are primitive operations on variables.

2.7.6 Programming by Rehearsal

Programming by Rehearsal [Finzer and Gould 1984] was developed to aid nonprogrammers in creating educational software. This system defines a theater metaphor where a director can audition performers with various attributes, place required performers on a stage and define how they respond to cues. Programming by Rehearsal was programmed in Smalltalk-80 and is basically another metaphor for object-oriented programming. In this case, the objects (performers) are grouped by their

attributes into "troupes", i.e. basic data types (text, integers) are the BasicTroupe, pictures are the GraphicsTroupe, clock forms are the TimeTroupe, etc. The director uses a mouse to move the performers to the appropriate positions on the stage or in the wings, where they are hidden from the programs end user. The director defines the interaction of the performers by the way that they respond to cues from other performers, on- and off-stage. In this system, textual commands define both the actions and interactions of the various objects while their graphical representation can be manipulated to provide the appropriate "view" of the objects, that is, their presentation to the user. Programming By Rehearsal was developed at the Xerox - Palo Alto Research Center (PARC).

The use of the theater metaphor and the object oriented, message passing nature of the program are the key elements of Programming by Rehearsal.

2.7.7 PiP

An important visual programming language is the Programming in Pictures (PiP) system, developed at the University of Southern California in 1984 [Raeder 1984, Raeder 1985]. In this system, pictures are used to represent the operations on input data. The user creates these pictures in a freehand manner that then represent the operations. The data pictures are then combined by pointing at them and then using operators to combine them into functions. Control structures are available to combine small functions into larger ones. The system also uses Backus's functional programming model.

Because the PiP system allows users to create their own pictures, the graphics may more adequately represent to the user his problem domain. However, the metaphors suggested by the pictures can be misleading, since the level of completeness of meaning is also be defined by the user.

2.7.8 FGL and GPL - Data Flow Languages

Two graphical programming languages that were designed in conjunction the University of Utah data flow machine project, DDM 1, are FGL (Function Graph Language) [Keller and Yen 1981] and GPL (Graphical Programming Language) [Davis and Lowder 1981]. These systems are two approaches to the development of a visual programming language are based on the data flow model. Each system takes a slightly different approach to the data flow implementation [Davis and Keller 1982].

FGL and GPL are graphical languages very similar to the system proposed here. The papers don't present sufficient information about the programming environment and implementation to contrast it with this system. Both allow the creation and manipulation of a data flow graph as the primary means of programming. (See also [Maguire 1983]).

2.7.9 PROGRAPH

One of the most interesting visual programming languages is PROGRAPH, developed at the University of Ottawa [Matwin and Pietrzkowski 1985]. The language model for PROGRAPH is a combination of a number of functional programming languages; namely, LISP [McCarthy 1965], APL [Iverson 1962], FP [Backus 1978], and GPL (GPL is described above).

PROGRAPH contains many important features:

- It is essentially functional in nature.
- It operates on data of arbitrary complexity.
- It does not require the declaration of variables.
- It provides a database structure that is compatible with PROLOG.
- It is suitable for use with parallel processing systems by providing some important functions, like APPLY TO ALL.

• The graphical representation is also the basis of run time animation for program visualization.

With respect to its graphical programming aspects, PROGRAPH allows the user to manipulate a set of primitive operations that are assembled into a definition. Each of the user's definitions can then be manipulated just like a primitive operation, to be combined into further definitions.

PROGRAPH's importance stems not only from its use of a graphical programming and execution environment, but also from its ability to support and encourage parallelism, its list manipulation facilities (similar to LISP) and its ability to interact with PROLOG and PROLOG databases. These features all combine to make it one of the most powerful, state-of-the-art systems. In terms of visual programming, much of the graphical elements rely on text to identify the function and the control structures rather than utilizing some unique graphic portrayal of these characteristics. However, this is appropriate since the problem domain of PROGRAPH is general programming and the terms and text identifiers are appropriate for the targeted users, who would be familiar with general, control flow-style programming.

2.7.10 ThinkerToy

ThinkerToy is a visual language environment developed at the University of Massachusetts at Amherst that was developed for modeling decision support systems [Gutfreund 1987]. Decision support systems are those put in place to analyze semi-structured decision tasks such as bond trading, setting budgets, capital acquisition analysis and time study.

ThinkerToy is an object-oriented system that supplies a number of different "toolkits" with different dimensional characteristics:

0d: Scalar Operations such as trig. and log functions, detection, injection.

1d: Array Operations for manipulating tabular data.

2d: Chart Operations for curve fitting and extraction nets.

3d: Terrain Map Operations for managing terrain maps and thematic maps.

These operations are applied through the use of Panels which allow access to the various primitive functions. The panels are manipulated with a mouse to directly apply the tools to data. ThinkerToy actually supports two types of tool application: direct visual analysis and scripting. An example of direct visual analysis is the action of placing a ruler tool over a map or a curve fitter tool over a data chart. Scripting is a collection of generic tool handling instructions, that correspond to complex mouse operations on a tool or set of tools.

ThinkerToy seems quite similar to Programming By Rehearsal, with a Panel in ThinkerToy being equivalent to a Troupe in Programming By Rehearsal. This demonstrates one of the attributes of visual languages, which is their focus on a specific problem domain.

2.7.11 ARK

The Alternate Reality Kit (ARK) was developed at the Xerox Palo Alto Research Center (PARC) by Randell Smith [Smith 1986]. ARK is an animated simulation environment that was designed to enable the development of intuitive understanding of abstract physical laws.

ARK is an object oriented system implemented in SmallTalk-80. It consists of seven major objects which are the fundamental tools and building blocks of the language. These objects include:

- A mouse-directed hand for picking, placing and moving objects.

• Buttons that are moved and placed by the had over objects that they can then send messages to. The buttons name describes the message it sends.

• The warehouse is the store of all of the prototype objects available to an alternate reality. They are extracted from the warehouse for use.

• Representatives are objects that display a picture or some text describing the SmallTalk object they represent in the alternate reality.

• Message Boxes allow the development of user defined messages which can be sent to objects that are defined to respond to the particular message. Message boxes employ visual "plugs" to specify the message receiver and the message arguments (if any). To eliminate syntax errors, plugs will not "stick" to any object that does not recognize or respond to the type of message being sent. Message boxes, once defined by the user, can be changed to buttons for repeated use.

• "Slider switches" are a graphical method of specifying numeric values to the objects that require numbers.

• Interactors are the fundamental method of defining an on-going "physical process" that occurs in the Alternate Reality. An example of this is a gravity interactor, which is constantly effecting the forces on other objects in the Alternate Reality that have a defined property of Mass. Any number of interactors can be be specified in an Alternate Reality and then the properties that they represent are available for inspection and manipulation. The hand can be used to pickup and "throw" objects, for example, to study the effects of their direct manipulation. The properties defined by the interactors in the Alternate Reality can be adjusted to see how these properties effect the objects that can be manipulated by the hand.

ARK uses graphical images and the ability to manipulate them in a tangible way to aid in its simulation of physical properties. The combination of objects provides the definition of the properties of an environment or object.

2.7.12 LabVIEW

LabVIEW is a commercial software package designed to support data acquisition and data processing on the Macintosh [Kodosky and Dye 1987]. It combines many of the features of visual programming with data flow programming to provide an environment described as a virtual instrument.

LabVIEW has been designed to handle the tasks of data acquisition instrument control, data acquisition, and data processing. It uses a data flow-like paradigm of wiring a circuit, providing graphical components that represent logical, mathematical, statistical, string and I/O functions. These are assembled in an "Diagram" window that represents a single function or operation. The I/O functions provide control of instruments that are connected to the Macintosh via a GPIB bus. Other available functions provide data flow extensions that provide intuitive methods for specifying conditional operations, iterative operations and "apply to all" procedures.

Once functions have been defined, the "Panel" window carries the the metaphor of the virtual instrument by placing graphic switches, knobs, and meters which can represent one source of I/O to the functions. Once simple virtual instruments have been defined, they can be readily combined to create more complex data processing systems.

LabVIEW demonstrates the major contribution of visual languages. It is intuitive to personnel that are used to working with instruments and it matches well with their problem domain. It provides a "rear view" of an instrument, complete with its wired interior circuits and a "front view" that contains the switches and meters.

2.8 Summary

The major problems to be solved at this stage in the development of visual programming languages have to deal with the development of the appropriate paradigms. Pure visual programming systems seem to indicate that standard programming language paradigms (i.e. imperative, procedural languages) cannot be transformed into good visual languages. Because the user interacts to a high degree in the design and specification of the graphical objects, declarative or object-oriented approaches may be better suited to visual languages. The PICT system seems to suffer because it tries to hold to a procedural language approach.

Graphical programming language techniques have evolved in two different branches. One branch, that of the programming visualization systems, exemplified by the PECAN system and the OMEGA system, will develop systems with many "views" of a program (some graphical and some textual) so that the appropriate information about the program's performance will be available to the programmer, along with systems that help enforce rules of good structuring, high reliability and good maintainability.

The other branch will be that of "pure" visual programming environments, where the programmer will manipulate graphical objects. One of the difficulties to be overcome is the context of graphical objects, the fact that they convey information abstractly (through their appearance) and may be misunderstood or misinterpreted. The need exists to develop a symbol system or rules and constraints for designing symbols that are clear and unambiguous. Also the management of the graphical databases that are generated will have to be optimized.

Visual programming languages can provide programming systems that will solve a wide range of problems. They will allow programs to be written by persons with an understanding of the problem's domain by allowing symbols to be developed that represent exactly the problem rather than first having to redefine the problem with a set of unrelated symbols in a standard computer programming language.

Missing Page

Chapter 3

An Example of Graph Creation With FlowGraph

3.1 Introduction

This chapter gives a demonstration of FlowGraph as it used to create a data flow program of the classical factorial algorithm. It demonstrates many of the features of FlowGraph and many of the techniques used to create data flow graphs and programs that will execute on the data flow simulator at RIT [Benjamin 1988].

3.1.1 Some Basics

FlowGraph operates on a Macintosh computer as a preprocessor for the data flow simulator at RIT. Its purpose is to allow the user to draw a data flow graph in a very intuitive manner and then translate the graph into a textual assembly language for processing with the data flow simulator.

FlowGraph operates through the framework of many of the standard user interface paradigms of the Macintosh [Rose 1985]. The terms used in this chapter are those found in the standard Macintosh vernacular. For the user unfamiliar with the Macintosh, the principle terms are defined in the Glossary.

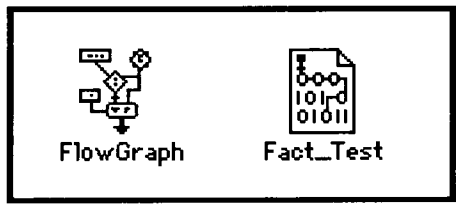


Figure 3.1 - the FlowGraph Program and Document Icons

3.1.2 Starting FlowGraph

FlowGraph can be launched by selecting its icon (Figure 3.1) in the Macintosh Finder and selecting Open from the File menu. Alternately, double clicking on the FlowGraph icon will launch the program. The FlowGraph screen display is shown in Figure 3.2.

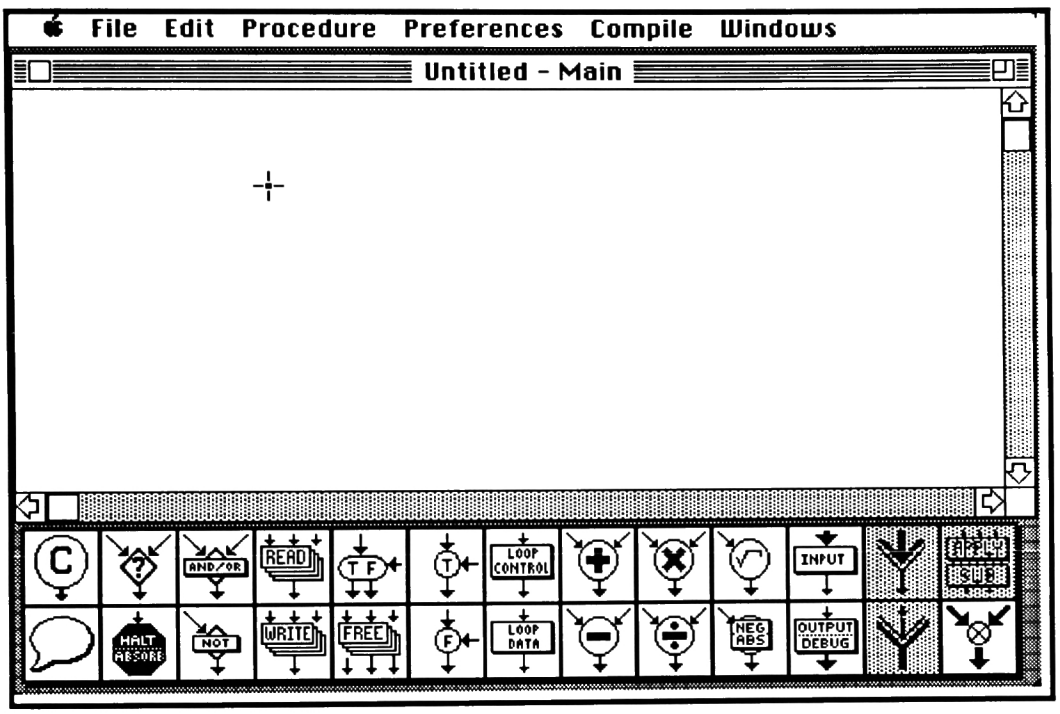


Figure 3.2 - The FlowGraph Screen

The FlowGraph screen display consists of four major objects: the menu bar at the top of the screen, a graph window, a node palette and the cursor. The menus in the menu bar perform functions of global nature such as opening files, translating graphs into textual simulator commands, and setting user preferences. A graph window is where the nodes and arcs making up a data flow graph area drawn, edited, and manipulated. There may be multiple windows on the screen at one time, but only one window will be active at a time. There is one window representing the main program.

It is the first window listed in the window menu. The graphs in the remaining windows represent procedures.

As a demonstration of FlowGraph, a graph program representing the calculation of a factorial will be drawn. The recursive algorithm for a factorial is:

```
mainline
     input(n)
     factorial(n)
     output("factorial = ",n)
end mainline

factorial(n)
     if (n = 1) then
          return (n)
     else
          return (n * factorial(n - 1))
     end if
end factorial
```

A data flow graph representing this algorithm is shown in Figure 3.3.

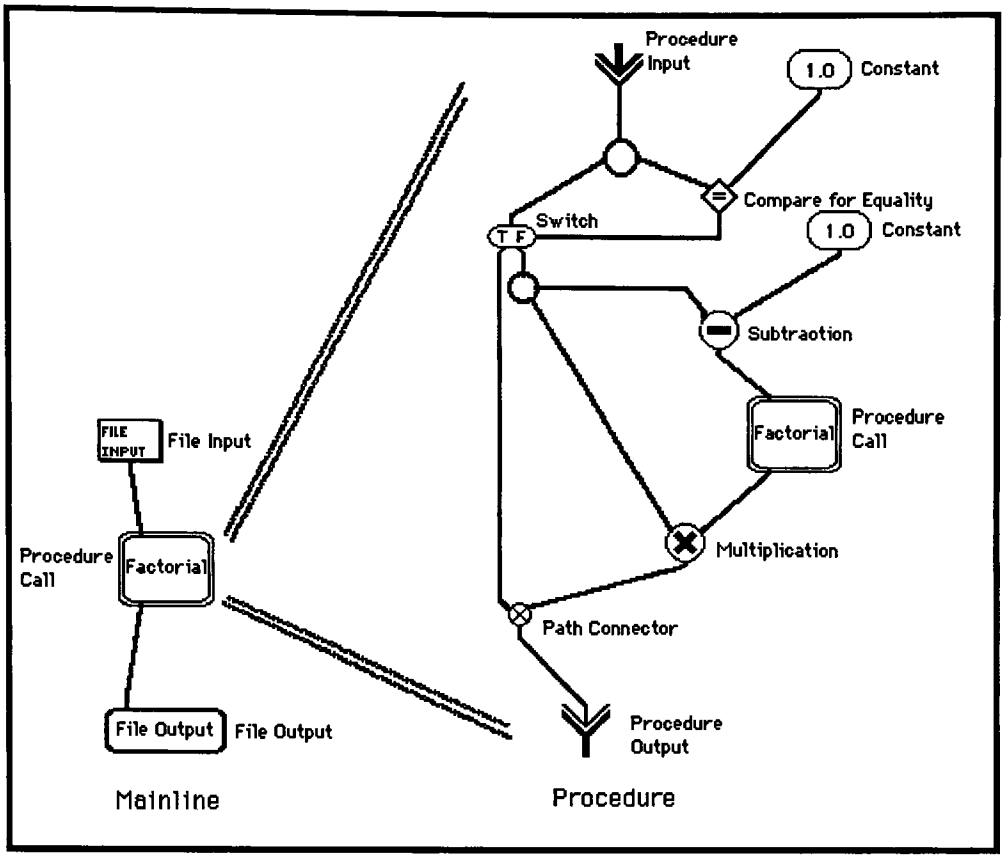


Figure 3.3 - A Data Flow Graph representing the Recursive Factorial Algorithm

3.1.3 Definitions

The definition of some fundimental terms is appropriate at this point. A node is the pictorial representaion of an operator that typically modifies data or the flow of data. An arc connects nodes together, describing the path of data between nodes. A port is the point of connection of a arc to a node. The type and semantic meaning of the data are defined by the port that accepts or emits the data token to / from the node. A graph is a collection of nodes and arcs that describe the flow of data during the execution of the program that the graph represents.

3.1.4 Creating a Procedure

It is most convenient to create all of the procedure graphs prior to creating the main program graph. It is not absolutely necessary to create the entire procedure graph, however. It is only necessary to specify how many input and output ports are required of each procedure. Then the appropriate number of ports will be represented on the procedure node when it is specified in a graph.

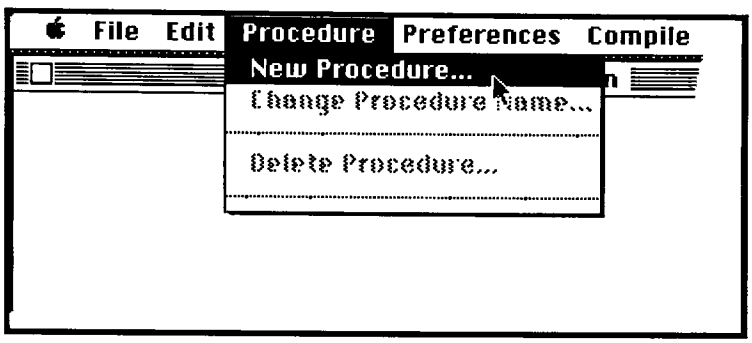


Figure 3.4 - Creating a New Procedure Window

To create a procedure graph, create a new procedure window with the "New Procedure" command in the "Procedure" menu. (Figure 3.4). This creates a new window with the name "Procedure #1". As new procedures are created they are identified with sequentially higher numbers. To change the name of a procedure, use the "Change Procedure Name..." command in the "Procedure" menu. A dialog box is used to type in the new name for the procedure that is the currently active window (Figure 3.5). Press the OK button to accept the new name or press the cancel button to cancel the name change.

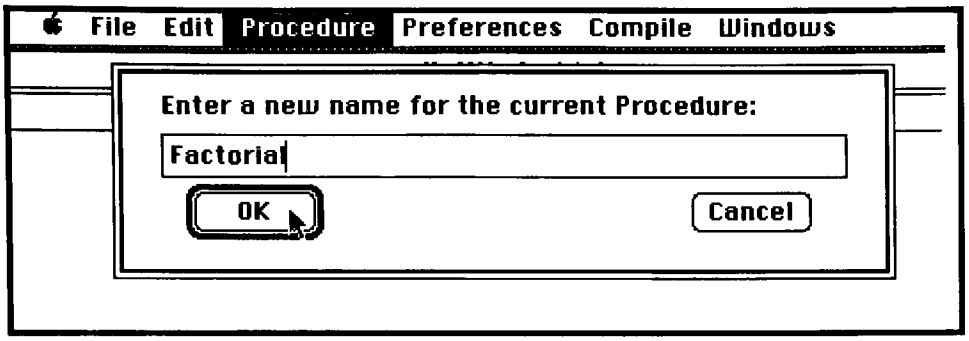


Figure 3.5 - Setting a Procedure's Name

In the dialog boxes presented to the user by FlowGraph, there is usually one button that is outlined. That indicates the button that is "pressed" when the return key or the enter key is typed on the keyboard. This is a useful shortcut in text entry dialog boxes.

3.1.5 Working with the Node Palette

The actual creation of the graph is accomplished by moving the appropriate nodes from the node palette into the graph window. To pull a copy of a node off of the palette, move the cursor over the desired node in the palette, hold down the mouse button, drag the node to its desired location on the palette, and then release the mouse button. While you are dragging a node, the node will appear as a square outline (Figure 3.6). When the mouse button is released, the node will appear in the graph.

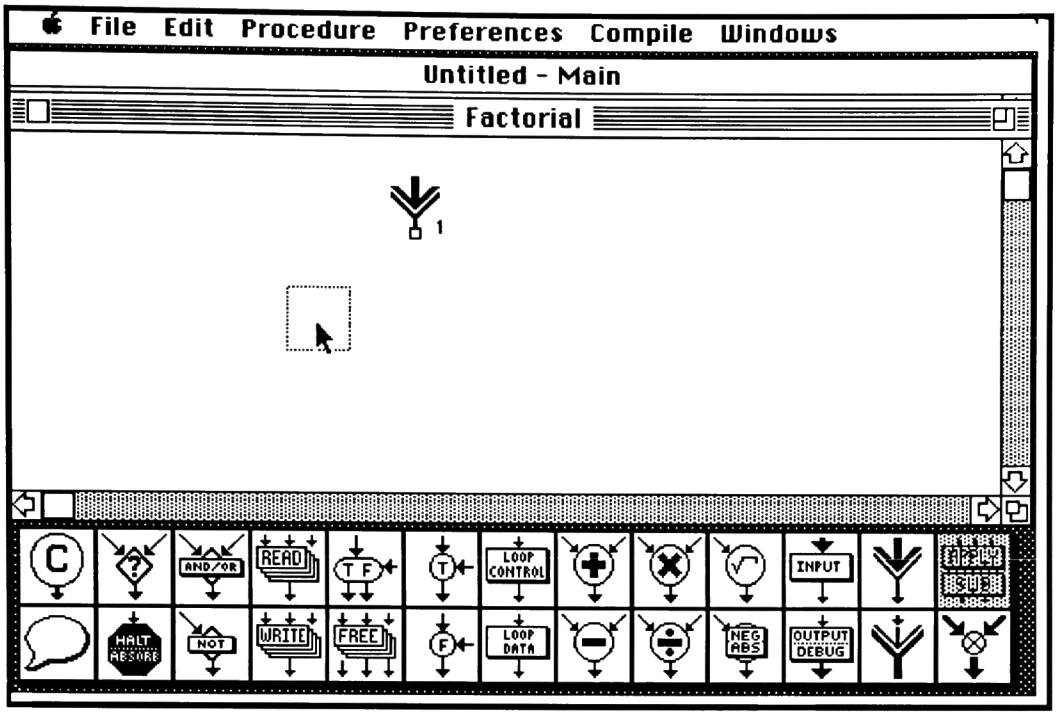


Figure 3.6 - Placing Primative Operation Nodes from the Palette into the Graph Window

To create the Factorial procedure, place an input port connector node and a switch node in the Factorial graph window as shown in Figure 3.7.

