# The Design and Implementation of Perseus, a Visual Programming Language

by

Salik Rafiq

A Thesis
Submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba

ISBN 0-315-77789-3

# THE DESIGN AND IMPLEMENTATION OF PERSEUS, A VISUAL PROGRAMMING LANGUAGE

BY

SALIK RAFIQ

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

## Abstract

*Visual programming is a rapidly growing and improving field of computer science which deals with the manipulation of visual objects while programming. It essentially started from user interface research being done at XEROX PARC. As part of this thesis, a classification of visual programming systems is presented, and some existing visual programming systems are surveyed.*

*This thesis also presents the design and implementation of a visual programming language called Perseus developed by this author. Perseus is an iconic visual programming language which uses icons to represent individual programming elements. These can be spatially arranged on a canvas, and connected in such a way as to create a program. This program can then be directly executed. Perseus, unlike other similar systems, actually executes these icons and outputs the result. It is based on the functional language paradigm and on the textual language FP in particular. A short description of FP is included for comparison. Perseus is better than FP and a discussion of why this is so is included in the thesis.*

*Perseus is implemented in C++ using object-oriented programming techniques. An overview of the implementation and its object hierarchy is presented, along with and evaluation of the suitability of C++ as a development language for this project.*

# Acknowledgments

I would first like to first thank my thesis supervisor, Professor Daniel Salomon, who suggested the topic and took the time to proof-read my thesis. I would also thank Prof. Salomon for providing me with a small bursary for most of my graduate study. In addition I would also like to thank Prof. Neil Arnason who employed me during the summer months, and paid me quite well. I would also like to thank the other graduate students, whose companionship made graduate studies an entertaining time. Specifically I would thank Richard Lukes (whom I constantly barbed), Ken Hotz, Renate Scheidler, Vivi Le and Sean Kalynuk (for the pool games).

# Contents

# List of Figures

# Chapter 1

# An Introduction to the Perseus System

## 1.1 Introduction

A great deal of research has been done on making computers easier to use. Computers were once a mystery to everyone except a few elite. This was not a problem since very few people could afford to own one. The invention of the microcomputer has made computers available to the general public. The majority of these microcomputers, however, still require the learning of an unfriendly command line interface. To persuade more people to purchase a computer, the industry has sought to make computers easier to use. One area of research has concentrated on the interface between the user and the computer. Some success has come through the development of the graphical user interface (GUI, pronounced "gooie"). GUIs, such as the Macintosh's interface, OpenLook, and Microsoft Windows, are examples of such success. All GUIs have evolved from ideas developed at Xerox PARC in its Smalltalk project and share the common features of windows, menus, icons and a pointing device. The term WIMP (Windows, Icons, Menus and Pointer) is the accepted generic term for such

user interfaces. WIMP interfaces can allow the use of word processors, databases etc. by the novice computer user.

With the success of the GUI, people began to wonder if similar graphical techniques could be applied to make the programming of computers easier too. This spawned the development of visual programming languages and environments. Languages such as Omega [32], PiP [33], HighVisual [50, 30] , ICONLISP [15] and Tinkertoy [12] are a few examples of research in visual programming languages.

These languages have features in common. They generally involve the moving and placement of screen icons which are then connected together — similar to soldering electronic circuits — to form a program or function. One simple example of a visual language is a graphical flowchart language. The user connects together the different shapes which perform the appropriate actions. The user then types, from the keyboard, appropriate parameters inside the shapes to complete the program. Because the user must still use the keyboard, this cannot be considered a completely visual programming language. A completely visual programming language would allow the user to perform all actions using the pointing device, without ever using the keyboard or any conventional coding. Such languages are rare.

## 1.2 Advantages of Visual Programming

People tend to relate easily to graphical representations, and use them often. For example, programmers draw pictures of linked lists and trees. This is a more natural representation for expressing algorithms, designs and data structures. A more productive approach to program creation involves an increase in our ability to express

requirements to the computer. One approach would be to create a language that is closer to the user's natural language. Another approach is the development of visual languages.

The majority of programming environments today use text as their medium of construction. Text hampers understanding because it is not unique visually — the readability of many text languages is low since they allow a free format for token placement. Text is also not an optimal representation since the program's semantic properties cannot be easily determined without converting it to another form such as a parse tree. Text is also not a good representation for editing because we wish to edit program elements (ie: statements, structures, types etc.). Currently one must use a text editor, and manipulate words, lines and characters.

A visual programming environment can aid the programmer by providing the following characteristics: [33]

- Readability — a graphical representation that shows how data is modified and the flow of control can make the program more understandable and simplify correction.
- Parallelism — a visual animation of the concurrent components gives the programmer a better idea of the parallelism in his program.
- Animation — One should be able to use graphics to test and debug. Animation can be effective in helping the user find bugs and incorrect logic in programs.
- Abstraction — A programming system should treat software as a hierarchy of neat abstractions. A visual system is best for this since it offers

freedom in representing program navigation, for example a zoom feature to examine program components in detail.

- Typing — the type of an entity in a program usually cannot be determined by its name. It is however easy to categorise objects by a visual system, through size, shapes, or colours.

## 1.3 Representation

Most of the current visual programming environments are functional in nature, or are visual representations of functional languages. This is perhaps because imperative languages are difficult to visualise cleanly. Functional languages, however, can be relatively easily represented as blocks with only inputs and an output. Backus's FP language applies a function to its argument and produces a single output. Thus I feel that it is a good candidate to be represented in a visual manner.

My representation of an FP-based visual language is contained in figure 1.1. A short description of the symbols in the language follows.

These symbols comprise the entire FP language object set [2] which consists of: atoms, sequences, primitives, constants, functionals (IF, WHILE, Construction, Composition, $\alpha$ and /) and definitions (DEF). User defined functions are created by using the DEF operation.

The atom and sequence icons represent data. Their pointed left sides can only be attached to the input of a function via the "V" connector.

The square connector on the left side of the functions represents the function's output.

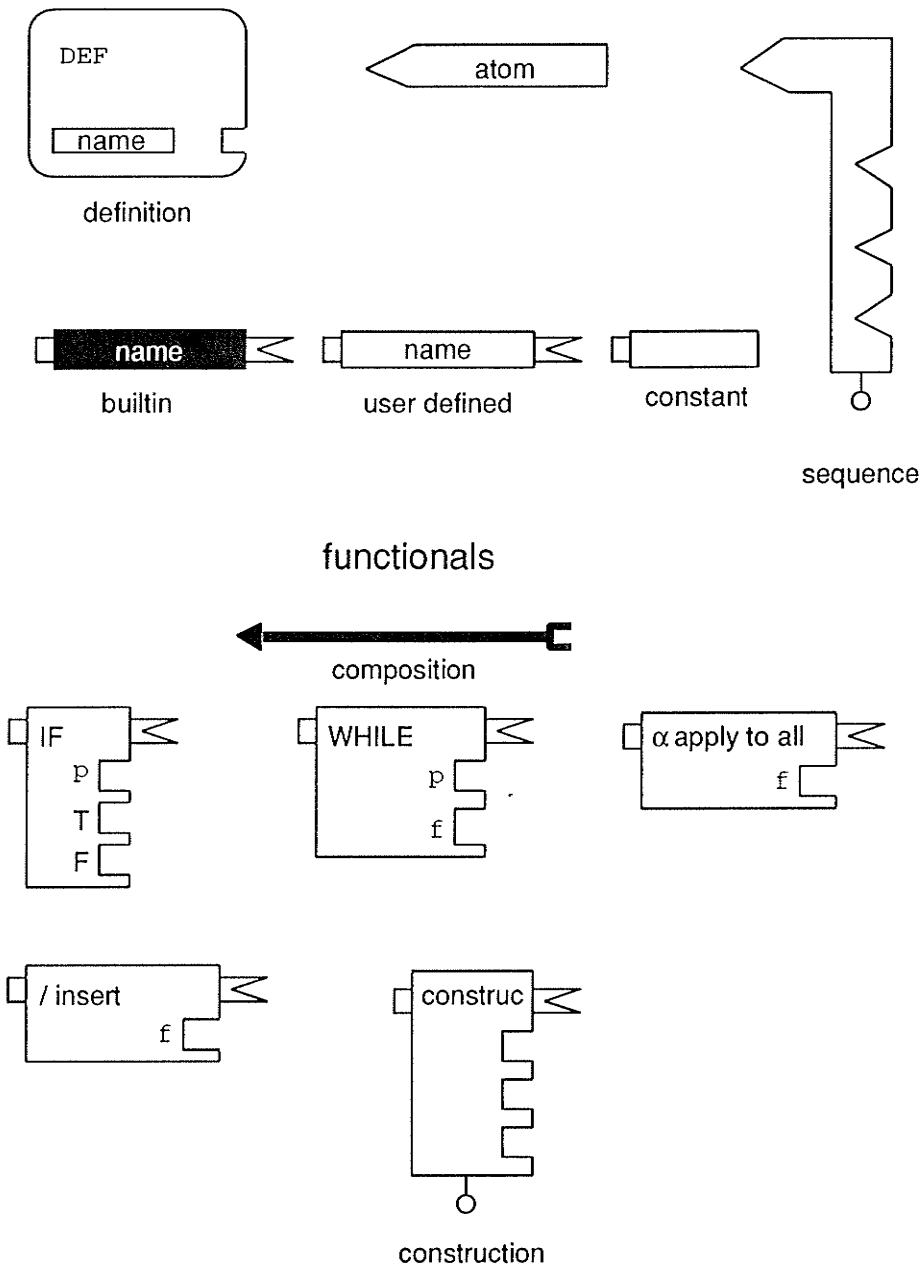



**Figure 1.1:** Proposed representation for visual FP

Functionals are objects that combine existing functions to define new functions. The indented square connectors on the functional are where parameters to the functional are connected. This connector will only allow a function, or functional, to be connected as a parameter.

The composition functional lets the user connect the output from a function to the input of another function.

The pull-down loop on both the construction and sequence icons, operates in a similar fashion to the loop on a window blind. The pull-down loop on the construction icon allows the user to increase or decrease the number of connectors for the connection of functions. The pull-down loop on the sequence icon performs a similar function by creating "V" connectors for the attachment of atoms or sequences.

## 1.4 Perseus an Experimental FP Language

The current Perseus system has been developed as a small prototype running under the XWindows system. It is currently implemented on Sun SPARCstations running the SUN version of the UNIX operating system (SUN OS).

Perseus currently consists of approximately nine thousand (9000) lines of C++ code. It was developed using object-oriented techniques wherever possible. The graphics and user interface of Perseus were created with the aid of InterViews [26], a object-oriented toolkit based on XWindows. InterViews was available to the author free of charge. The choice of operating system, language and toolkit should provide Perseus with portability across different UNIX platforms.

The Perseus system is a menu-based system, with a large working area, or canvas, where the user creates the programs. Buttons are provided to scroll around the canvas, the actual area working of which is quite large and should prove sufficient for any user's needs. Dialog boxes are provided to choose the available built-in functions during editing, and for altering the values of atoms. Execution is interpreted, starting

from the object the user selects with the mouse. Execution results are provided by this object. Currently, output is provided in textual form to the user's terminal.

## 1.5 Outline of This Thesis

Chapter 2 —In this chapter the term *Visual Programming* will be defined. A framework of *Visual Programming* will also be presented along with a short survey of several selected visual programming systems.

Chapter 3 —This chapter contains an introduction to the Perseus language, with each of its elements. Use of the Perseus editor is also described with two small programming examples.

Chapter 4 —This chapter contains an introduction to the FP language along with a comparison to Perseus. The advantages, and disadvantages, of Perseus versus textual FP are also discussed.

Chapter 5 —This chapter contains a short description of the implementation of Perseus. The method of execution used will also be described. As well, experiences with object-oriented design methods used during implementation are also discussed.

# Chapter 2

# Survey of Current Visual Programming Systems

## 2.1 Introduction

This chapter contains an introduction to visual programming, a framework for visual programming, and a short survey of several visual programming systems.

## 2.2 What is Visual Programming

Systems that use visual languages to construct programs are becoming more common. The initial research in visual languages was spured on by cheap computers capable of high resolution graphics. But what is *visual programming*?

To some people visual programming means that the objects manipulated by the language are visual — languages for processing visual information. Or perhaps the language is itself visual — languages for programming with visual expressions. However, one can generally define visual programming as *The meaningful manipulation of visual objects in the programming process*, Shu[42].

This definition can include such things as organising objects in such a way that the computer can interpret them. This definition may also include the type of programming environment, the visual display of data, the display and execution of programs or it can mean the display of program specifications. Visual programming can cover all aspects of programming.

## 2.3 Framework

There are two main types of visual programming, *visual environments* and *visual languages*. These categories have been broken down further into a framework of visual programming. Figure 2.1 displays this framework with the respective categories as developed by Shu[42]. This section discusses this framework in more detail.

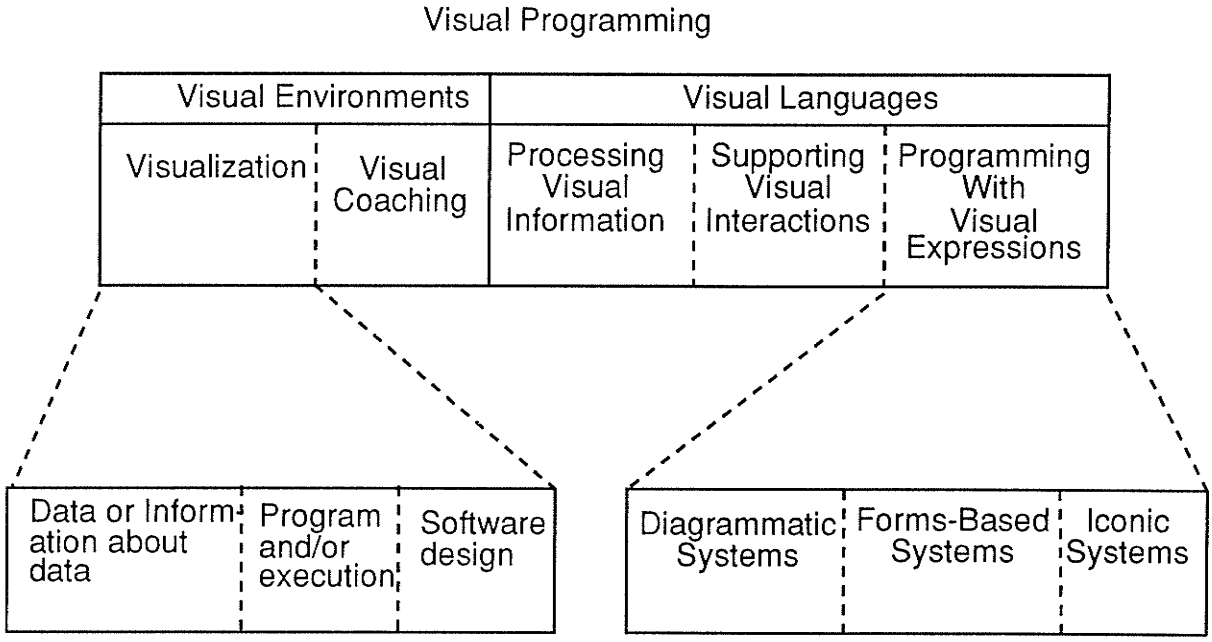



**Figure 2.1:** Visual Programming Framework

**Visual Environments** — The systems which fall into this category provide the user with a visual environment which allows the user to interact with the computer. They provide nothing new in terms of the language aspects of programming; the emphasis is on the human-computer interaction.

**Visualization of Data or Information About Data** — These systems display data in a graphical form using a spatial framework to emphasise its hierarchical structure. Users can then traverse the framework and "zoom in" to obtain greater detail. The data, however, is still stored in traditional databases and users use the visual interface to retrieve the information.

**Visualization of Programs and Execution** — These systems provide graphical support for the visualisation of programs. The programs are still written in textual languages, or are eventually written into textual languages. Such system include features such as "pretty printing", graphical tree displays of data and program, flowchart or Nassi-Shniederman views, and syntax-directed editors. The object of such systems is to make program development and debugging easier.

**Visualization of Software Design** — This final category under *Visual Environments* deals with the visualisation of software specifications, data structures, design decisions and system modularization. These elements are displayed in some graphical form for users to design, document, or maintain a software project.

**Visual Coaching** — In a visual coaching system, the user constructs programs by showing the computer how to perform the desired actions. Program construc-

tion relies almost totally on user interaction. This style of interaction mimics the way we describe programs and algorithms, with pictures, diagrams, and pointing actions. Because visual coaching and visual languages are very close, people tend not to make a distinction between the two.

**Visual Languages** — A visual language system provides the user with a language that is visual, or that produces visual output. The user creates the program, or sees the result, via the connection and spatial arrangement of visual objects on his, or her, screen. These systems have been categorised into three basic types:

**Languages for Processing Visual Information** — These languages process visual information such as picture databases. They are motivated by the need to have easy-to-use languages for the manipulation and query of pictorial data. The languages allow one to directly reference pictures. However, even though the information handled relate to pictures, the languages still employ a textual interface to the visual data base.

**Languages for Supporting Visual Interactions** — Advances in hardware have paved the way for the use of graphical objects as a means of human interaction with the computer. However, without software, interactive visual environments could not become a reality. The languages in this category are specifically designed to handle the creation and manipulation of visual objects. Although the languages in this category support visual representations and interactions, they are still textual.

**Languages for Programming with Visual Expressions** — These systems represent the true *visual languages* since users are programming with the

visual elements of the language. The visual language has both a syntax and semantics. These languages accomplish, using visual objects, what would normally be done using traditional (textual) languages. Visual languages are also be broken down into three types:

**Form-Based Systems** —These languages provide the non-programmer with a simple or natural interface based on fill-in-the-blank type forms that are common in this society. These systems are not an attempt to make electronic paper executable, but to provide a program with a comfortable user interface. Most provide the user with a "sketch and go" type language which looks much like the output form.

**Diagrammatic Languages** —These languages use diagrams to represent programs. They evolved from programmers using diagrams on paper to define or document their programs. The prevalence of cheap graphics terminals has allowed these programming tools to become visual programming languages.

**Iconic Languages** — These languages are designed to play a deliberate role in programming. They have been designed from the ground up to replace textual languages. Iconic languages represent programming constructs and data using graphical icons. The iconic language elements represent their purpose, syntax and semantics graphically using shape, colour, actions and perhaps sound. The user constructs programs by the spatial arrangement and connection of these icons. Examples of such systems include logic design systems such as Log-

icWorks, HI-VISUAL(§2.4.3.3.2) and audio synthesis systems. The author's system will be another example of an iconic language.

The above framework is not rigid; there are not always clear boundaries between categories. Some visual programming systems do not fit into their assigned categories perfectly. They may contain aspects present in other parts of the framework. However, we must base our placement on what are the major goals and features of the system.

## 2.4 Summary of current systems

This section will survey a select few systems in a few of the categories under the framework discussed in the previous section.

### 2.4.1 Visualization of Programs and Execution

This subsection will survey systems that fit into the framework as visual programming systems that accomplish *visualization of programs and execution.*

As defined earlier these systems provide programmers with graphical support for the visualisation of programs. The systems can provide novice programmers with an understanding of how their programs work by using animation.

The simplest visualization system is *pretty printing.* Pretty printing shows the programmer the structure of programs through indentation and fonts. Such a representation can possibly show errors in the code and thus be a useful tool during debugging.

A much more sophisticated visual programming system uses multiple views of a program, or programs. The goal of these systems is not only to show a user his, or her, program, but also help the user in the programming process. A tool that can show multiple views of the program can be very helpful in assisting the user in this regard. Examples of views include flow-charts, parse-trees, syntax-directed editors and others.

There have been many systems developed in this area—for example PECAN[35], InterLisp[47], BALSA, MESA and Think Pascal. Only InterLisp will be surveyed here.

#### 2.4.1.0.1 The XEROX InterLisp System

The InterLisp environment developed by Teitelman[47] is an interactive programming environment based on the LISP programming language. Similar environments also developed at XEROX PARC include Smalltalk, MESA and Cedar[46]. They all use a multiple-view multiple-window interface.

InterLisp is a system that is designed for programmers who are interested in large artificial intelligence tools. Such programs begin with sketchy specifications and evolve with experimentation.

