1989

A Program Visualization System That Supports the Program Understanding Process.

Brady R. Rimes
Louisiana State University and Agricultural & Mechanical College

Order Number 9002167

A program visualization system that supports the program understanding process

Rimes, Brady R., Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A Program Visualization System That Supports the Program Understanding Process

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Brady R. Rimes

B.S., University of Southern Mississippi, 1974
M.S., University of Southern Mississippi, 1978
May 1989

ACKNOWLEDGMENTS

I am very appreciative to Dr. Doris L. Carver for serving as my major professor, guiding me through my research efforts, and investing many hours towards the final results. I would also like to thank the members of my committee Dr. Donald Kraft, Dr. S. Sitharama Iyengar, Dr. Leslie Jones, and Dr. Warren Waggenspack for serving as members and for the contributions they made to my research.

I am grateful to my parents for their encouragement throughout my life. I would like to thank Mr. Danny Carter for giving me the opportunity to pursue graduate studies and the support he gave during my efforts. I appreciate the time Mr. Carter spent with me discussing ideas and problems during this period of time. Finally, I would like to acknowledge my wife, Pam, and my sons, Toby and Tyson, for their love and for their support through the trials and tribulation we encountered. Their sacrifice was far more than I could have asked of them. I will be forever grateful to my wife for her ongoing encouragement and her desire for my success which inspired me to continue through the final stage. Thank you all for everything that has helped me in reaching my goal.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The goal of this research is to provide a graphical system that supports the program understanding process by representing the program's control flow, the code and the identifiers local to a specific point within the program. By having more information local to the point of interest, the programmer can maintain continuity in developing program understanding. The programmer can see loops, procedure calls, and other structures with respect to their execution order and can view them in the environment or the context in which they will execute. The Peec system supplies a graphical representation of the program's control flow in which the control structures are represented as tiers. The tiers are arranged in a three-dimensional space representing the program's operational flow. The body of the procedure or function is nested within the reference tier so that the programmer views the routine local to its reference point. Also, a list of live identifiers is displayable for the current tier element. The advantage is that the routine's text and the identifier list are local to the area of study and the programmer does not have to look elsewhere for the program text and the identifier definition. The programmer can maintain a continuity in developing program understanding using information local to the point of interest.

The Peec system consists of the Peec compiler which transforms a Pascal program into tier and identifier information, and the Peec environment for modeling the program's operational flow image. The Peec environment provides the programmer many interactive capabilities. These capabilities consist of browsing the flow model, displaying text, displaying identifiers and transforming the three-dimensional flow model into appropriate views. These features are aimed at assisting the programmer in the process of developing program understanding.

CHAPTER I

INTRODUCTION

Graphics in Problem Solving

Humans, by nature, are symbol oriented. Individuals acquire and transfer information through the use of vision by either viewing a physical object or a representation of that object. In general, humans are better able to understand a concept if a graphical medium is used in the communication process. Today, graphical representations are used in many domains to communicate ideas in a clear and concise way. The use of graphics is found in the industrial as well as the educational environments. In education, graphics is used to explain existing ideas and concepts whereas industry may use graphical representations to express new ideas and to develop new products.

Many of the concepts that people deal with today are supported by a mental model that the individual has developed. A mental model is an image, within the individual's mind, that he uses to support his understanding process. The human's mental model is used to support both physical and abstract concepts found in his environment. A graphical representation can be used to either enhance an existing mental model or it can be used to develop new models (Weber & Kosslyn, 1986). In either case, the model assists the individual in understanding the idea or concept. A misunderstood idea is often cleared up with a simple, unambiguous drawing. The graphical images are valuable to humans in communicating and in comprehending the meaning of many concepts.

One area in which graphical enhancements are increasingly being used is in the programming environment. A programmed algorithm represented in its normal textual

format may require a significant amount of time for the user to develop an understanding of the algorithm's implementation. The difficulty of program analysis is dependent on the complexity of the code and the experience of the programmer. Incorporating graphics in the programming environment can assist the programmer in the program comprehension process.

There are numerous ways that graphics can be incorporated into the programming environment. Some approaches use graphics by enhancing existing languages with graphically represented data structures and control structures while other approaches are based on developing new languages which use graphical images as their medium for programming. With improved hardware facilities and heightened program complexities, the use of graphics in the programming environment has become an active area of research.