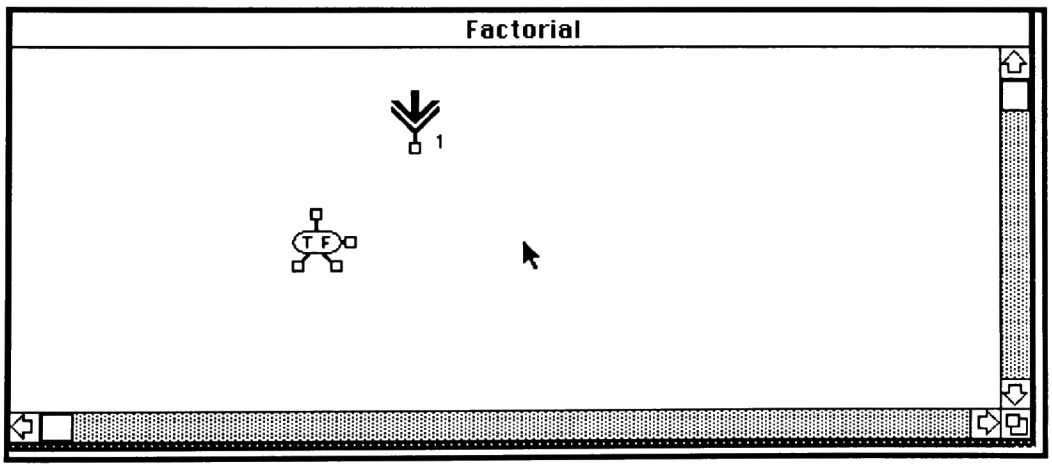


Figure 3.7 - Some Primative Operation Nodes in the Graph Window

All of the possible nodes are not shown in the palette. Nodes tend to be grouped into "equivalent groups", which are groups of nodes with very similar characteristics. Examples of equivalent groups include the mathematic operators, the conditional operators and the boolean operators (and, or). Members of an equivalent group can be substituted for each other in the graph.

To place a node representing a conditional test for equivalence on the graph, a copy of the general conditional is pulled off of the palette and its outline is used to position it in the graph. When the mouse button is released, a "mini-palette" appears over the left side of the node palette (Figure 3.8). This mini-palette contains the members of the equivalent group for the conditional node. By clicking on the appropriate conditional, that conditional is placed in the graph.

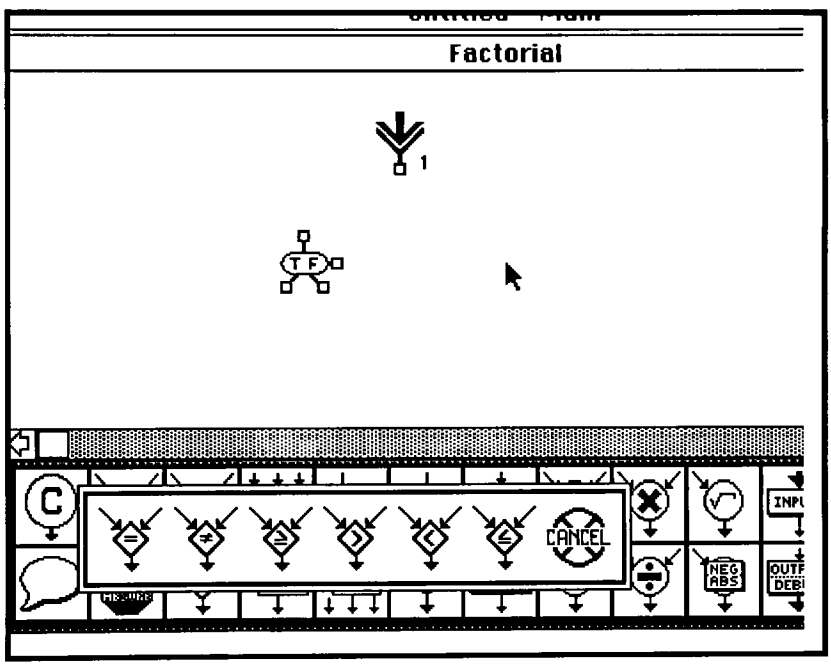


Figure 3.8 - "Mini-Palette" for Selecting Conditional Nodes

3.1.6 User Preferences

A number of user preference features exist in FlowGraph. One of these is the ability to make an invisible grid in the graph window that the nodes will "snap" to when they are placed or moved on the graph. This allows a cleaner, more organized graph. Select the "Grid Mode..." command from the "Preferences" menu. A dialog box will appear as shown in Figure 3.9. The spacing is the number of screen pixels between grid points. A screen pixel is 1/72 of an inch. Thus a quarter inch square grid can be specified with a setting of 18 pixels in both horizontal and vertical. Click the OK button or press return to accept the entered value.

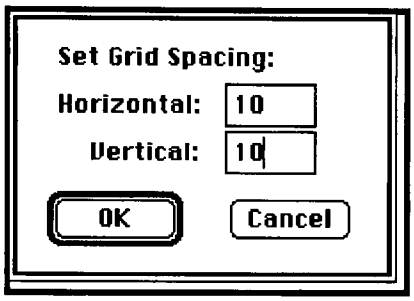


Figure 3.9 - Grid Dialog Box

3.1.7 Constant Nodes

When a constant node is placed in the graph, it has a default value of zero. To set the value of a constant node, double click on the node. The dialog box shown in Figure 3.10 is used to set both the value and the type of the constant. Typing a number sets the constant to have a floating point value. By clicking on the True or the False buttons, a boolean constant is defined.

Constant nodes are displayed in the graph with an attempt to display its value as well. Figure 3.11 shows the value displays for a number of constant nodes with different values. If a real value cannot be displayed in the node because of its length, the symbol "*V*" is placed in the node.

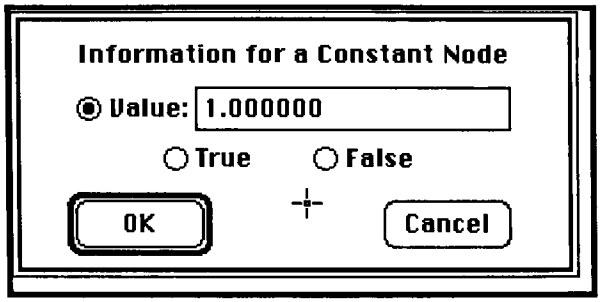


Figure 3.10 - Setting the value of a Constant Node

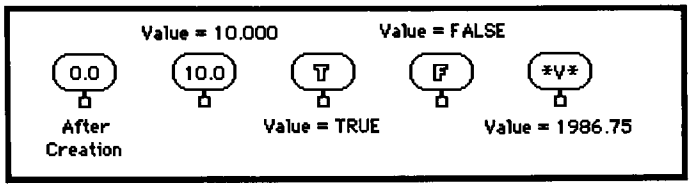


Figure 3.11 - Various Displays of a Constant Node

Place a constant node on the graph as shown in Figure 3.12. Set its value to 1.0.

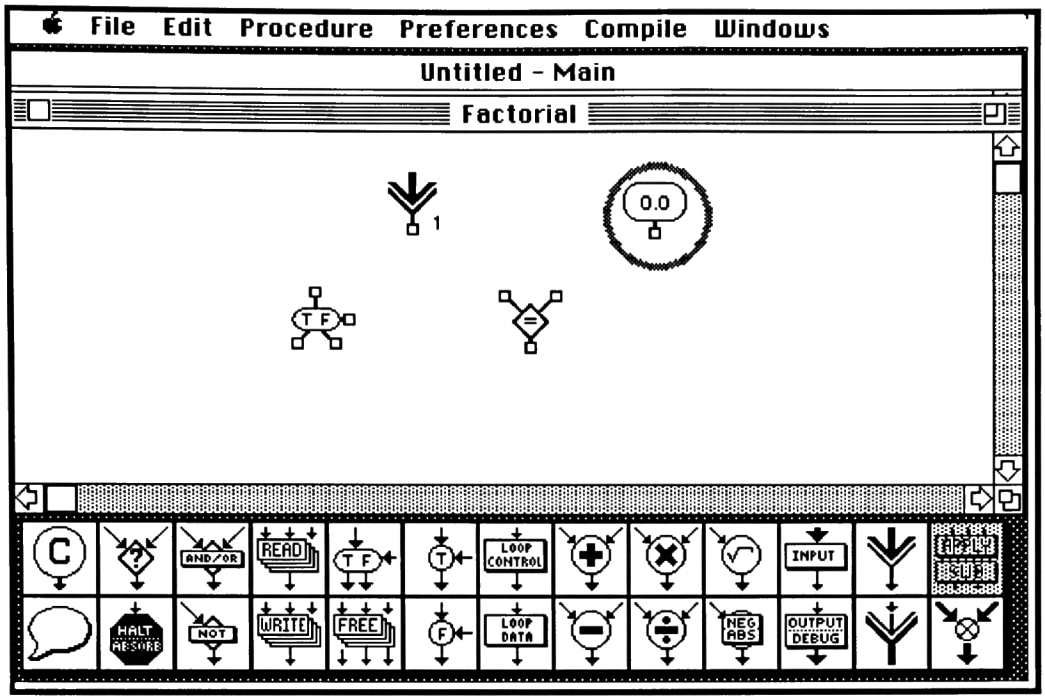


Figure 3.12 - Adding a Constant

3.1.8 Changing Nodes on the Graph

Place an addition operator node on the graph as shown in Figure 3.13.

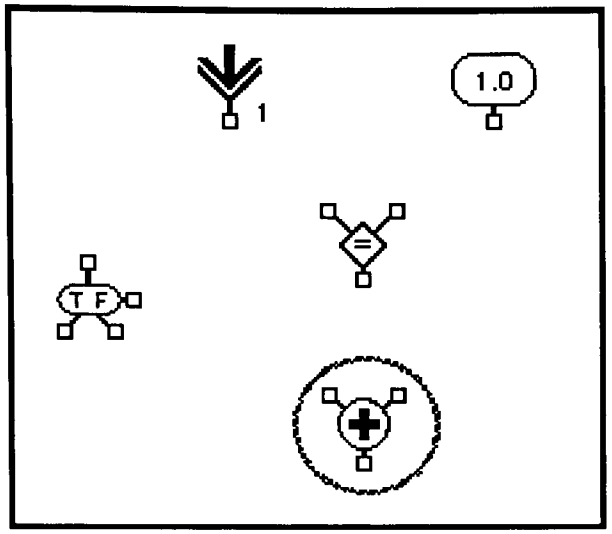


Figure 3.13

Actually, a subtraction operator is appropriate for the Factorial algorithm. To change a node on the graph to a different "equivalent" type, select the node by clicking once in it. The node will indicate that it has been selected by being displayed with a grayed "body" as shown in Figure 3.14. In the "Edit" menu, select the "Change Node Type..." item. (Figure 3.15). A mini-palette will appear showing the equivalent types for the addition operator. (Figure 3.16). Click on the subtraction node to substitute it for the addition operator (Figure 3.17).

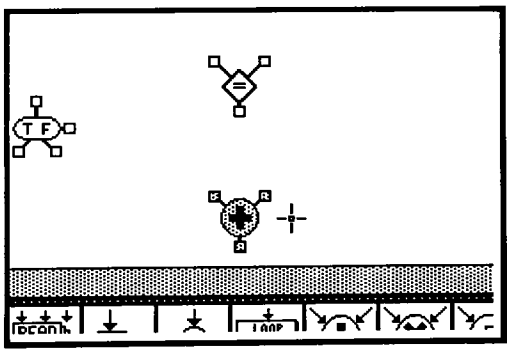


Figure 3.14- A Selected Node is displayed with a Grayed "Body"

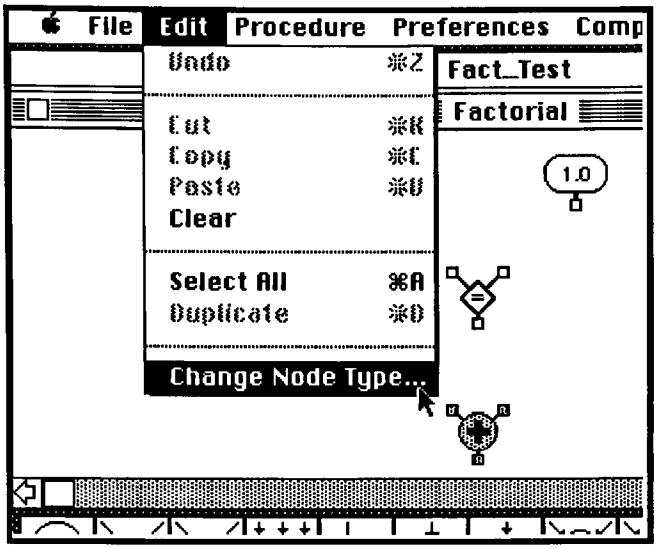


Figure 3.15 - The "Change Node Type..." from the Edit Menu allows changing Selected Node to another "Equivalent" Type.

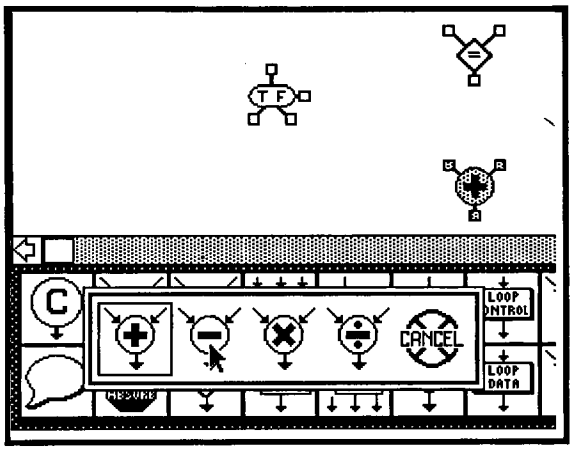


Figure 3.16- The Mini-Palette displaying the "Equivalent" Types

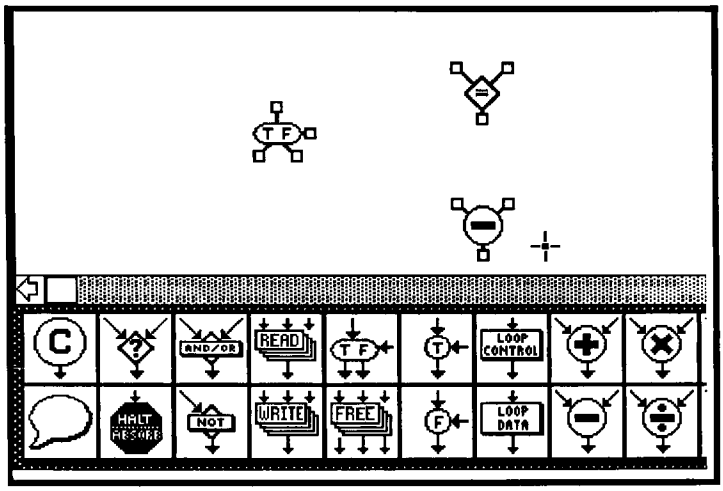


Figure 3.17 - The Graph after the Change.

3.1.9 Adding the Rest of the Nodes

To finish adding the rest of the elementary operator nodes, refer to Figure 3.18 for their type and location. To move about on the graph, use the horizontal and vertical scroll bars.

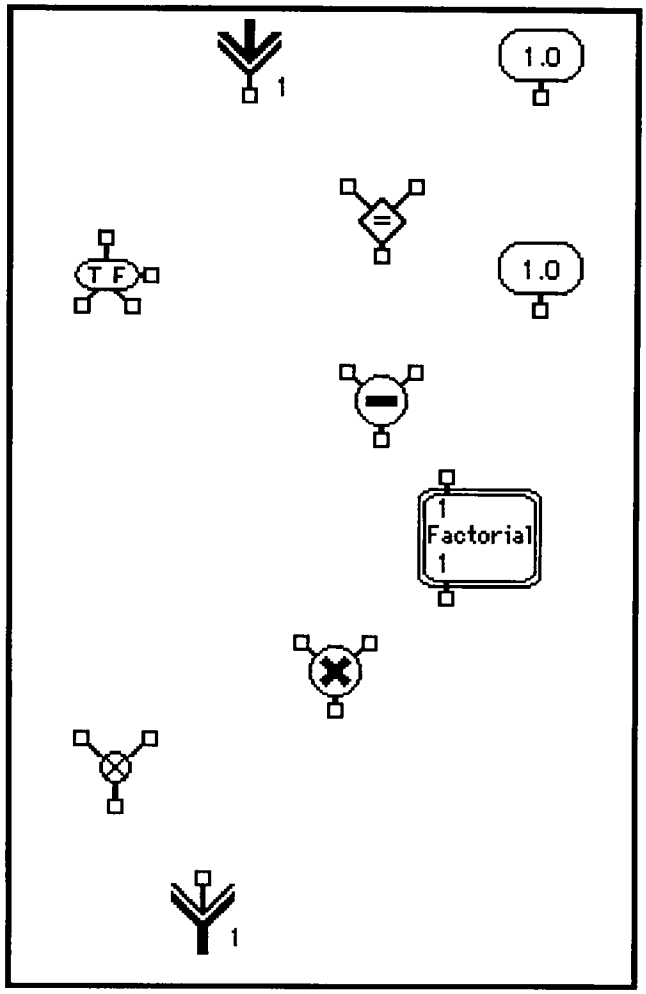


Figure 3.18 - Node Layout for the Factorial Graph

The scroll bars allows the user to place the graph window over any position on the large virtual graph plane. Moving the slider in a scroll bar moves a proportional amount of distance in the "allocated graph plane" while pressing the arrowheads moves the window over the virtual graph plane in small increments. The allocated graph space is the the portion of the virtual graph plane that has been allocated to the particular graph. To increase the allocated plane dimensions of the graph, press the arrowheads at either end of the graph. This will allocate small amounts of the virtual graph plane to the graph and scroll over that area of the plane. Figure 3.19 shows the releationship of the concepts of the virtual graph space, the allocated graph space and the graph window.

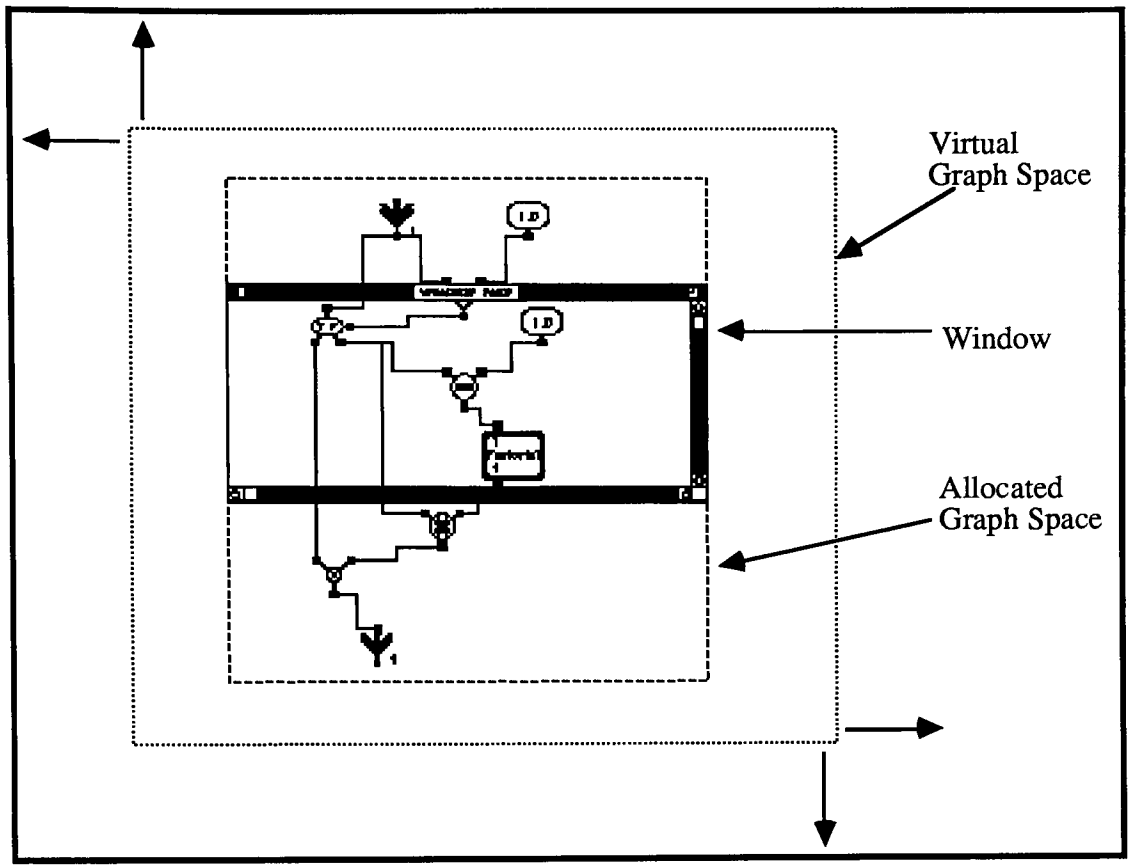


Figure 3.19 - Comparison of Virtual Graph Space, Allocated Graph Space and the Graph Window

3.1.10 Adding Nodes Representing Procedures

To add a procedure to the graph, use the Apply operator. As each procedure is created, its name is added to the "Procedure" menu. By selecting the name of the procedure you want to add from the procedure menu, a node representing the procedure can be pulled off of the node palette using the Apply operator (Figure 3.20).

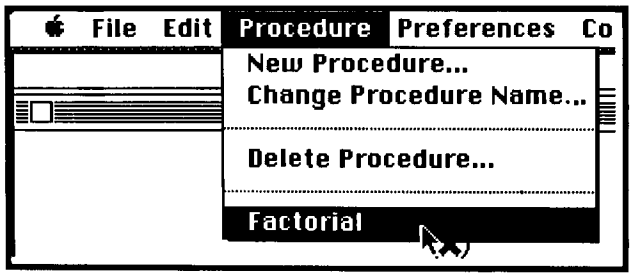


Figure 3.20 - Selecting a Procedure Name to be Added to the Graph with the Apply Palette Operator

When there is no procedure is selected in the "Procedure" menu, the Apply operator is grayed on the palette, indicating that it cannot be selected. When a procedure is selected, a check mark appears by it in the menu and the Apply operator becomes available for use (Figure 3.21).