The development environment is designed with three considerations in mind.

- The typical Lisp programmer is engaged in experimental programming,
- They are willing to use computer resources to improve productivity,
- Users would prefer sophisticated tools, even at the cost of simplicity.

The developer felt that the environment should provide powerful tools for devel-

opment. That choice makes InterLisp difficult to learn, but worth the effort, because the complexity eventually aids productivity.

InterLisp contains a now common set of user facilities, and a couple that are not common. Common facilities include automatic error correction, a syntax-directed editor, a debugger and a sophisticated filing system.

The three most useful InterLisp aids are: DWIM, a Do-What-I-Mean error correction facility, MASTERSCOPE, a module bookkeeping system, and the PROGRAMMER's ASSISTANT, a command history system.

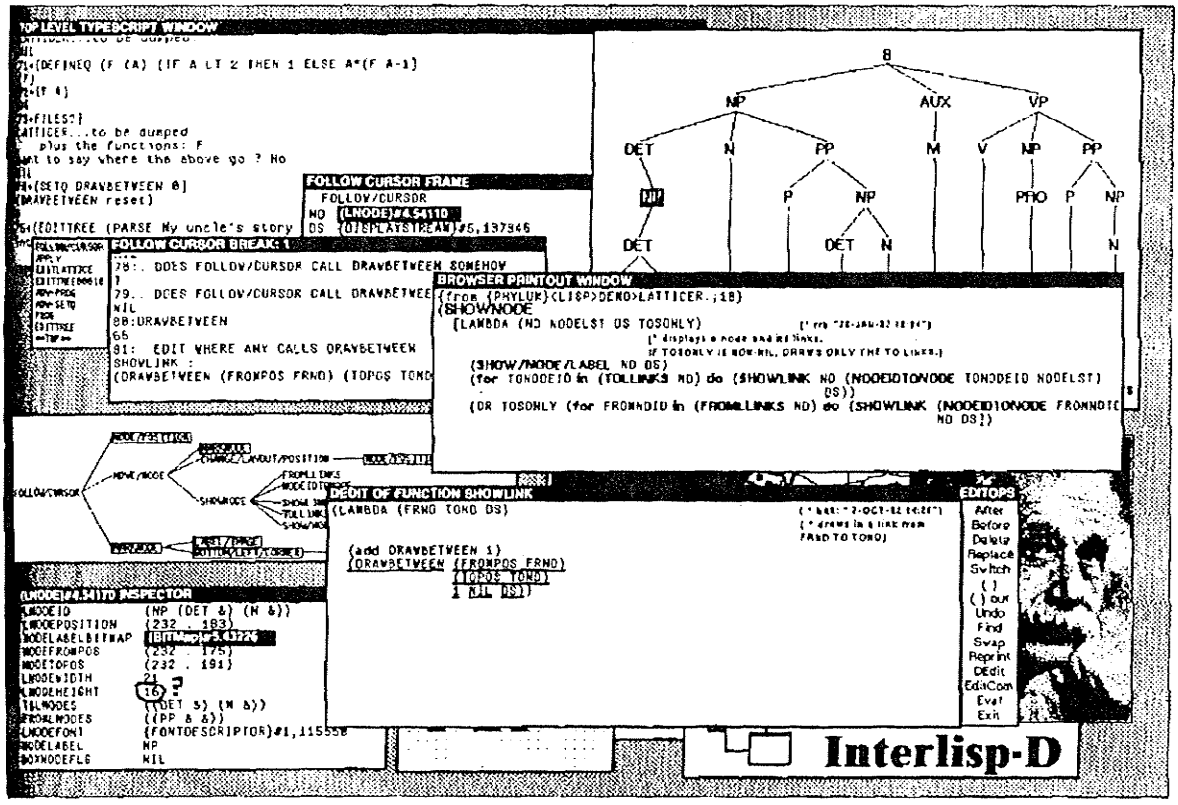



Figure 2.2: The InterLisp Programming Environment

Figure 2.2 shows an example of the InterLisp windowed environment.

### 2.4.2 Visualizing Program Design

The goal of these systems is to provide tools to specify programs, data structures and design decisions for large software projects. Clearly, an understanding of program design is important at the start of a project — and is more important as the project increases in size. A software designer generally uses drawings and text to both specify and maintain a project. This is the impetus for the graphical visualisation of software design.

There are two significant systems in this area. The Program Visualization (PV) system [6] and the PegaSys [31] system. These two systems have two different approaches to software design. PV provides users the ability to create arbitrary views of their design. PegaSys is more formal and provides a more rigorous environment. Only the PegaSys environment is surveyed here.

#### 2.4.2.0.2 The PegaSys Environment

PegaSys (*Programming Environment for the Graphical Analysis of Systems*) is an experimental system which uses graphics to aid in the formal documentation of programs[31]. Unlike other systems which simply use graphics to display program structures, PegaSys uses graphics and text to *document and verify* programs.

A program description in PegaSys consists of a hierarchy of related pictures. These pictures display dependencies between concepts such as processes and data-objects. A PegaSys developer can define his own concepts using these primitives. The user then constructs pictures to describe his formal software specifications using these concepts. These pictures are known as *Formal Dependency Diagrams* or FDDs.

PegaSys has the ability to reason about the FDDs within a single logic framework. This framework ensures that a user's picture has meaning both syntactically and semantically. The logic structure of a picture is represented internally using *form calculus*. A syntactically correct picture in PegaSys is known as the *form of a program*.

Each icon in PegaSys corresponds to one predicate in the form calculus. For example these primitives:

process(host1), process(host2),
module(line), type(pkt),
write(host1,line,pkt),
read(host1,line,pkt)

correspond to figure 2.3.

PegaSys constrains user manipulations to those described by the form calculus—similar to a structure-oriented editor. This constraint will ensure that graphical operations make sense. For example if a predicate has been defined to take two processes as its arguments, PegaSys ensures that both arguments are provided and that both are processes.

PegaSys allows one to build a hierarchy of pictures to reduce the complexity of a specification. Levels can be formed by successive refinement of concepts, or FDDs, in the higher levels. PegaSys checks that these successive levels preserve relationships at the higher levels.

The PegaSys verification process is performed invisibly. The user is only required to establish correspondences between the elements of a picture and the actual program constructs.

Each correspondence is specified by two selections, one from the picture, the other

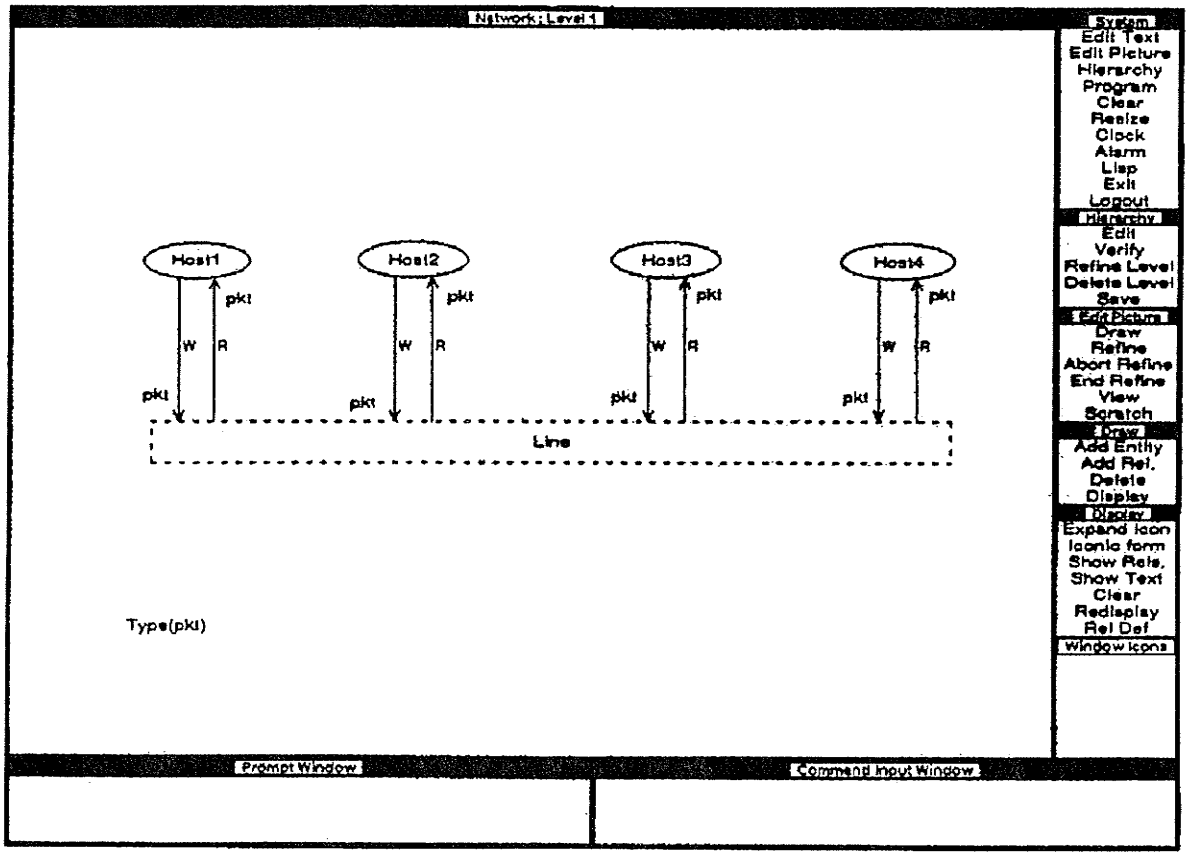



**Figure 2.3:** The PegaSys Documentation Environment

from a program. PegaSys requires that each entity (a lowest-level item) is associated with one, and only one program construct. For example Figure 2.4 shows the picture object Send_Host_Pkt is associated with the code segment SEND_HOST_PACKET. Once this step is complete, PegaSys can then attempt to verify that the program and the FDD hierarchy are logically consistent with the program it is intended to describe. This verification requires the proof of several form calculus formulas, which is done quickly and invisibly (due to the decidability of the form calculus).

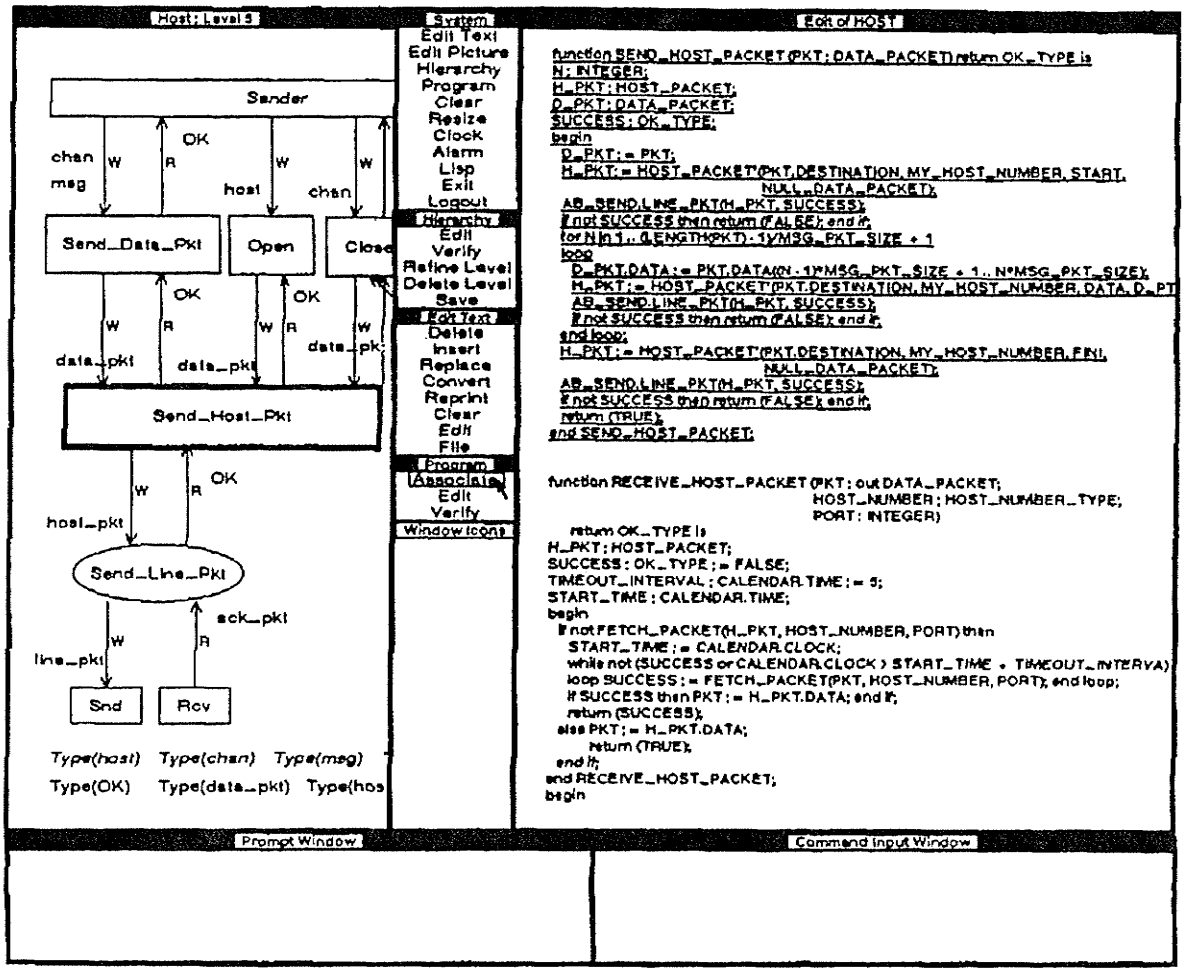



Figure 2.4: Example showing Module "Send_Host_Pkts" Associated Code Segment

### 2.4.3 Visual Programming Languages

Visual languages allow users to create and run programs with graphical objects or expressions. They are more commonly called *visual programming languages.*

A *visual programming language* can be defined as a language which uses some visual representation to accomplish what would normally have required a traditional textual programming language.

The individual components of a visual language posses both a syntax, and in

many cases semantics. The following sections survey languages contained under the framework category *programming with visual expressions*. This covers languages in diagrammatic, iconic and forms-based languages.

#### 2.4.3.1 Diagrammatic Systems

Diagrams have long been used for the documentation of programs and illustration of data structures and algorithms. These graphical aids did not comprise the program themselves, and they were not executable. The falling cost of graphical workstations has allowed these "paper tools" to jump to the screen and become visual languages.

Among the first diagraming tools is the *flow chart*, originally developed for assembly language programmers. Since assembly language cannot enforce structured programming, the flow chart's purpose was to show programmers the structure of their programs. This prevented them from becoming lost in the intricate details of their code.

With the advent of high-level languages, which enforced structure to some degree, flow charts were imbued with structured extensions. Example systems which use *structured flowcharts* include FPL and Pascal/HSD. Probably the best known *structured flowcharts* are Nassi-Shniederman diagrams. Other types of charts include data-flow and state-transition charts.

I have chosen to survey Robert Jacob's state transition chart language which is designed to build user interfaces. Other examples of diagrammatic systems include GRASE[1] and PIGS, which are based on Nassi-Shniederman diagrams. Jacob's system is a good example of a state-diagram language system.

#### 2.4.3.1.1 State-Diagram Language for Visual Programming

Robert Jacob chose a state-transition diagram language for visual programming[24]. In particular he has designed a language for the creation and specification of user interfaces. State diagrams have been used by computer programmers — using paper and pencil — to describe algorithms; in particular, to describe user input to programs. State-transition languages are a good representation of user interfaces because they offer three properties:

- In each state, they make explicit the interpretation of all possible inputs.
- They show explicitly how to change to a state solely on the user input.
- They emphasise the time-life sequence of user and system actions in the dialogue.

The language discussed here is an extended version of state transition diagrams. The language is based on the conventional graphical diagrams that are used in finite automata. A set of nodes (states) with links between them (transitions). Each transition is associated with a single token of the user input. From a state, an input token initiates a transition that is labeled with that token. A transition may also be labeled with an output token — which provides output to the user, or processing which is performed by the system during that transition.

The state diagram language is part of a methodology for designing and specifing user interfaces. The method is outlined below, it consists of three levels:

**Semantic level.** — The semantic level describes the functions performed by the system. It defines *meanings* rather than *forms* or *sequences*, which are left to

the lower levels. The semantic level manipulates internal variables, no input or output operations are actually described. However, the manipulation of values read in from input and generation of values for output are described. For execution, these functions are coded in a conventional (textual) programming language.

**Syntactic level.** — The syntactic level describes the rules by which sequences of tokens are formed into sentences. This level describes the sequence of the logical input, output and semantic operations, but not their internal details. A state transition diagram is a syntactic representation. A logical input or output operation is a *token*. Its internal structure is described at the lexical level. The syntactic level calls a token by name — like a subroutine. As an example, a syntactic representation of a desk calculator is shown in figure 2.5.

**Lexical level.** — The lexical level determines how input and output tokens are actually formed from the primitive hardware operations. This level identifies the devices, display windows, and positions with which each token is associated, and the primitive hardware operations that constitute them. For an input token the sequence of key presses, as well as any lexical output, is specified by the lexical-level description. For an output token, the lexical level specifies how the output token (windows, formats, positions, etc.) looks to the user.

The lexical level is represented in the same state transition diagram notation as the syntactic level. This consists of a separate state diagram for each input or output token, each of which can be called from the syntactic diagrams.

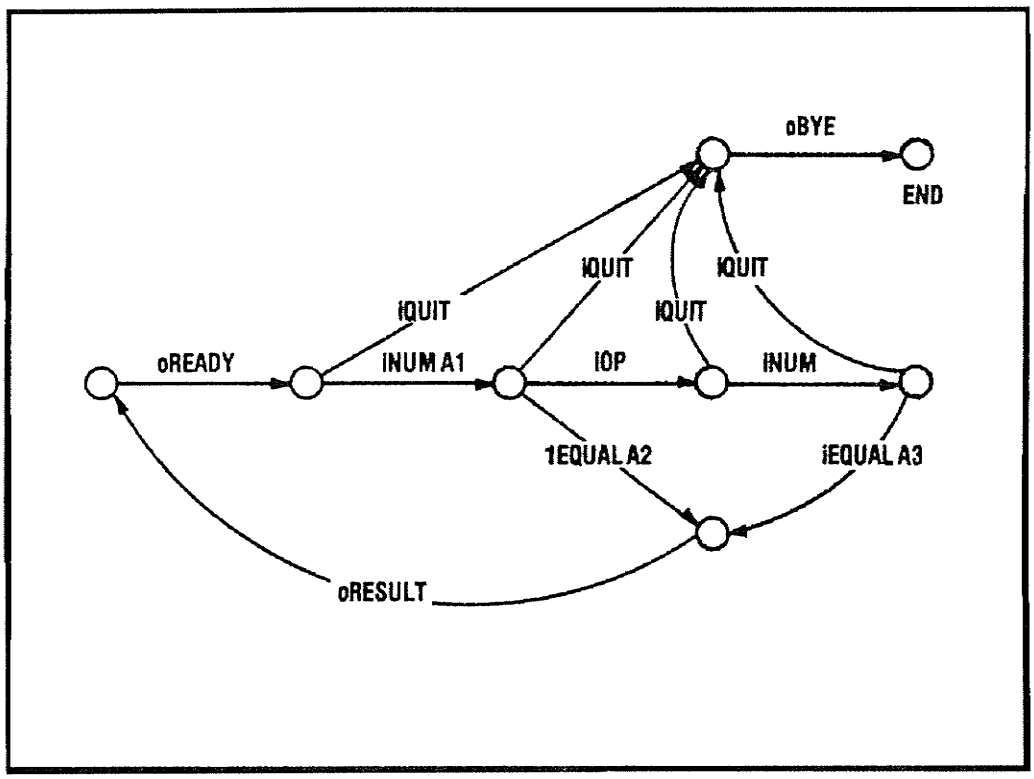



**Figure 2.5:** directed graph language

When execution of a diagram is first started, the top-level diagram pops up in its own window. The current state in the diagram is highlighted using inverted colours. As each state is invoked, a separate window may pop up containing that state's diagram. The current state of the program is represented by the lowest-level diagram (top-most window).