Graphics has been incorporated in many areas of the programming environment. In order to establish a baseline for describing the combination of graphics and programming, we define some concepts found in the programming environment. These concepts are the basis for the graphical interface found in many of the programming languages and software systems discussed in Chapter II. The following sections define concepts and properties of programming and the use of images to visually support the programming environment.

Graphics and Software

Computing mediums found in many research environments today include high-performance, graphics-based, personal workstations. Often, these modern computing environments are only used to support the traditional modes of programming such as

designing, coding, debugging, and maintaining software. The benefits of visualization in the programming environment are still being explored.

Researchers are studying methods of using graphics to enhance the user's ability to interface with all types of software systems. In particular, graphics are used in the design and development of software systems in numerous areas including debugging, performance monitoring, and non-textual program displaying. Other uses of graphics are found in data base and information systems where graphics are used as a medium for displaying information. These visual capabilities facilitate the user's understanding of software systems, programs, and data characteristics. Software systems with visualization can be used effectively to create new software systems and to enhance or alter existing systems. The incorporation of graphics into the software development process continues to mature as the complexity of software continues to rise and as graphical hardware continues to improve.

Concepts of Programming

There are many reference points from which programmers view software systems. The most primitive view is that of a black box with input and output processes. A second view is as a set of data structures, manipulated by a group of control statements. By applying the control statements to the data structures in some orderly fashion, the desired results are produced. Many programmers design and develop programs by defining the data structures first. The programmer starts by defining a set of data structures and then designing the operational flow using the language's control structures to manipulate the data, thus creating a program.

A third view is from the perspective of the programming language. The systems programmer views software as bits, bytes, registers and addresses at the machine level,

while a high-level language programmer envisions software at the algorithmic level. Software systems can also be classified by the complexities and facilities within the system. These features include the ease of developing software systems, the convenience of maintaining them, and the ease of understanding the software after it is developed. Regardless of the approach taken in studying software systems, graphics can be used to enhance the programmer's perception of these systems. With such enhancements, the programmer increases his understanding and willingness to use, develop, or change such software systems.

Properties of Programming

Software can be analyzed by studying the tangible and the intangible properties within programs. Examples of tangible properties are the language's data types and control structures. Although these properties may not appear to be tangible, they can be associated with a physical representation such as lexemes, syntaxes, machine addresses, and memory. The programmer can associate a mental image with these tangible entities. For example, a programmer can envision registers where calculations are performed or memory addresses as boxes within blocks of memory where data values or program code resides.

Intangible properties are features that cannot be represented explicitly by some physical object. A FOR loop is a tangible object, but its meaning is intangible. For example, a FOR loop is considered a tangible property since the loop has an initial value, ending value, increment value, and a body. All of these are entities used to define and describe the loop construct. The programmer can see a loop with its attributes and visualize an image of the loop with its increments and termination constraints, arranged at appropriate locations of the loop. However, the intangible property of the loop is its meaning

within the context of the program. Does the loop sum a series of numbers or output a string of characters, one character at a time? To understand the loop's objective, the statements in the body of the loop must be examined. By decomposing the loop into meaningful objects, such as the individual statements in the body, the programmer develops and associates meanings with each statement. The programmer then reconstructs the loop, using the statements and associated meanings, and forms a clearer understanding of the loop as a whole.

The concept of the FOR loop is well-defined in programming languages. However, the intangible or abstract meaning of the loop requires the programmer to develop a relationship between the program constructs and the data it is manipulating within the body of the loop. The loop examination example also demonstrates the abstraction process a programmer performs when developing the meaning of code. The abstraction process is applied to blocks of code, procedures, packages, entire programs, or groups of programs. The programmer develops understanding by abstracting from different parts of the software and from different levels of abstraction in order to understand the program or program segments (Basili & Mills, 1982; Soloway et al.,1983). The levels of abstraction may be at the statement level, control structure and procedure level. Once the programmer understands a block of code, he then views the block and its meaning as a single entity of the program. This meaning is used to support higher levels of understanding. The ability to abstract meaning at different levels is an important tool used by the programmer in all phases of the programming environment.