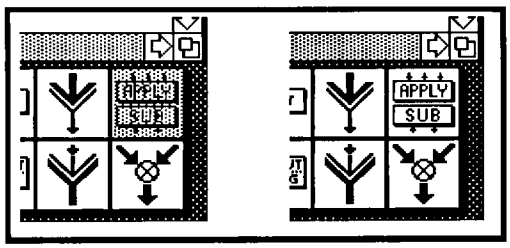


Figure 3.21 - An Inactive Apply Operator and an Active Apply Operator

When the Apply operator is dragged off of the palette and released in the graph window, a node representing the selected procedure is created. It is represented with a double, round corner box containing the name of the procedure and numbered input and output port connections. Place a Factorial procedure node on the graph as shown in Figure 3.22 to allow the use of the procedure recursively. Any number of nodes representing a selected procedure may be added to the graph this way. To add a different procedure, select a new name from the "Procedure" menu and continue to

use the Apply operator from the palette. To deactivate the Apply operator, select the current procedure from the menu again.

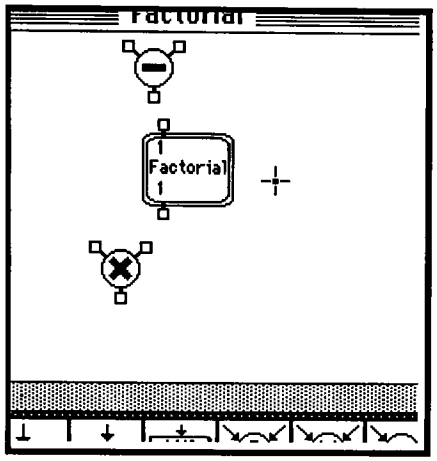


Figure 3.22 - A Procedure Node

3.1.11 Adding Arcs

Arcs are drawn in the graph by moving the cursor into a node port, pressing and holding the mouse button, and dragging the cursor to an appropriate port on another node. While the cursor is being dragged, the arc is represented by a "rubber band" line (Figure 3.23). If the mouse button is released while the cursor is over an acceptable port, the program redraws the arc as a jagged line with darkened ports at either end, as in Figure 3.24.

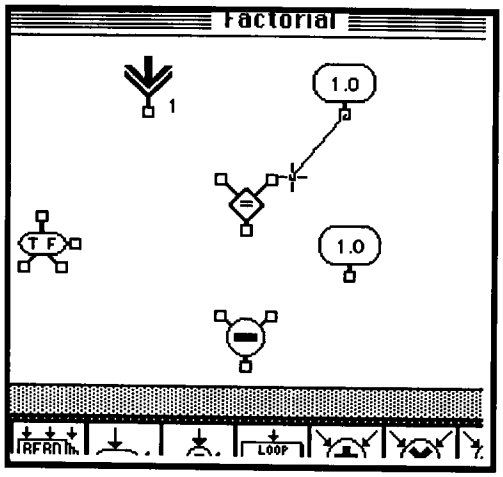


Figure 3.23 - A "Rubber Band" Arc

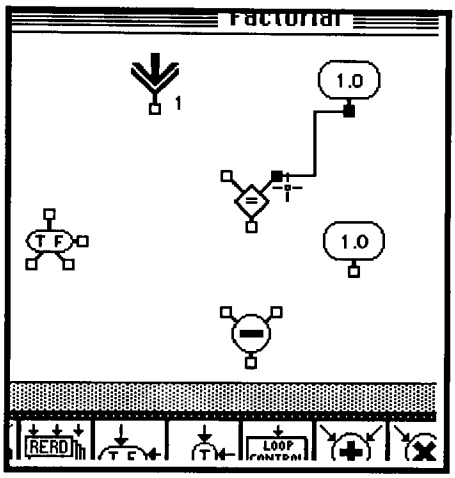


Figure 3.24 - A Final, Connected Arc

FlowGraph uses a simple set of rules for adding arcs to a graph:

- A arc must connect an output port of one node to an input port of another node.
- Multiple arcs can eminate from any output port (except from a constant node).
- Only one arc can be connected to any input port.
- The ports being connected must have the same data type (boolean or real).
- Since the graph windows do not autoscroll when an arc is dragged outside the

window area, the ports to be connected must both be visible in the graph window at the same time.

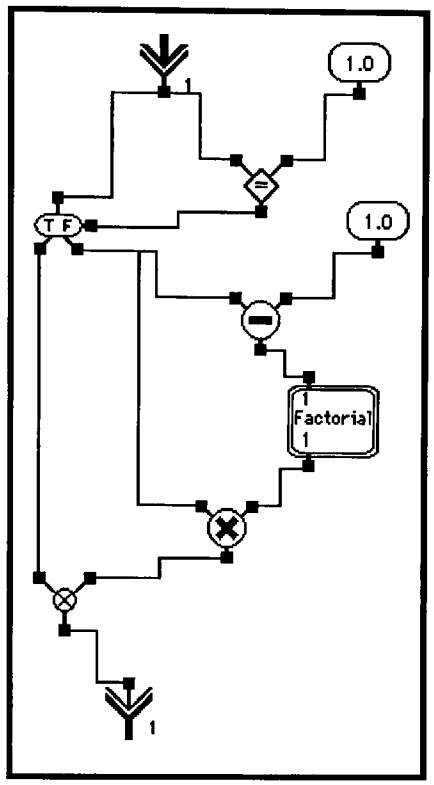


Figure 3.25 - Connected Factorial Graph

In the Factorial example, connect the nodes as shown in Figure 3.25.

Based on the rules for connecting arcs, there are a number of approaches that give the user a fair amount of flexibility. For example, if a Switch operator is used as a decider for two different data paths, and the results of either one or the other data path needs to be connected to the input port of a node, there is no way to directly connect the two arcs to the single input port. However, a connection node exists that has two input ports and one output port which can be used to channel the two alternate data streams to a single input port. Of course, during the program execution, only one data path can supply tokens to the input port. See the Factorial graph for an example of this. This is the method of creating "if - then - else" structures.

Some nodes can process either boolean or real data. If an arc is drawn between two ports that are not defined to have a specific data type, a dialog box will be presented to pick the type for the arc. This data type will then be assigned to the other relevant ports on the node as well. For example, the Switch operator node can switch the input data based on the value of a boolean input. The input data can be either boolean or real. Once an arc is connected to either the input port or one of the two output ports, the data type for that arc determines the data type that must be connected to the other ports on the node. If an arc representing a real data type is connected to the input port of the Switch operator node, the output ports of the Switch will then check for connections to ports of another node that accept real values. It is possible to turn off the type checking with the "Preferences" menu in FlowGraph. In the data flow simulator, all data objects are processed as floating point numbers, where a value of zero is returned by a boolean expression that evaluated to false and a non-zero value is considered to be true. Circumventing type checking is not recommended, however.

To help connect ports in a large graph, the graph windows can be zoomed out to fill the entire screen space using the zoom box in the upper right hand corner of a graph windows title bar. Figure 3.26 shows the standard window which exposes the node palette while Figure 3.27 shows the window zoomed out to fill the screen.

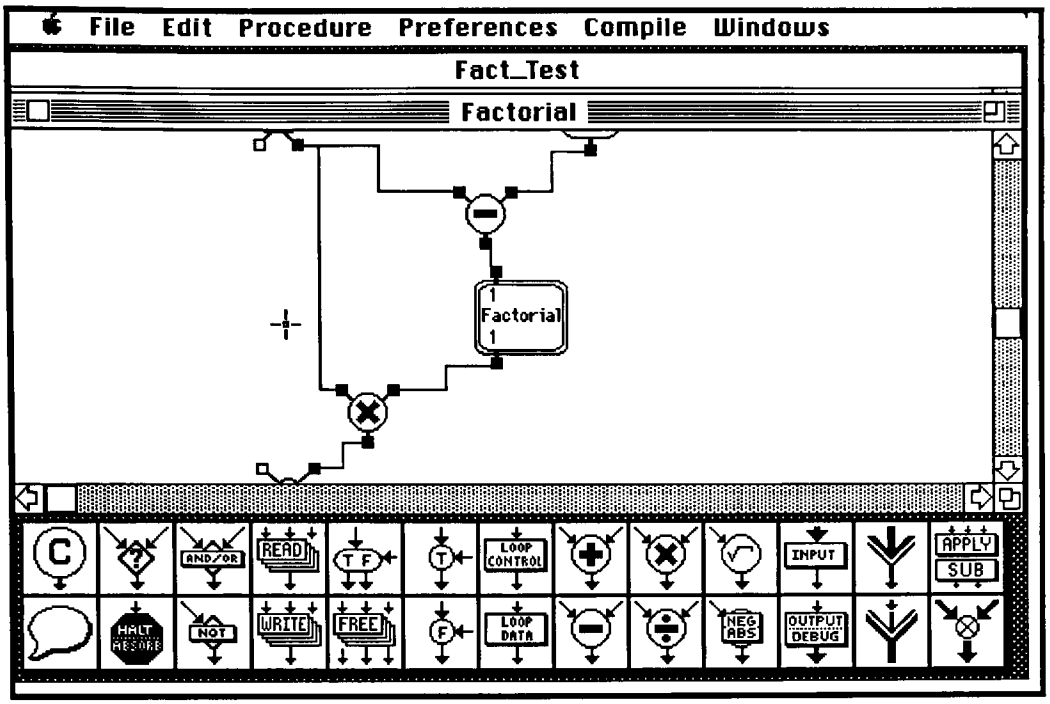


Figure 3.26 - The Standard Window Size Exposes the Palette

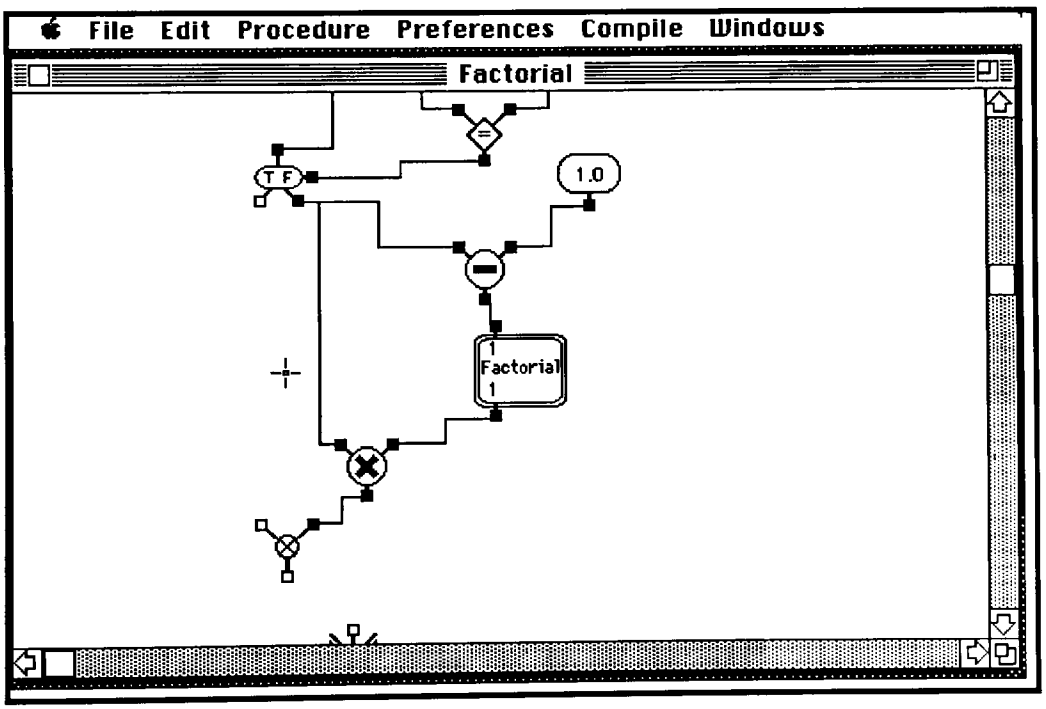


Figure 3.27 - The Window Zoomed to Full Size Covers the Palette

3.1.12 Editing Arcs

Figure 3.28 shows an arc that has been incorrectly connected from the true side of a Switch operator to the multiplication operator. To select an arc, click on the input port that it connects to. A selected arc is displayed as a grayed line (Figure 3.29). To remove the arc, select the "Clear" menu item from the "Edit" menu or press the backspace key. This approach of select an object and deleting it works for nodes as well. If the shift key is held down while selecting objects, multiple objects may be selected at one time.

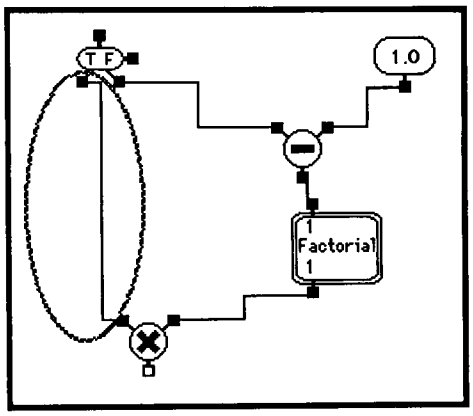


Figure 3.28 - The arc between the "True" output port of the Switch Node should not go to the Multiplication node.

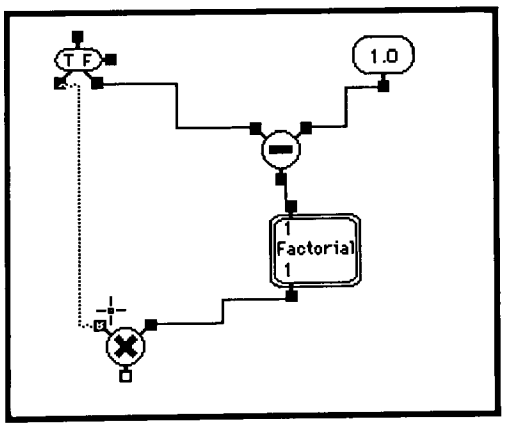


Figure 3.29 - Select An Arc By Clicking on the Input Port it is Connected to.

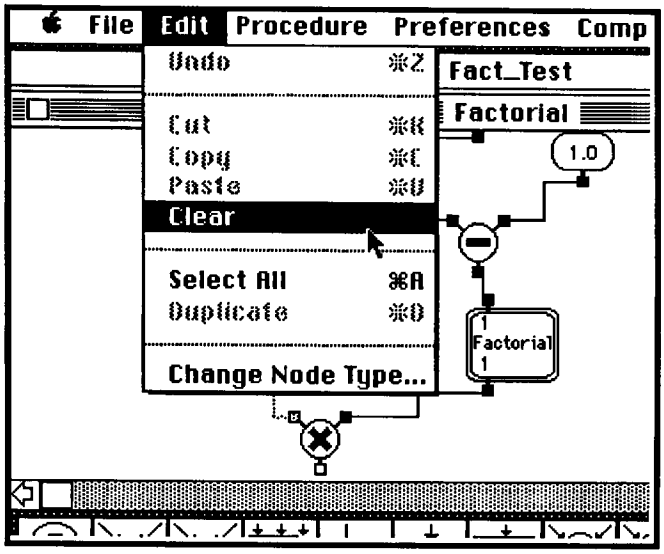


Figure 3.30 - Delete a Selected Arc (or Node) by using the Clear menu item.

3.1.13 The Main Program Graph

Figure 3.31 shows the main program in its graph window. After creating the Factorial procedure graph, activate the main window by clicking somewhere on the window or by selecting the main graph window from the "Windows" menu. The main graph window is always the first window listed in the "Windows" menu.

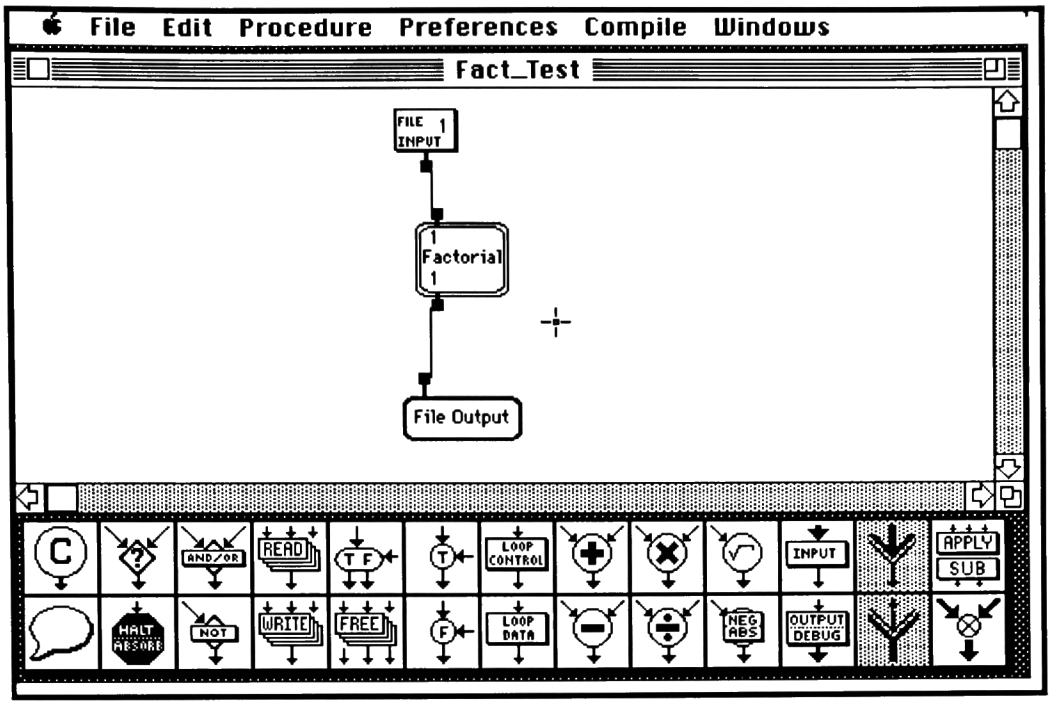


Figure 3.31- The Main Program Graph

As Figure 3.31 shows, the procedure input / output connectors in the node palette are grayed because they are not available in the main program graph.

The main program should have a file input node which will be used as the trigger for the graph execution by the simulator.

When the output node is created, the user will be asked to supply the number of I/O ports required. For this example, one port was specified. By double clicking on the file output node, a window will open to enter the text of any message to be printed with the output value(s).

3.1.14 Setting Trace Options

Before generating the simulator input files, any simulator tracing options must be set. Select the "Set Trace Function..." menu item from the "Preferences" menu. The dialog box shown in Figure 3.32 allows the user to select any trace options for the simulator trace functions. (Be aware that the trace file generated by the simulator can grow very large very quickly during the simulator execution.) Clicking OK sets the selected options when the graph is translated into the simulator assembly language.

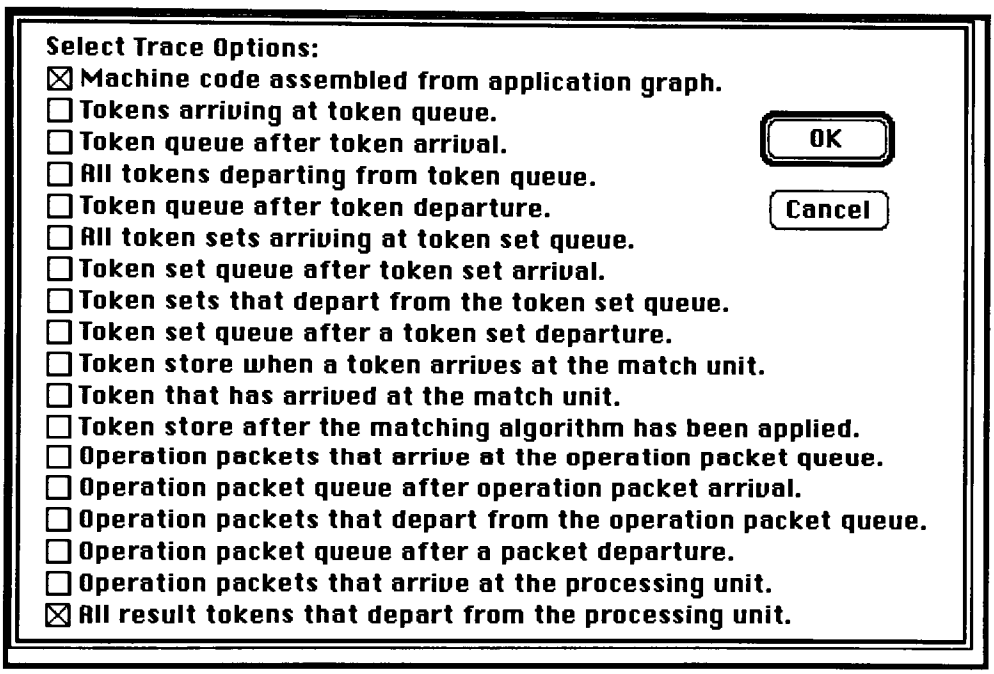


Figure 3.32 - The Trace Option Dialog Box

3.1.15 Saving a FlowGraph Graph File

Select the "Save" item in the "File" menu to save the the graph(s) to some storage media. Figure 3.33 shows the standard Macintosh Save file dialog. Enter the name

of the file and click the save button. The file name that was selected will become the name of the main graph window.

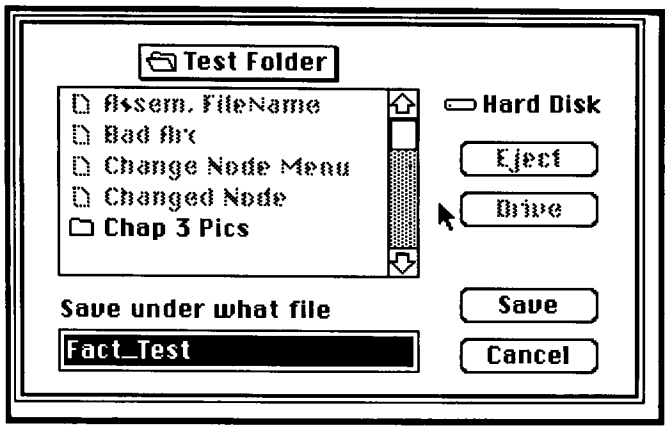


Figure 3.33 - The Standard Output File Dialog for the Saving the Graph

3.1.16 Translating the Graphs into the Simulator Input Files

Select the "Assemble..." command in the "Compile" menu to begin the translation process. Two file dialogs will be presented. The first will be request for the name of the assembly language text file to be saved. The default name will consist of the name of the main program graph window with ".prog" appended to the end. The second file window will request the name of the file containing the input data for the "File Input" nodes that are in the graph. The default name will consist of the name of the main program graph window with ".dat" appended to the end.