During execution the programmer can move the interpreter to a specific state by pointing at the desired state with the pointing device. Since the system is interpreted, arbitrary editing of a diagram can occur at anytime — even during execution.

The language described so far is static — the diagrams were first designed then entered as text, and then executed by an interpreter.

A visual programming environment for this language is nearly complete. In this version the programmer creates the state diagram using a graphical editor. The user then affixes the appropriate labels and actions (using the keyboard). Each of the nonterminals, tokens, are diagramed in their own window to reduce object complexity. Each object's semantic actions are written in a textual language, compiled and linked to the objects. Once the diagram is complete, execution can begin.

Another approach being considered by Jacob, is a visual interpreted language. The user is supplied with a pre-compiled basic set of actions. The programmer may then combine these actions to perform more complex actions.

#### 2.4.3.2 Forms-Based Systems

Because forms are used in everyday life (e.g. order forms, registration forms, application forms, etc.) the forms-based approach is perhaps the most natural interface between the user and data. The popularity of electronic speadsheets testifies to the success of the forms-based approach. Today the forms approach is used in menu-driven user interfaces, data entry, and database query systems.

One reason for using forms languages is the relative ease with which they can handle complex data structures, particularly hierarchical data. Most form-based languages recognise the need to represent and manipulate complex tables other than simple flat tables.

The system surveyed here, FORMAL, is designed to handle hierarchical data tables. Other systems, such as FORMANAGER, use an underlying flat relational table system, such as SQL, to store and retrieve the data. Most systems can display the retrieved data in a hierarchical form. FORMAL is much more flexible and usable

than other systems however.

#### 2.4.3.2.1 FORMAL

FORMAL is an experimental programming language developed by Shu[41].The goal of FORMAL is to provide non-programmers with a high-level environment to computerise their data processing tasks. FORMAL (Forms-Oriented MAnipulation Language) provides these data manipulation tools simply and usefully via a recognizable forms interface. It is designed for use by people in data-processing with minimal training.

FORMAL uses stylised form headings as visual representations of data structures. Figure 2.6 shows an example of a form heading and instances of PERSON data.

```
I-------------------------------------------------------------------------|
I                                (PERSON)                                 |
I-------------------------------------------------------------------------|
IENO|DNO|  NAME  |PHONE|JC|   (KIDS)   |       (SCHOOL)          |SEX|LOC|
I   |   |        |     |  |------------------------------------|   |   |
I   |   |        |     |  | KNAME |AGE|  SNAME  |   (ENROLL)   |   |   |
I   |   |        |     |  |       |   |         |--------------|   |   |
I   |   |        |     |  |       |   |         |YEARIN|YEAROUT|   |   |
I=========================================================================|
I05 |D1 |SMITH   |5555 |05|JOHN   |02 |PRINCETON|1966  |1970   |F  |SF |
I   |   |        |     |  |MARY   |04 |         |1972  |1976   |   |   |
I-------------------------------------------------------------------------|
I05 |D1 |SMITH   |5555 |05|JANE   |01 |         |      |       |F  |SF |
I-------------------------------------------------------------------------|
I07 |D1 |JONES   |5555 |05|DICK   |07 |SJS      |1960  |1965   |F  |SF |
I   |   |        |     |  |JANE   |04 |---------------------|   |   |
I   |   |        |     |  |       |   |BERKELEY |1965  |1969   |   |   |
I-------------------------------------------------------------------------|
I11 |D1 |ENGEL   |2568 |05|       |   |UCLA     |1970  |1974   |F  |LA |
I-------------------------------------------------------------------------|
I             .                                        .                  |
I             .                                        .                  |
I             .                                        .                  |
I-------------------------------------------------------------------------|
```

**Figure 2.6:** Example output of a FORMAL process

The *form heading* is on the top line, and the lower-level components are grouped

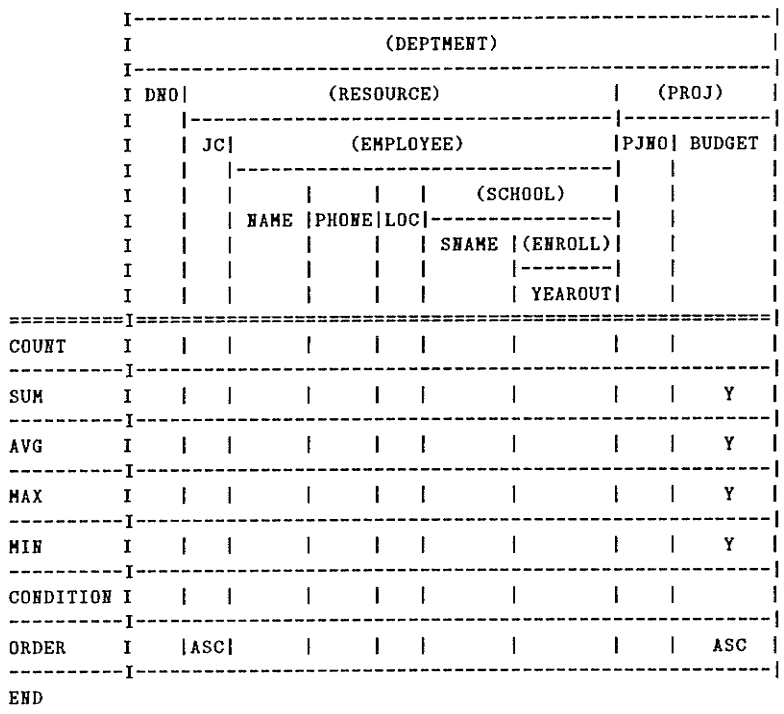



**Figure 2.7:** Example definition of a FORMAL form process

below it. The names of components of groups are placed under the associated group names. Groups may be either repeating or nonrepeating, a repeating group is denoted by parentheses.

To view the data, values are displayed under the form heading. The small size of the form heading enables the visualization of many instances at one time. This is advantageous since in the data-processing environment, many instances of data may need to be manipulated as one.

To perform data manipulation in FORMAL, a form process—in addition to form headings—must be defined. A FORMAL program consists of one or more form specifications. Each *form specification* can either define a form or a *form process.* In

general, each *form process* takes one or more forms as input and produces another form as output.

To define a form, the user starts with the *form heading* and proceeds to fill in the data types of each field. Optionally the user may specify whether the field is a *key field*, by using the keyword KEY, and may specify the ordering of the field using the keyword ORDER. An example of defining a form is shown in figure 2.7.

```
DEPTMENT: CREATE DEPTMENT

          I-------------------------------------------------------------|
          I                        ( DEPTMENT )                         |
          I-------------------------------------------------------------|
          I DNO|                (RESOURCE )            |    (PROJ)     |
          I    |---------------------------------------|---------------|
          I    |    JC  |          (EMPLOYEE)          | PJNO  | BUDGET|
          I    |        |------------------------------|       |       |
          I    |        |    |      |     |  (SCHOOL)    |       |       |
          I    |        |NAME|PHONE| LOC |--------------|       |       |
          I    |        |    |      |     |SNAME|(ENROLL)|       |       |
          I    |        |    |      |     |     |--------|       |       |
          I    |        |    |      |     |     | YEAROUT|       |       |
==========I=============================================================|
SOURCE    I         PERSON                          |PROJECT| < 1 > |
----------I-------------------------------------------------------------|
<1>       I PROJECT.PROJ.COST TIMES 1.5                                  |
----------I-------------------------------------------------------------|
DATATYPE  I                                                  | NUM(9)|
----------I-------------------------------------------------------------|
MATCH     I   PERSON.DNO,  POROJECT.DNO                                  |
          I   ELSE PERSON, PROJECT PREVAIL                               |
----------I-------------------------------------------------------------|
CONDITION I    |GE '05'|          | 'LA'|                               |
          I    |       |          | 'SJ'|                               |
----------I-------------------------------------------------------------|
ORDER     I ASC|  DES  | ASC|                                           |
----------I-------------------------------------------------------------|
END
```

**Figure 2.8:** Example of a form process and its output below

To construct a *form process* the user starts again with a *form heading* and fills in the information required to perform the data processing.

The keyword SOURCE specifies where the data for the specified fields is to come from. The source may be specified in many ways:

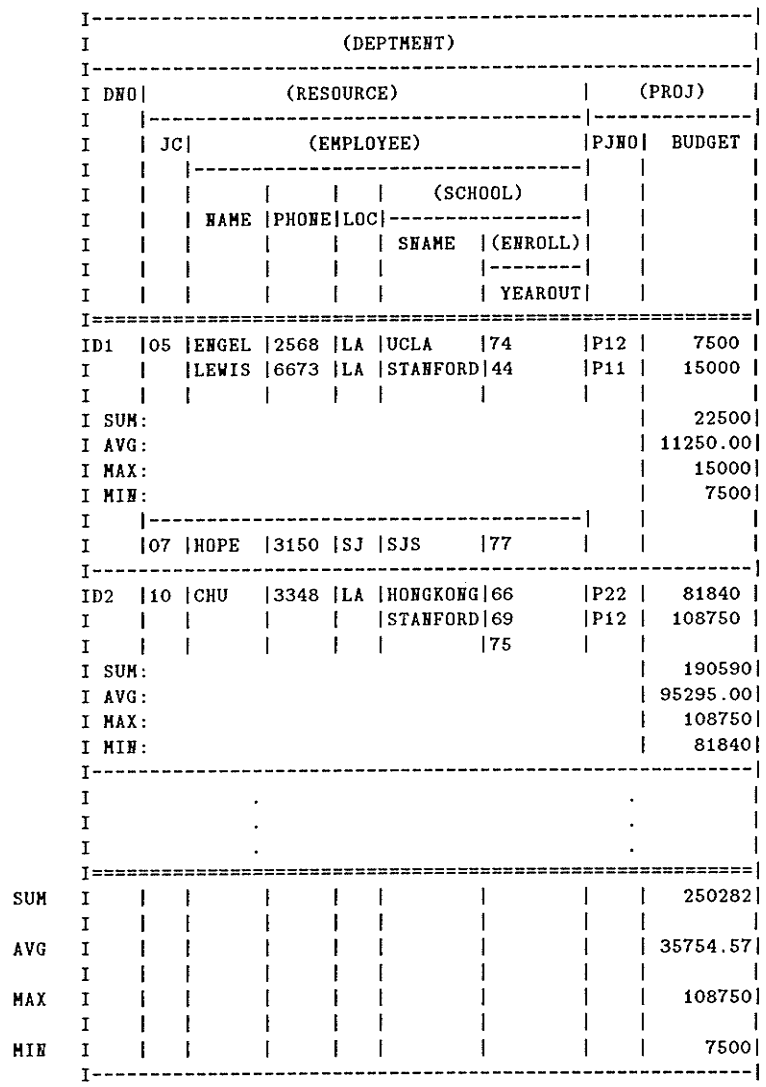



**Figure 2.9:** Example of form output by form process in Figure 2.8

- An asterisk specifies that the value of the field(s) is to be supplied at run time.
- A form name specifies that the value of the component(s) is to be obtained from the corresponding component(s) on the named form.

- An expression specifies the computation of a new value.
- A varying assignment specifies a case-by-case evaluation.
- A name of a user program to be invoked.

The notation $< n >$ in Figure 2.8, is a "note" expression, it is used to denote a case-by-case varying assignment according to the expression contained in the note.

The CONDITION keyword is a way of specifying the rules for selecting the appropriate instances from input. Conditions specified in two or more rows are ORed, and conditions for different fields are ANDed.

The MATCH keyword is used to bind two input forms together at the specified fields. This makes multiply sourced forms much more meaningful. For example we want to tie together the project information in a department with the employee information for the same department.

Figure 2.8 and figure 2.9 show an example of a FORMAL form process and its output.

#### 2.4.3.3 Iconic Systems

Iconic language systems use icons exclusively, or extensively. They are specifically designed as programming tools. Historically the term icon is associated with religious images. In computer science it is a pictorial representation of an object that can be used in direct data manipulation operations.

Icons are designed to represent either the data that the icon holds, or the action that will be performed when the icon is activated. Unfortunately, not all icons can be designed in this way — some icons are inherently ambiguous.

Icons have found their way into visual programming, and especially visual languages. This is because programming constructs can be represented more intuitively to the programmer. Rather than having to look at lines of text and form an image of what his program structure looks like, the iconic system can draw its structure.

Another advantage is in educating beginning programmers. Because iconic languages can animate the programmer's algorithms and allow him to change them on the 'fly', programming is easier to learn. This is the goal of many systems such as Pict and 'Show and Tell'. Other iconic languages include Tinkertoy[12], IDEOSY[16], HI-VISUAL[50, 30], THINGLAB[5].

At the lowest level they are all designed to use icons as programming constructs, yet all have different approaches to this goal.

I have chosen to survey TinkerToy, HI-VISUAL and Pict. TinkerToy is perhaps the best example of a visualisation of the Lisp language. HI-VISUAL was an early entry into visual programming, it has evolved to its current ('86) level. Pict is an excellent system for beginner programmers, it uses almost no keyboard input.

#### 2.4.3.3.1 Tinkertoy

The Tinkertoy visual language developed by Mark Edel[12] is one of several iconic languages based on Lisp. Others include ICONLISP[15] and VENNLISP.

The Tinkertoy icons resemble building blocks, they have input and output connectors on them to which other icons may be connected to build programs or structures. Examples of Tinkertoy icons are shown in Figure 2.10. The output of a Tinkertoy program is an iconic structure, which can be manipulated in the same way as normal icons.


**Figure 2.10:** Examples of TinkerToy icons.

Editing and creating programs in Tinkertoy involves the picking up, moving and connecting of icons. Programs in Tinkertoy are "built" by snapping icons together. This snapping gives important feedback to the user, because only syntactically correct objects sequences will snap together. This helps the user reduce errors in the programming process.

The user does most of his work with the mouse. However, typing is still needed as the user must choose his functions by typing its name. Thus some knowledge of Lisp is required. The user may also type in a Lisp expression and Tinkertoy will convert it to its iconic form.

Edel notes that there are significant problems with Tinkertoy:

- Compactness. This is a common problem with iconic languages. Tinkertoy programs are 20 percent larger than their text equivalents, according to Edel. (This figure seems rather low but Edel does not explain precisely how he computes it. See section 4.3.2 of this thesis for a more credible size comparison.)

- The Tinkertoy prototype can't execute the icon program, Tinkertoy must first convert the program to Lisp and then execute it.

- The graphic pretty printer (router) is far too slow for use in a highly interactive environment where you are continually opening and closing windows.

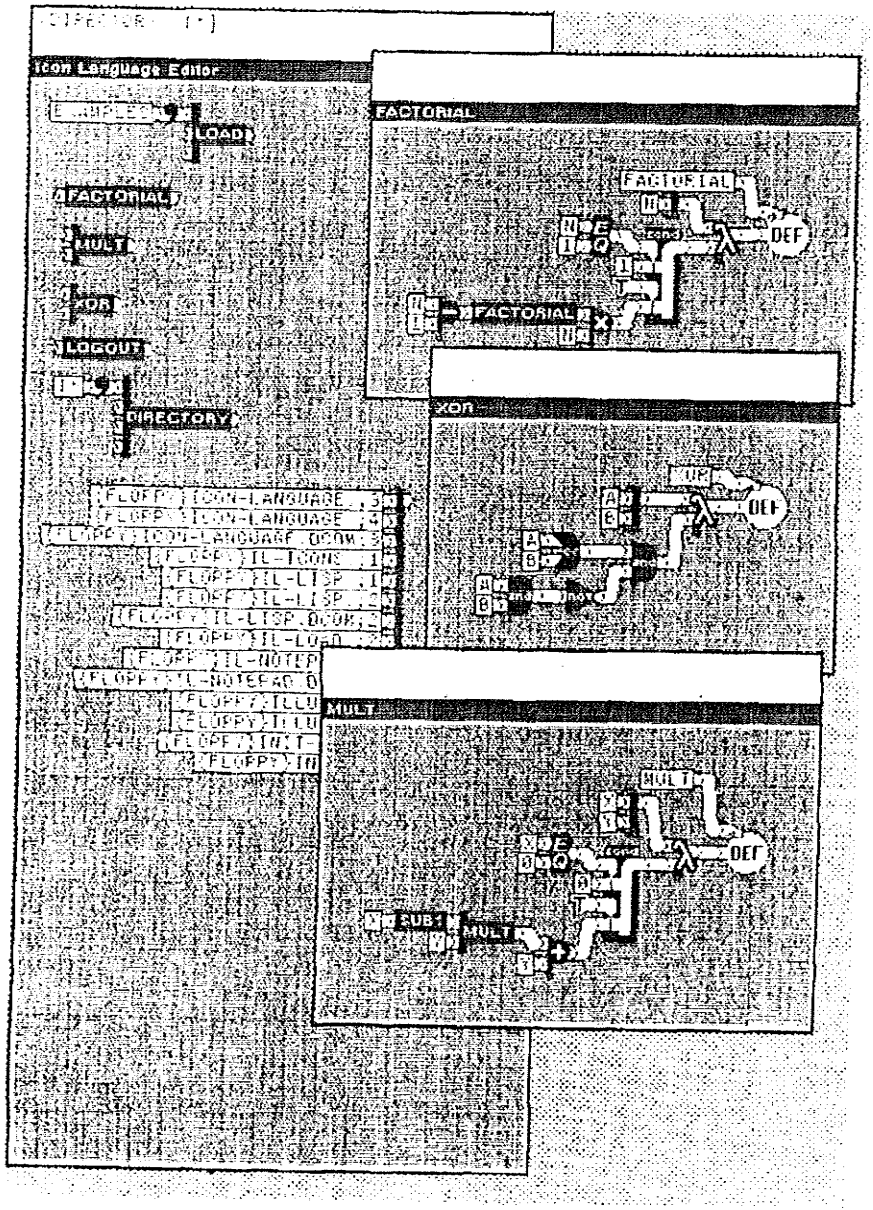



**Figure 2.11:** The TinkerToy programming environment.

An example of a Tinkertoy session is shown in figure 2.11. The large central

window is a system interface, the three smaller windows are program editors. At the top of each window is the text area where the user types in his expressions. Functions and structures occupy the same workspace and can be used identically.

#### 2.4.3.3.2 HI-VISUAL

The HI-VISUAL system was originally proposed as a visual language for supporting visual interaction in programming[30]. It has been extended into a general purpose visual language by providing facilities for iconic programming[50]. HI-VISUAL provides an interactive iconic programming environment using the data-flow paradigm. Each icon is regarded to be a function module which has inputs and outputs. The program is constructed by specifying the connections between inputs and outputs of the icons. In this way the data flows from the initial icon through the following icons until it reaches the end, where it is output to the user.

In HI-VISUAL there are seven types of icons representing the following (Figure 2.12):

a) DATA – Characters, numbers and images. It may also represent an intermediate result. Structured data may be defined hierarchically by combination of existing data.

b) DATA TYPE – A specific class of data.

c) PRIMITIVE – A built-in function supplied by the system.

d) PANEL – A set of icons without interconnections.

e) PROGRAM – A program created by the user.

f) CONTROL – The flow of control, such as loop, conditional branch, input or output.

g) COMMAND – A system function such as run, edit etc.