Essence of Programming

Brooks(1987) refers to an intangible entity as the essence of software. The essence is the natural inherent difficulty found in software. For example, the interlocking concepts

between data sets, relationships among data items, algorithms, and initiations or referencing of functions are the intangible entities that give meaning to software. These are conceptual constructs the programmer must deal with in designing programs or in understanding existing programs. The properties associated with the essence of software are complexity, conformity, changeability, and invisibility.

The complexity of software is more than just the repetition of basic elements of smaller systems. By linearly increasing the number of elements within a program, the interaction between these elements increases nonlinearly, thus the program's complexity is magnified as the size of the program increases.

The conformity property relates to matching the software to the user's needs. Software conformity is the fitting of software interfaces to the needs and capabilities of the user. Unlike physics, which is a study of well-defined relationships based on the laws of nature, software systems are not so well defined. Software is developed by humans to be used by humans and therefore must conform to the human user's needs. The results of conforming a system to the user contributes to much of the complexity found in software systems.

Programs are used to support many facets of operations within the environment we live. Software is used in business, industrial, research, and educational areas. As the operations with these areas evolve, so must the software that supports them. Changeability is an essence property where software must be altered as operational procedures are updated. Since humans are involved with developing and using software and with the continued improvements to operational procedures, the changeability of software systems is inevitable.

The last property found in the essence of programming is the unvisualizable features rather than the invisible features. For example, land can be represented by maps, silicon

chips by diagrams, and computers by connectivity schematics; but software cannot be represented by such physical features. We can represent the structures of a program using various types of diagrams, but we cannot inherently embed software into a space and give a physical representation of the meaning for which the software reflects.

The essence of programming is much more than syntax of the language or arrangement of control structures within a program. It requires conceptual constructs which are not defined by any laws and do not occupy any physical space. The programmer must abstract these concepts from the program's text. There are studies ongoing in which graphics is being used to represent and express some of these features for the purpose of supporting the programmer in the understanding process (Shu, 1986). Researchers are attempting to define and design systems which convey to the programmer more meaning about the program in an easy and concise way.

There are applications in the programming environment that requires an understanding of the textual form of the algorithm. If a programmer needs to modify a program, he must first understand its implementation or understand the textual structures before performing any alterations. The same inference can be used if the programmer needs to create a new program by extracting sections of code from other programs. To help the programmer in the understanding process, he uses some type of model which assists him in managing the organization of the program's execution order as defined by the coded algorithm. This model does not necessarily reflect the algorithm as defined by the program code, but as an abstraction of the order the program tasks execute. With a representation of the program's operational flow and the data it affects, the programmer develops an understanding of the algorithm's implementation which allows him to alter or use the code more effectively.

The goal of this research is to provide the programmer an environment to assist him in understanding a coded program. The environment supports a model of an abstract concept, the program's execution flow. The execution flow model represents the order the control structures are executed. The flow model itself is abstracted from the program's code which allows the user to see the implementation of the algorithm based on how the code is written. It shows the relationship the control structures have with one another such as nested, sequential, or optionally executed statements. The environment also supplies information about the variables being affected by these control structures. The model and the environment allow the programmer to develop program understanding in much the same manner he currently does, but with more detail and more information available.

Summary

Software is more than well-defined data structures and control structures or the other tangible entities of programming. Many of the complexities of software are invisible and must be abstracted from the text before an understanding of the software is obtained. The programmer must be aware of the data entities, control structures, the operational flow of these structures, and the interactions between the control structures and data entities. The use of graphics can assist the programmer in both developing and maintaining software systems by improving his ability to understand the software.

The research presented in this paper focuses on program visualization using graphical representations. A visualization of the program's execution environment is used in the programming environment to assist the programmer in understanding programs. As part of the comprehension phase, the programmer abstracts from the program's textual structure the program's execution environment or the program's control flow. The program environment supports the comprehension problem for delocalized code

(Letovsky & Soloway, 1986). The graphical representation can assist the programmer in developing an understanding of the program's operations and the order these operations are performed based on the control structures. Program understanding is essential in the areas of program development, debugging, maintenance, and code migration. A visual image of the program's control flow assists the programmer in these areas.

In Chapter II, we investigate the motivation for using graphics in software design, development and maintenance. We review the manner in which graphics is currently being merged with the programming environment. Included in the review are systems that cannot be classified as programming languages, but have a graphical perspective to them. These systems make use of graphical interfaces to assist the user in both using and interpreting the results within the software system. They are cited because they contribute to the understanding process through the use of visual representation.