Figure 3.34 shows the simulator assembly language output generated by the graph of the Factorial program. It is important to remember that the assembly language output is based not only on the nodes and the arcs but also on the order of creation the nodes in the graph. Thus the order of the commands in the assembly language file may vary between two different users drawing the same graph, based on the order in which they created the nodes. Since the entire description of the graph must

be read into the simulator prior to its execution, the ordering of the statements is not critical as long as they describe the same graph.

```
# Dataflow Assembly from: Fact_Test

# Trace Flag:
#          1 2 3 4 5 6 7 8 9 a 1 2 3 4 5 6 7 8
trace      1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

#  Main

a           1          1           2          4
a1          2          1           3 1
output      3          1

# Procedure: Factorial

link       12                      7 1        6 1
pbeg        4          1          12 1        5 2
link       13                      8 1        9 1
switch      6                      5 1       13 1
eq          7                      6 2
const                 1.000000     7 2
sub         8                     10 1
mul         9                      5 1
a          10          1          11          4
a1         11          1           9 2
const                 1.000000     8 2
pend        5          1
```

Figure 3.34 - The Assembly Langauge File

Figure 3.35 shows the input data file. For each of the "File Input" nodes that are in the graph, one line is generated. Since FlowGraph has not implemented a method of getting the actual input values from the user, this file uses a placeholder of the form <valuenn>, where nn is the number in the "File Input" nodes that are in the graph. A text editor on the Macintosh or on the UNIX system where the simulator resides can be used to edit this file, substituting the actual input values for the <valuenn> placeholders.

```
<value1>        1 1
```

Figure 3.35 - Output File for Simulator Data Input

3.1.17 Printing the Graphs

FlowGraph can print the entire graph in the currently active window selected by the user through the "Page Setup..." and the "Print..." commands in the "File" menu. First select "Page Setup...". Figure 3.36 shows the Page Setup Dialog box for an ImageWriter dot matrix printer. Most settings should be picked based on the paper size loaded into the printer and the orientation that is desired. The "Tall Adjusted" option should always be selected to give the correct aspect ratio for the output. For a large graph, the "No Gaps Between Pages" may also be appropriate.

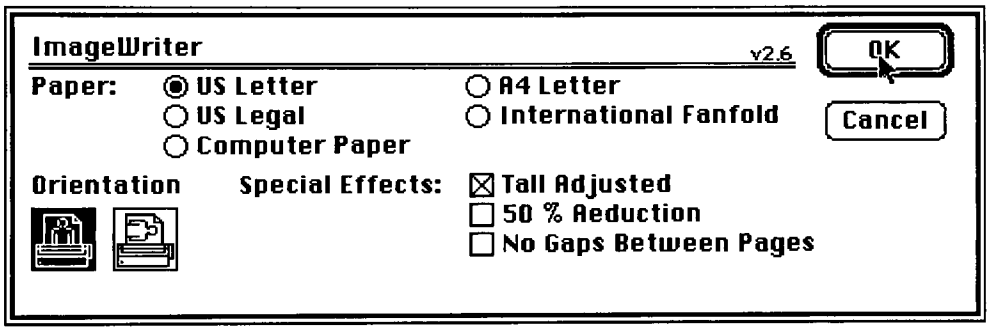


Figure 3.36 - The Page Setup Dialog - Use Tall Adjusted for all FlowGraph graphs

Figure 3.37 show the print dialog presented for the "Print..." command. On the ImageWriter, the Faster quality setting seems to work better for FlowGraph graphs than the Best setting. Nothing will be printed if draft is selected.

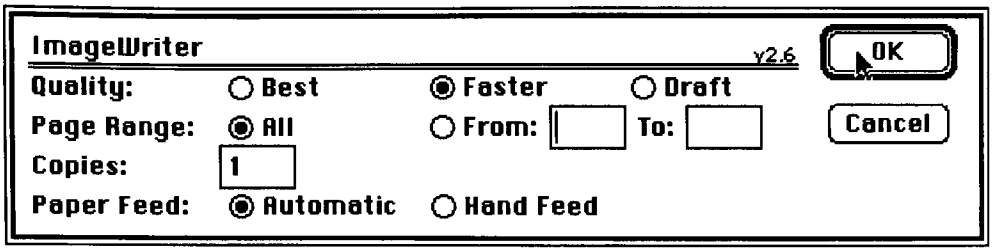


Figure 3.37 - The Print Dialog - Faster Quality is Usually Sufficient.

3.1.18 Running the Simulator

Exit FlowGraph using the "Quit" command in the "File" menu. If there have been some changes to the graph since it was last saved, it will check with the user to see if the changes should be saved as well. If the user clicks the Save button, the standard file dialog will be displayed if the graph has never been saved. Otherwise, the changes will be saved to the currently open file.

The assembly language file and the edited input data file can be sent to the simulators host system using as variety of text file transfer methods associated with the various terminal emulators available for the Macintosh.

To run the simulator, type:

```
$ dfw
Program file name: <assembler program file name>
Input file name: <input data file name>
Output file name: <simulator output file name>
```

If any trace options were selected, a file named "trace.lis" will be created as well.

Chapter 4

Project Description

4.1 Introduction

In an attempt to better understand the issues involved in visual languages and their implementation, a simple visual language environment was developed for this thesis. The environment consists of a program called FlowGraph, which is implemented on a Macintosh computer.

FlowGraph allows a user to manipulate graphical elements representing the nodes and arcs of a data flow graph. The language elements are based on the requirements of the data flow simulator developed by [Benjamin 1988] at RIT. The user of FlowGraph creates data flow graphs on the Macintosh which are translated by FlowGraph into the proper assembly language representation of the graph that can then be used as input to the data flow simulator.

A data flow graph consists of nodes representing operations on data tokens passing through the node. Arcs define the paths of data tokens around the graph by connecting nodes together. The basic premise of data flow systems is that the operations represented by the nodes can be carried out whenever all of the input data tokens are available to the node. This allows multiple processor systems to process data flow programs with a high degree of parallelism.

4.2 Project Implementation

As described in chapter 2, a visual "language" actually consists of three elements: the language syntax, the language semantics and the user interface that allows the user to manipulate the elements representing the syntax in a fashion that supports the

language semantics. The next sections describe the user interface design of FlowGraph, the data structures that are manipulated in the program, and the overall organization of the program.

FlowGraph was written in Pascal on a Macintosh Plus computer using the Lightspeed Pascal™ environment by THINK Technologies,Inc[1]. This language implementation allowed source level debugging in an incrementally compiled environment, which helped speed up the edit - compile - execute - debug - edit cycle.

4.2.1 The User Interface Design of FlowGraph

FlowGraph implements four major elements in its user interface design: menus, the node palette, the graph window and user dialogs. The manipulation of all elements of FlowGraph is through the mouse. There are very few operations that utilize the keyboard. The keyboard operations are only those that require the naming of an object or the indication of the number of certain objects. The user works with FlowGraph to develop the graph using the mouse 95% of the time. The screen display presented by FlowGraph is shown in Figure 4.1.

[1]Lightspeed Pascal is a registered trademark of
THINK Technologies, Inc.
420 Bedford Street
Lexington, MA 02173

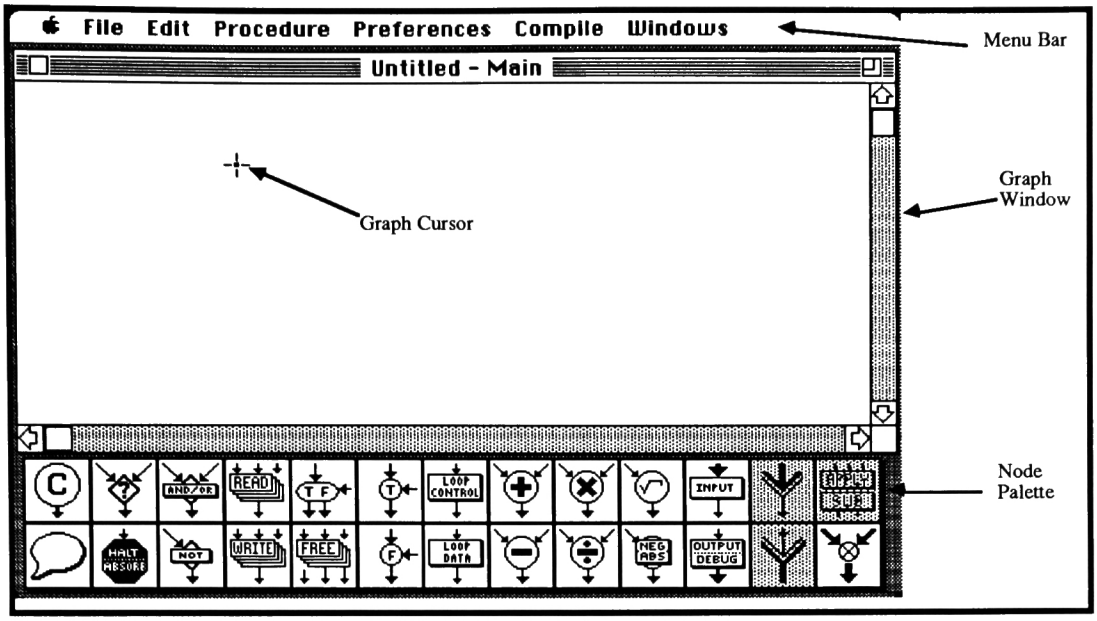


Figure 4.1 - Parts of the FlowGraph Display

4.2.1.1 Menus

Pull down menus contain the commands that are used to manipulate the overall environment of FlowGraph. The type of commands that are in the menus are those for file manipulation (loading files, saving files and printing graphs), editing operations such as element deletion, and the commands to check and compile graphs to a textual assembly language accepted by the simulator. Menus also provide the user with the means to set environment preference variables such as the size of an invisible grid used to align the nodes of the graph. Figure 4.2 shows the menu commands available in FlowGraph.

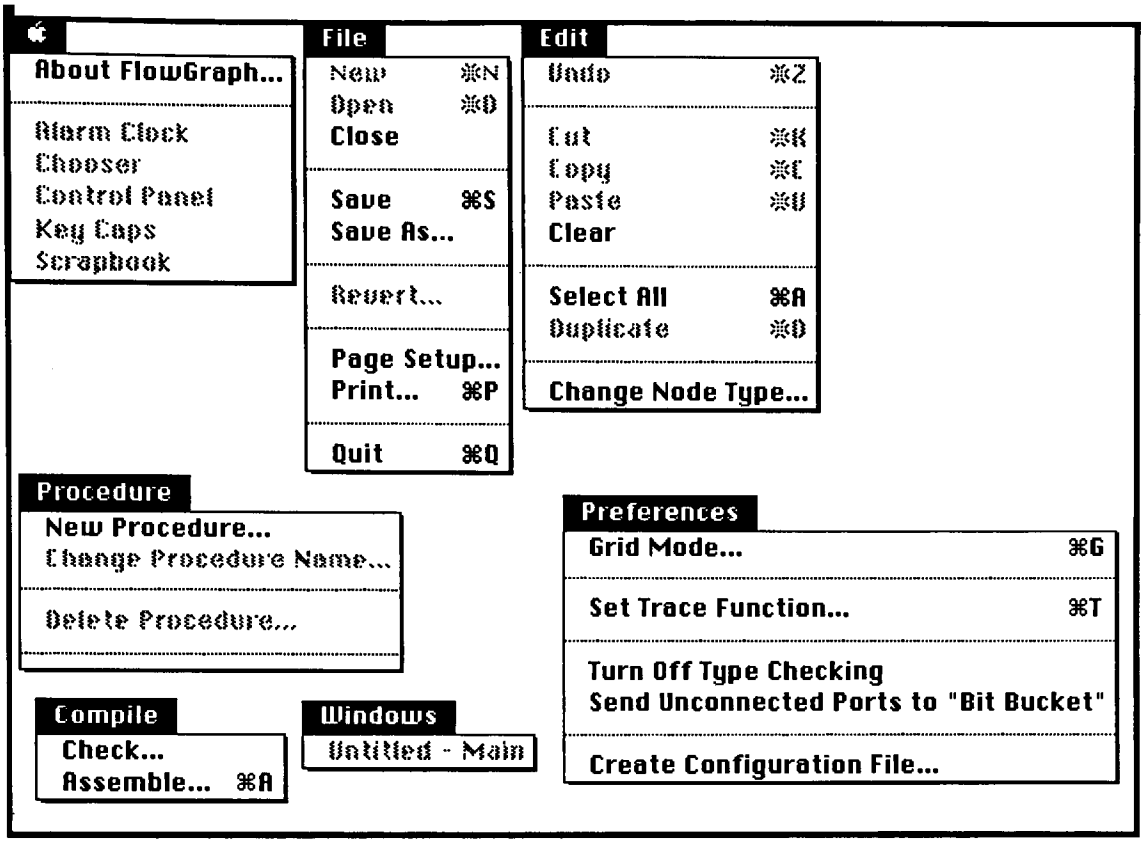


Figure 4.2 - The Menus in FlowGraph

4.2.1.2 The Node Palette

The node palette is a window that presents the user with the nodes that are available to form a graph. Since the node represents the major syntactic element of a data flow graph, the node palette is a method for minimizing errors by giving user only the elements of the language that are correct. When nodes are placed into the graph, the user is further constrained from making errors by active type checking as the nodes are connected. These two methods essentially eliminate syntax errors in the graph language. The user is left with the responsibility for the semantics of the program being written. Figure 4.3 shown the node palette with the various types of nodes identified.

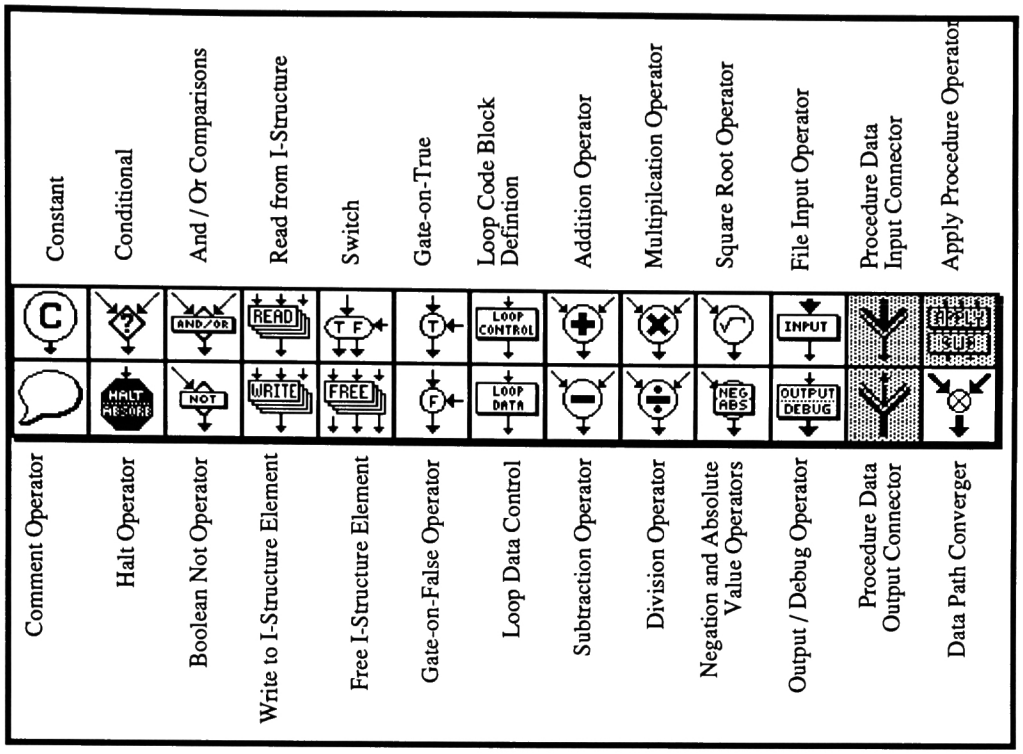


Figure 4.3 - The Node Palette

4.2.1.3 Graph Windows

The graph windows are the method of representing procedure or subprogram elements of a graph program. The user creates new windows to draw graphs that represent procedures that can then be used in other graphs. One window represents the main program. Figure 4.4 shows the components of a graph window in FlowGraph.

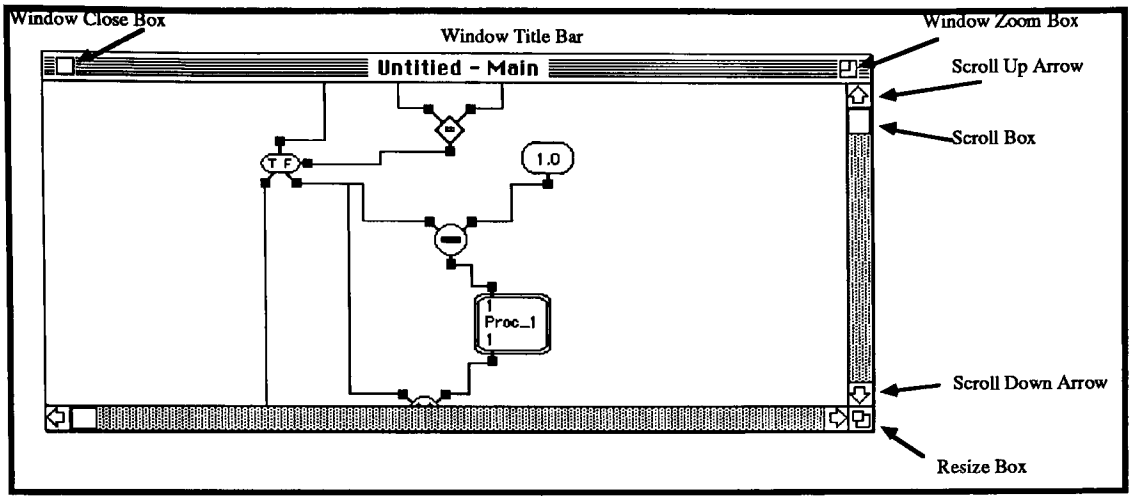


Figure 4.4 - Parts of the Graph Window

Because of the nature of graph programs, it is possible to extend this paradigm to entire programs. That is, data flow graphs, by their nature, can always become elements in other data flow graphs. Thus, entire data flow files could be used as elements in other data flow graphs. Although this is not implemented in FlowGraph, it is appropriate for a full data flow system. This is the technique utilized in the LabVIEW software package, where the data flow paradigm is used to control laboratory instruments and data acquisition systems and to program the data analysis routines. Once programs are developed for a certain application, they can be utilized as sub-elements in other programs.

4.2.1.4 Dialogs and Alerts

The last element of the user interface of FlowGraph are dialogs and alerts. These are windows that are presented to the user to request a specific piece of information, such as the value of a constant, or to present a specific piece of information such as an error message. Dialogs and alerts are usually modal, presenting information at

specific situations. In general FlowGraph is not modal, allowing the user to perform operations in almost any order. Some typical dialogs and alerts are shown in Figure 4.5.

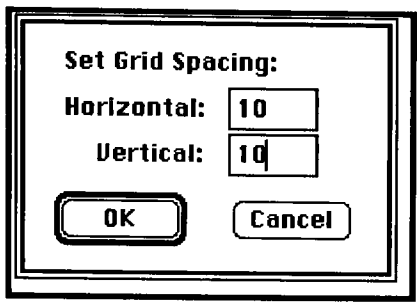


Figure 4.5 - A Typical FlowGraph Dialog Box

4.2.2 Data Structures

A significant part of programming the Macintosh is interacting with the software "managers" in the Macintosh ROM and the System file. These managers make up the operating system and the "user interface toolbox" (referred to as "the Toolbox"). The software developer uses the over 400 procedures and functions in the managers to manipulate nearly a hundred previously defined data structures.

In this section, the data structures developed for FlowGraph are presented. It is important to briefly discuss several data structures that are part of the Toolbox in order to understand all of the elements of the data structures developed for FlowGraph. Throughout this discussion of the implementation of FlowGraph, data structures from the Toolbox will be presented as necessary. The complete description of these are contained in [Rose 1985],[Apple 1986], and [Apple 1988].

4.2.2.1 Significant Toolbox Data Structures

The following are some data structures that are previously defined in the Toolbox and are utilized by data structures developed for FlowGraph. They are presented as Pascal Constants, Types and Records.

The first group defines some of the basic Macintosh data types.

```
TYPE
    SignedByte = - 128..127;  { any byte in memory }
    Byte = 0..255;              { unsigned byte }
    Ptr = ^SignedByte;          { blind pointer }
    Handle = ^Ptr;              { pointer to master pointer }

    Str255 = String[255];     { maximum string size }
    StringPtr = ^Str255;      { pointer to maximum string }
    StringHandle = ^StringPtr;   { handle to maximum string }
```

There are two significant points of the Macintosh architecture illustrated in these data types. First is the concept of a `handle`, which is a `pointer` to a `pointer`. `Handles` are the basis for memory management on the Macintosh. They allow memory management of the heap defining objects on the heap that can be moved around by the memory manager during memory allocation operations. This is discussed in detail in a following section of this chapter. Most objects defined in FlowGraph are done so in terms of `handles`.

Because the Toolbox was conceptualized in Pascal, strings that are manipulated by Toolbox routines are limited to 255 characters. This is because Pascal defines strings using a length value prefixed to the string. Since this is a `byte`-length object, it can have a value no greater than 255.

The next set of data structures define some of the graphic entities on the Macintosh.

```
TYPE
    VHSelect    = (v, h);

    Point = RECORD              { a point }
        CASE INTEGER OF
            0: (v: INTEGER;           {vertical position}
                h: INTEGER);      {horizontal position}
            1: (vh: ARRAY [VHSelect] OF INTEGER);
```

```
        END;

Rect = RECORD      { a rectangle }
        CASE INTEGER OF
            0: (top:      INTEGER;
                left:     INTEGER;
                bottom:   INTEGER;
                right:    INTEGER);
            1: (topLeft:  Point;
                botRight: Point);
        END;
```

These are two of the graphical elements defined by the Toolbox, the `point` and the `rectangle`. These define mathematical entities that exist in the Macintosh graphics package. The integer values define locations on a two dimensional coordinate plane.

4.2.2.2 Data Structures in FlowGraph

There are seven data structures that are fundamental to FlowGraph. These are interrelated in a manner shown in Figure 4.6.