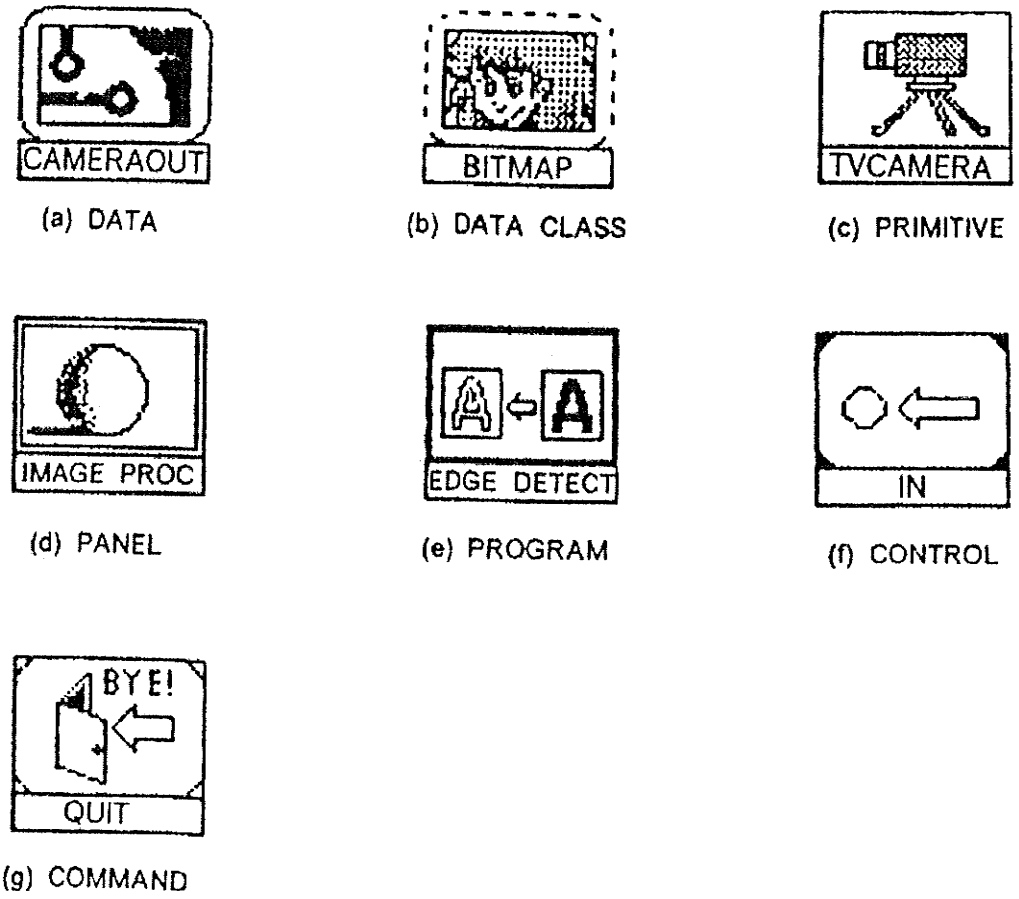



**Figure 2.12:** The HI-VISUAL programming icons

As an example of programming in HI-VISUAL, lets look at how a program is constructed for image processing.

While program development takes place the HI-VISUAL window is broken into three parts:

- The program specification area

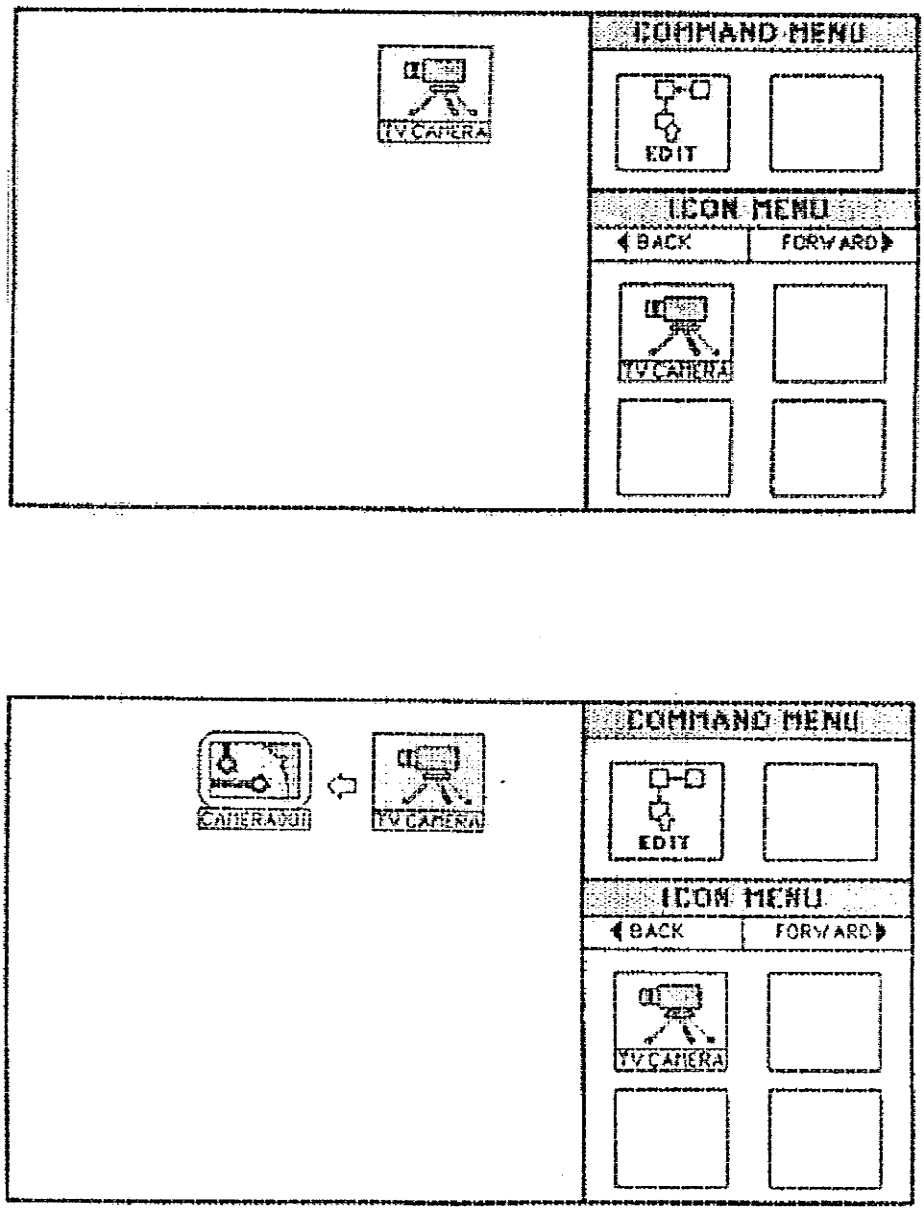



**Figure 2.13:** The HI-VISUAL programming screen

- a menu for choosing the programming icons, and
- another menu for choosing command icons.

The user selects an icon from the programming icon menu area and places it in

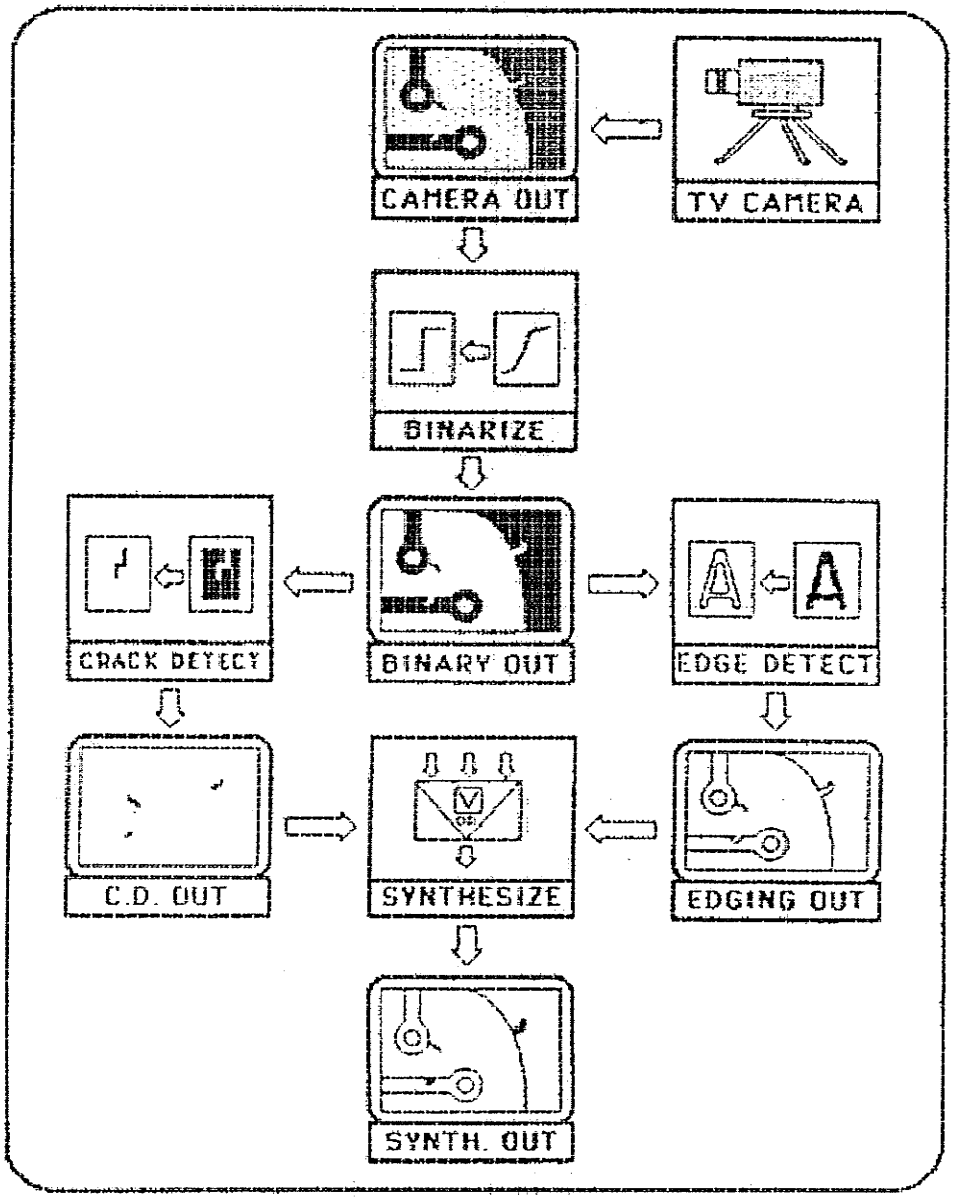



**Figure 2.14:** Program construction in HI-VISUAL

an appropriate place in the progam area. The user then attaches the icon to its data source, if necessary. The icon is immediately activated and executes itself returning a DATA icon representing the resultant data. In figure 2.13 the user has chosen the TV CAMERA icon, and has placed it in the program area. A DATA icon, CAMERA

OUT, is created representing the images from the camera. If the TV CAMERA is not operating a DATA TYPE icon of the expected type is displayed instead of the actual data.

Programming in this way is effective since the user is always referring to the previous output to guide his, or her, next actions.

Programming continues in the same manner until the user has completed the program (Figure 2.14). If the user receives unexpected results he can replace previous icons with other ones until he achieves the expected output.

HI-VISUAL also provides the programmer with a way to navigate through program development. This is useful when user knows the types of input and output data required, but does not know what functions will produce the output from the input. HI-VISUAL will provide a list of candidate icons which match either just input or both input and output types specified.

#### 2.4.3.3.3 PICT

The Pict programming system is a highly iconic visual language developed by Glinert and Tanimoto[19]. Pict is a graphical, flowchart based, interactive programming environment designed to aid in the implementation of programs rather than algorithm design. This approach is appropriate for two reasons:

- Many people find it hard to learn how to use existing (textual) programming languages.
- The vast majority of programs written by beginning students employ familiar algorithms.

Pict is designed to handle the small, but not trivial programs often assigned to introductory programming students.

The Pict philosophy of programming differs from the traditional routine of selecting, or designing the algorithm, then choosing names for the variables and encoding the algorithm into text to be compiled by the appropriate compiler. The Pict programmer creates his programs in the following manner:

- Select images that *visually represent* the data structures and variables needed.
- *Draw* the desired algorithm as a logically structured, multidimensional picture.
- If the program isn't doing what is expected, *see where* and *when* errors occur.

The Pict environment itself is designed to be nearly (with the exception of numbers and HELP messages) nontextual. Users sit in front of a colour display and draw their programs using an input device (a joystick). Users communicate with Pict through menus of icons. Program and subprogram names are icons, variables are single coloured icons. Control structures are represented by coloured directed paths that can actually be seen. User drawings are not free-form, they are similar to jigsaw puzzles with adjacent, predefined components and connectors.

Working in Pict is working at a language level similar to that of BASIC or Pascal, but is done visually. User programs may be recursive and contain chains of subprogram calls. During program testing PICT uses simple animation to make the program come to life. If revisions are required the user can easily return to the editing

subsystem and make changes.

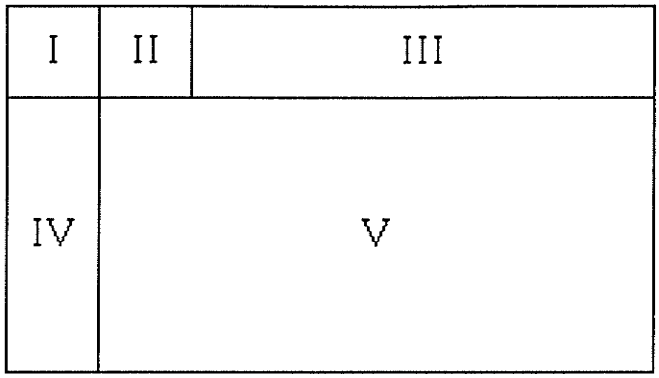



**Figure 2.15:** Diagram of PICT display areas

Figure 2.15 shows how the PICT display is broken into five parts:

I - program icon

II - subsystem indicator

III - help bulletin board/data structure display

IV - system menu/input keypad, and

V - user program easel.

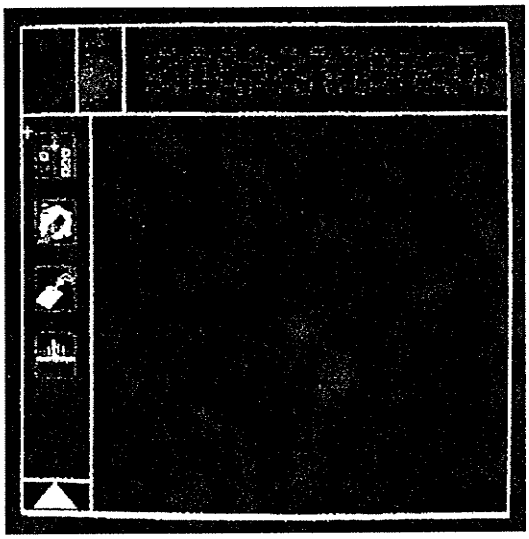


**Figure 2.16:** PICT/D Initial Screen

Figure 2.16 shows the Pict display when the user initially sits in front of the screen. The main menu along the left side show the four major subsystems; from top to bottom:

- *Programming*, represented by a small flowchart.
- *Erasing*, represented by a hand holding an eraser on a blackboard.
- *Icon editor*, represented by a hand writing on a sheet of paper and
- *User Library*, represented by a book shelf.

Figure 2.17 display a sample run of a program which calculates the factorial of an inputted number.

This example shows what happens. A numeric keypad appears along with red [1] $\times$ "reject" and green $\surd$ "accept" soft buttons. When a read operation is encountered the keypad flashes different colours to alert the user that input is required. In the top

[1] original images are in colour

display area, the four numeric registers (corresponding to the four colour variables available in Pict), red, blue, green and orange, are also displayed.

Once input is accepted a white box moves along the path showing flow of control. Eventually the program reaches the call to the factorial function (Figure 2.17, top right), This causes the code for the factorial function to be displayed (Figure 2.17, bottom left). When control reaches a *stop* sign, execution stops and a green check mark appears in the program name area (not shown). Notice the runtime stack on both sides of the programming easel which indicates the depth of execution to the user.

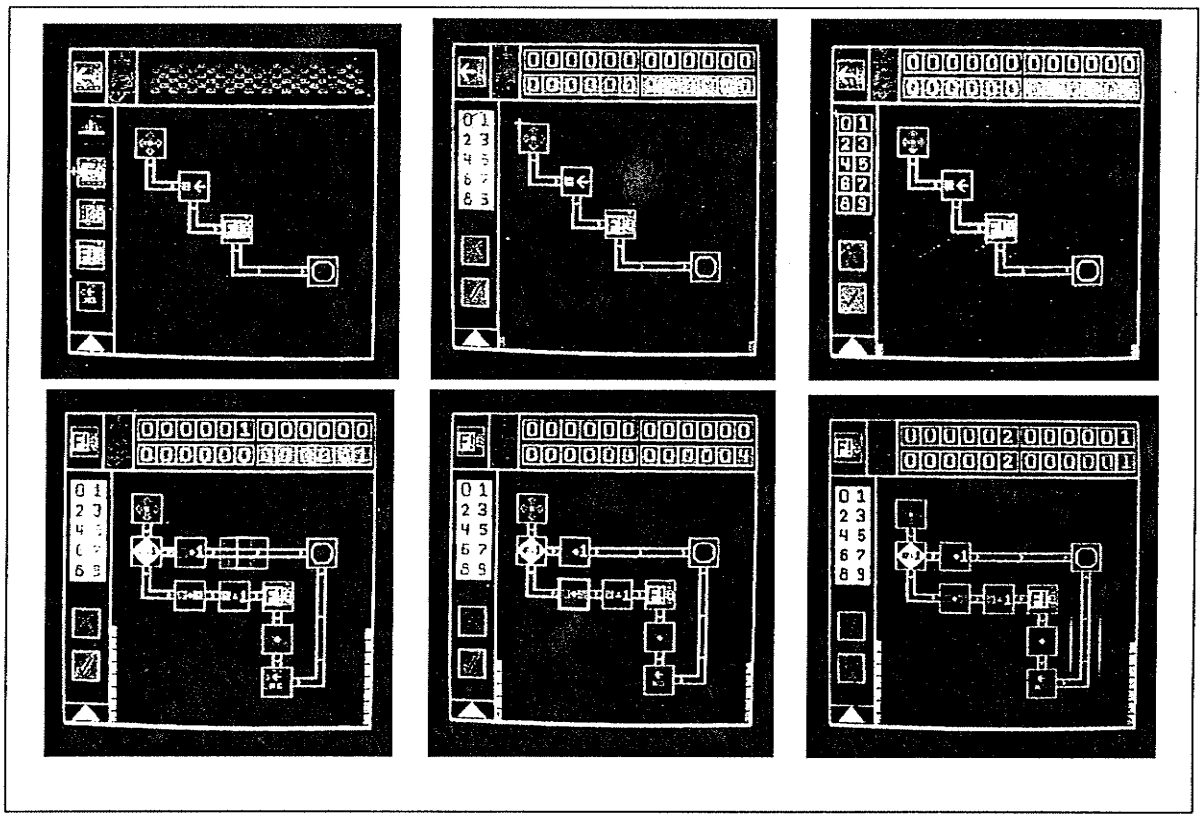


**Figure 2.17:** Sample run of PICT factorial program

Pict provides easier and more natural environment for novices to learn programming. The use of colour graphics and runtime animation and a simple user-interface is a factor in its success. Pict users never use a keyboard and do all their programming by manipulation and selection of icons. However users must still understand the basics of programming, variables, flow-of-control, recursion, algorithms and run-time stacks.

## 2.5 Conclusions

This chapter has provided a survey of a small number of visual programming systems. I have defined what *visual programming* is, visual programming is the meaningful manipulation of visual objects in the programming process. In addition a visual programming framework developed by Shu[42] was presented along with a description of each category.

The surveyed systems were placed in this chapter according to the category into which they best fall. Tinkertoy (§2.4.3.3.1) for example is a iconic visual language and was placed accordingly. PegaSys (§2.4.2.0.2) is a visual environment for the documenting of programs, and so was placed under *visualizing software design*. However, most systems contain aspects from other parts of the visual programming framework. For example HI-VISUAL (§2.4.3.3.2) provides a means to manipulate visual information, and so could also be placed there.

Most current work in visual programming has been focusing on the practical applications of visual programming. Work has increased in the formal aspects of visual programming, Chang's work on pure icons[8] and work by others is establishing

a formal foundation.

Most of the surveyed systems were implemented on hardware that is now considered obsolete, current hardware runs circles around them. With the increasing power of workstations new developments should increase the general applicability of visual programming making it palatable to the commercial developer.

# Chapter 3

# The Perseus Language and System

## 3.1 Introduction

In this chapter the author will introduce the reader to the Perseus programming language. The meaning and visualisation of each of the language elements will be described. Sample programs in Perseus will be presented as well as the construction of a complete program. The discussion of design issues will be interspersed throughout the chapter wherever an important design decision was made.

The Perseus system itself is named after the star for the rescuer of Andromeda.