Chapter III presents the specific objectives of the research. Chapter IV discusses the process a programmer uses in developing program understanding. From this procedure we are able to define the elements used in modeling a program's execution environment and the format or the arrangement of these elements. The model's elements are abstracted from the program's source statements. Chapter V discusses the abstraction process and their supporting algorithms.

Chapter VI discuss the generation of the program's execution flow image and the environment for studying the program. Chapter VII discusses the interactive features and functionally of the system. Chapter VIII discusses the empirical evaluation conducted with the Peec system. Finally Chapter IX gives a summary of the research as well as suggestions for extending this work.

CHAPTER II

VISUAL PROGRAMMING ENVIRONMENTS

Human and Computer Graphics Imagery

The human mind has a strong visual orientation. It can acquire information at a significantly higher rate by using graphical representations than it can by using textual representations. A text representation can be considered as a one-dimensional stream of words used to point to things in the real world. Pictures, on the other hand, better reflect the real world by borrowing from it such properties as shape, size, color, texture, direction and distance (Reader, 1985). When the human mind is processing information in textual form, the mind tries to develop a mental image of the idea or concept. By associating an image with a concept, the mind can develop a stronger understanding of the concept and retain related knowledge for longer periods of time. A study by Bugelski (Nicholas, 1977) showed that a person could immediately recall 30 to 40 words if he integrated them into a picture or an image. Without the support of an image, an individual could, at best, recall 10 words immediately.

The mind also uses its imagery system to develop and design new ideas or concepts. An individual who is designing a program will first develop a high-level, logical order of the program's flow as a mental model. The designer then takes the mental image and "externalizes" it into some other form (Weber & Kosslyn, 1986). External mediums include flowcharts, textual programs or drawings of data structures. The designer must develop a mental model of the program's operational flow before he proceeds with the other program design phases.

With the advent of computer graphics, the human imagery system has been enhanced by capitalizing on features of computer generated images that are lacking in the human system. Computer graphics can support the individual's ability to learn, understand and express his ideas in a clear and concise way. Weber and Kosslyn (1986) made a comparison of human imagery and computer graphics in which they explored how a graphical system could be used in externalizing the human's mental image. They also investigated how to enhance the human imagery system using ideas from computer graphics. Their research demonstrated that there exists features in the mental imagery system that are not found in computer graphics and features in computer graphics that are not found in the mental system. The human imagery system seems to be object-oriented for both short and long term memory. It has the capability of zooming from a large overview of a concept into the smallest details defining the concept. The most obvious advantages the human imagery system has is the ability to learn, program, repair, and integrate imagery information with semantic information. A computer graphics system, on the other hand, has a larger capacity (i.e., number of detailed objects displayed or maintained at one time) and a longer retention rate than is attainable by the human system. The results of their research showed that a properly designed system using computer graphics with modern technology can enhance human imagery and contribute to a better visual communication of a concept's meaning.

Visual Concepts

Weber (1986) suggests that the human imagery system be enhanced by icons to help reduce mental memory workload. The icon replaces many textual phrases and reduces the comprehension time frame needed to understand and develop a concept. An icon is defined as "an image; figure; representation; picture" (Webster, 1983). In the computing

environment, the term has evolved to mean a symbolic representation denoting a common object or location, or to direct data manipulation operations (Korfhage and Korfhage, 1986). Several studies (Rohr, 1986; Korfhage & Korfhage, 1986; Lodding, 1982; Montalvo, 1986; Carroll & Thomas, 1982) have been done on the upper and lower limits of information an image has before it becomes too complex for humans to manage. Rohr showed that a user must maintain a sense of the underlying meaning of the icon's representation so that he can use the icon effectively to accomplish a designated task.

Rohr (1986) and Rosch (1978) studied the use of visual concepts by analyzing how people represent different aspects of reality. The human imagery system works with a mental image as one form of its representations and can encode information into a spatial organization. The individual references items in his mental memory independently of whether the input was originally presented in text or graphical form (Rosch, 1978). The recall of information is independent of the number of elements or the order the elements were received (Rohr, 1986). This type of memory is relational encoding. It allows the individual to randomly access information similar to holistic representation where a representation is placed in memory and, when a retrieval is needed, a scan of the elements is made. A computer generated image should augment the human system by using icons that the user relates to and understands their representational meaning.