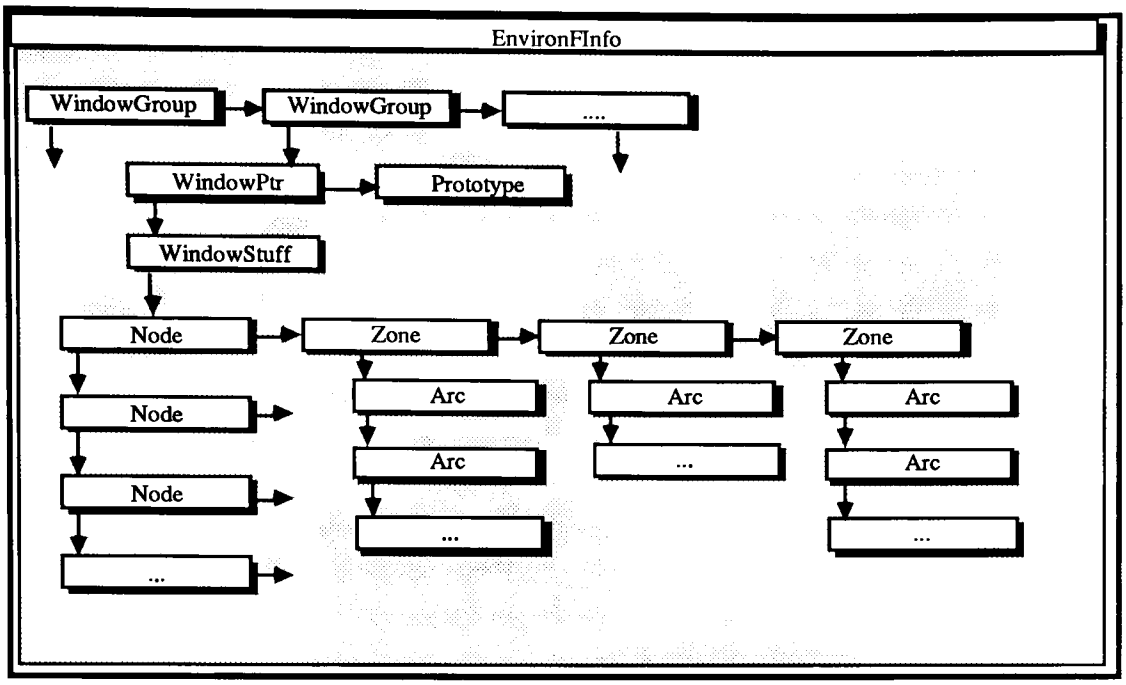


Figure 4.6 - FlowGraph Data Structure Inter-Relationships

Many variables significant to the overall environment of FlowGraph are contained in the environment structure, `EnvironFInfo`. These include the number of windows created for the graph program (i.e. the main program window and any procedure windows), the current window that the user is working in and user preference settings.

```
TYPE
    EnvironFInfo = RECORD
        EFINumWGroups: integer;    {Number of Window Groups}
        EFICurWGroup:      integer;    {Current Window Group}
        EFITraceMode:      longInt;    {Bits for Trace Flags}
        EFIGridH:          integer;    {Horiz. Window Grid Spacing}
        EFIGridV:          integer;    {Vertical Window Grid Spacing}
        EFINumProc:        integer;    {Simulator config file data}
        EFIMatchSize:      longInt;    {Simulator config file data}
        EFIInstSize:       longInt;    {Simulator config file data}
        EFIProcSize:       longInt;    {Simulator config file data}
        EFIInsCount:       integer;    {Simulator instruction count}
        EFIInputCount: integer;    {Simulator file input count}
    END;
```

The WindowGroup structure contains some of the information necessary to support the creation of node from a graph window. It is the link between the graph window and the Prototype structure. It contains flags to indicate that a graph window has been updated in a way that effects the nodes representing that graph window.

```
TYPE
    wGrpPtr = ^WindowGroup;
    wGrpHndl = ^wGrpPtr;

    WindowGroup = RECORD
        wgName:         Str255;{Window Title}
        wgNameChanged: boolean;    {Window Title Has Changed}
        wgUpdate:           boolean;
                                {Window Prototype needs an update}
        wgType:         integer;    {Window Type - Main or Proc}
        wgInsNum:           integer;
                                {Current Simulator Instruct. Number}
        wgPrototype:        PrototypeHndl;
                                {Prototype for the window}
        wgProtoID:      LongInt;        {Prototype ID number}
        wgWindow:           WindowPtr; {the Window}
    END;
```

The Prototype structure contains the information about the basic attributes of any node that has either been previously defined by FlowGraph or has been defined in a graph window as a procedure by the user. The Prototype structure indicates the type of the node (one of the previously defined types or user defined), the size of the rectangle required to contain the node representation (i.e. its icon) when it is drawn, the WindowGroup that the Prototype was created from (if it was user-defined) and a description of all of the ports that are associated with the node.

```
TYPE

    PrototypePtr = ^Prototype;
    PrototypeHndl = ^PrototypePtr;


    Prototype = RECORD
        PType:          integer;    {Type of Node being Prototyped}
        PRect:          Rect;       {Screen Area Required by Node}
        PSource:        integer;    {WindowGroup for the Node}
        PZoneList: ZoneHndl;  {Ports (Zones) for the Node}
    END;
```

Another structure associated with each graph window is the WindowStuff structure. It contains values such as the current graph window origin. The current graph window origin is the point in the coordinate space defined for the entire graph that is currently the point in the upper left corner of the graph window. The WindowStuff structure also contains a handle to the first node in the graph being drawn in the window as well as some miscellaneous counters.

```
TYPE
    WSPtr = ^WindowStuff;
    WSHandle = ^WSPtr;

    WindowStuff = RECORD
        theOrigin: Point;       {Current Window Origin}
        StartNode: NodeHndl;    {Start of Node Linked List}
        InputCount:     integer;
                        {Number of Param. Inputs in Proc. Window}
        OutputCount:    integer;
                        {Number of Param. Outputs in Proc. Window}
        WGIndex:        integer;
                        {Index of the Window Group for this Window}
    END;
```

A Node data structure exists for each node in every graph of the program. A Node is an element of the linked list that represents all of the nodes in each the graph in memory. It contains the type of the node, the rectangle that contains the node icon, and a handle to the linked list of port descriptions for the ports associated with the node. A Node also contains other information and a handle linking it to the next Node in the list. Note that the linked list of Nodes is built up as nodes are added to the graph by the user. This list does not attempt to represent the actual graph being drawn. When a Node is created, the information in the Prototype for that type of node is used to initialize the fields of the structure.

```
TYPE
    NodePtr = ^Node;
    NodeHndl = ^NodePtr;

    Node = RECORD
        NType:          integer;        {Node Type}
        NRect:          Rect;           {Location in Graph}
        NSource:        integer;
                                {Window Group that node represents}
        NProtoID:       LongInt;        {Prototype ID number}
        NInsNumber:     integer;        {Simulator Instruction Number}
```

```
    NSelect:       boolean;        {Node Select Flag}
    NZoneList: ZoneHndl;       {Port (Zone) linked list}
    NSubIONumber: integer;         {Procedure IO number}
    NComTitle: StringHandle; {Comment Title}
    NComText:      StringHandle; {Comment Text}
    NValue:    real;           {Constant Value}
    NBoolean:      boolean;        {Constant Type}
    NID:           LongInt;        {Node ID Number}
    NNextNode: NodeHndl;       {Next node in list}
END;
```

The ports of a node are represented by structures called Zones. A Zone contains information describing the port, including the rectangle that contains it in the graph space, the data type of the data moving through the port, the direction that the data flows through the port (i.e. whether it is an input port or an output port) and other miscellaneous information. A Zone is also an element of a linked list, where the list represents all of the ports on a particular node. When a Prototype structure is created, the Zone list is also created. This list is then copied to any Node created from the Prototype.

```
TYPE
    ZoneDirection = (INPUT, OUTPUT);

    ZonePtr = ^Zone;
    ZoneHndl = ^ZonePtr;

    Zone = RECORD
        ZRect:         Rect;           {Screen Area for Port}
        ZPoint:    Point;          {Arc Connect Point}
        ZDirect:       ZoneDirection;{Data Flow Direction}
        ZDataType: integer;        {Port Data Type}
        ZPortID:       integer;        {Procedure Node Port Number}
        ZSelect:       boolean;        {Selection Flag}
        ZNode:         NodeHndl;       {Handle to Owner Node}
        ZArcList:      ArcHndl;        {Start of Linked List of Arcs}
        ZID:           LongInt;        {Unique Zone ID Number}
        ZNextZone: ZoneHndl;       {Handle to Next Zone in List}
    END;
```

The last principle structure in FlowGraph is the Arc. This structure represents the connection of ports in the data flow graph being drawn by the user. An Arc contains handles to the Zones that it is "connected" to. An Arc is also a linked list element. The linked list is maintained by the output port that the arc originates from. In this

manner, FlowGraph represents the duplication of data in the data flow graph for a single output port to multiple input ports.

```
TYPE
    ArcPtr = ^Arc;
    ArcHndl = ^ArcPtr;

    Arc = RECORD
            ASelect:        Boolean;    {Selection Flag}
            AStartZone:     ZoneHndl;   {Staring (Output) Zone}
            AEndZone:       ZoneHndl;   {Ending (Input) Zone}
            ANextArc:       ArcHndl;    {Next Arc in Linked List}
        END;
```

These are the principle data structures in FlowGraph. In the section on Program Structure in this chapter, the organization of the code that is used to create, manipulate, and delete these structures is described.

4.2.3 Program Organization

FlowGraph is organized to provide the user with as much flexibility as possible. It accomplishes this by implementing an event oriented approach to its structure. Events are generated by the actions of the user and as consequences of actions by the user. The principle program organizational "structure" is the event loop.

4.2.3.1 The Event Loop

The Event Loop is the central dispatcher of activities in FlowGraph. When FlowGraph is launched, it first initializes its global variables and then drops into a loop that waits for things to happen. This is the event loop. The Macintosh Toolbox provides data structures and procedures to support this event loop concept. The major Toolbox types and procedures related to event handling are listed below.

```
CONST
    NullEvent = 0; { no other type of event occurred}
    mouseDown = 1; { the mouse button went down }
    mouseUp = 2;       { the mouse button went up }
```

```
    keyDown = 3;        { a keyboard key went down }
    keyUp = 4;      { a keyboard key went up }
    autoKey = 5;        { a key is being held down to autorepeat }
    updateEvt = 6; { a windows contents needs to be redrawn }
    diskEvt = 7;        { a disk was inserted or ejected }
    activateEvt = 8;    { a window has been activated/deactivated }
    networkEvt = 10;    { a network event occurred }
    driverEvt = 11;     { a driver event occurred }
    app1Evt = 12;       { an application-defined event occurred }
    app2Evt = 13;       { an application-defined event occurred }
    app3Evt = 14;       { an application-defined event occurred }
    app4Evt = 15;       { an Operating System event occurred }

TYPE
    EventRecord = RECORD
        what:           INTEGER;    { the type of event (see above) }
        message:        LONGINT;    { info. specific to event type }
            when:       LONGINT;    { time when the event occurred }
        where:          Point;      { global point of mouse cursor }
        modifiers: INTEGER;     { other event information }
    END;
```

The Toolbox function `GetNextEvent` is the principle part of the event loop. It is called once each time through the loop and returns an `EventRecord` that describes the event that occurred. FlowGraph checks the `EventRecord.what` field and then calls the handler for that type of event. When the handler returns, the loop repeats until a flag is set indicating the user wants to quit the program.

The event loop also performs three other "housekeeping" functions. Each time through the loop, a time slice is given to any drivers or Macintosh desk accessories that the user has activated. Also, the position of the cursor on the screen is checked and it is changed to the appropriate shape. Finally, some periodic memory management operations take place.

4.2.3.2 Mouse Events

Every time the user presses the button on the mouse, an event is generated that FlowGraph must interpret. The mouse event handler is probably the most extensive part of FlowGraph. This follows from the fact that 90% of the operations in

FlowGraph are initiated by the mouse. The basic structure of the mouse event handler is outlined here.

First, the location of the cursor on the screen is determined and the "system area" that the mouse is located in is identified. The system area is defined here to be either the menu bar, the content region of a FlowGraph window or one of the standard window parts (close box, re-size box, zoom box or title bar) in some arbitrary window. If the mouse was in the menu bar, the menu handler is called. The menu handler determines which menu and menu item were selected and calls the appropriate procedure. If the mouse event occurred in some part of a window and the window was inactive, it is activated. If the window is currently active (indicated by the highlighted title bar) then handlers are called depending on which part of the window was "hit". The standard parts of any window are handled the same way, in terms of the code necessary to close, zoom, drag or re-size a window. If the mouse event occurred in the "content" region of window, more checking is required.

The content region of a window is that portion of a window that is application-dependant on its contents. In this case, both the node palette and graph window contain content regions handled by FlowGraph. If the mouse event is in the content region of the node palette window then a handler is called that eventually creates a new node in a graph window. If the mouse event occurs in the content region of a graph window, it must now be determined whether the event occurred in the scroll bars or in the actual graph area of the window. This is because scroll bars are actually controls that are independent of the window and are considered part of the application that is not handled in any standard

fashion by the Macintosh toolbox. This is in contrast to window parts such as the close box or the title bar, which is used for dragging the window around the screen.

If the event occurred in the actual graph area of the graph window, it is again necessary to locate the exact point of the event relative to the existing objects in the graph. It now also becomes necessary to assess the condition of certain state modifiers such as the shift key on the keyboard. It now becomes necessary to traverse the entire node list of the window to determine if the event took place inside the rectangle bounding the node. If the location was in a node, the zone list for the node is traversed to find whether the point was in one of the node's ports.

A mouse event in a zone is then checked for three possible conditions: (1) was the mouse button "clicked" (pressed and released) in the zone to indicated selecting that zone; (2) was the mouse button "double clicked" to indicate a request for information about the zone (such as its data type); or (3) is the mouse button still being pressed and the mouse being dragged out of the zone, indicating the users desire to create an arc. It is then necessary to check, when the mouse button is released, exactly where the mouse was it when it was released and to again traverse the window's node list and, if necessary, the zone list of a node. Once a second zone has been identified, error checking must be done. Depending on the outcome of the error checking, either the error message is displayed to the user or the arc is created.

The previous description of a mouse event demonstrates the type of programming required in a highly interactive program such as FlowGraph. The mouse event is

probably the most complex to be handled because of the amount of semantic information that must be derived from each event. The description above really describes the activities that occur when a user wants to draw an arc between the ports of two nodes. There are many other mouse-driven events that are not described here, such as node creation and dialog handling but this gives a flavor for the activity involved. Of course, once an arc has been created, as in this example, control returns to the event loop for handling the next events.

4.2.3.3 Update Events

The Macintosh system uses update events to handle drawing to the screen. The handler for update events is designed to look at the data structures created by user and recreate the screen image from those data structures whenever an update event is generated. This supports features such as overlapping windows. When the ordering of windows changes on the screen or a window is dragged over other windows, uncovering some parts of the background windows, update events are generated. Part of the information that is available from an update event is area of the window that needs updating, so the update handler can check to see if anything it can draw is in that area.

As an example, when the user creates a node in a graph window, the node data structure is created in memory. Rather than drawing on the screen directly at that time, the rectangular area on the screen that contains the node is "invalidated". Any other operations such uncovering the window may invalidate additional areas of the window. An update event is posted for any particular window when some area of the window is invalidated. Any additional areas that become invalidated before the update event is handled are just added to the update region of the window. Once the update is processed, these areas become "valid" again and the update region is cleared.

For FlowGraph, an update event in a graph window causes the update handler to traverse the linked list of nodes for the window being updated. As it checks each node, if the rectangle containing the node intersects with the window's update region, the node and its zones (ports) are redrawn. Then a bounding rectangle is calculated for each arc in the graph and it is checked for intersection with the window's update region. The clipping region is set to the windows content area (minus the scroll bars) to prevent drawing outside of the window.

4.2.3.4 Activate Events

Activate events are generated for a window whenever it is made active or inactive. There is only one active graph window on the screen at any time. It is indicated by the highlighting of its title bar. Activate events usually come in pairs, one for the window being made active (an activate event) and one for the window being made inactive (a deactivate event).

In FlowGraph, deactivate events are used in procedure windows to make sure the `Prototype` structure representing the window is up to date. Both activate and deactivate events set flags indicating the state of the window to the program.

4.2.3.5 Keyboard Events

The last type of event handled routinely by FlowGraph are keyboard events. In FlowGraph, keyboard events are used mainly for two purposes: clearing selected items and issuing menu commands. When a keyboard event occurs, if the backspace key was pressed, the Clear command from the Edit menu is executed to clear any selected nodes or arcs. If the command key on the keyboard was being pressed when the keyboard event was generated, FlowGraph attempts to process this as a menu command. Keyboard menu equivalents are given in the menus next to the command name.

All other keyboard handling, such as for dialogs, is handled by the system software.

4.2.4 Program Structure

The previous section described the types of logic flow in FlowGraph and some of the principle data structures. This section briefly outlines the code blocks that were written for FlowGraph and their overall organization.

FlowGraph was written in an extended version of Pascal. The principle extension of this version of Pascal is the availability of the UNITS construct that allows separately compiled blocks of code. The structure of the UNITS construct is:

```
UNIT unitname;
    INTERFACE
        USES external_unitnames;
        CONST
            externally_available_constants;
        TYPE
            externally_available_types;
        VAR
            externally_available_variables;

        Procedure_and_function_headings;

    IMPLEMENTATION
        CONST
            internally_available_constants;
        TYPE
            internally_available_types;
        VAR
            internally_available_variables;

        Procedure_and_function_declarations;

END.
```

FlowGraph consists of twenty-seven different UNITs, broken roughly into functional units that operate on a particular set of data structures or perform some set of operations.

GlobalDataTypes.pas,CompileTypes.pas,PaletteGlobals.pas: These units contained most of the global constants, type declarations, and variables for FlowGraph.

WindowStuff.pas: This unit contains all of the routines to manage the `WindowStuff` data structure.

MenuUpdate.pas: This unit controls the appearance of the menus, including enabling and disabling the appropriate menu items given a certain state of the program.

WindowGroup.pas: This unit contains all of the routines to manage the `WindowGroup` data structure.

DialogSupport.pas, DialogStuff.pas, InfoDialog.pas, StructDialog.pas: These routines manage the interaction of the user with the various dialogs in FlowGraph.

Preferences.pas: This unit manages the preference settings of the user.

NodeStuff.pas: This unit manages the creation, deletion and management of the `node`, `zone`, and `arc` data structures.

EditOperations.pas: This unit contains the code for clearing selected graph items and also changing the node types of selected nodes.

WindowMaker.pas: This units controls the the creation of windows.

FileHandler.pas: This unit contains all of the code for saving and reading graph files.

SubPrototype.pas: This unit is used to create the icons representing procedure windows.

IOPrototype.pas: This unit is used to create the icons for output operators and debug operators.

InitStuff.pas: This unit initializes all of the system variables and calls a myriad of other units to set up the environment of FlowGraph when the program is launched.

NodeDrag.pas: This unit handles all of the operations involved in dragging a node.

GraphStuff.pas: This unit handles all of the graph drawing operations as well as the semantic analysis of the principle mouse operations.

PaletteStuff.pas: This unit manages the creation and use of the node palette.

WindowControls.pas: This unit controls the scrolling operations of the graph windows.

PrintGlobals.pas: This unit manages the printing of graphs to the printing device.

CompPass1.pas: This unit is the first pass of the graph "compiler", used to do type checking and some other global error checking. It also numbers the nodes in preparation for the simulator assembly file creation.

CompPass2.pas: This units creates the simulator assembly language files from the graph.

EventHandler.pas: This unit contains the main event loop and many of the main event handlers.

FlowGraph.pas: This is the main program, which essentially exists to initialize the system and then start the main event loop. See below.

```
PROGRAM Flowgraph;
{$I-}
    USES
        MacPrint, ROM85, EventHandler, InitStuff;

BEGIN
    InitThings;
    MainEventLoop;
END.
```

Beside the Pascal UNITs in FlowGraph, many of the visual "objects" in FlowGraph are created as "resources". Resources are a major concept in Macintosh programming and provide the programmer with the ability to separate a programs logic and code from the data needed by the program. In the case of FlowGraph, all of the menus,

icons, dialogs and windows are contained in resources. This allows them to be edited or modified separately from the code. (Incidentally, the code of a program is keep as a resource of type CODE). One feature of resources is the ability to substantially localize a program for use in another language without ever changing the code.

4.2.5 Other Issues

There were two fundamental issues that were encountered in the development of FlowGraph. First was the issue of the programming language to be used. Originally, FlowGraph was to have written in C. However it became obvious that since all of the Toolbox for the Macintosh was written in Pascal, the translation of the Pascal interface into C was to be a stumbling block. Also, the Pascal language environment that was chosen provided a very good source level debugger and an excellent integrated environment. However, the sacrifice that was made was in the amount of code that needed to be generated. Since Pascal is more verbose than C, FlowGraph became quite large in terms of the code written. FlowGraph consists of approximately 10,000 lines of code. Also, some execution speed was probably lost.

The second issue became the priority of providing good memory management. In FlowGraph, there is a high degree of segmentation in the code such that unneeded code could be purged from memory if the space on the heap was needed. Each time through the main event loop, the code segments are marked as purgable, although they are only purged if the memory is actually needed.

Another method of memory that is utilized is the extensive use of handles, which are pointers to pointers. Almost all data structures are created with a call to the function `NewHandle`, which is very similar to the standard Pascal function `New`, except that a pointer to a pointer is returned. The pointer that is returned by `NewHandle` points to a master pointer block location in memory. The Macintosh OS manages the heap memory by moving objects in memory through the master pointer block. When the

programmer is accessing the contents of a handle, the handle can be designated as locked so that the OS won't move it. Once the handle is no longer being accessed, it is unlocked and allowed to be moved if the memory manager requires a large block of contiguous free space. Since the master code block never moves, the programmer can always access the contents of the handle, even if it has been moved in memory. By using this feature of the Macintosh OS, FlowGraph can best utilize the available heap memory to store all of the data structures used in the graph being created.

Chapter 5

Conclusions

5.1 Introduction

This chapter presents the conclusions reached in this thesis. These conclusions are presented with regard to the viability of visual programming languages, the use of a visual languages for data flow programming and the general development of a visual language. Some suggestions for improvements in FlowGraph are also made.

5.2 The Viability of Visual Programming Languages

Progress in the development of visual programming environments has been quite rapid in the last three years alone. This is being driven substantially by two areas of development, the cost and availability of graphics hardware and development of object-oriented programming languages. Object-oriented languages seem particularly suited to developing visual programming environments since the concepts such as object attributes and inheritance are usually a key part of visual languages. It would also make the programming of visual languages that manipulate objects with a visual attribute much more logical in terms of the organization of the program.

Visual languages can play a large part in providing an environment that allows the user to provide information in a more natural, tactile manner and to allow the user to deal with problems in a manner that may be more closely related to the original problem domain. As an example, ARK provides simulations of physical phenomenon that substantially extent the users ability to control the conditions of the phenomenon and to be able to interact with the phenomenon.

Another example is LabVIEW, which provides the paradigm of "wiring" experiments and data acquisition equipment and then controlling them with collections of knobs and sliders. It can provides a substantial increase in productivity and understanding since the user is not required to learn fundamentally new concepts to solve problems. This is the principle motivation behind all visual programming languages.

5.3 The Use of Visual Languages in Data Flow Programming

Data flow programming provides an excellent opportunity to develop a limited visual programming environment. This is because the objects being manipulated in a data flow graph have a limited set of attributes that must be provided in the environment. Visual environments such as ARK and LabVIEW must, of their nature, provide a much greater collection of object attributes to provide their functionality. Data flow programming in a visual language can be reduced to the connection of objects that have only one or two attributes: nodes define an operation on the data and may define an acceptable data type for the operation. In the data flow simulator, there is no requirement for data typing, although this may limit some problems that can be developed. Certainly extensions can be added to a data flow graphing system that can become more complex, such as execution simulation in a graphical manner.

5.4 Issues in the General Development of Visual Languages

There are a number issues that must be resolved in the development of any visual language system. The principle issue is that of the paradigm to be used to represent objects or concepts in the problem domain that the visual language is targeted for. ARK uses buttons and sliders like LabVIEW but the principle object seems to be the interactor, that "object" that defines a particular attribute of the physical environment being simulated, such as gravity. In LabVIEW, the principle concept is that

of circuit wiring and components that can be built up to any level from some primitives.