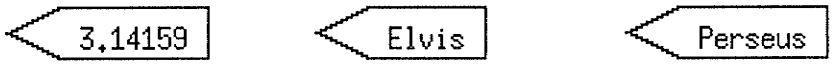
## 3.2 A Tutorial in the Notation and Semantics

Perseus is an *iconic visual programming language*. Rather than typing in lines of code using the keyboard, the user constructs programs using graphical objects, icons. These icons can also be executed producing output. Perseus implements a functional programming language. There are no variables in Perseus data enters a Perseus program at one end, and the result will exit at the other. Also there are no cycles in the connection of icons in Perseus, although a looping construct does exist.

The individual programming icons in the Perseus language consist of atoms, constants, sequences, functions, composition, and functionals. The visualisation of these can be seen from the figures. In the following paragraphs the author will describe the visual shape and the use of each of these icons. Any important design issues regarding each icon will be discussed as well.

The first section will describe the element of Perseus known as an atom.

### 3.2.1 Atoms

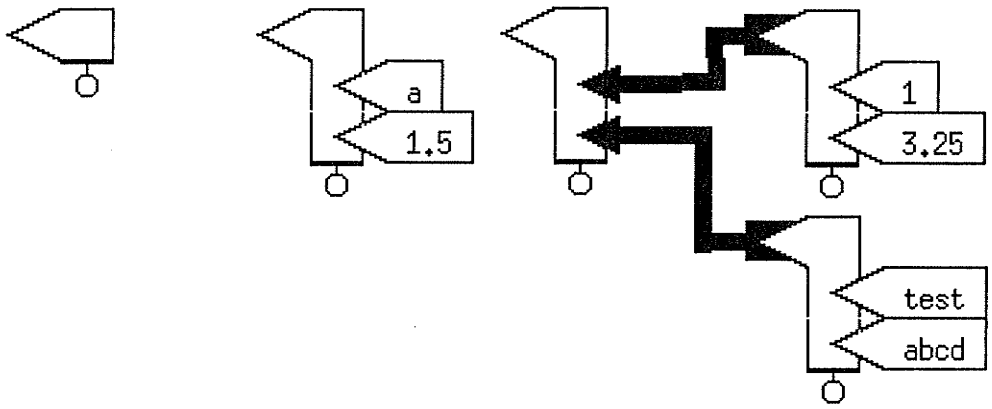



Atoms contain the data in Perseus. They are used as input to Perseus programs. In Perseus the atom icon is a box with a pointed end on the left side. The current value of the atom is centered within the box. The point on the left side is a connector which allows the user to connect the atom to other Perseus items with the appropriate connector.

An atom contains a string of characters. Some of these strings can be treated as numbers, or as the booleans "T" or "F", which represent True and False respectively. The value of an atom may contain no spaces; and is changed by a dialog box. An atom containing no characters is considered empty and will cause an undefined value to be output when executed.

*Undefined* is a empty value in Perseus. It indicates that an error has occurred in the program somewhere. It is output as the string "Undef'd" since Perseus currently prints output to a text terminal.

### 3.2.2 Sequences




Sequences are the other data icons in Perseus. They are used to create data structures with either no items, or many items. These items can be atoms, or other sequences. A sequence icon is initially a rectangle similar to an atom, but with a window-blind loop attached at the bottom.

Items can be attached to the sequence by the input connectors along the right side of the icon. These connectors are shaped to accept only the connection of an atom or a sequence. The user can vary the number of inputs available by pulling up or down on the blind loop at bottom of the icon.

If a sequence is sized so that there are no inputs, the sequence is considered to be NULL, and is then considered to be both an atom and a sequence. Any unconnected—or empty—inputs will render the entire sequence invalid, and cause an error to be output when executed.

### 3.2.3 Functions


The Function icons in Perseus are black rectangles with an input on the right and a output on the left. The function performs the named operation on the input and outputs a result. The name of the function is centered in the rectangle. The user chooses the function via a dialog box.

Each function has only one input and output connector. The input is triangular in shape to allow only an atom, or sequence to be connected. The user will usually attach a function to data and then execute it. Should the input be empty, the function will not execute and an undefined value will be output.

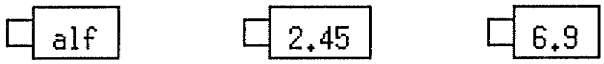
A function icon gives no visual clues as to the format of data it accepts. It is up to the user to know the format of input accepted by a function. Most functions in Perseus, however, accept either an atom or a sequence of two atoms and Perseus currently has a small number of functions.

Notice that the output connector is square in shape and hence incompatible with a function's input connector. To build programs, one must connect together a series of functions from the output of one to the input of the next. To achieve this, Perseus contains an icon called *Composition*, this element is described later.

Below is a list of functions which are currently in Perseus.

| | | |
|---|---|---|
| / | – | * |
| + | NULL | LENGTH |
| REVERSE | TAIL | HEAD |
| FIRST | SECOND | THIRD |
| FOURTH | FIFTH | SIXTH |
| SEVENTH | EIGHTH | NINTH |
| TENTH | AND | OR |
| NOT | EQUALS | ROTATE DOWN |
| ROTATE UP | ATOM | GREATER THAN |
| LESS THAN | GREATER OR EQUAL | LESS OR EQUAL |

### 3.2.4 Constants




The *Const* object is similar in appearance to the atom. Instead of a pointed connector, it has a function output connector. Its primary use is to supply a parameter to functionals. To the user, the interaction and data rules of *Const* objects are exactly those of atoms.

### 3.2.5 Composition

To construct programs in Perseus the user must connect together function icons. Notice, however, that the output of a function is not compatible with the input of a function. How is program construction done? It is done using the composition object.

The *Composition* object will connect the output of a function to the input of another function. Perseus will do its best to prevent looping caused by an illegal connection, but it is not guaranteed to always detect loops.

Once the user has constructed a legal connection, the composition object will be displayed as a solid black line, with an arrow at one end, and a rectangular magnet at the other. These represent the output and input of the composition. The arrow will fit into the input of the function, and the other end will fit over the output of another function. Composition will then route the output of one function to the input of the next function.

There is no way to split *Composition* to send output to more than one function. The construction functional—described later in this chapter—can be used to approximate this action.

### 3.2.6 Functionals

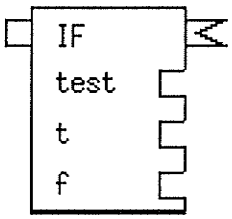
A functional is an object that combines other functions to operate on input in a new way. The composition object of the previous section is a simple example of a functional. A functional in Perseus has an input connector for data, and one or more function connectors for function parameters. These parameters will act on the data supplied by the functional and produce a result.

The functional icons are similar in appearance to a function's, except that they are white. The output and input are in the same positions, but below them are

rectangles each containing an indented connector on the right side. These connectors are the functional parameters and are designed to accept only function outputs.

Perseus contains five functionals, namely: If, While, For Each, Insert and Construction. Each of these will be described in one of the following subsections.

#### 3.2.6.1 If

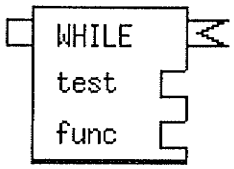



The *If* functional has three parameters, a boolean parameter, and one parameter to execute for each of the cases where the boolean evaluates to true, or false. The *If* icon thus has three parameter connectors and is shaped accordingly.

When executed *If* passes the input data to the boolean parameter. The function attached operates on the given data and must return a boolean result, either "T" or "F". If the result is true then the function attached to the "t" parameter is executed and the result returned as the result of the *If*. Otherwise the function attached to the "f" parameter is executed and the result returned similarly.

If an error occurs in any of the parameters, then the entire functional stops executing and returns the value *undefined*. This action holds true for all the other functionals as well.

#### 3.2.6.2 While

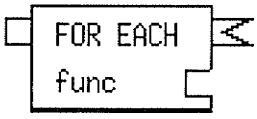



The *While* functional has two parameters, a test parameter and a func parameter.

When executed *While* will first apply the test parameter function to the input data. If the result is true,"T", then the func parameter is executed on the same data. The result of the func function then replaces the copy of the input, and is used as the input to the test function. This looping continues until the test result is false,"F", and the current copy of input is output as the result.

An example of using while is searching for a non-zero atom of a sequence, and then returning the remainder of the sequence.

#### 3.2.6.3 For Each




*For Each* is one of the most straightforward of functionals in Perseus. It has only one parameter, a func function.

The *For Each* will take its input—which must be a sequence—and apply the parameter function to each element of the sequence. The output is a sequence of the same length as the input sequence, each of whose elements was obtained by applying the parameter to the corresponding element of the input sequence.

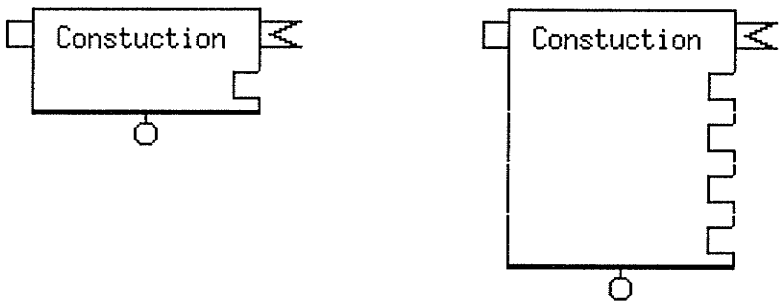
#### 3.2.6.4 Insert


The *Insert* functional takes only one parameter, a function func.

The input to the *Insert* functional must be a sequence and the parameter function must be able to be applied to a sequence of two items. The *Insert* applies the parameter function to pairs of items in the input. The last two items of the input will be passed to the parameter and the result of the invocation will replace these two elements in the input. Execution will continue in this manner until the input sequence contains only a single item. This item is returned as the output of the *Insert* functional.

An example of insert is finding the sum of a sequence of any length. A user would attach the "+" function as the parameter, then attach a sequence as input.

#### 3.2.6.5 Construction




The *Construction* functional can have from zero (0) to as many parameters as the user desires. The user can use the loop at the bottom of the *Construction* icon to resize the *Construction* to obtain the desired number of parameters.

When executed the *Construction* functional passes a copy of the input data to each of the parameter functions for execution. The result of the functional is a sequence made up of the results of applying each of the function parameters to the input data. The parameters are invoked from top to bottom. The results appear in the same order in the result sequence. Once all parameters have finished execution then the entire result sequence is output.

If an error occurs in any invocation of a parameter then the result sequence is deleted and the *undefined* value is output.

#### 3.2.6.6 Functional Design Considerations

The functionals were designed the way they are as both an experiment and to simplify the prototype implementation.

One problem with the design is that the user may be confused about from exactly where the inputs to functional parameters get their values. This may cause the user to incorrectly connect the data to the parameter functions, and not the functional of which they are a parameter. Other designs for the functional icons were considered such as:

- Encasing the parameter functions inside the functional block. this would be quite cumbersome to implement, since the functionals need to be dynamically sized to fit the icons attached to the parameters.
- Another alternative was to attach a dummy object to the rightmost input on the functional's parameter functions, if there exists one. This would be drawn in automatically by the system and clue the user not to connect

data there.

- Yet another possibility would be a connection line similar to composition, which would connect the functional to the input of the parameter function. this could be automatically provided by the system. this would explicitly clue the user as to where the input to the parameter functions was coming from, and prevent connection of data.

In the end the simplest form of functionals was chosen until more experience is gained with visual programming.

Another problem with Perseus is the method of execution of *While*. It can be quite confusing for the user.

Also the shape of the icons in general do not provide any clues to the internal data manipulation or the form of input expected. Inventing shapes that clearly represent the purpose of every function, or the form of expected input, or generated output is not an easy task.

## 3.3 Using the Perseus Language Editor

To start up the Perseus system; the user types "`Perseus`" at the terminal prompt. The Perseus system comes up on the screen and looks initially like figure 3.1. The frame is provided by the window system; Perseus is contained within this frame.

Within the frame the user can see the two pull-down menus *file*, and *edit*, on the menu-bar. The large area below the menu-bar is the programming canvas where the user creates and edits his or her programs. At the bottom left side there are four directional scrolling buttons, zooming buttons, and a panner object. The scrolling

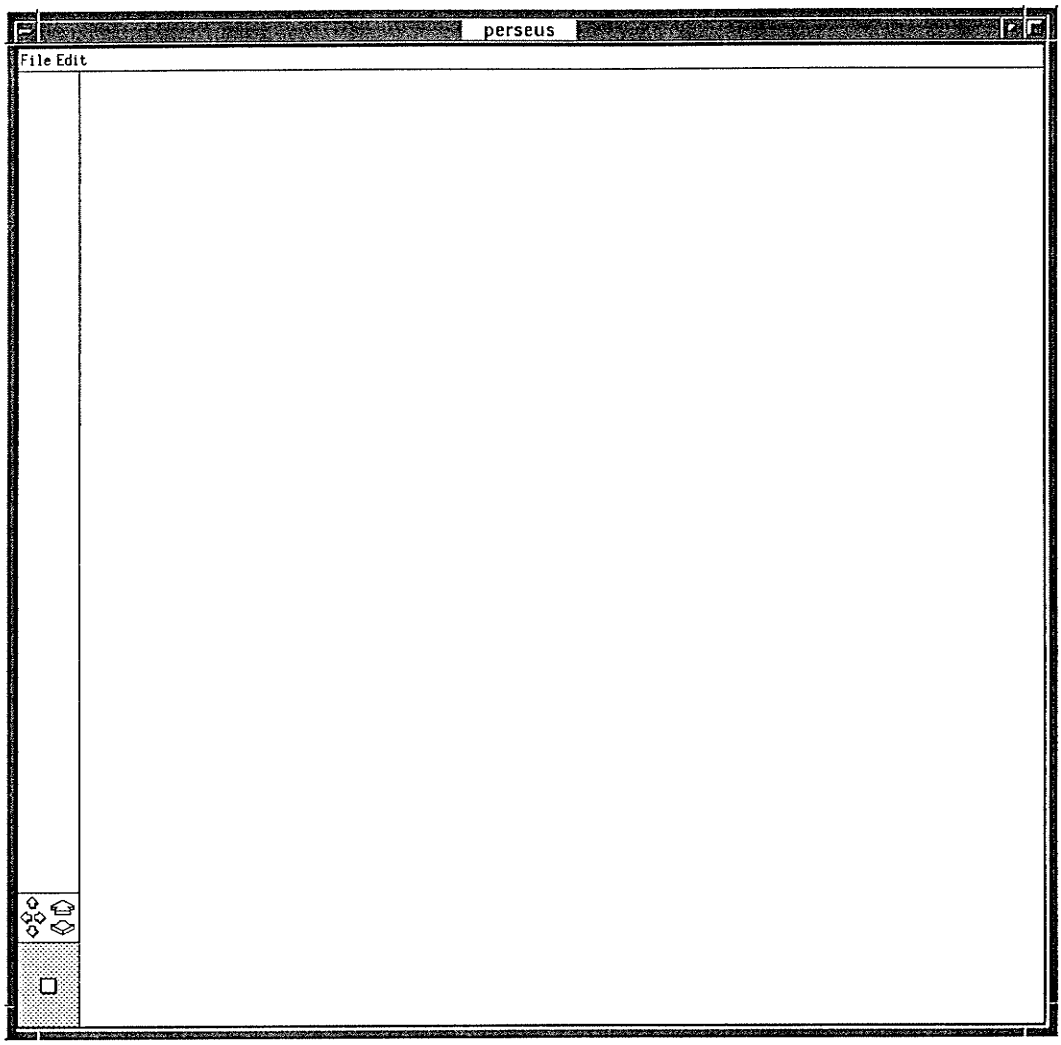



**Figure 3.1:** Initial screen of Perseus system.

buttons move the canvas in the four directions. The panner object shows the user the position of the visible portion of the drawing canvas. The zooming buttons are disabled, and have no effect.

The Perseus editor system is a mode-based editor; the user must select the correct mode from the menu before selecting an object on which to perform the action. It is not a selection editor where the user selects an object and then selects an operation from the menu.

To quit the Perseus system, the user pulls down the *file* menu. It contains only one item, "quit", which ends the user's session.

The other menu, "edit" contains operations to create a function, delete objects, edit an object's value, resize an object, and to execute a program (figure 3.2). The "function" menu item brings up a dialog box from which the user chooses the desired function to be created (figure 3.3). The sub-menus are available from the "Create" and "Functional" menu entries. This hierarchy helps to simplify the menu structure.

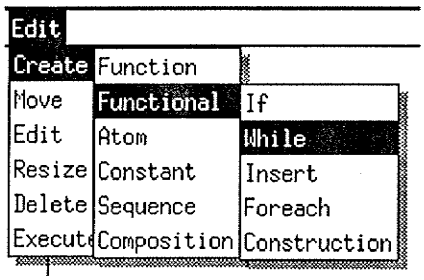



**Figure 3.2:** Perseus edit menu

The menu item "Resize" deals with the resizing of the objects with pull-down loops on them. Presently these are the objects *Sequence* and *Construction*. To stretch one of these objects, first select the menu operation "Resize", and then move the mouse pointer to the loop on the object to be stretched. Press and hold the left mouse button. A resize rectangle appears, and by moving the mouse up and down, the user can increase or decrease the number of connectors that object has. Releasing the mouse button creates or deletes the connectors as desired. Figure 3.4 shows an example of resizing *Construction*. The same procedure applies to *Sequence* objects.

As an example of using the Perseus editor, the author will present a sample construction of a small program. This program will add two numbers together.

Select the "Function" menu item, which brings up the dialog box. Select the "+"

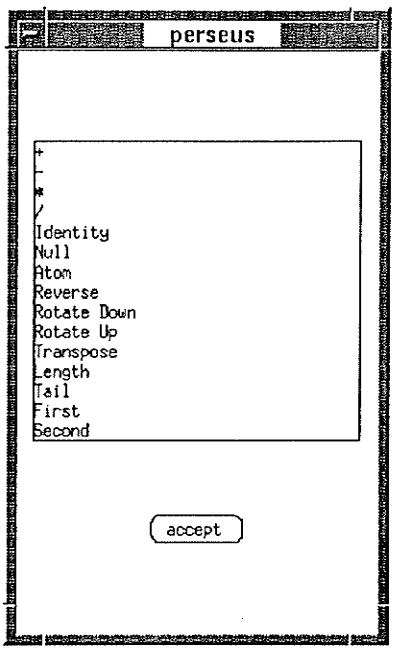



**Figure 3.3:** Function selection dialog box

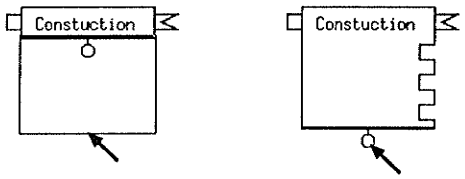



**Figure 3.4:** Example of stretching *Construction* object

item, and press "accept". By pressing and holding down the right mouse button, the user drags a bounding box of the function object. Releasing the button within the canvas places the function on the canvas at the current point.

Select the "Sequence" menu-item and follow the same procedure as above, but this time place the object on the input connector of the function. This *Sequence* initially is NULL; it contains no connectors for attaching other items. The "+" function accepts a sequence of two numbers, thus the *Sequence* must be stretched using the "Resize" operation to create two connectors. Place two *Atom* objects into the connectors on

the *Sequence*. Initially the *Atom* is empty, the user must use the edit dialog to alter the value of the *Atom*. select the "`Edit`" item and click on any one of the *Atom* objects. This brings up a dialog box similar to the one shown in figure 3.5. Type in a number in the text window and press the "`accept`" button. Do the same thing to the other *Atom*. This sample program should look similar to the one in figure 3.6.

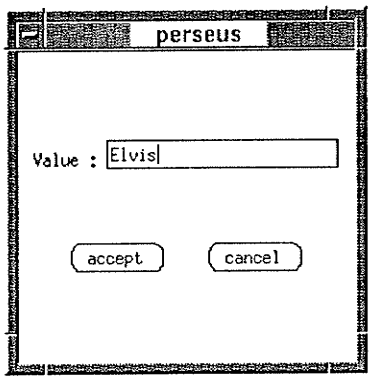



**Figure 3.5:** Example editing box

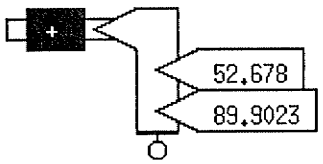



**Figure 3.6:** Example program to add two numbers in Perseus

This example shows the user the basics of creating a program in Perseus. The creation and placing of objects, and the connection of objects which will snap together. Recall that only compatible connectors can be connected. Also notice that typically program construction in Perseus will take place from right to left.