A number of existing visual systems are based on icons. Predefined or user defined icons are used for object representation. Icons are used to represent action or processing features in a language. The icons representing actions or processes are more abstract in nature and are not as flexible and convenient for the user to manage. Jones (1983) showed that icons representing complex, abstract concepts were made up of a combination of several pictorial representations, either super-imposed, linked, or even hidden. Icons of this nature tend to be misunderstood, thus causing misconceptions about the properties and

attributes that the icon was designed to represent. In either representation, icons are used to communicate ideas and actions between the user and the software system. Lodding (1982) classifies icons by design and function. The image of an icon should relate to the idea or action based on its resemblance (picture to object) or as an analogy.

In some cases, an inherent drawback associated with icons can evolve. An icon can have an erroneous interpretation. An ambiguous interpretation is made based on the icon's design or on a composite icon structure. In other cases, icons can take on different meanings depending on the context in which they are used. Another reason for misinterpreting an icon is because it contains an unmanageable amount of detail to be understood by the individual. The Korfhages (1986) point out that there is no universally accepted set of icons, but that icons just evolve through the development of visual features in systems. Many authors state that a user can use icons provided they are simple and perform simple functions. This claim is supported in studies conducted by Rohr (1986) and by Jones (1983). Their study revealed that if a pictorial representation of more complex structures or actions is defined by a complicated icon, the icon became too involved or too abstract for the user to manage. The user lost time and momentum in dealing with complex icons which would ideally assist him in his problem domain. Thus, the user required more time to perform a task or to develop a multistep algorithm using the more complex icons. Research in these areas has shown that the user spends a significant amount of time interpreting the icon rather than concentrating his efforts on solving the problem.

Rohr (1986) studied visual symbols, or icons in visual languages, from a cognitive psychology point of view. Rosch's study of visual concepts was based on the human imagery and the set of icons currently used in some visual languages. It showed that physical objects which occupied space tended to have a strong pictorial support within the

human imagery system. The icons representing such objects were completely understood by the user. The user had complete command of these icons and could manipulate these objects in an orderly, algorithmic way.

Humans learn and develop conceptual understanding by first decomposing large problems into smaller ones, developing an understanding of the smaller entities, and then reconstructing the smaller entities and their associated meaning into the overall intent of the concept. A graphical representation or an icon system can enhance the human's normal features and contribute to his ability to understand an idea in a shorter time frame. Rohr's study (1986) confirmed that essential functional components of the software structure should be made visible. But any complex representation should be decomposable in order to develop a semantic understanding of the underlying function representing the image. A visual system should depict certain system components using images that yield a clear understanding of the software and, if a conflict occurs, then the visual representation should be decomposable.

Visual Program Classification

Currently, numerous research efforts are directed to the incorporation of graphics or a visiual aspect into the computer programming environment. These efforts include using visual support, either internal or external with respect to programming languages, developing languages that are partially visual, and developing languages that are completely visually based. These range from general-purpose to special-purpose languages. Many of these visual systems represent software in both static and dynamic modes. Other systems make use of two and three dimensional images and accent them with coloring and highlighting (Grafton, 1985).

We can group visual languages and systems several ways. Shu's (1986) classification terms are *visual languages* and *visual environments*. Some authors use the terms *program visualization* and *visual programming* (Grafton, 1985); some use *visualization* and *program visual design* (Clarisse & Chang, 1986); while others group systems into *information processing* or *information displaying* systems. Another concept for grouping visual languages and systems is based on their design principles (Shu, 1986). These three broad categories are executable graphics (Lakin, 1986), graphics-enhanced, and software systems with graphics as an integral part (Shu, 1986). The use of vision in the programming environment is still relatively new and many of the concepts found in this area are not yet well defined.

Several visual systems use many of the standard graphics models that have been employed since the beginning of the programming era. These models consist of flowcharts and structured charts. The models impose a rigid structuring on the program's control flow used in the program design phase. However, they lack the ability to show graphical expressions, procedure calls, data and functions in a way most beneficial to the programmer. State diagrams and augmented transition networks are used for simple, automaton-type program segments, but they suffer the same drawbacks as flowcharts. Dataflow graphs do not convey semantic information beyond node and arc connection, and they lack pictorial representation for data structures and high-level control structures. Many of these models are uninteresting to the user in light of today's architecture and its graphical capabilities.