The next issue is the need to develop an entire user interface management system to implement the visual language. Visual language semantics are often driven by what the user does with or to a particular object. It becomes the burden of the UIMS to associate meaning to these actions.

Because the user is actively participating with a visual language, issues such as speed of interaction and rendered detail of the objects may become a critical part of the language. In FlowGraph, for example, speed is much more critical because the user is almost constantly interacting with a multitude of objects on the screen.

Another issue is completeness of the environment. The users will demand a complete set of operators for object handling and manipulation. The development of these tools in the language is a major part of the "language development" and they are critical to supporting the paradigm that is being implemented.

5.5 Improvements on FlowGraph

The FlowGraph environment is certainly incomplete, especially in terms of editing tools. Full object cut, copy and paste operations need to be provided. Operational Undo commands are also important additions to FlowGraph. The speed of the interactions could be increased as well by optimizing the handler calls.

The ultimate improvement would be to incorporate the simulator directly into the FlowGraph code and to support many of the aspects of program visualization to assist in programming data flow systems.

5.6 Final Conclusions

Developing a visual programming language requires an extensive development of user interface techniques which are not often simple or straightforward to implement. There is an extensive overhead involved in implementing a visual language to remove many of the normal programming burdens from the user.

Appendix A

FlowGraph Users Manual

A.1 Introduction

FlowGraph is an experiment in the design of a visual programming language. FlowGraph allows the user to write a data flow program through the use of a data flow graph that the user creates and manipulates on the screen. The data files created by FlowGraph are text files containing the textual representation of the data flow graph. This is written in the assembly language format of a data flow simulator at RIT. See [Benjamin 1988].

FlowGraph was written on a Macintosh computer and takes advantage of many of the aspects of the user interface system available on the Macintosh. FlowGraph is implemented through an interface that includes movable windows, pull-down menus, and a palette of graph primitives.

This appendix describes some details of using FlowGraph that are not described in detail in either Chapter 3 or Chapter 4 of this thesis. Please refer to those chapters first and then look here for more detail.

A.2 General Operation

A.2.1. The Parts of FlowGraph.

FlowGraph consists of a number of principle components: menus,the node palette, graph windows, and operations with the mouse. Menus provide commands that effect the overall environment of FlowGraph. The node palette provides the primitive operators for FlowGraph. Graph windows are used to access the graphs and, in themselves, represent procedural structures. Other operations in FlowGraph utilize

the mouse to a high degree; for moving nodes on the graph, for drawing arcs, for selecting graph parts and for querying parts for information. Each of these areas will be discussed here in the following sections in detail.

A.2.2 Operating Environment

FlowGraph executes on a Macintosh computer with the following configuration:

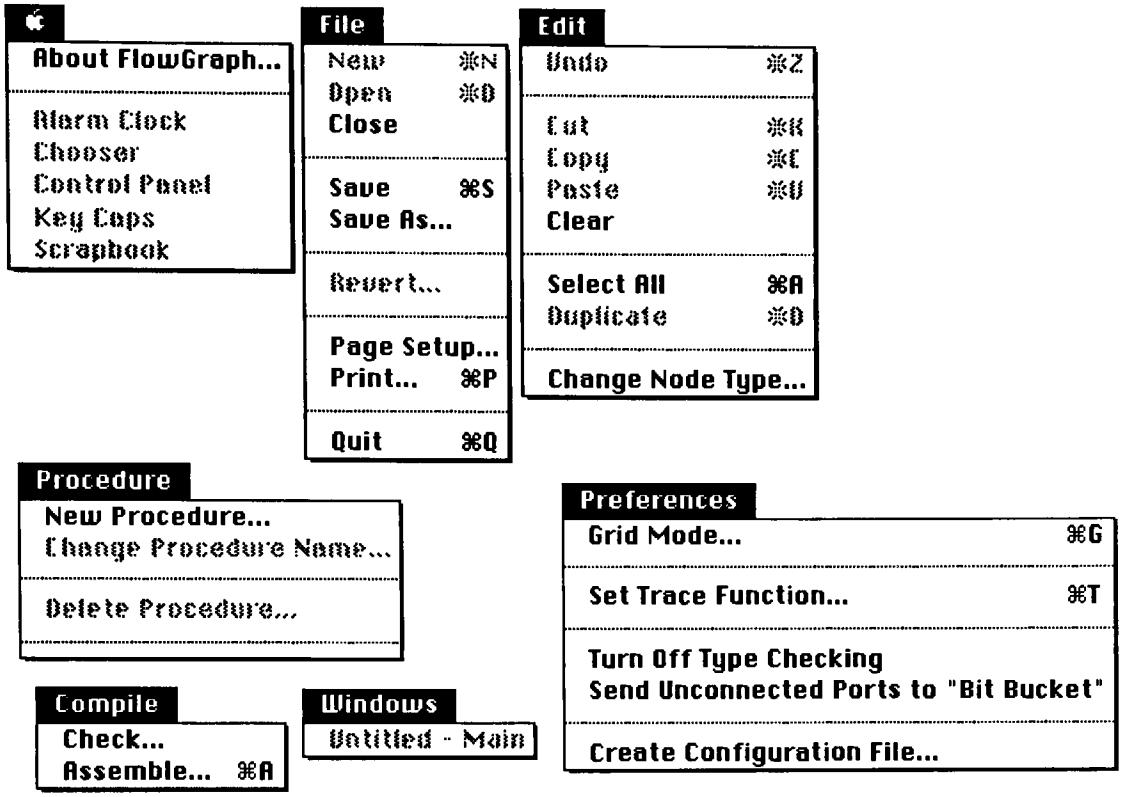
- 1 Megabyte of RAM (or greater)
- 128K Macintosh ROM (or greater)
- System Update 5.0 (System File version 4.2 or greater)

FlowGraph was developed and tested on a standard Macintosh Plus with a Hard Disk.

A program for transferring the simulator output text files to the host system of the simulator is also required.

A.3 FlowGraph Menu Commands

In this section, each of the FlowGraph menus are listed and each of the commands in the menus are described.



A.3.1 The Apple Menu

The Apple menu contains the About FlowGraph... Command and the desk accessory drivers available in the current System file.

About FlowGraph...

This menu item gives information about FlowGraph, including the author, the motivation and the version number. It also shows the amount of available memory remaining to FlowGraph. Since FlowGraph keeps all of the information about the graph in memory, this number will get smaller as the size of the graph.

Desk Accessories

The Apple Menu is also the point of access for any desk accessories loaded into the system file. See the Macintosh owners manual for details on desk accessories. Whenever a desk accessory is active, the Edit menu commands Undo, Cut, Copy, Paste and Clear are enabled. Most other FlowGraph menu commands are disabled whenever desk accessories are operating.

A.3.2 The File Menu

The File Menu provides all of the file management commands, including file creation and printing of graphs.

New

The New command is used to create a new graph program. It is enabled when the current graph program file has been closed.

The New command is automatically entered when FlowGraph is started, if no data files were passed on start-up. Data files can be passed to FlowGraph on start-up by selecting an existing FlowGraph file from the Finder and either double clicking on the data file icon or using the Finder's Open command.

Open

The Open command is used to open existing FlowGraph data files. It is enabled when the current graph program file has been closed.

When the Open command has been selected, it will present a dialog box that will allow the user to select any available FlowGraph data files. It uses the standard Macintosh file dialog, which can be used to navigate through the Macintosh file

system to find FlowGraph files. Only FlowGraph files are displayed in the Open dialog box.

Close

The Close command is equivalent to the Close box on any currently open window. If the Close command is executed when a Procedure window is front-most, the window will be hidden and the next window on the screen will be displayed. To reopen the procedure window, select the window name in the Windows menu.

If the front-most window is the main program window, the Close command will attempt to close the file. If unsaved changes have been made to the file since it was opened, a dialog box will be presented asking whether changes to the file should be saved. This dialog box can also be used to cancel the command if the user does not really want to close the file.

Save / Save As...

The Save commands are used to save the contents of the graph to a file for later use by FlowGraph. The Save command will save the graph to the currently open graph file. If the file is a new file, it will execute a Save As command.

The Save As command will present the standard Macintosh Save dialog box to allow the user to specify a file name to save the current contents of the graph file. The user will be warned if a file name is specified that already exists.

Revert

The Revert (to the contents of the file as it was last saved) has not been implemented. To accomplish this task, close the file and do not save the changes. The open the file again.

Page Setup...

The Page Setup command is used to set up certain aspects of the current printing device.

For the Apple ImageWriter dot matrix printer, set the command Tall Adjusted for the best printing of FlowGraph graphs.

Print...

This command is used to print the graph in the front-most window. For the Apple ImageWriter dot matrix printer, set the Faster mode for the best printing of FlowGraph graphs.

Quit

This command quits FlowGraph. If unsaved changes have been made to the file since it was opened, a dialog box will be presented asking whether changes to the file should be saved. This dialog box can also be used to cancel the command if the user does not really want to quit FlowGraph.

A.3.3 The Edit Menu

The Edit Menu provides a number of different editing commands for editing the graphs being created.

Undo

The Undo (the last command or action) command is not implemented in FlowGraph. It is available when desk accessories are active.

Cut

The Cut (the selected objects to the clipboard) command is not implemented in FlowGraph. It is available when desk accessories are active.

Copy

The Copy (the selected objects to the clipboard) command is not implemented in FlowGraph. It is available when desk accessories are active.

Paste

The Paste (the current objects in the clipboard to the graph) command is not implemented in FlowGraph. It is available when desk accessories are active.

Clear

The Clear command deletes any selected nodes and/or arcs from the graph in the current window. Pressing the Backspace key on the keyboard will also call this function. Deleting a node will also clear any arcs connected to it.

Select All

The Select All command will select all nodes and arcs in the current graph window.

Duplicate

The Duplicate (the selected node) has not been implemented. Use the node palette to create all nodes.

Change Node Type...

This command is used to change a selected node to a different node in its "Equivalent Group". There are six Equivalent Groups in FlowGraph.

- Conditionals - <,>,≤,≥,=,≠
- Math - +,-,*,/
- Boolean - And, Or
- Halt, Absorb
- Absolute Value, Negation
- True Gate, False Gate

When a node in one of these groups has been selected and the Change Node Type command is executed, a "mini-palette" is displayed in the lower left corner of the screen with all of the nodes in the proper group. By clicking on an icon in the mini-palette, it will be substituted for the selected node in the graph.

If the selected node is not in an equivalent group, a message will be displayed.

A.3.4 The Procedure Menu

The Procedure menu contains the commands to create, rename, delete and select procedure graph windows.

New Procedure ...

The New Procedure command creates a new procedure graph window and makes it the front-most window on the screen. It also place the procedure name at the end of the Procedure menu and at the end of the Windows menu. When the name is selected from the Windows menu, that window is made the front-most. When the name is selected from the Procedure menu, the procedure node can be placed in the graph with the Apply operator on the node palette.

Change Procedure Name...

This command will be enabled if a procedure graph window is the front-most on the screen. It will present a dialog box allowing the user to name the current Procedure. That name will appear in the window title box.

Delete Procedure

This command will be enabled if a procedure graph window is the front-most on the screen. When selected, it will delete the procedure and its graph window. It will present a dialog box allowing the user to confirm or cancel the command.

Procedure Names

Any procedures created in the current graph file will be displayed in the Procedure menu. By selecting a procedure name from the menu, that procedure will become the procedure that will be placed into a graph with the Apply operator in the node palette. It is good practice to make sure all of the input and output nodes for the procedure have been defined before it is "applied" into a graph. If the input or output nodes are changed in the procedures graph window, arc connections to any nodes representing that graph will become invalid.

A.3.5 The Preferences Menu

The Preferences menu is used to set user operating preferences in FlowGraph. Some of these preferences apply to the graph window and some apply to the simulator assembly language file or its compilation process. A Processor Configuration File for the simulator can also be created from this menu.

Grid Mode...

This command allows the user to create an invisible grid in a graph window that nodes will "snap" to when they are created or moved. This will assist in creating a cleaner, more organized graph.

The dialog that is presented allows the setting of the grid spacing, in pixels, for both the horizontal and the vertical directions. The grid setting applies to any graph window in a graph file while it is set. To "turn off" the grid settings, set each value to 1.

Set Trace Function...

This command displays a dialog box that allows the setting of the simulator trace operations that are desired. See Appendix B, section B.6, for more information.

Turn Off Type Checking

Normally, FlowGraph checks to see that arcs connect ports that have the same types. However, the simulator converts every token value to a real number. Boolean True values are represented as a value of one while the boolean False is a zero. If the programmer wants to take advantage of this part of the simulator implementation, the type checking can be turned off during the creation, checking and assembly of the graph. A check mark appears beside the menu entry when this mode is in effect.

Send Unconnected Ports to "Bit Bucket"

Normally during the graph checking process and the assembly process, all ports on all nodes must be connected. This setting allows a short hand operation by compiling any unconnected output ports to the null address destination, the "bit bucket". Normally, if the null address is desired, the output port should be explicitly connected to an Absorb node. This will allow the complete type checking process to occur.

Create Configuration File...

This command presents a dialog box to allow the setting of the four values required in the simulator Processor Configuration File. The default values of this file are presented in the dialog box and can be changed by the user. A discussion of these values is given in Appendix B, section B.7.1.

When the OK button is clicked, a standard Macintosh Save File dialog box is presented with the file name preset to config, the name required by the simulator. Clicking the Save button will save the properly formatted processor configuration file.

A.3.6 The Compile Menu

The Compile menu is used to create the simulator assembly language data file and the input value data file.

Check

This menu item causes some consistency checks to be performed on the data flow graph, such as checking for data type consistency and the connection of all ports on the nodes in the graph. These same consistency checks are performed when the file is "assembled".

Assemble...

This item starts the process of creating the Simulator assembly language text file from the data flow graph. The user shall be presented with some Standard File System dialogs to name the two files created. The assembly language file has the text ".prog" appended to the main graph window name, while the file containing the template for the input data values to the simulator has the suffix ".dat".

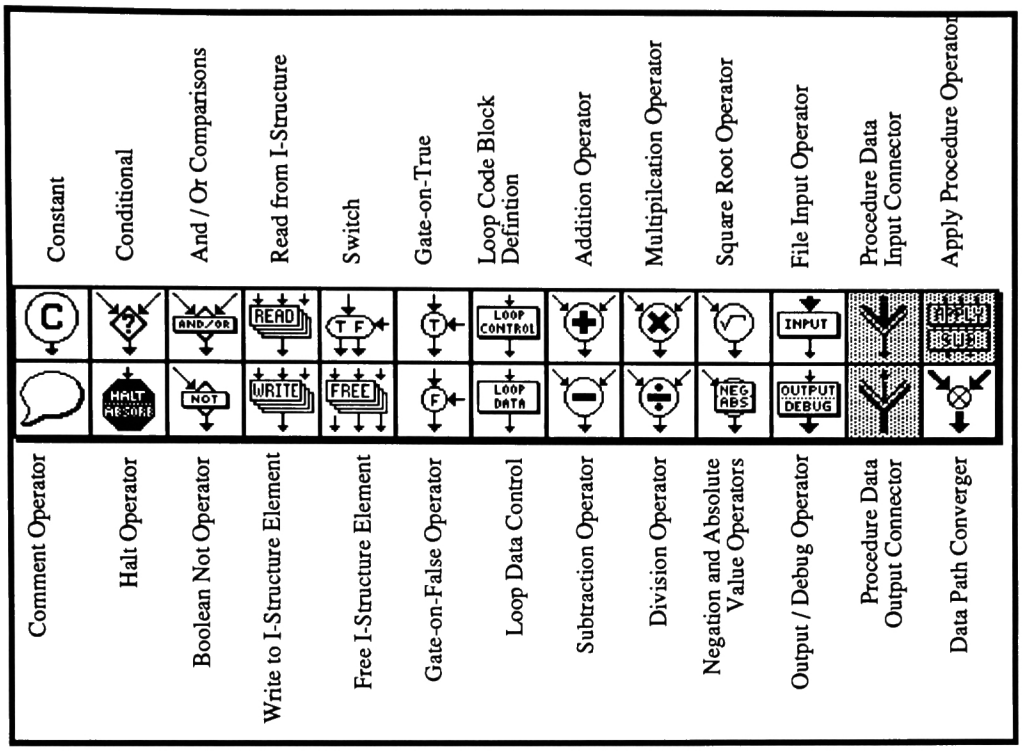
A.3.7 The Windows Menu

The Windows menu is used to help manage the various graph windows that can be created in FlowGraph.

FlowGraph supports two kinds of windows, a single main graph program window and multiple procedure graph windows. The main graph program window is always listed first in the Windows menu. The menu item for the window that is currently front-most on the screen is disabled. Selecting a window name from the Windows menu will make that window the front-most window, opening a closed procedure window if necessary.

A.4 The Node Palette

The node palette is the device used in FlowGraph to create nodes that are used in the graph windows. Each of the operators available in the data flow simulator are presented on the node palette. The two exceptions are the use of the procedure window and the Apply operator, which generate the four procedure operators, and the ability to draw multiple arcs from an output port rather than using the link operator directly.



To use the node palette, place the cursor on the node in the palette to placed on the graph. Press and hold the mouse button and drag off an outline of the node. Place the outline on the graph at the desired position and release the mouse. The node will be placed on the graph at that location. If the icon on the node palette represented an "equivalent group" (see section A.3.3 above), a "mini-palette" will be displayed in the lower left corner of the screen. This is used to select the type of node to be placed in the graph.

The node palette is a fixed window placed at the bottom of the screen. It cannot be moved or have its size changed. It can be covered by graph windows if they are dragged over the palette or enlarged to full size. To get access to the node palette, the graph windows must be moved or resized such that the required node icon is visible.

A.4.1 A Description of Each Node Type

Each of the nodes available through the node palette are described below.

Nodes are structures with a number of attributes. Besides the type of operation that the node performs, its attributes include the number and type of ports associated with each node. The input ports are located on the top and right sides of a node. They are numbered from the top left to the right side in a clockwise fashion. The output ports are located along the bottom of the node and are numbered from left to right.

CONSTANT - The value of a constant is available at its output port whenever it is required during the execution of the data flow program. Because of some limitations of the simulator, only one output arc is allowed from a constants output port.

COMMENT - This nodes allows a comment to be placed into the graph. The comments title text is displayed under the node in the graph display. The full text of the comment can be displayed in a dialog box by double clicking on the node.

A comments node may be placed anywhere on the graph or it may be placed inline in the arcs of the graph.

The text of the comment will appear in the assembly language text of the program, based on when the comment node is encountered in the assembly process.

CONDITIONAL - The node palette contains a "dummy" node that is used to gain access to all of the conditional nodes. When the "dummy" conditional is placed on the graph, the mini-palette containing the actual available conditionals is displayed.

The conditionals accept real data at input ports one and two and generate a boolean result at the output port.

The available conditionals include equal to, not equal to, less than, greater than, less than or equal to, and greater than or equal to.

HALT AND ABSORB - The halt and absorb operators are paired together in one mini-palette. The halt operator absorbs a single data token and then halts execution of the program. The absorb node will just absorb the data token at its input port. The absorb token is used to route the data token to address 0,0 in the simulator.

AND/OR - The AND and OR nodes are grouped together. They perform the boolean functions on two boolean inputs and produce a boolean output.

NOT - The unary NOT function is applied to the boolean input to form the boolean result.

I-STRUCTURE READ - The I-Structure in the simulator can be considered a two dimensional array of real values. The real values at input ports 1 and 2 of the Read node identify the i-structure cell. The output port yields the value of the cell, after it has been written to.

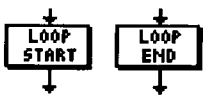
I-STRUCTURE WRITE - The real values at input ports 1 and 2 of the Write node identify the i-structure cell. The output port yields a boolean true if the value of the cell was successfully set to the value in input port three.

I-STRUCTURE FREE - The i-structure Free command reinitializes an i-structure cell. The real values at input ports 1 and 2 of the Free node identify the i-structure cell. The value at input port three is not used by the command but the values are all passed to the corresponding output ports. In this fashion, the Free command can be followed by a write command to clear and then fill a cell.

SWITCH - Send input port 1 to the "true" output port if input port 2 is true, send input port 1 to "false" output port if input port 2 is false.

GATE IF TRUE - Pass input port 1 if input port 2 is True; otherwise absorb input port 1.

GATE IF FALSE - Pass input port 1 if input port 2 is false; otherwise absorb input port 1.

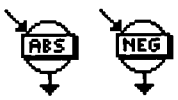


LOOP CODE BLOCK DEFINITION - There are two nodes paired together that are used to define the context of a loop. These allow the loop contents to be "unwound" and parallel processed. The Loop Start Node adds a codeblock to the activity name of the token at input port 1. The Loop End Node removes the codeblock from the activity name of the token at input port 1.

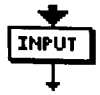
LOOP DATA CONTROL - The Loop Data Control node pair is used to mark different instantiations of a loop variable. The data increment node increments the data instantiation number while the data reset node reset the instantiation number back to 1.

MATH OPERATORS - The standard math operators for addition, subtraction, multiplication, and division are implemented on the palette. They exist as an equivalent group and can be changed from one operator type to another.

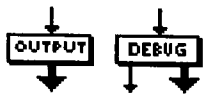
SQUARE ROOT OPERATOR - The square root operator calculates the square root of the token at the input port and returns the result.



NEGATION AND ABSOLUTE VALUE OPERATORS - This group provides operators to negate or provide the absolute value of the input token.



FILE INPUT OPERATOR - This node is used to indicate a constant that is read from a file. When the graph is compiled to its assembly language code, an input data file is created of a form described in section 3.1.16. The numbered nodes in the graph correspond to the placeholder values in this file.



OUTPUT / DEBUG OPERATORS - These operators creates nodes for file output. The debug operator also creates output ports that pass the data as well as sending the data to a file. The output operator sends the input data to a file and then absorbs the token(s). When the node is created, the user is requested to provide the number of input values, up to 20, to be created for the node.

PROCEDURE DATA INPUT AND PROCEDURE DATA OUTPUT CONNECTORS - These nodes are available on the node palette when a procedure graph window is active. They are used to define the input and output data paths for the procedure. As they are created, they are sequentially numbered and the numbers correspond to port numbers displayed on the procedure node when it is created with the apply operator. The IO points should be the first items defined when creating a procedure so that the procedure nodes can be properly created with the apply operator, especially if the procedure is recursive. If IO points are added to a procedure graph after the apply

operator has placed the procedure node into any graph, any existing arcs to these nodes may be cleared and will have to be redrawn.

APPLY PROCEDURE OPERATOR - This operator is used in conjunction with the Procedure menu to place node in a graph that represent user created procedures. The nodes that are created have the name of the procedure on them and have the number of IO ports specified by the procedure data input and output connectors used in the procedure graph.

The apply operator can only create one procedure graph node at a time. The procedure node to be created is selected by the user using the list of procedure names in the Procedure Menu. When no procedure has been selected, the apply operator is not available on the node palette.