To execute this program, select the "`Execute`" menu item and click, with the left mouse-button, on the object to execute. In this example that would be the "+" function. The results will be printed to the terminal from which Perseus was started. The results will be sandwiched between "`Running...`" and "`Finished`" text. If an error occurs the text "`Undef'd` will be output. *Sequences* will be encased between parentheses, "(...)", and items are separated by commas, ",".

The next example shows the construction of a more complex program to calculate the inner-product of two vectors.

This program will require the creation of a *Sequence* of *Sequences*, the connection of these sequences using the composition connectors, and the use of composition to connect functions together. As well the creation and connection of *Functionals* is also used.

To calculate an inner-product the program needs two vectors of the same length. The first element of the first vector must be multiplied by the first element of the second vector, and so on. These numbers are then summed to produce the result which is the inner-product of the two vectors.

In Perseus, the input will be a *Sequence* of two *Sequences* each of the same length and each containing only *Atoms*. The first thing to do is to pair up the input, so the first *Atom* of the first *Sequence* is multiplied by the first *Atom* of the second *Sequence*. To do this the user can use the function "transpose". When given a *Sequence* containing two *Sequences* of the same length, "transpose" will produce a *Sequence* containing *Sequences* of pairs of *Atoms*. The user may experiment with "transpose" to see how it works. Choose the "`function`" menu item and choose and place the "transpose" function on the canvas.

The "transpose" function outputs a *Sequence* of *Sequences*, each having a pair of *Atoms*. These *Atoms* must be multiplied together, so for each of these pairs the multiply function must be applied. This can be done with the *For Each* functional with "*" as its parameter.

Choose "`Foreach`" from the "`Functionals`" menu and place it to the left of the "transpose" function. Create and place the "*" function into the parameter connector of the *For Each* object. The user will now have something similar to what is shown in figure 3.7.

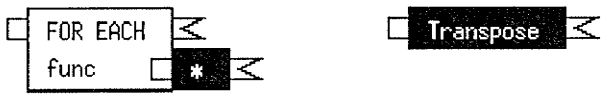



**Figure 3.7:** Example before using "`Composition`".

Recall, that to connect two functions together in Perseus, the user needs to use the *Composition* object. In the editor to create a composition between functions the user selects the "`Composition`" menu item. To create a connection between two objects first click the left mouse-button where the line is to start and click the middle mouse-button to end. Any direction changes can be made by clicking with the left mouse-button where ever a corner is desired. Figure 3.8 shows the *Composition* as the user is dragging and after release and connection of the *Composition*. *Composition* will not connect if the two connectors at the start and end are not compatible, or if the connection creates a loop.

So far, the program multiplies the pairs of elements and produces a *Sequence* of numeric *Atoms*. It is necessary to sum this sequence to obtain the result. To sum up

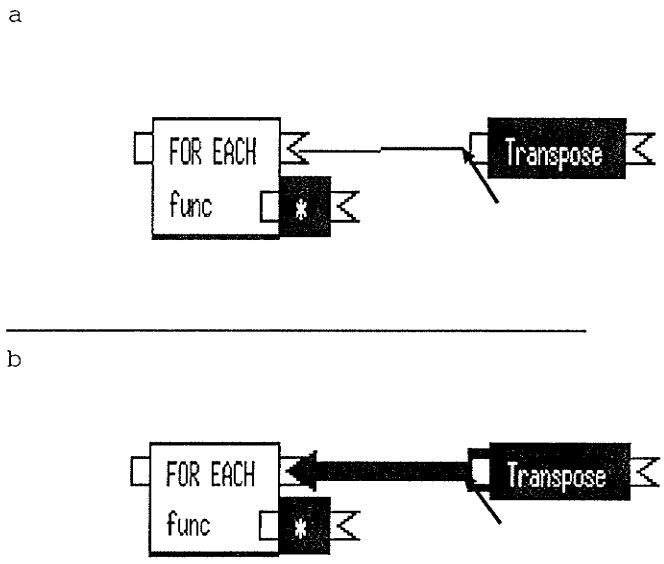



**Figure 3.8:** Connection procedure using "Composition"

the results of the *For Each*, the functional *Insert* is required. Recall that *Insert* takes a function which operates on pairs, and allows it operate on a sequence of any length. Create and place the *Insert* to the left of *For Each* and connect using composition to the output of *For Each*. Attach the "+" function as the parameter.

The program is now almost complete; all that is needed is to attach the input. This can be done similarly to the previous step. To space the data for easy readability, *Composition* can be used to connect the data connector of one *Sequence* to the data output of another. Then attach the *Atoms* to the appropriate connectors, and edit their values. The completed program with attached data looks as in figure 3.9.

Execution is exactly the same as for the previous example, clicking on the object from which output is desired. The program will be executed to that point. This is excellent for debugging. The user can just click on the object anywhere in the program and obtain the results to that point.

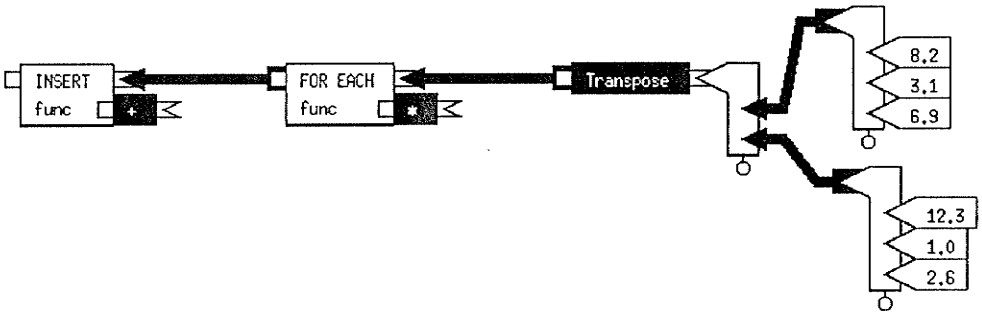



**Figure 3.9:** Completed inner-product program in Perseus.

# Chapter 4

# Comparison of Textual and Visual Programming

## 4.1 Introduction

In 1987 John Backus, the originator of FORTRAN and BNF grammars, detailed in a Turing Award lecture a functional language on which he had been working [2]. That language is called FP.

FP is a very small, very simple language for functional programming. In functional programming there are no variables, no intermediate results, and no looping. Data is supplied to the program at the start, and the result is output at the end.

Perseus is based on FP and implements many of the constructs available in FP. In the previous chapter no mention was made of FP to emphasise to the reader that Perseus is a stand-alone language. One can use it without any knowledge of FP.

During the next paragraphs I summarise the FP language using Backus' notation. In the examples, Perseus code will be shown for a comparison with textual FP.

## 4.2 Introduction to the FP Language

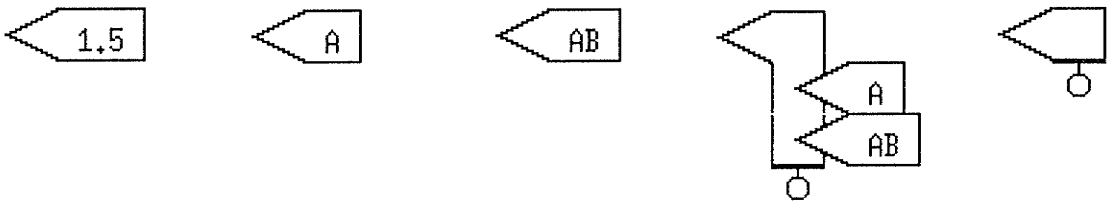
Backus' FP language system is composed of the following: a set of objects, functions, functional forms, and definitions.

An object is either an atom or a sequence. A sequence is represented by a list of objects enclosed in angle brackets, as: $<x_1, \ldots, x_n>$, where each of the $x_i$ are objects or are undefined. An undefined object is a object which has no value and is represented by "$\perp$". Atoms can consist of letters, digits and symbols not used by FP. Some of the strings belong to the set of atoms called "numbers". The atom $\phi$ denotes a empty sequence and is both a atom and a sequence. The atoms T and F denote "true" and "false" respectively.

Some examples of FP objects are:

1.5, A, AB, <A,AB>, $\phi$

The same objects in Perseus appear as:




Notice that the atom $\phi$ is denoted as a sequence icon with no connectors. It looks just like a atom but also has the pull-down loop that a sequence has, emphasising that it is also a atom and a sequence.

The FP language has a single operation, application. The notation

$f{:}x$

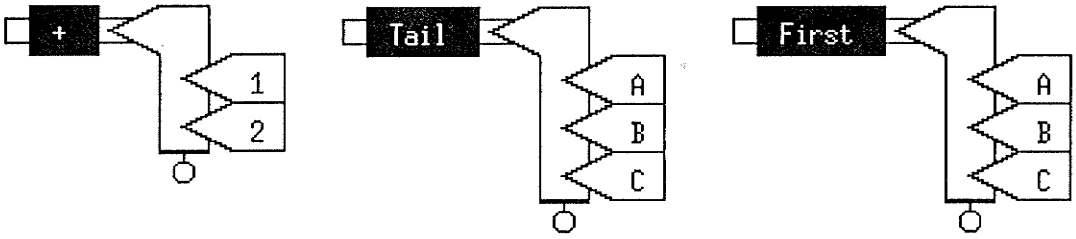
denotes that function $f$ is being applied to an object $x$ producing another object as

output.

Some examples of application are:

$+:<1,2>$, $\text{tl}:<A,B,C>$, and $1:<A,B,C>$

The same applications in Perseus look like the following:




Notice how the user can clearly see in Perseus where the input to the program is provided.

The functions in FP map objects into objects. They are roughly equivalent to imperative language functions. FP contains a robust set of functions, rather than a set of weak functions. Backus defined FP so that it contained a powerful set of functions to do real work, rather than have the user create these functions using the definition function.

Some examples of functions in FP are:

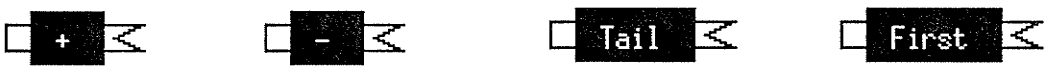
$+:<x,y>\rightarrow x+y$

$-:<x,y>\rightarrow x-y$

$\text{tl}:<x_1,\ldots,x_n>\rightarrow<x_2,\ldots,x_n>$

$1:<x_1,\ldots,x_n>\rightarrow x_1$

The same functions in Perseus take the following form:



Notice that the Perseus function object has an input to attach the data and an output connector for results. Unfortunately just like FP neither show the type of input expected by the function. Perseus does have this potential since it is graphical and such a feature could be added to it.

FP also contains a set of functional forms. A functional form accepts existing functions as parameters and combines them to build a new function. Below are a number of functionals which are a included as part of FP.

The composition functional, denoted by "o", takes two functions as parameters. It generates a new function that is equivalent to the application of the first parameter to the result of the second. It is defined as

$(f \circ g):x \equiv f:(g:x)$

An example of its use is obtaining the last item of a list by reversing it and removing the first element

e.g:

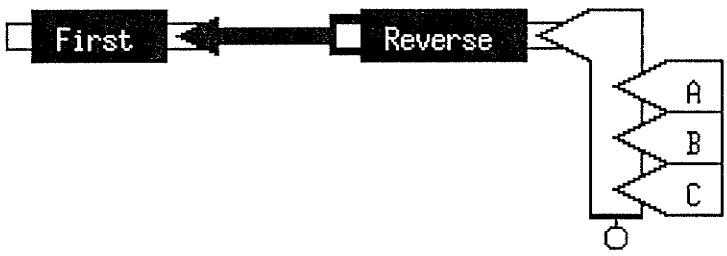
1 o reverse:$<A, B, C>$

1: reverse: $<A, B, C>$

1: $<C, B, A>$

C

The same functions in Perseus appear as:



Note how the Perseus composition line indicates that the output of "Reverse" is being sent to the input of "First".

The construction functional, denoted by "[ ]", takes $n$ functions as parameters and yields a function that applies each function to the same objects, and forms a sequence of the result. Its general form is :

$[f_1, f_2, \ldots, f_n]{:}x \equiv <f_1{:}x, \ldots, f_n{:}x>$

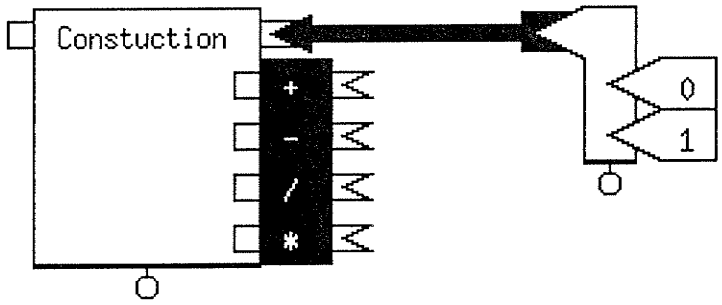
Examples using construction are:

$[+, -, /, *]{:}<0, 1>$

$<+{:}<0, 1>, -{:}<0, 1>, /{:}<0, 1>, *{:}<0, 1>>$

$<1, -1, 0, 0>$

The same program in Perseus appears as:




Notice, in the Perseus code, how the user can clearly see that the parameters are being filled by the appropriate functions.

The insert functional, denoted by "/", takes a function as a parameter and generates a new function that applies only to sequences. It applies the parameter functions to successive elements of the sequence. It is used to distribute functions taking two parameters over a sequence of $n$ elements. A small example of insert is using + to find the sum of a sequence of any length:

/+:<1, 2, 3>

+:<1, /+:<2, 3>>

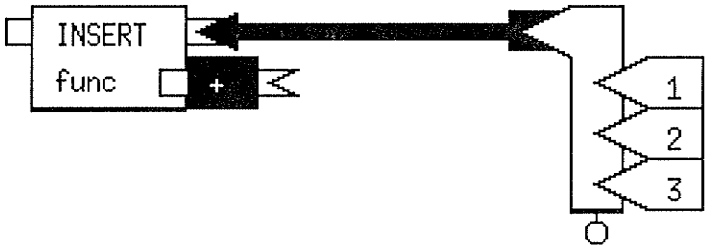
+:<1, +:<2, /+:<3>>>

+:<1, +:<2, 3>>

+:<1, 5>

6

The same program in Perseus is:




The constant functional takes an object as a parameter and produces a function which outputs that object. It is defined as:

$\overline{x} : y \equiv x$

For example:

$\overline{0} : y \equiv 0$

$\overline{T} : y \equiv T$

$\overline{1} : y \equiv 1$

The same objects in Perseus:


The apply-to-all functional, "$\alpha$", takes its parameter and generates a function, applicable only to sequences. It applies the function to each element of the argument

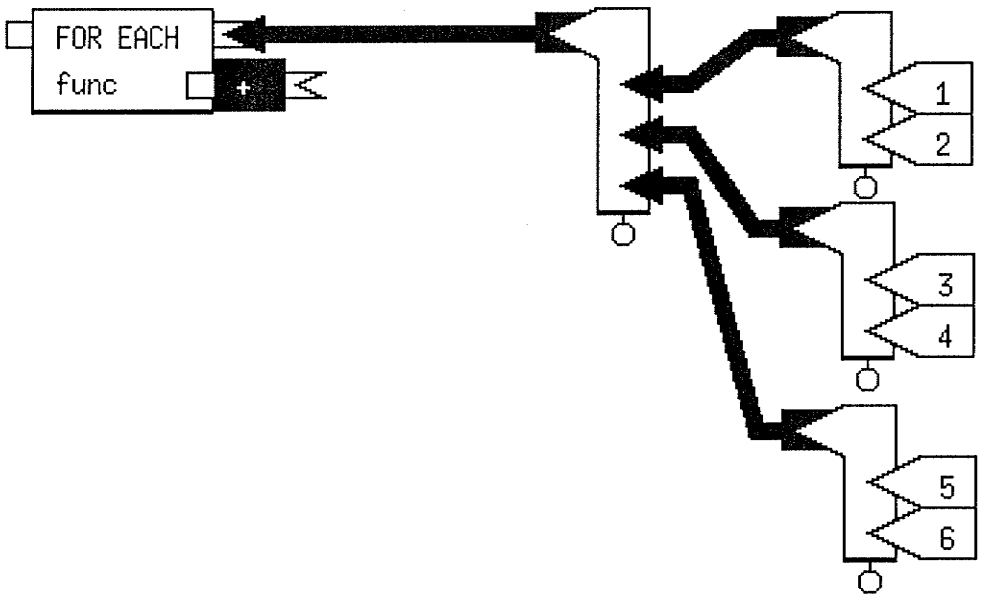
forming a sequence of the results.

e.g: $\alpha$+:<<1, 2>, <3, 4>, <5, 6>>

<+:<1, 2>,+:<3, 4>,+:<5, 6>>

<3, 7, 11>

The same program using *For Each* appears in Perseus as:

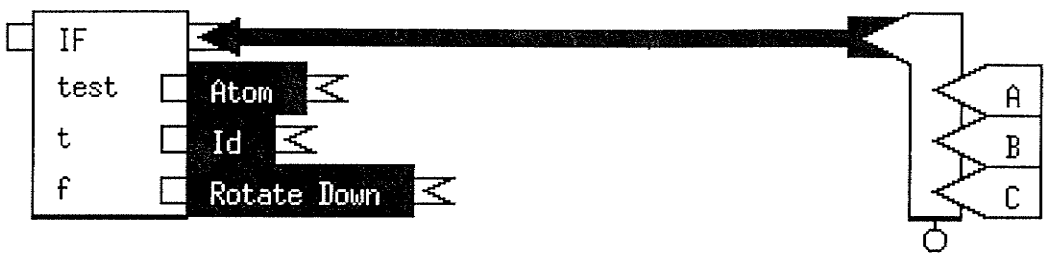



The condition functional is defined as $(p \rightarrow f;g)$. It will execute its second or third parameter depending on whether or not the result of the first function is the atom T. This is similar to the conditional found in imperative languages, but the parameters are all applied to the same input argument.

For example:

(atom $\rightarrow$ id; rotr):<A, B, C>

The same function in Perseus has the following appearance.




The function "rotate down" pushes the object on a sequence down one rung and moves the bottom object to the top. This is equivalent to a rotate right in textual FP.

The functional "while", defined as (while $p$ $f$):$x$, will execute its second parameter as long as the first parameter yields T. It applies the argument to the first parameter, then applies the second argument to the parameter. The result of the parameter is used as the new argument to the first, and so on.

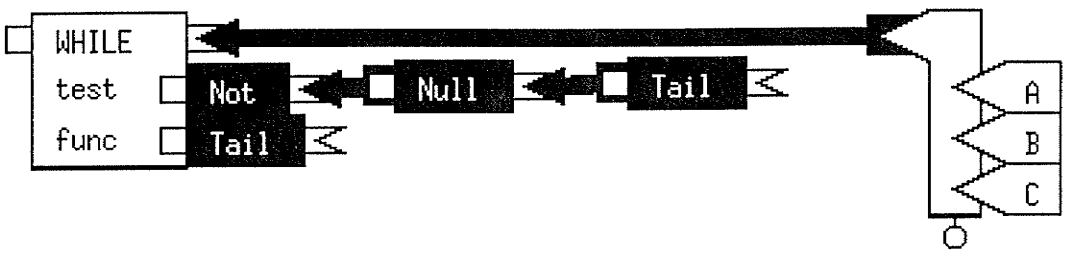
An example of using a while is to obtain a sequence containing the last object of a sequence by successively removing the objects in front of it:

(while (not ∘ null ∘ tl) tl):$<A, B, C>$

<C>

This function executes by testing the tail of the argument until the argument is null ($\phi$). It then returns the argument prior to the test, the argument is unaffected by the test parameters.

The same *While* functional in Perseus appears as:




Notice how easily distinguishable the parameters to the "While" functional, as compared to the textual code.

## 4.3 Comparison of Perseus vs. Textual FP

Now that the reader has seen both Perseus, a visual language based on FP, and the textual FP defined by Backus[2], The author can now compare the properties of Perseus and FP.