DATA PATH CONVERGER - This node is used to bring together two alternate data paths as input to a single port. This is most often used in conjunction with a Switch operator. A simulator execution error will occur if the graph causes two tokens to arrive at a single input port so this operator can only be used at the end of alternate data path

A.5 The Graph Window

The Graph Window is used as the construct to define a graph as a unit or entity. Two types of graph exist, the main program graph and procedure graphs. The graph windows that represent these structures are functionally identical.

Graph windows can be manipulated in a fashion similar to any window on a Macintosh. The can be moved (dragged) using the title bar, zoomed to full screen size using the zoom box or resized to any other size using the resize (or grow) box.

The Graph space to be used by the graph is allocated using the arrow buttons on the scroll bars. See Section 3.1.9 for more information.

A.6 Mouse Operations

Four mouse operations are used extensively when creating a graph. The first is creating nodes with the node palette, which has been described.

Creating arcs involves pressing and holding the mouse in one port and dragging the mouse to another port and releasing the mouse. The ports must be of the same type and one port must be an input port and another must be an output port.

Nodes can be moved around the graph by dragging. Any connected arcs will also be dragged to the new node position.

Nodes and ports can be queried for information about themselves. By double clicking on a node or a port, a dialog box of information is displayed. Comment nodes will display the comments and output or debug nodes will allow entry of the text message that can accompany the token data output. Ports will display the data type that is accepted at that port. Information about the node can also be accessed from the information dialog box about a port.

A.7 Error Messages

Error messages are displayed in dialog boxes at the time they are detected. The dialog box is cleared by pressing the OK button. Error messages are designed to be sufficiently descriptive of the error condition.

Appendix B

Data Flow Simulator Information

B.1 Introduction

This section provides a cursory overview of the assembly language of the data flow simulator developed at RIT. For a more complete discussion of the language, refer to the [Benjamin 1988]. The following sections are taken from that document.

B.2 Instruction Set Design

The instruction set for the simulator is taken largely from [Dennis 1975] in which he defines a data flow program as a bipartite directed graph where the two types of nodes are called links and actors.

He regards the arcs of a data flow program as channels through which tokens flow carrying values from each actor to other actors by way of the links. The instruction set used on the simulator differs from [Dennis 1975], in that data is permitted to travel directly from one actor to another actor. Links are used only to replicate tokens with multiple destinations.

The simulator instruction set also includes actors necessary for implementing the U-Interpreter [Arvind and Gostelow 1982], and for implementing I-structures [Arvind and Thomas 1980].

LINK: replicates its input token and distributes the copies to its output destinations.

OPERATOR ACTOR: applies its function to its two input tokens (one input token for unary functions) and sends the result to its output destination. Operator actors in the simulator include negation, square root, absolute value, addition, subtraction, multiplication, and division.

DECIDER ACTOR: applies its predicate to its input tokens and sends the resulting control token (true or false) to its output destination. Decider actors provided by the simulator are less than, greater than, less than or equal to, grater than or equal to, equal or not equal.

BOOLEAN ACTOR: applies its boolean function to its input tokens and sends the resulting control token (true or false) to its output destination. Boolean actors provided by the simulator are and, or, and not.

T-GATE CONTROL ACTOR: passes its input token to its output destination if it receives the value true at its control operand; the data operand is discarded if false is received.

F-GATE CONTROL ACTOR: passes its input token to its output destination if false is received, discards its input if true is received.

SWITCH CONTROL ACTOR: allows a control value to determine which of two output destinations its input should be passed to. A true control value will cause the input data to be routed to the T-destination; a false value will cause the data to be routed to the F-destination.

LOOP ACTORS (L,L1,D,D1): manipulate context-sensitive information in the token tag, making it possible to concurrently execute several iterations of a loop.

The L actor adds new contexts to the token tag when loops are entered; L1 removes contexts added by L when loops are exited.

The D actor adds 1 to the initiation value; D1 resets the initiation value to 1.

APPLY ACTORS (A,PBEG,PEND,A1): operate on context-sensitive information in the token tag, making it possible to concurrently execute several instantiations of a procedure.

The A instruction adds a new context to the token tag each time a procedure is invoked, sends its input tokens (procedure arguments) to the PBEG instruction, and sends the A1 instruction address (return address) to the PBEG instruction.

The PBEG instruction is always the first instruction of a procedure, it collects the procedure arguments and distributes them to the statements of the procedure, and sends the A1 address (return address) to the PEND instruction.

The PEND instruction is always the last instruction of a procedure, it collects the procedure result tokens and sends them to the A1 instruction.

The A1 instruction removes the context added by A and distributes the tokens to the statements of the calling procedure.

I-STRUCTURE ACTORS (IREAD,IWRITE): manipulate I-structures. The IREAD operation retrieves the data value of I-structure x0 at selector i. The

IWRITE operation appends the value v to I-structure x0 at selector i. It, also, satisfies any pending reads.

B.3 Simulator Instruction Set

The data flow assembler language is a statement description of the data flow graph. The assembler commands and their function are summarized below.

Assembler Command Summary

#	comment; characters from # to end of line are skipped
a	add subprogram context; invoke subprogram
a1	remove subprogram context; distribute result tokens
abs	absolute value
add	addition
and	boolean and
const	store instruction constants
d	increment loop iteration counter
debug	output token values at particular point in graph
d1	reset loop iteration counter to 1
div	division
eq	equal
fgate	propagate token if control signal is false
ge	greater than or equal to
gt	greater than
halt	halt program
ifree	re-initialize i-structure cell
iread	read i-structure cell
iwrite	write i-structure cell
l	add loop context
l1	remove loop context
le	less than or equal to
link	duplicate token
lt	less than
mul	multiplication
ne	not equal
neg	negate value
not	boolean not
or	boolean or
output	output token value(s)
pbeginitiate	procedure

pend	terminate procedure
sqrt	square root
sub	subtraction
switch	choose token path depending on control signal
tgate	propagate token if control signal is true
trace	trace simulator execution

B.4 Simulator Syntax Summary

Summarized below is the syntax for each instruction as it appears in a simulator assembly program.

Assembler Syntax Summary

Key

addrinstruction address; address range is from 1 to 500;
0 specifies a null address
dest destination instruction address and port id
numarg number of procedure arguments (maximum 20)
numtoken number of tokens (maximum 20)
[ ] optional field
m{}nfrom m to n of field

#	comment text		
a	addr	numarg	a1-addrpbeg-addr
a1	addr	numarg	numarg{dest}numarg
abs	addr		dest
add	addr		dest
and addr		dest	
const		value	dest
d	addr		dest
debug	addr	numtoken	numtoken{dest}numtoken [label]
d1	addr		dest
div	addr		dest
eq	addr		dest
fgate	addr		dest
ge	addr		dest
gt	addr		dest
halt addr			
ifree addr		3{dest}3	
iread	addr		dest
iwrite	addr		dest
l	addr	codeblock	dest
l1	addr		dest
le	addr		dest
link addr		2{dest}2	
lt	addr		dest
mul addr		dest	

```
ne          addr                    dest
neg         addr                    dest
not         addr                    dest
or          addr                    dest
output      addr    numtoken        [label]
pbegaddr    numarg          numarg(dest)numarg pend-dest
pend        addr    numarg
sqrt addr                   dest
sub         addr                    dest
switch      addr                    true-dest false-dest
tgate       addr                    dest
```

trace f1 .. f18

where fn is 0 for off
1 for on

f1	trace instruction store
f2	trace token queue arrival
f3	trace token queue after arrival
f4	trace token queue departure
f5	trace token queue after departure
f6	trace token set queue arrival
f7	trace token set queue after arrival
f8	trace token set queue departure
f9	trace token set queue after departure
f10	trace token store before arrival
f11	trace token store arrival
f12	trace token store after arrival
f13	trace operation packet queue arrival
f14	trace operation packet queue after arrival
f15	trace operation packet queue departure
f16	trace operation packet queue after departure
f17	trace processor arrival
f18	trace processor departure

B.5 Detailed Description of the Simulator Instruction Set

This section details the form and function of each instruction in the simulator assembly language. The instructions are grouped according to function.

B.5.1 Math Operators

ABS addr dest

addr - address in the graph
dest - result token destination address and port

Find absolute value of port 1.

ADD addr dest

addr - address in the graph
dest - result token destination address and port

Add port 1 to port 2.

DIV addr dest

addr - address in the graph
dest - result token destination address and port

Divide port 1 by 2.

MUL addr dest

addr - address in the graph
dest - result token destination address and port

Multiply port 1 by port 2.

NEG addr dest

addr - address in the graph
dest - result token destination address and port

Multiply port 1 by -1.

SQRT addr dest

addr - address in the graph
dest - result token destination address and port

Find square root of port 1.

SUB addr dest

addr - address in the graph
dest - result token destination address and port

Subtract port 2 from port 1.

B.5.2 Predicate Instructions

EQ addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is equal to port 2; false otherwise.

GE addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is greater than or equal to port 2; false otherwise.

GT addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is greater than port 2; false otherwise.

LE addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is less than or equal to port 2; false otherwise.

LT addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is less than port 2; false otherwise.

NE addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is not equal to port 2; false otherwise.

B.5.3 Boolean Instructions

AND addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 and port 2 are true; false otherwise.

NOT addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1 is false; false otherwise.

OR addr dest

addr - address in the graph
dest - result token destination address and port

Send true token if port 1, port 2, or both are true; false otherwise.

B.5.4 Branch Instructions

FGATE addr dest

addr - address in the graph
dest - result token destination address and port

Send port 1 if port 2 is false; otherwise absorb port 1.

HALT addr

Halt program execution.

LINK addr 2{dest}2

addr - address in the graph
dest - result token destination address and port;

Replicate port 1 token; send the token copies to dest 1 and 2.

SWITCH addr true-dest false-dest

addr - address in the graph
true-dest - result token destination address and port; if control signal is true
false-dest - result token destination address and port; if control signal is false

Send port 1 to true-dest if port 2 is true, send port 1 to false-dest if port 2 is false.

TGATE addr dest

addr - address in the graph
dest - result token destination address and port

Send port 1 if port 2 is true; otherwise absorb port 1.

B.5.5 Loop Instructions

D addr dest

addr - address in the graph
dest - result token destination address and port

Increment port 1 initiation number.

D1 addr dest

addr - address in the graph
dest - result token destination address and port

Reset port 1 initiation number to 1.

L addr codeblock dest

addr - address in the graph
codeblock - loop code block number
dest - result token destination address and port

Add codeblock to port 1 activity name.

L1 addr dest

addr - address in the graph
dest - result token destination address and port

Remove codeblock from port 1 activity name.

B.5.6 Procedure Instructions

A addr numarg **A1**-addr **PBEG**-addr

addr	- address in the graph
numarg	- number of procedure arguments
A1-addr	- A1 instruction address
PBEG-addr	- procedure **PBEG** instruction address

Send arguments in ports 1 to numarg to **PBEG**-addr; send **A1**-addr (return address) to **PBEG**-addr; add addr to token activity names.

A1 addr numarg numarg{dest}numarg

addr	- address in the graph
numarg	- number of procedure arguments
dest	- result token destination address and port

Send arguments in ports 1 to numarg to corresponding dest; Remove **A** addr from token activity names.

PBEG addr numarg numarg{dest}numarg **PEND**-dest

addr	- address in the graph
numarg	- number of procedure arguments
dest	- result token destination address and port
PEND-dest	- procedure **PEND** address and port

Distribute port tokens 1 to numarg to procedure instructions; send **A1**-addr (return address) to **PEND**-dest.

PEND addr numarg

addr	- address in the graph
numarg	- number of procedure arguments

Send result parameters in ports 1 to numarg to the **A1** instruction address.

B.5.7 I-structure Instructions

IFREE addr 3{dest}3

addr - address in the graph
dest - result token destination address and port

Re-initialize i-structure cell located by port 1 and port 2; send port 1 to dest 1, port 2 to dest 2, and port 3 (icell value) to dest 3.

IREAD addr dest

addr - address in the graph
dest - result token destination address and port

Send a copy of the value stored at the i-structure cell located by port 1 and port 2 to dest.

IWRITE addr dest

addr - address in the graph
dest - result token destination address and port

Store the value in port 3 at the i-structure cell located by port 1 and port 2.

B.5.8 Output Instructions

DEBUG addr numtoken numtoken{dest}numtoken [label]

addr - address in the graph
numtoken - number of tokens to be output
dest - result token destination address and port
label - character string prefix to output tokens

If a label exists output the label; output the tokens in ports 1 to numtoken; send the tokens in ports 1 to numtoken to the corresponding dest.

OUTPUT addr numtoken [label]

addr - address in the graph
numtoken - number of tokens to be output
label - character string prefix to output tokens

If a label exists output the label; output the tokens in ports 1 to numtoken.

B.5.9 Comment and Constant Instructions

\# comment text

Characters from the # to the end of the line are ignored by the assembler, and may be used for program comments.

CONST value dest

value - constant value
dest - instruction address and port where constant value is to be stored

Stores value at dest.

B.6 Debugging

The simulator provides two kinds of debugging tools. The **DEBUG** and **OUTPUT** instructions are used for debugging assembler programs written by the application programmer. The **TRACE** instruction is used by the simulator's maintainer to debug the simulator itself. Debugging an application program is best approached by studying the program graph, and choosing nodes where knowing the token values will provide clues to solving the problem.

The **DEBUG** or **OUTPUT** instruction can be inserted to display these token values. If the **DEBUG** instruction is used then the tokens will be propagated to the next node in the program graph, the **OUTPUT** instruction will absorb the tokens. .pp The TRACE instruction, though it may provide some insight into the application program,

is used to debug the simulator itself. The **TRACE** instruction has available eighteen options of which any combination may be employed.

TRACE f1 f2 f3 .. f18, where fn is enabled if set to 1 fn is disabled if set to 0

All trace results are written to the "trace.lis" file. This file has a tendency toward being very large.

Whereas the **DEBUG** and **OUTPUT** instructions only display the token data, the **TRACE** instruction also displays the token tag.

TRACE Instruction Flags

f1	displays the machine code assembled from the application program.
f2	displays all tokens that arrive at the token queue.
f3	displays the token queue after a token arrival.
f4	displays all tokens that depart from the token queue.
f5	displays the token queue after a token departure.
f6	displays all token sets that arrive at the token set queue.
f7	displays the token set queue after a token set arrival.
f8	displays all token sets that depart from the token set queue.
f9	displays the token set queue after a token set departure.
f10	displays the token store, prior to applying the matching algorithm, when a token arrives at the match unit.
f11	displays the token that has arrived at the match unit.
f12	displays the token store after the matching algorithm has been applied to the token.
f13	displays all operation packets that arrive at the operation packet queue.
f14	displays the operation packet queue after an operation packet arrival.
f15	displays all operation packets that depart from the operation packet queue.
f16	displays the operation packet queue after a departure.
f17	displays all operation packets that arrive at the processing unit.
f18	displays all result tokens that depart from the processing unit.

B.7 Other Files Required by the Simulator

The simulator requires a number of configuration files be available in the same directory with the simulator whenever it runs. The processor configuration file and the instruction configuration file are required along with the users assembly language file and the input data file. The processor configuration file can be created by FlowGraph while the instruction configuration must be created by the user if it is not available.

B.7.1 Processor Configuration File

The processor configuration file gives the simulator the information it needs to simulate any number of available "processors", which are simulated with coroutines. This file can be created with FlowGraph. In use, it must be named config and it must reside in the same directory with the simulator during execution.

The processor configuration file consists of one line with four values, each separated by a space. The first value is the number of processor coroutines to be created. The second value is the working set size of the match unit coroutine. The third value is the working set size of the instruction unit coroutine. The fourth value is the working set size of the processor unit coroutines. A typical configuration file is

```
20 200000 200000 200000
```

B.7.2 Instruction Set Configuration File

The data flow simulator was designed to have an extendable instruction set. To support this, an instruction set configuration file is necessary to describe the characteristics of each of the currently available commands. FlowGraph was designed to support the original simulator instruction set and is not as easily extendable.

The instruction set configuration file must be called commandTable and must be in the same directory as the simulator at execution time. The instruction set configuration file that corresponds to the instructions that are generated by FlowGraph is shown below. If it is not available, it must be created by the programmer.

```
#           100     0       0
A           1       0       1
A1          2       0       0
ABS         31      1       1
ADD         3       2       1
AND         4       2       1
CONST       101     0       0
D           5       1       1
D1          6       1       1
DEBUG       34      0       0
DIV         7       2       1
EQ          8       2       1
FGATE       9       2       1
GE          10      2       1
GT          11      2       1
HALT        32      1       0
IFREE       33      3       3
IREAD       12      2       1
IWRITE      13      3       1
L           14      2       1
L1          15      1       1
LE          16      2       1
LINK        17      1       2
LT          18      2       1
MUL         20      2       1
NE          21      2       1
NEG         22      1       1
NOT         23      1       1
OR          24      2       1
OUTPUT      19      0       0
PBEG        25      0       0
PEND        26      0       1
SQRT        27      1       1
SUB         28      2       1
SWITCH      29      2       2
TGATE       30      2       1
TRACE       102     0       0
```

The instruction set configuration file is structured such that one instruction is declared per line. For each instruction line, column 1 contains the instruction mnemonic, column 2 contains the instruction opcode, column 3 contains the instruction enable count, and column 4 contains the instruction destination count.

B.8 Running the Simulator

The assembly language file and the edited input data file can be sent to the simulators host system using as variety of text file transfer methods associated with the various terminal emulators available for the Macintosh.

To run the simulator, type:

```
$ dfw
Program file name: <assembler program file name>
Input file name: <input data file name>
Output file name: <simulator output file name>
```

If any trace options were selected, a file named "trace.lis" will be created as well.

B.9 Simulator Errors

An application program may cause the simulator to abort with one of the following errors.

Error 1: unknown command: address = n

The application program contains an unknown assembler command at address n, or a syntax error prior to address n.

Error 2: instruction store overflow

The application program contains an instruction address beyond the range of the instruction store.

Error 3: file not found

The file name given at the prompt could not be found by the simulator.

Error 4: port collision: address = n: port = m

A token was destined for an instruction port that was already occupied by another token.

Error 5: Istructure[i,j] collision

An IWRITE was issued for an i-structure cell that was not empty.

Appendix C

Data Flow Graph Example

C.1 Introduction

As an example of data flow programming with FlowGraph, this appendix presents the algorithm for calculating the approximate planar area using the trapezoidal rule.

C.2 Background

The method of using the trapezoidal rule to calculate a planar area is given as:

Divide the planar area S into n strips by equidistant parallel chords of length $y_0, y_1, y_2, \cdots, y_n$ (where y_0 and/or y_n may be zero), and let h denote the common distance between the chords. The approximate area is given by:

$$S = h(\tfrac{1}{2}y_0 + y_1 + y_2 + \ldots + y_{n-1} + \tfrac{1}{2}y_n) \quad [1]$$

In the example given here, the limits of the area are defined by a and b, and the values of y are given by the function

$$f(y) = y^2 \quad [2]$$

such that:

$$y_0 = f(a) \quad [3]$$

$$y_n = f(b) \quad [4]$$

and

$$h = (b - a) / n \quad [5]$$

Thus, from [1], [3], and [4],

$$S = h(\tfrac{1}{2}f(a) + f(a+h) + f(a+2h) + \ldots + f(b-h) + \tfrac{1}{2}f(b)) \quad [6]$$

C.3 Implementation

The implementation for this set of equations in a generic control flow language is:

```
begin main
     input(a,b,n)
     h = (b - a) / n
     S = ( f(a) + f(b) ) / 2

     x = a + h
     for i = 1 to n - 1
          S = S + f(x)
          x = x + h
     end for

     S = S * h
     output(s)
end main

f(x)
     return (x * x)
end f
```

For the example given here, with a equal to 2, b equal to 4, and n equal to 3, figure C1 shows the problem graphically.

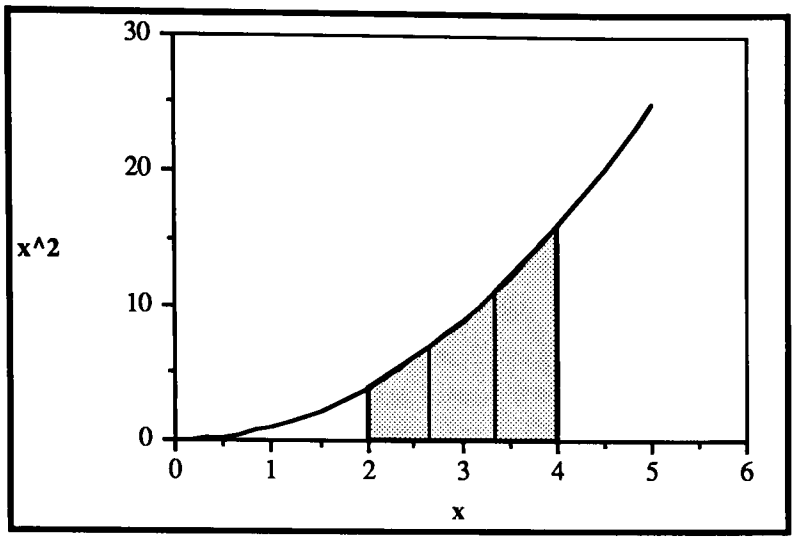


Figure C1 - Graph of the Trapezoidal Rule Example

The data flow graph that was created using FlowGraph for the main algorithm is shown in figure C2 and the graph for the function f(x) is shown in figure C3. (Note that color output devices show nodes in blue, boolean ports and arcs in red and real ports and arcs in green. Untyped ports and arcs would be shown in black).

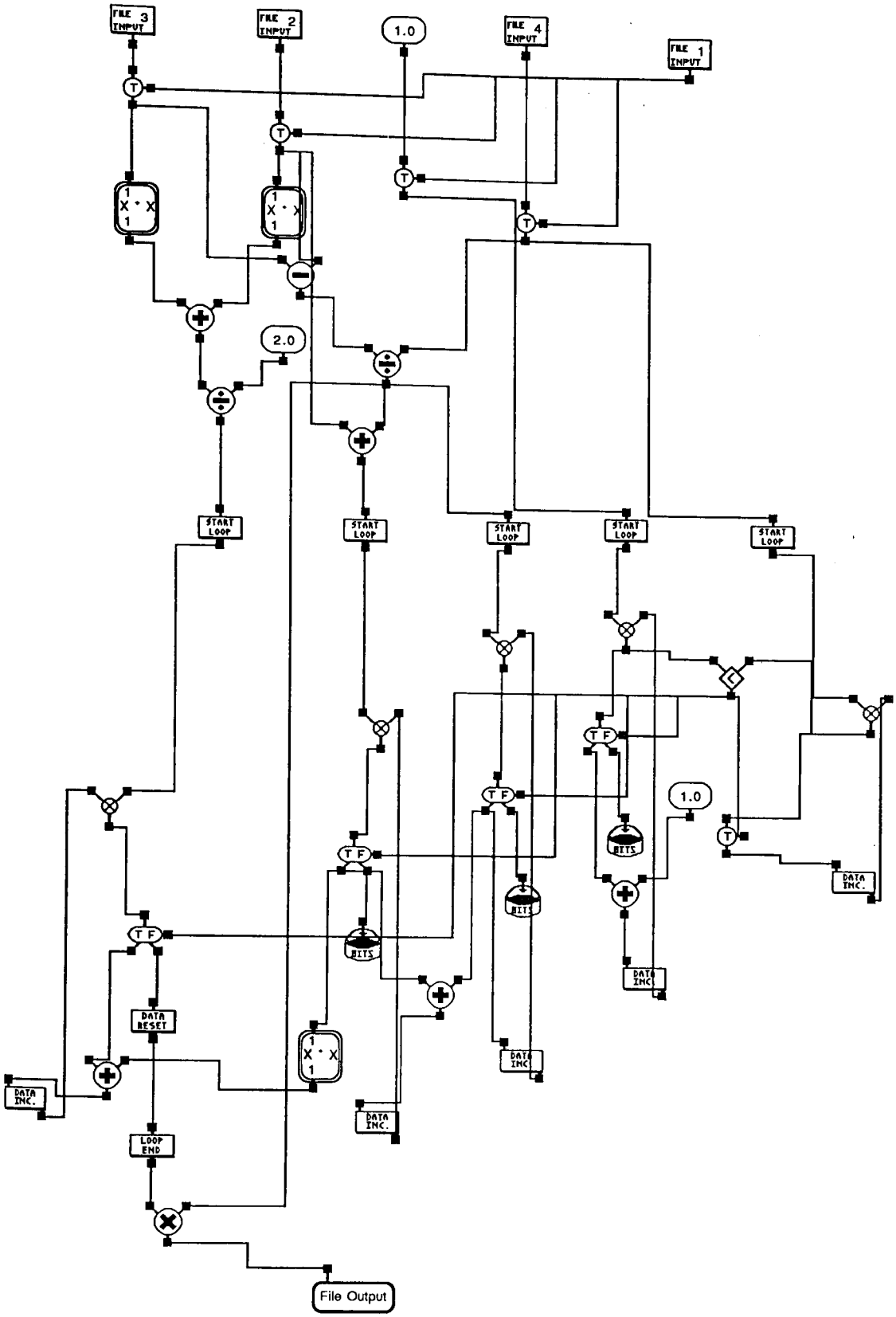


Figure C2 - Data Flow Graph of the Trapezoidal Rule Made by FlowGraph - Main Program

Figure C3 - Data Flow Graph of the Trapezoidal Rule Made by FlowGraph - SubProgram for Calculating x^2

There are four input data values in the data flow graph. Input value 1 is a boolean used to trigger the processing of the graph. Input values 2, 3 and 4 correspond to a, b, and n, respectively.

The assembly language version generated by FlowGraph for the data flow simulator is shown in figure C4. An example of the simulator test data and the output of the simulator are shown in figures C5 and C6, respectively.

```
# Dataflow Assembly from: Trapezoid

# Trace Flag:
#       1 2 3 4 5 6 7 8 9 a 1 2 3 4 5 6 7 8
trace   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

#  Main

link    42                      2 2     1 2
link    43                      3 2    42 1
link    44                      4 2    43 1
link    45                      5 1     9 1
tgate    1                     45 1
link    46                      9 2    13 1
link    47                      7 1    46 1
tgate    2                     47 1
tgate    3                     17 1
link    48                     10 2    18 1
tgate    4                     48 1
const          1.000000        3 1
a        5      1               6      39
a1       6      1              11 1
a        7      1               8      39
a1       8      1              11 2
sub      9                     10 1
link    49                     16 1    33 2
link    50                     13 2    49 1
div     10                     50 1
add     11                     12 1
div     12                     14 1
const          2.000000        12 2
add     13                     15 1
l       14      1              19 1
l       15      1              20 1
l       16      1              21 1
link    51                     22 1    23 1
```

```
l         17      1             51 1
link      52                    23 2     24 1
l         18      1             52 1
switch    19                    29 1     30 1
link      53                    26 1     27 1
switch    20                    53 1      0 0
link      54                    26 2     36 1
switch    21                    54 1      0 0
switch    22                    25 1      0 0
link      55                    19 2     24 2
link      56                    20 2     55 1
link      57                    21 2     56 1
link      58                    22 2     57 1
lt        23                    58 1
tgate     24                    34 1
add       25                    35 1
const           1.000000        25 2
add       26                    37 1
a         27      1             28       39
a1        28      1             29 2
add       29                    38 1
d1        30                    31 1
l1        31                    33 1
output    32      1
mul       33                    32 1
link      59                    23 2     24 1
d         34                    59 1
link      60                    22 1     23 1
d         35                    60 1
d         36                    21 1
d         37                    20 1
d         38                    19 1

# Procedure: X * X

link      61                    41 1     41 2
pbeg      39      1             61 1     40 2
mul       41                    40 1
pend      40      1
```

Figure C3 - FlowGraph Output File for Simulator

```
1          44  1
2           2  1
4           1  1
3           4  1
```

Figure C4 - Test Data File for Simulator

```
        18.814817
```

Figure C5 - Simulator Output File

C.4 Conclusions

It is interesting to note that the control flow version of the program takes only about 1 minute to type into an editor while the data flow graph takes about 45 minutes to draw using FlowGraph. This is an inherent limitation of data flow programming, especially where the algorithm does not lend itself to abstraction through subprograms. In this case, the algorithm shown, as well as the input data, is taken directly from [Benjamin 1988 - section 4.5.4] as a test of FlowGraph.

Appendix D

Glossary

active window: The frontmost window on the desktop; the window where the next action will take place. An active window's title bar is highlighted.

Backspace key: A key that backspaces over and erases the previously typed character or the current selection.

button: A pushbutton-like image in dialog boxes where you click to designate, confirm, or cancel an action. See also mouse button.

Cancel button: A button in dialog boxes that you click to cancel a command.

check box: A small box or circle associated with an option in a dialog box. When you click the check box, you may change the option or affect related options.

click: To position the pointer on something, and then to press and quickly release the mouse button.

close box: The small white box on the left side of the title bar of an active window. Clicking it closes the window.

command: An instruction that causes the computer to perform some action. A command can be typed from a keyboard, selected from a menu with a hand-held device (such as a mouse), or embedded in a program.

Command key: A key that, when held down while another key is pressed, causes a command to take effect. When held down in combination with dragging the mouse, the command key lets you drag a window to a new location without activating it. The

command key is marked with a propeller-shaped symbol. On some keyboards, the command key has both the propeller symbol and the Apple symbol on it.

crossbar: A type of pointer used in FlowGraph to draw arcs and manipulate nodes.

desktop: The Macintosh's working environment- the menu bar and the gray area on the screen.

dialog box: A box that contains a message requesting more information from you, or that contains alternatives from which you can choose. Sometimes the message warns you that you're asking the computer to do something it can't do or that you're about to destroy some of your information. In these cases the message is often accompanied by a beep.

dimmed command: A command that appears gray rather than black in the menu. You cannot choose a dimmed command, usually because the command would be unable to act on anything or because the command has not been implemented in FlowGraph.

double-click: To position the pointer where you want an action to take place, and then press and release the mouse button twice in quick succession without moving the mouse.

highlight: To make something visually distinct from its background. An item is usually highlighted to show that it has been selected or chosen.

I-beam: A type of pointer used in entering and editing text.

icon: A graphic representation of an object, a concept, or a message. FlowGraph uses icons for its representation of nodes, the data operators of the graph language.

menu: A list of choices presented by a program from which you can select an action. Menus appear when you point to and press menu titles in the menu bar. Dragging

through the menu and releasing the mouse button while a command is highlighted chooses that command.

menu bar: The horizontal strip at the top of the screen tht contains menu titles.

mouse: A small device you roll around on a flat surface next to your Macintosh. When you move the mouse, the pointer on the screen moves correspondingly.

mouse button: The button on the top of the mouse. In general, pressing the mouse button initiates some action on whatever is under the pointer, and releasing the button confirms the action.

pointer: A small shape on the screen that follows the movement of the mouse. In FlowGraph the pointer can be an arrow, a crossbar, or an I-beam.

press: To position the pointer on something and then hold down the mouse button without moving the mouse.

scroll: To move a document in its window so that a different part of it is visible.

scroll arrow: An arrow on either end of a scroll bar. Clicking or dragging in the scroll bar causes the view of the document to change.

scroll box: The white box in the scroll bar. The position of the scroll box in the scroll bar indicates the position of what's in the window relative to the entire document.

select: to designate where the next action will take place. To select, you click or drag across information.

selection: The information or items affected by the next command. The selection is usually highlighted.

size box: A box on the bottom right corner of some active windows that lets you control the size of the window.

title bar: The horizontal bar at the top of a window that shows the name of the window's contents and lets you move the window.

window: Displays information on the desktop. You view documents through windows. You can open or close them, change their size, edit their contents, scroll through them, and move them around on the desktop.

zoom box: A small box with a smaller box enclosed in it found on the right side of the title bar of some windows. Clicking the zoom box expands the window to its maximum size; clicking it again returns the window to its original size.

Bibliography

Albizuri-Romero 1984a. M.B. Albizuri-Romero. "A Graphical Abstract Programming Langauge," SIGPLAN Notices, Vol. 19, No. 1, January 1984, pp.14-23.

Albizuri-Romero 1984b. M.B. Albizuri-Romero. "GRASE - A Graphical Systax Directed Editor for Structured Programming," SIGPLAN Notices, Vol. 19, No. 2, February 1984, pp.28-37.

Apple 1986. Apple Computer, Inc.. *Inside Macintosh Volume IV*, Addison-Wesley, Reading, MA, 1986.

Apple 1988. Apple Computer, Inc.. *Inside Macintosh Volume V*, Addison-Wesley, Reading, MA, 1988.

Arvind and Gostelow 1982. Arvind and K. P. Gostelow. "The U-Interpreter," Computer, Vol. 15, No. 2, February 1982, pp. 42-49.

Arvind and Thomas 1980. Arvind and R. E. Thomas. "I-Structures: An Efficient Data Type for Functional Languages," Technical Report MIT / LCS / TM-210, Laboratory for Computer Science, MIT, September 1980.

Backus 1978. John Backus. "Can Computer Programming be Liberated form the von Neuman Style," Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 613-641.

Benjamin 1988. Steven I. Benjamin. "DATAFLOW: Overview and Simulation," M.S. Thesis, Rochester Institute of Technology, April 1988.

Brown, Meyrowitz, van Dam 1983. Marc H. Brown, Norman Meyrowitz, and Andries van Dam. "Personal Computer Networks and Graphical Animation: Rational

and Practice for Education," ACM SIGCSE Bulletin, Vol. 15, No. 1, February 1983, pp. 296-307.

Davis and Keller 1982. Alan L. Davis, Robert M. Keller. "Data Flow Program Graphs," Computer, Vol. 15, February 1982, pp. 26-41.

Davis and Lowder 1981. Alan L. Davis, S. A. Lowder. "A Sample Management Application Program in a Graphical Data-Driven Programming Language," Digest of Papers Compcon Spring 81, February 1981, pp. 162-167.

Dennis 1975. J. B. Dennis. "First Version of a Data Flow Procedure Language," MAC Technical Memorandum 61, Project MAC, Massachusetts Institute of Technology, May 1975.

Finzer and Gould 1984. William Finzer, Laura Gould. "Programming By Rehearsal," BYTE, June 1984, pp.187-210.

Foley and Wallace 1974. J.D. Foley and V.L. Wallace. "The Art of Natural Graphic Man-Machine Conversation," Proceedings of the IEEE, April 1974, pp.462-471.

Foley, Wallace and Chan 1984. J.D. Foley, V. L. Wallace and P. Chan. "The Human Factors of Computer Graphics Interaction Techniques," IEEE Computer Graphics & Applications, November 1984, pp.13-48.

Glinert and Tanimoto 1984. E.P. Glinert, S.L. Tanimoto. "PICT: An Interactive Graphical Programming Environment," Computer, November 1984, pp. 7-25.

Gutfreund 1987. Steven H. Gutfreund. "ManiplIcons in ThinkerToy", Special Issue of the SIGPLAN Notices, Vol. 22, Num. 12, December 1987, pp. 307-317.

Halbert 1984. David Halbert. "Programming-by-Demonstration," PhD Dissertation, Xerox Technical Report, OSD-T8402, December 1984.

Hamiltion and Zeldin 1976a. M. Hamilton and S. Zeldin. "Integrated Software Development System / Higher Order Software Conceptual Description," Research and Development Technology Report ECOM-76-0329F, US Army Electronics Command, Fort Monmouth, NJ, 1976.

Hamilton and Zeldin 1976b. M. Hamilton and S. Zeldin. "Higher Order Software - A Methodology for Defining Software," IEEE Transactions on Software Engineering, SE-2, No. 1, 1976, pp. 9-32.

Hamilton and Zeldin 1983. M. Hamilton, S. Zeldin. "The Functional Life Cycle and It's Automation : USE.IT," Journal of Systems and Software, Vol. 3, No. 1, March 1983, pp. 25-62.

Iverson 1962. K. Iverson. *A Programming Language*, Wiley, New York, 1962.

Keller 1977. Robert M. Keller. "Semantics of Parallel Program Graphs," Technical Report UUCS-77-110, Department of Computer Science, University of Utah, July 1977.

Keller and Yen 1981. Robert M. Keller and Wu-Chien J. Yen. "A Graphical Approach to Software Development Using Function Graphs," Digest of Papers Compcon Spring 81, February 1981, pp. 156-161.

Knuth 1963. Donald E. Knuth. "Computer Drawn Flowcharts," Communications of the ACM, Vol. 6, No. 9, Sept. 1963, pp. 555-563.

Kodosky and Dye 1987. Jeff Kodosky and Robert Dye. "Graphical Programming", Computer Graphics World, December 1987, pp. 77-80.

Lawson 1986. Edith A. Lawson. "DIVA, a Data Flow Language," M.S. Thesis, Rochester Institute of Technology, May 14, 1986.

Maguire 1983. G. Q. Maguire. "A Graphical Workstation and Programming Environment for Data Driven Computation," PhD Dissertation, University of Utah, Salt Lake City, Utah, 1983.

Martin 1985. James Martin. *System Design from Provably Correct Constructs*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

Matwin and Pietrzkowski 1985. S. Matwin and T. Pietrzkowski. "PROGRAPH: A Preliminary Report," Computer Languages, Vol. 10, No. 2, pp. 91-126, 1985.

McCarthy 1965. J. McCarthy. *LISP 1.5 Programmers Manual*, 2 ed., M.I.T. Press, Cambridge, MA, 1965.

Myers 1986. Brad A. Myers. "What are Visual Programming, Programming by Example, and Program Visualization?," Proceedings of Graphics Interface '86 / Vision Interface '86, May 1986, pp. 62-65.

Powell and Linton 1984. M.L. Powell and M.A. Linton. "Visual Abstraction in an Interactive Programming Environment," SIGPLAN Notices, Vol. 17, No. 6, June 1984, pp. 14-21.

Raeder 1984. Georg Raeder. "Programming in Pictures," PhD Dissertation, University of Southern California, Los Angeles, CA., November 1984, USC Technical Report TR-84-318.

Raeder 1985. Georg Raeder. "A Survey of Current Graphical Programming Techniques," Computer, Vol. 18, No. 8, August 1985, pp.11-25.

Reiss 1984a. Steven P. Reiss. "Graphical Program Development with PECAN Program Development Systems," SIGPLAN Notices, Vol.19, No. 5, May 1984, pp. 30-41.

Reiss 1984b. Steven P. Reiss. "An Approach to Incremental Compilation," SIGPLAN Notices, Vol. 19, No. 6, June 84.

Rose 1985. Caroline Rose. *Inside Macintosh*, Addison-Wesley, Reading, MA, 1985.

Rubin, Colin and Reiss 1985. R.V. Rubin, E.J. Colin, S.P. Reiss. "ThinkPad: A Graphical System for Programming by Demonstration," IEEE Software, March 1985, pp. 73-78.

Schwartz, Delisle and Begwani 1984. M.D. Schwartz, N.M. Delisle and V.S. Begwani. "Incremental Compilation in Magpie," SIGPLAN Notices, Vol. 19, No. 6, June 1984.

Smith 1986. Randall B. Smith. "The Alternate Reality Kit - An Animated Environment for Creating Interactive Simulations," 1986 IEEE Computer Society Workshop on Visual Languages, 1986, pp. 99-106.

Sutherland 1963. Ivan E. Sutherland. "SKETCHPAD: Man Machine Graphical Communication System," Proceedings of the Spring Joint Computer Conference, Detroit, Michigan, May 21-23, 1963, pp. 329-346.

Torsone 1985. Carol M. Torsone. "Simulation of a Data Flow Computer," M.S. Thesis, Rochester Institute of Technology, April 2, 1985.

Treleaven, Brownbridge and Hopkins 1982. P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins."Data Driven and Demand Driven Computer Architectures," Computing Surveys, Vol. 14, March 1982, pp. 93-143.

Futher Readings

The following references are provided for further readings in the area of data flow languages and systems:

William B. Ackerman. "Data Flow Languages," Computer, Vol. 15., February 1982, pp. 15-25.

Arvind and K. P. Gostelow. "A New Interpreter for Dataflow Schemas and it's Implications for Computer Architecture," Technical Report 72, Department of Information and Computer Science, University of California, Irvine, October 1975.

J. B. Dennis. "On Storage Management for Advanced Programming Languages," Computational Structures Group Memo 109-1, Project MAC, M.I.T., October 1974, (revised November 1, 1974).

J. B. Dennis. "On Storage Management for Advanced Programming Languages," Computational Structures Group Memo 109-1, Project MAC, M.I.I., October 1974 (Revised November 1974).

J. B. Dennis. "A Language Design for Structured Concurrency," Computational Structures Note 28-1, Laboratory for Computer Science, Massachusetts Institute of Technology, February 1977.

J. B. Dennis. "The Varieties of Data Flow Computers," Proceedings of the First International Conference on Distributed Computing Systems, Toulouse, France, October 1979, pp. 430-439.

J. B. Dennis. "Data Flow Supercomputers," Computer, Vol. 13, November 1980, pp. 48-56.

I. Watson and J. Gurd. "A Prototype Data Flow Computer with Token Labelling," Proceedings of the National Computer Conference, AFIPS Press, New Jersey, 1979, pp. 623-628.

The following references are provided for further readings in the areas of user interaction and graphics.

Edward Anson. "The Semantics of Graphical Input," Computer Graphics, Vol. 13, No. 2, August 1979, pp. 113-120.

David R. Barstow, Howard E. Shrobe and Erik Sandewall. *Interative Programming Environments*, McGraw-Hill, New York, 1984.

Willian Buxton. "Lexical and Pragmatic Considerations of Input Structures," Computer Graphics, January 1983, pp.31-37.

Patrick P. Chan and Michael A. Malcolm. "Learning Considerations in the Waterloo Port User Interface," Proceedings of the IEEE Computer Society Conference on Office Automation, December 1984, pp. 33-40.

Andrea A. diSessa. "A Principled Design for an Integrated Computational Environment," Technical Memo MIT/LCS/TM-223, M.I.T., 1984.

Alistair D. N. Edwards. "Visual Programming Languages: the next generation?," SIGPLAN Notices, Vol. 23, No. 4, April 1988, pp. 43-50.

A. Giacalone, M.C. Rinard and T. W. Doeppner, Jr. "IDEOSY - An Ideographic and Interactive Program Description System," Computer Graphics, 1984.

Ephriam P. Glinert. "Towards "Second Generation" Interactive, Graphical Programming Environments," 1986 IEEE Computer Society Workshop on Visual Languages, June 1986, pp. 61-70.

H. Hanusa, H. W. Kuhlmann and G. E. Pfaff. "On Constructing Interactive Graphics Systems," *Eurographics 82*, (Ed. by D. S. Greenway and E. A. Warman), North-Holland, Amsterdam, 1982, pp. 237-248.

Robert R Korfhage and Margaret A. Korfhage. "The Nature of Visual Languages", 1984 IEEE Computer Society Workshop on Visual Languages, December 1984, pp. 177-183.

Lloyd K. Konneker. "A Graphical Interaction Technique Which Uses Gestures," Proceedings of the IEEE Computer Society Conference on Office Automation, December 1984, pp. 51-55.

Henry Lieberman. "There's More to Menu Systems Than Meets the Screen," Computer Graphics, Vol. 19, No. 3, July 1985, pp.181-189.

Brad A. Myers. "The User Interface for Sapphire," IEEE Computer Graphics & Applications, December 1984, pp.13-23.

Brad A. Myers. "Incense: A System for Displaying Data Structures," Computer Graphics, Vol. 17, No. 3, July 1983, pp. 115-125.

D. R. Olsen, Jr., E. P. Dempsey. "Syntax Directed Graphical Interaction," Proceedings of the SIGPLAN 1983 Symposium on Programming Languages Issues, June 1983.

Dan R. Olsen, Jr., William Buxton, Roger Ehrich, David J. Kasik, James R. Rhyne and John Sibert. "A Context For User Interface Management," IEEE Computer Graphics & Applications, December 1984, pp.33-42.

D. R. Olsen, Jr. and E. P. Dempsey. "SYNGRAPH: A Graphical User Interface Generator," Computer Graphics, Vol. 17, No. 3, July 1983, pp.43-50.

D.R. Olsen,Jr., E.P. Dempsey and R. Rogge. "Input / Output Linkage in a User Interface Management System," Computer Graphics, Vol. 19, No. 3, 1985, pp. 191-197.

D. C. Smith. "Pygmalion: A Creative Programming Environment," PhD Dissertation, Dept. of Computer Science, Stanford University, Tech. Report STAN-CS-75-499, 1975.

Steven L. Tanimoto and Ephriam P. Glinert. "Designing Iconic Programming Systems: Representation and Learnability," 1986 IEEE Computer Society Workshop on Visual Languages, June 1986, pp. 54-60.

Thomas H. Taylor and Robert P. Burton. "An Icon-based Graphical Editor", Computer Graphics World, October 1986, pp. 77-82.

Iris Vessey and Ron Weber. "Structured Tools and Conditional Logic: An Empirical Investigation," Communication of the ACM, Vol. 29, No. 1, January 1986, pp. 48-57.

Thomas Lee Williams. "A Graphical Interface to an Economist's Workstation," IEEE Computer Graphics & Applications, August 1984, pp.42-47.

Moshe M. Zloof. "Classification of Visual Programming Language", 1984 IEEE Computer Society Workshop on Visual Languages, December 1984, pp. 232-235.

The following general references are provided for further readings in the areas of Macintosh programming.

Steven Chernicoff. *Macintosh Revealed, Volume One - Unlocked the Toolbox*, Hayden Publishing, Hasbrouck Heights, New Jersey, 1985.

Steven Chernicoff. *Macintosh Revealed, Volume Two- Programming with the Toolbox*, Hayden Publishing, Hasbrouck Heights, New Jersey, 1985.

Scott Knaster. *How tot Write Macintosh Software,* Hayden Publishing, Hasbrouck Heights, New Jersey, 1986.

Scott Knaster. *Macintosh Programming Secrets,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

Apple Computer. *Macintosh Technical Notes,* (published periodically), Apple Programer and Developers Association, Renton, Washington, 1986, 1987, 1988.