In the previous section of this chapter, the reader was shown the differences and similarities of FP constructs in FP and Perseus. A short comparison between the two was also made at that time. The author wanted to show that a user would be better served by the visual nature of Perseus.

One clear advantage of Perseus is that the user is not tied to the textual FP, and is able to organise the program on the screen to better show its structure and program flow. In textual systems, the user is sometimes limited to placing the FP program on one continuous unbroken line of text. This style can impede a programmer from understanding the operation of even the programmer's own code, after some time has passed. This is especially a concern when nested functionals are used. The multiple levels of parentheses can confuse a user, it may not be easy to discern which functions

are, or are not, a parameter of a functional.

Perseus does not enforce formatting of programs; icons are laid out and connected to one another as the user sees fit. The Perseus user can thus layout the icons and connect them in such a way as to enhance the readability of Perseus programs. This feature will help the user understand faster what the program is doing. This type of construction allows a user to break the Perseus program into logical "modules". The user can separate major portions of a program by long connectors, and can concentrate on each of these portions.

There is no confusion as to which functions are or are not the parameters of a functional; it is clearly visible. A user can see the nesting of the functionals and the parameters to each of them easily.

The Perseus system allows a user to click on any object in a program and obtain the results up to that point. Contrast this to most textual FP systems where a user must type in the portion of the program that he or she wants to execute.

It is obvious from the examples in the previous subsection that Perseus does take up more space than textual FP. Most of the screen space is taken up by connections between objects to prevent overlapping and obscuring icons. Also large icons and oddly shaped connector layouts tend to spread the programs out. Similar problems exist in other iconic languages such as Tinkertoy[12]. In Tinkertoy, icon layout is performed by a layout algorithm. The algorithm tends to "fan-out" the program and uses up much more space. Edel states that a Tinkertoy program is 20% larger than its equivalent Lisp program. Edel does not state precisely how this figure is achieved, but this author finds this figure quite low.

Perseus does not use auto-layout because a very complex algorithm is needed to

achieve even mediocre results. User placement often produces better results than a layout algorithm. Also auto-layout can be distracting as objects are moved, and lines are being added and removed. Auto-layout would be needed if iconic output were added to Perseus. In that case, the output icons would be composed of sequence and atom icons, and could be manipulated in exactly the same way. This is an advantage over the textual FP where output must be re-typed to be used as input to another FP program. In Perseus the user can also connect the output from one program to the input of the next program, thus creating a new program.

Perseus's menu-based and dialog-box-based user interface makes life easier for a user. A user can choose the function to add and to edit the value of an atom or constant through a dialog box. Many of the objects are created by choosing them from a menu. Dialog boxes and menus are common in today's GUI's and a user of Perseus would learn their use very quickly. Compared to a textual FP system which has special symbols to represent $\phi$, $\circ$, $\perp$, $\alpha$ and constants ($\overline{n}$). These symbols must be represented using the 128 characters in the ASCII standard. These selected characters are then not allowed to be used anywhere else. Perseus allows all available characters, except blanks, to be used in an atom or constant object.

One feature currently lacking in Perseus is the "Def" function or "definition". The Def function allows a user to create a program in FP and give it a name. This is similar to defining functions in imperative languages. Adding this name into a program would execute the actions defined by the definition, just like a function call in an imperative language. Given more time such a feature could be added to Perseus.

### 4.3.1 Evaluation of Visual Systems

In order to properly evaluate if Perseus is better than textual FP, an experiment would be required. Such an experiment could involve the recruiting of paid subjects and a performance measurement scheme. The volunteers should have limited exposure to computing, perhaps equivalent to a second-year university computer science standard. It is hoped that none of the volunteers would have exposure to the FP or Perseus language.

An experiment then could take one of two forms:

- There would be two groups. One group would spend all their time using textual FP, the other using Perseus. Relative performance would be measured.
- There would be two groups. One group would start using FP, the other using Perseus. At the mid-point of the experiment each group would switch to the other system. Relative performance differences would be measured.

Each volunteer would be given identical programming assignments to complete on the system on which they are currently assigned to work. During the completion of the work, the volunteer would be asked to time themselves for each part of the assignment. Also each would be asked to evaluate the performance of the system and whether a visual or textual system would have performed better. These reports along with each volunteer's comments could be combined to form a result in favour, or not in favour, of visual programming.

Such an experiment is beyond the scope of a masters thesis, and so it was not

attempted.

### 4.3.2 Size Comparison

To compare the relative differences in size of the textual FP and the graphical Perseus system. The author will reuse the examples presented in the previous section. The Perseus examples will be "formatted" to take up a minimal amount of space, yet keep the program readable to the user. The textual FP examples were also spaced for easy readability. The author believes that this is a fair comparison between the two systems.

Size differential will be measured by taking the total screen area used by the same program in Perseus and textual FP. For the textual FP programs, the author will use screen text which is approximately the same size as the text used in Perseus icons.

To measure the textual FP programs the author simply took the height of the text on the line, accounting for the space between lines, and calculated the total area. For the Perseus programs the author drew a boundary around the object leaving a slight gap between the objects and the boundary. The total area of the boundary was then calculated. A total of seven examples were used.

After completing the measurements the average Perseus program is approximately a factor of 6.4 larger than its textual FP equivalent, given the above stated conditions.

Some reasons for the size differential of Perseus over textual FP are that the function names tend to be longer than those in textual FP. Also the connectors add extra spacing around the objects. The functionals are much larger because of the extra space for each of the parameter connectors. It is the opinion of the author, however, that this size increase is justified for the increase in useability of Perseus

over textual FP.

### 4.3.3 Rapid Learning

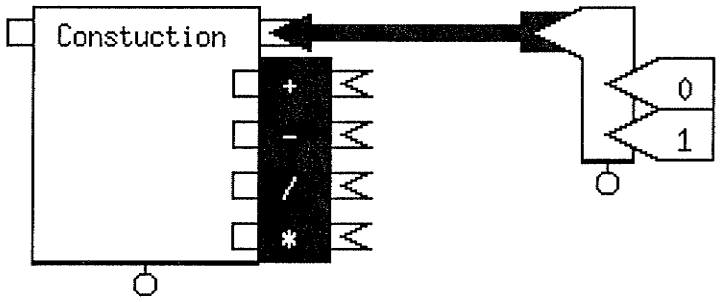
The Perseus system uses a GUI as its base environment. The GUI is the fastest developing environment for computers today. The goal of GUI's is to make computing easier for novice users. Such GUI's as the Macintosh, SUN's Solaris environment, and Microsoft Windows are oriented to provide powerful, yet easy to learn, software at the hands of users. Although it would take an experiment to show that the Perseus environment is easier to use, the author is confident—given the recent trend—that the Perseus GUI is better than the textual FP environments found today.

### 4.3.4 Visual Parallelism

Backus' original design of FP employed parallelism. This was his way of avoiding the Von Neumann bottleneck. Parrallelism was primarily achieved through the construction functional, which simultaneously executes all of its parameters. Unfortunately the textual syntax of construction does not give any visual clues to the user that these parameters are being executed in parallel. The user simply sees a number of functions encased between "[ ]" characters. Construction may also be nested so parallelism may be even harder to discern.

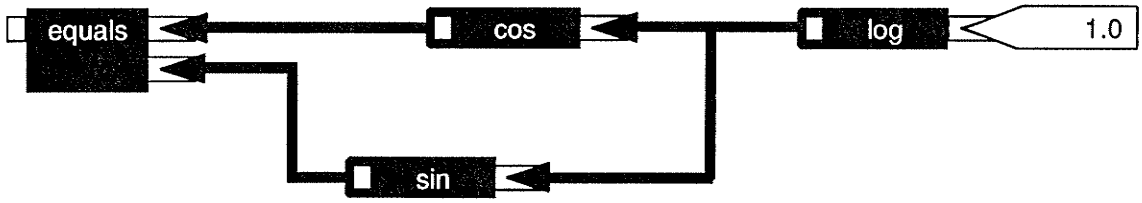
Since Perseus is a visual language, parallelism is easily seen in the configuration. The construction functional displays a number of parameter connectors, each of these will be connected to other functions. The vertical stacking of the connectors shows better to the user that these parameters can be executed in parallel. Also the vertical

orientation of the connector lines from the parameter connectors to the functions, is further emphasis that parallelism exists in the system. As an example look at the example construction from the previous section.




### 4.3.5 Shared Output

One advantage of visual languages is the possibility of taking one output from a function and distributing it to two or more inputs to other functions. This output splitting is not part of Backus' original FP [2]. A visual implementation could further enhance the parallelism in a environment, by visually separating the paths of data flowing through the system. An example of how sharing an output could appear in Perseus can be seen below. This action is similar to sharing variables among many functions, but without actual variables. A similar approach can be used in Lisp, but extra lambda expressions need to be used. The visualisation of this provides a much simpler interface for sharing.




Such a visual implementation can also enhance the debugging of the program by a user. Currently in Perseus, *Construction* comes close to shared outputs, but

all parameters are executed when construction is executed. In a split output, each "branch" or line of the execution path can be separately executed for a much finer level of debugging.

# Chapter 5

# Design of the Perseus System

## 5.1 Introduction

In this chapter the author will discuss the design and implementation of the Perseus system. Also a section will be used to discuss the merits of object-oriented programming as it relates to this project. The author will describe experiences during the implementation, and whether the object-oriented paradigm helped or hindered the progress of the implementation.

## 5.2 Organisation and Implementation of the Perseus System

The Perseus system is currently written in approximately nine thousand lines of C++ code. During coding the author concentrated on using object-oriented programming techniques wherever possible. This section will describe several of the important objects used in the Perseus system. Full details of Perseus's implementation must be left to a technical report, due to space limitations.

To ease the author's workload, a graphical user interface(GUI) toolkit called InterViews [26] was used. This toolkit is available free. InterViews is an object-oriented GUI toolkit also written in C++. It provides common GUI elements such as frames, buttons, dialog boxes, scroll bars to name a few. It also contains a graphics library which the author used to draw the graphics in Perseus. Several of the objects in Perseus were inherited from the InterViews toolkit objects. For example the dialog boxes were derived from the generic InterViews object *DialogBox*.

### 5.2.1 Objects and Object Hierarchy

There are several objects used in Perseus system. This subsection will cover the six main objects used.

The first object hierarchy used in Perseus is the *FPObject* hierarchy. The classes making up this hierarchy are diagramed in figure 5.1. Each of the leaf classes represent a programming-language element of Perseus—except for *function*, which is not a leaf class but is a programming-language element. The *FPObject* hierarchy was designed according to the interaction of the object with the user. For example the class *construction* and *sequence* have different purposes, but are elements which are stretchable, thus they are grouped under the *resizable* parent class.
What follows is a short description of what each class provides:

FPObject —The base class for the programming elements in Perseus. provides not only basic operations, but also provides storage of attributes, temporary results, and a bitmap graphic of itself.

Function —This class represents the *function* element which the user

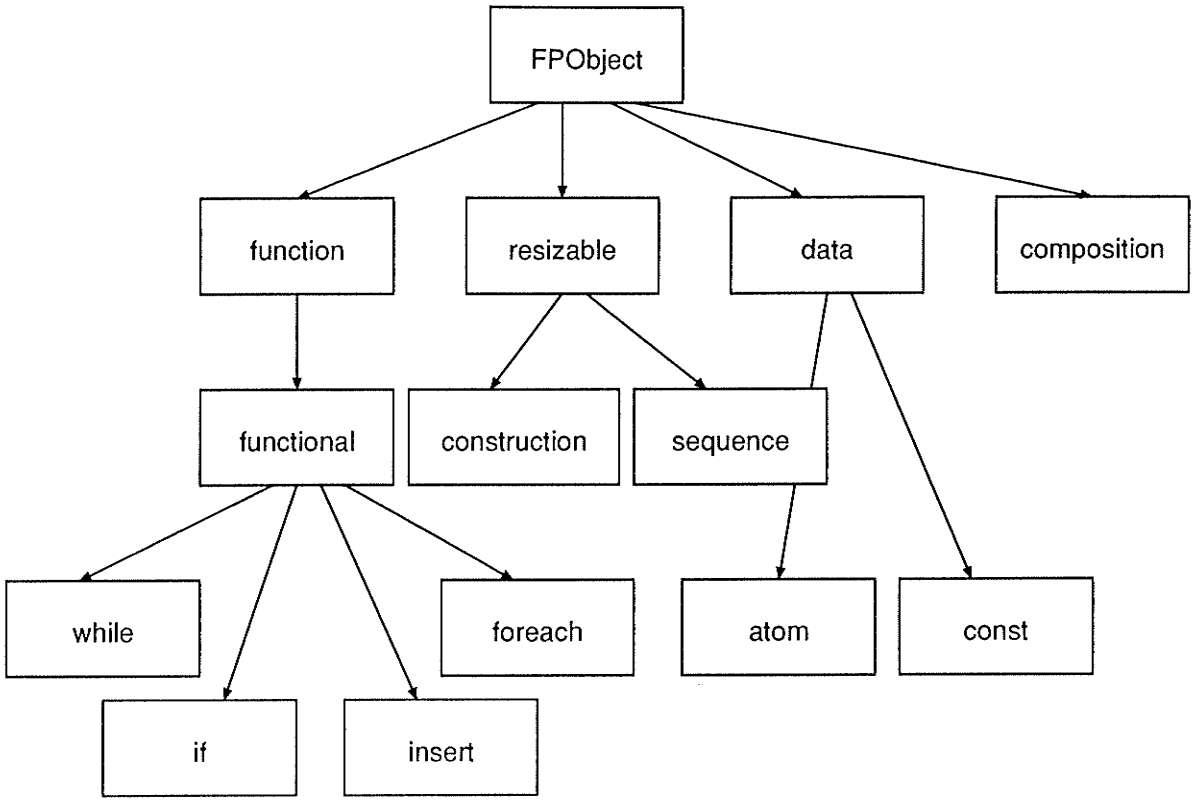



**Figure 5.1:** Diagram of FPObject hierarchy

uses to construct programs. It contains the appropriate function code to be performed during execution.

Functional —This class provides basic properties of each functional, except *Construction*. It contains extra information about parameters. It inherits the characteristics of *function* because the user interaction is similar to a function, except for the addition of parameters.

Below *functional* are the classes *While*, *If*, *Insert*, and *Apply*. Each contains extra information regarding the number of parameters it has. As well as the necessary code to perform the

named operation.

Resizable —This class represents the Perseus objects which are stretchable and have a pull-down loop at the bottom. It contains the appropriate methods to resize the graphic image of itself, and to make the necessary changes to the internal data structures.

*Resizable* has only two children, namely *Construction* and *Sequence*. These elements execute differently and thus their execute methods override the execute method of their parent.

Data —This class defines the properties of the two data objects in Perseus, *Atom* and *Const*. It provides methods to the alter the internal stored value of the object, and to redraw the object to represent the current internal value.

Composition —This class contains information about which objects it is connected to, and contains methods to draw the appropriate connector at each end and a line in between them. It also has methods to transport the data from one connector to the other.

Another object hierarchy used in Perseus is the connector hierarchy, named *FPconnec*. This hierarchy contains classes which describe input, output and composition connectors. Each object contains the code to draw itself and also methods to perform syntax checking. Each connector knows what types of connectors with which it can be correctly connected. It also contains information so that a clean connection can be made. This is the information that is used to "snap" the objects together. The

hierarchy is pictured in figure 5.2.

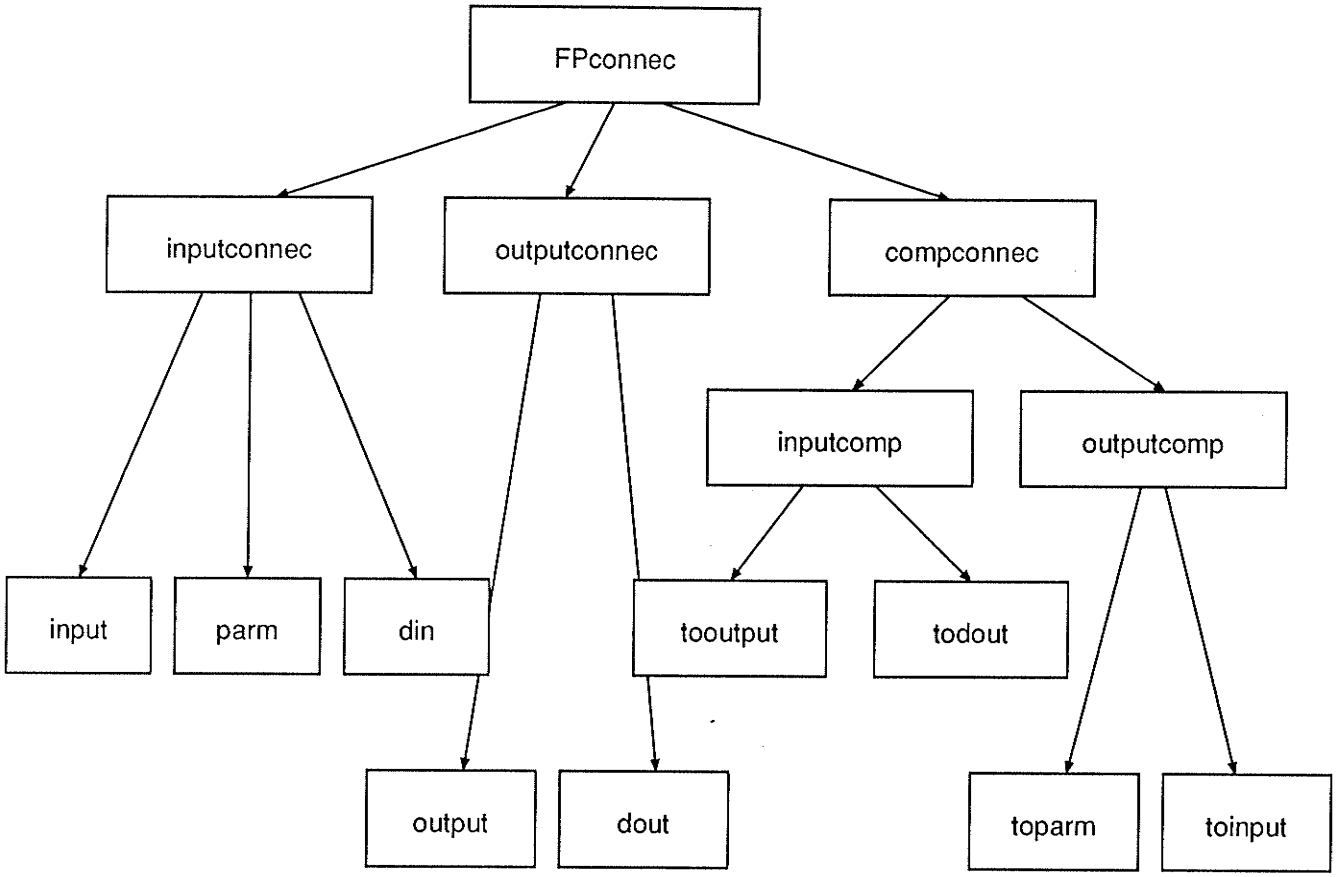



**Figure 5.2:** Diagram of FPconnec hierarchy

As the reader can see the hierarchy is broken into three distinct types—input, output and composition connectors. Within composition there are also input and output types. Each of the different type of connectors exists as a child of one of one of the subclasses.

FPconnec —This is the base class of the hierarchy, this contains information regarding whether this connector is connected to another connector, and which one. It also specifies which *FPObject* this connector is a part of.

Inputconnec —This class contains common methods for execution and proper

connection of these types of connectors, namely:

Input —is used on functions and functionals,

parm —is used as parameter connector on functionals and construction, and

din —is used on the input side of the sequence object.

Outputconnec —Is similar to inputconnec, except these are output connectors:

output —is used on functions and functionals,

dout —is used as the connector on sequence and atom.

Compconnec —Is the connector at the end of composition. It has a different way of snapping into other connectors, thus it has its own hierarchy. The input and output children of *compconnec* are quite similar to the *inputconnec* and *outputconnec* of regular connectors.

Another major class used in Perseus is a class called "Program". The *Program* class is an integral part of the Perseus system. As its name implies this class maintains the current status of all the programming elements currently in use by the Perseus system. When a user creates an object, and places it down on the canvas, *Program* performs actions to add the new object to the current program. This may involve possibly connecting the new object with another object, via the connectors. Also if an object is deleted by the user, *Program* may also need to delete other objects, such as *Composition*. It is the *Program* object which also performs the loop checking when a user connects two objects together using *Composition*.

The most important class hierarchy in Perseus is the "FPData" hierarchy. This hierarchy is the internal data structure used to store and manipulate results. The base class *FPData* provides no functionality, it merely unites the two subclasses so that an object of class *FPData* can be used to store either subclass. The *FPData* hierarchy contains two children classes:

FPDataAtom —Stores the value equivalent to an atom, that is a string of characters with no spaces. All data manipulation in Perseus is performed using strings.

FPDataList —This stores a list of *FPData* objects, which could be either *FPDataAtom* objects, or other *FPDataList* objects.

Each object in the *FPObjects* hierarchy contains a *FPData* member, except *Composition* which has no need of one. The *FPData* objects can also print their contents, which is how output is achieved.

Several other smaller, but no less important, classes are also used in Perseus. One class called "funcclass" stores the code used to execute a particular function. When the user selects a function to be created, the appropriate *funcclass* object containing the proper code would be used in creating the *function* object. Also the "FPeditor" class provides the object interaction facilities, such as creating, moving, deleting and resizing objects. *FPeditor* also pops up the appropriate dialog boxes when a user decides to create a *function* or alter the value of a *Data* object. Lastly the entire Perseus system is encapsulated within a "FPViewer" class which creates and forms the frame, menus, canvas and scroll buttons. The *FPViewer* object also creates *FPeditor*, *Program* and other involved objects, and starts system execution.

*FPViewer* is derived from an InterViews class.

The author has just skimmed over the basic objects comprising the Perseus system, more detail would perhaps have confused the reader. The next subsection covers the method by which Perseus programs are executed.

### 5.2.2 Method of Execution

To execute a program in Perseus, the user clicks on an executable object and the user's program is executed up to that point. This subsection will describe the process by which an object is executed, and how, in general, a program is run.

When the user selects an object to be executed, the system invokes a method in that object called "run". This starts the execution sequence in progress.

The first thing an object needs is to acquire some data, either from another function's results, or from a *Sequence* or *Atom* object. To an object this is the same thing since all data is stored in the "FPData" class. The connectors on an object provide the flow of data between objects, so an object invokes a method in its input connector to bring any data from the object to which it is connected. The input is connected to an output connector on another object. This output connector invokes the "run" method in the object of which it is a part, and returns the result. If this object is an *Atom*, or *Const*, then a copy of the internal data is passed as the result. If the object is a *Sequence*, it then constructs an internal sequence of all objects connected to it by invoking the "get_data" method on each of its connectors. Otherwise the above execution sequence is continued until either an *Atom*, *Sequence* or *Const* object is reached.

Once the data has been acquired then the object may execute. If the object is a

simple function, then the input is passed on to the function code stored within it in to be processed. The function will execute the internal code and return the results of its "run" method. The process for a functional is more complex since it has parameters which must use the input data for their own input. A functional must "pre-process" its parameters and inform them that their input is coming from within the functional. Then the functional can execute using the rules coded within it, and return a result.

When the "run" method is finished it returns a result, which either returns control to the "Program" object or continues execution by returning the result to the output connector which invoked that invocation. This result is then passed back to the input connector which invoked it, and is used as input to the function which invoked it, and so on until the last function is finished, concluding execution.

The result, which is an "FPData" object, can then be printed, using a method within the object to print itself out to standard output (the terminal that Perseus was started from).

Perseus deals with any errors by passing a NULL value, or zero pointer, back to the object which invoked it. Any following functions will not execute on an input which is NULL, they will pass back the NULL value. When control is returned to "Program" the text "Undef'd" is output to the terminal from which Perseus was started.

The execution method in Perseus is quite simple, as all objects can be treated generically. Each object contains a copy of the appropriate methods and it can override the generic method to perform a specific action.

## 5.3 Lessons From Object-Oriented Programming

In this section, the author will discuss the experiences gained from choosing and using an object-oriented style of design and implementation. This was the first time the author used object-oriented programming (OOP) techniques for a project.

### 5.3.1 Suitability of C++ For This Project

The author at an early stage decided that an object-oriented approach would be used to implement the Perseus system. With such a style each object would have two halves, one half is the visual image of the object. The other half is the data and code segment. This follows closely the theory put forth by Chang [8].

C++, being a language capable of object-oriented programming, was one language choice available to the author. With C++, the scheme of using objects with the two halves could be achieved much more easily than with a traditional language.

C++ also provided a speed advantage over other languages, such as Smalltalk, which would be a definite bonus. C++, being a descendant of C, is only slightly slower than C itself.

The Perseus system is an iconic language that uses graphics. A graphics library would thus be needed to perform the various drawing and screen management operations. A library would also be need to supply the menus, dialog boxes and other parts of the GUI. These libraries would need to link easily with the language that was chosen for implementation. Both Smalltalk and Eiffel have graphics libraries, but a separate GUI toolkit would be required. The author found a C++ toolkit called InterViews [26], which provides a graphics library, as well as GUI toolkit.

The author also considered the availability of languages in the university. Neither Smalltalk, or Eiffel were available to the author. A fee would have had to be paid to have either language installed at this site. C++ however was readily available and was in use throughout campus. Further support of C++ was not in doubt.

In addition, it was felt that experience with C++ would be of value in the author's job hunt following graduation.

### 5.3.2 Problems Using C++

Through the development process problems were encountered. Some of these problems were related to the C++ language. In this subsection, a few of the major problems with C++ as it stands now will be discussed.

The first and most important problem was with using multiple inheritance and virtual base classes. Multiple inheritance allows a class to inherit attributes from two other classes. Virtual bases classes prevent more than one copy of that class from being copied into a class. Its major use is in multiple hierarchy situations where the two parent classes are both subclasses of the same parent class. The diagrams in Figure 5.3 shows a simple example of multiple inheritance and virtual base classes. Notice the difference in the data structures when virtual base classes are, and aren't used. Using virtual base classes, the structure more accurately represents the class structure the programmer expects.

In the Perseus implementation, multiple inheritance and virtual bases classes, could have been used to make *Construction* a child of both *Functional* and *Resizable*. Unfortunately the Perseus implementation relied on deferring the definition of some methods until lower in the hierarchy—these are called "pure virtual functions"

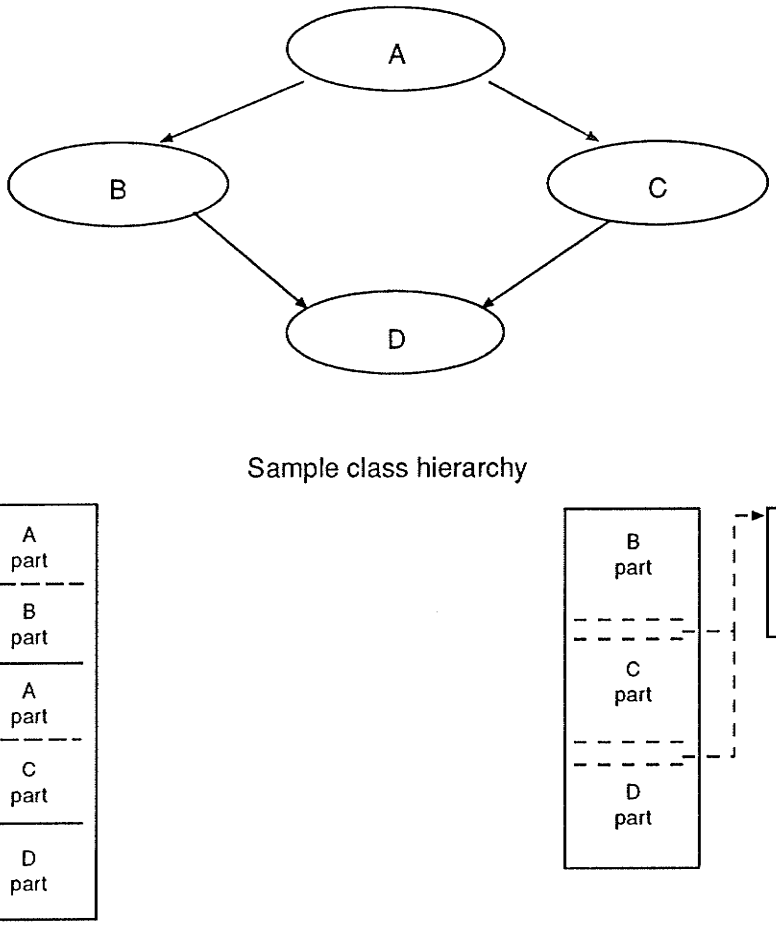



**Figure 5.3:** Example of effect of virtual base classes

because they define only a name and contain no code. This is what caused the problems. Since each of the base classes were incomplete, the compiler could not resolve the missing functions.

Another problem that the author had with virtual base classes, was with using them in an array of class objects. All the children of the base class could be stored in an array by just storing them using their base class definition. When it came time to extract a object from the array, the author would need to perform type-coercion

to use the full features of the object. This action is disallowed by the definition of C++.

With these two features of C++ unusable for Perseus, the class definitions in Perseus have some duplication in them. The classes of *Construction*, and *Sequence*, which would have used multiple inheritance, required the same methods as in *Functional* and *Data*—respectfully—duplicated within themselves.

One feature which is missing from C++ but which the author would have benefited from, is the ability to create a named object dynamically. That is a variable would contain a valid class name. This feature then could be used in a function to create an object of that class at run time. This feature exists in Smalltalk. In Perseus the user is given a dialog box to choose which of the available functions to create. With this feature the name of the function retrieved from the dialog box would be used to create a object of that name. Without such a feature, the Perseus implementation needed to use a large switch statment that could create an object of each possible kind. This negates some of the advantages of an object-oriented programming language, since they are supposed to remove that sort of inflexible coding.

# Chapter 6

# Summary

In this thesis the author has presented the use and design of a visual programming language called Perseus. The language is based on FP which was developed by John Backus [2]. The Perseus system is a member of the *iconic visual programming languages* category, one of the categories in the *visual programming framework* developed by Shu [42]. This framework is summarised in chapter 2 of this thesis.

The author also presented a small survey of several systems contained under the various categories of the framework. These systems cover a large area of programming, and each contains aspects of other categories in the visual programming framework. The basis for qualifying a system for the framework is that it meet the definition. That is, *visual programming is the meaningful manipulation of visual objects in the programming process*. Each of the surveyed systems follows this definition. The systems can then be placed into the categories based on their major properties and goals.

The Perseus system, like many of the current and past visual programming sys-

tems, was developed on a ad-hoc basis—due to the fact that currently in visual programming, there is very little theory to direct the developer of visual programming systems. But new theory, on the proper definition of icons, is being developed.

In chapter 3 the author introduced the Perseus system. Each of the icons currently implemented were described. The icons displayed a size, structure and consistency which the author hopes makes programming in Perseus better than programming in textual FP.

Some advantages to the objects in Perseus are that nesting of objects is much easier to see in Perseus. Also the programmer can see readily the number of parameters that a functional has. The user can also clearly see which functions are currently parameters to the functional. This is more advantageous for deep nesting as textual FP can become quite confusing. The syntax checking in Perseus is performed through the connectors and visually show to the user which objects can be connected to one another, by showing the type of object the connector expects. The composition object can also perform syntax checking and can prevent the formation of loops, which can only occur in such visual representations.

These visual properties are a distinct advantage of Perseus over textual FP. Chapter 4 is a comparison of textual FP versus Perseus. This made clear the readability advantage of Perseus by presenting several sample programs in Perseus and textual FP. The visual nature of Perseus can also make parallelism much clearer as the user which functions are to occur simultaneously. As well the adding of a shared outputs facility to allow sharing of outputs without variables, is a possibility.

The visual advantages of Perseus do come at a penalty. A Perseus program will, on average, be a factor of 6.4 larger than the identical textual FP program. These

observations were made using the measurement guidelines presented in chapter 4.

The advantages of Perseus outweigh the disadvantages, in the author's opinion.

The Perseus system is written in an object-oriented style. The object-oriented paradigm was a good choice for implementing Perseus as each icon could have its visual appearance and its data/code side. This is close to the theory of pure icons by Chang[8]. This approach allowed a much more generic means of performing operations. Each object would override the defaults if necessary and manage its own internal details. This would reduce the code duplication that may have occurred had the author used a traditional language. In any case the message selection details were left to the language, allowing more time to concentrate on the programming.

There is currently a great deal of research on visual programming, and a new journal in the area. This thesis has attempted to report on and advance this research.

# Bibliography


[1] Miren Begona Albizuri-Romero. GRASE - a graphical syntax directed editor for structured programming. *ACM SIGPLAN Notices*, 19(2):28–37, 1984.

[2] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[3] J. G. Bonar and B. W. Liffick. A visual programming language for novices. In Shi-Kuo Chang, editor, *Principles of Visual Programming Systems*. Prentice-Hall, 1990.

[4] Kjell Borg. Visual programming and unix. In *1989 IEEE Society Workshop on Visual Languages*, pages 74–79. IEEE, 1989.

[5] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM transactions on Programming Language and Systems*, pages 353–387, 1981.

[6] Gretchen P. Brown, Christopher F. Herot, and Paul Souza. Program visualization: Graphical support for software development. *IEEE Computer*, 18(8):27–35, 1985.

[7] Shi-Kuo Chang. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39, 1987.

[8] Shi-Kuo Chang. Principles of visual languages. In Shi-Kuo Chang, editor, *Principles of Visual Programming Systems.* Prentice-Hall, 1990.

[9] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from texual conditioning. In *1989 IEEE Society Workshop on Visual Languages*, pages 150–156. IEEE, 1989.

[10] C. Crimi, A. Guercio, G. Tortora, and M. Tucci. An intelligent iconic system to generate and to interpret visual languages. In *1989 IEEE Society Workshop on Visual Languages*, pages 144–149. IEEE, 1989.

[11] Serfim Dahl and Kjell lindqvist. Visual programming as an interface between program and user? In *1989 IEEE Society Workshop on Visual Languages*, pages 18–23. IEEE, 1989.

[12] Mark Edel. The Tinkertoy graphical programming environment. *IEEE Transactions on Software Engineering*, 14(8):1110–1115, 1988.

[13] Alistar D. N. Edwards. Visual programming languages: The next generation? *ACM SIGPLAN Notices*, 23(4):43–50, 1988.

[14] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[15] G.Cattaneo, A. Guercio, S. Levialdi, and G. Tortora. ICONLISP: An example of a visual programming language. In *1986 IEEE Computer Society Workshop on Visual Languages*, pages 22–25, 1986.

[16] Alessandro Giocalone, Martin C. Rinard, and Thomas W. Jr. Doeppner. IDEOSY: An ideographic and interactive program description system. *ACM SIGPLAN Notices*, 19(5):15–20, 1984.

[17] Ephraim P. Glinert. Towards "second generation" interactive graphical programming environments. In *1986 IEEE Computer Society Workshop on Visual Languages*, pages 61–70. IEEE, 1986.

[18] Ephraim P. Glinert. Nontextual programming environments. In Shi-Kuo Chang, editor, *Principles of Visual Programming Systems.* Prentice-Hall, 1990.

[19] Ephraim P. Glinert and Steven L. Tanimoto. Pict: An interactive, graphical programming environment. *IEEE Computer*, 17(11):7–25, 1984.

[20] Eric J. Golin and Steven P. Reiss. The specification of visual langugae syntax. In *1989 IEEE Society Workshop on Visual Languages*, pages 105–110, 1989.

[21] Kuan-Tsae Huang. Visual interface design systems. In Shi-Kuo Chang, editor, *Principles of Visual Programming Systems.* Prentice-Hall, 1990.

[22] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph and Ken Doyle. Fabrik: A visual programming environment. In *OOPSLA '88 Conference Proceedings*, pages 176–190. ACM, 1988.

[23] R. J. K. Jacob. Using formal specifications in the design of a human-computer interface. *Communications of the ACM*, 26(4):259–264, 1983.

[24] R. J. K. Jacob. A state transition diagram language for visual programming. *IEEE Computer*, 18(8):51–59, 1985.

[25] D. Ladret and M. Rucher. VLP: A visual logic programming language. *Journal of Visual Languages and Computing*, 2(2):163–188, 1991.

[26] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer*, 8(2):8–22, 89.

[27] Stanley B. Lippman. *C++ Primer, 2nd Edition.* Addison-Wesley, 1991.

[28] Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace and Ken Doyle. The fabrik programming environment. In *1988 IEEE Computer Society Workshop on Visual Languages*, pages 222–230. IEEE, 1988.

[29] Jun'ichi Miyao, Shin'ichi Wakabayashi, Noriyoshi Yoshida, and Yasushi Ohtahara. Visualized and modeless programming environment for form manipulation language. In *1989 IEEE Society Workshop on Visual Languages*, pages 99–104. IEEE, 1989.

[30] N. Mondon, Y. Yoshino, M. Hirakawa, M. Tanaka, and T. Ichikawa. HI-VISUAL: A language supporting visual interaction in programming. In *1984 IEEE Computer Society Workshop on Visual Languages*, pages 199–205, 1984.

[31] Mark Moriconi and Dwight F. Hare. Visualizing program designs through pegasys. *IEEE Computer*, 18(8):72–85, 1985.

[32] Michael L. Powell and Mark A. Linton. Visual abstraction in an interactive programming environment. *ACM SIGPLAN Notices*, 18(6):14–21, 1983.

[33] Georg Raeder. A survey of current graphical programming techniques. *IEEE Computer*, 18(8):11–25, 1985.

[34] J. R. Rasure and C. S. Williams. An integrated data flow visual language and software development environment. *Journal of Visual Languages and Computing*, 2(3):217–246, 1991.

[35] Steven P. Reiss. Graphical program development with PECAN program development systems. *ACM SIGPLAN Notices*, 19(5):30–41, 1984.

[36] Steven P. Reiss. Working in the Garden environment for conceptual programming. *IEEE Software*, 6(6):16–27, 1987.

[37] Steven P. Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, 11(3):276–285, 85.

[38] Yukari Shirota and Tosiyasu L. Kuni. Specification and automatic generation of intelligent graphical interfaces. In *1989 IEEE Society Workshop on Visual Languages*, pages 7–12. IEEE, 1989.

[39] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 19(8):57–69, 1983.

[40] Nan C. Shu. A forms-oriented and visual-directed application development system for non-programmers. In *1984 IEEE Computer Society Workshop on Visual Languages*, pages 162–170. IEEE, 1984.

[41] Nan C. Shu. FORMAL: A forms-oriented, visual-directed application development system. *IEEE Computer*, 18(8):38–49, 1985.

[42] Nan C. Shu. *Visual Programming.* Van Nostrand Reinhold, 1988.

[43] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[44] Kazou Sugihara, Tohru Kikuno, Noriyoshi Yoshida, and Masayuki Takayama. An approach to the design of a form language. In *1984 IEEE Computer Society Workshop on Visual Languages*, pages 171–176. IEEE, 1984.

[45] Steven L. Tanimoto and Ephraim P. Glinert. Designing iconic programming systems: Representation and learnability. In *1986 IEEE Computer Society Workshop on Visual Languages*, pages 54–60, 1986.

[46] Warren Teitelman. A tour through Cedar. *IEEE Transactions on Software Engineering*, SE-11(3):285–302, 1985.

[47] Warren Teitelman and Larry Masinter. The Interlisp programming environment. *IEEE Computer*, 14(4):25–34, 1981.

[48] Kazuyuki Tsuda, Masahito Hirakawa, Minoru Tanaka, and Tadao Ichikawa. Iconicbrowser: An iconic retrieval system of object-oriented databases. In *1989 IEEE Society Workshop on Visual Languages*, pages 130–137. IEEE, 1989.

[49] Steven P. Wartik and Maria H. Penedo. FILLIN: A reusable tool for form-oriented software. *IEEE Software*, 6(3):61–69, 1986.

[50] I. Yoshimoto, N. Monden, M. Hirakawa, M. Tanaka, and T. Ichikawa. Interactive iconic programming facility in HI-VISUAL. In *1986 IEEE Computer Society Workshop on Visual Languages*, pages 34–41. IEEE, 1986.