One classification of languages and systems that incorporate graphical representation provides for two categories that are based on the functional attributes of images. One category is the visual environment. Existing languages and software tools which have incorporated graphical representation are included in the visual environment

category. The second category is visual languages. Visual languages are concerned primarily with programming languages that have graphical control structures or graphically represented objects incorporated as part of the language itself (Shu, 1986). Shu has subdivided each of the broad categories into three subcategories. Table 2.1 shows the visual programming categories and subcategories. Additionally, a new subcategory under visual languages is included as a result of Chang's (1987) contribution.

Visual Environment
• Visualization of Program and Execution
• Visualization of Data or Information
• Visualization of System Design

Visual Languages
• Processing Visual Information
• Supporting Visual Interaction
• Programming with Visual Expressions
• Programming with Visual Expressions and Visual Information (Chang, 87)

Visual Programming Categories
Table 2.1

Numerous graphical systems are classified within the subcategory of visual environment. Visualization of program and execution allows the user to view a program from different perspectives. These systems may use multiple windows to display text, data

values and debugging statements. Visualization of data or information uses graphical images to depict the data structures and data values either in static or dynamic modes. The last subcategory, visualization of system design, uses icons in developing software systems. These systems use icons in designing software. The icons are then replaced with actual text statements resulting in the final program.

The visual language subcategories group languages based on how icons are used in developing and executing programs. One of these subcategories is *processing of visual (or image) information*. It uses icons that have an inherent visual representation for objects which are associated with certain logical interpretations. Areas using visual information processing systems are image processing, computer vision, robotics, image database management, and office automation (Chang, 1987). One of the earliest works using this type system is GRAIN (Graphics-oriented Relational Algebraic INterpreter) developed by S. K. Chang et al., (1978). A more recent example, which is an extension of System R's SQL language, is Pictorial Structure Query Language (Roussopoulos & Leifker, 1984), PSQL. We describe these systems in more detail below.

The second subcategory of visual languages is designed to support visual interaction. The user interacts with a data base through icons and the results are displayed in graphical form. Simple icons and table input formats control and instruct the software system to perform specific actions on the data.

The third and more aggressive category is programming with visual expressions. The language constructs are themselves visual. Expressions in the language are depicted visually even though many of them do not have inherent visual characteristics. Visual languages are applied in computer graphics, user interface design, database interface design, form management, and computer-aided design (Chang, 1987). These visual language systems are reviewed and compared in a later section.

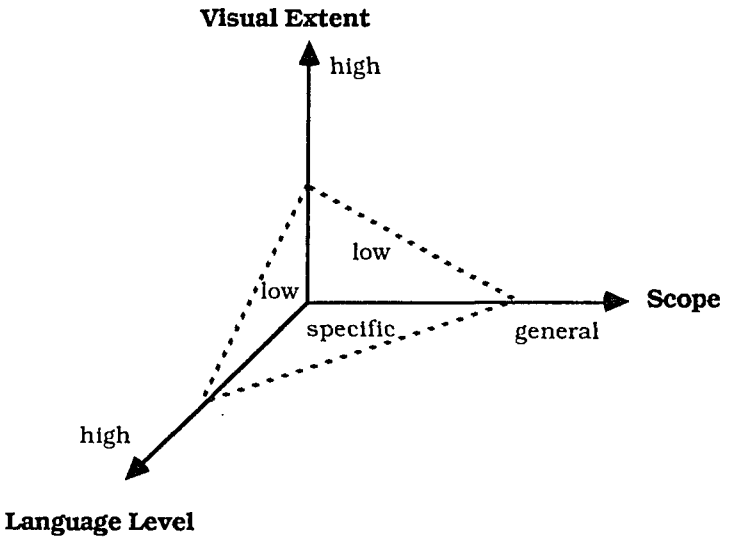
The three subcategories just described for visual environments and visual languages are Shu's language categories. Chang (1987), adds a fourth subcategory to visual languages. This category is termed the *iconic visual information processing languages*. The features of an iconic visual language have both objects and language constructs visually represented. The languages that fall into this category use icons to represent the objects and a set of processing icons to implement an algorithm.

In addition to Shu's (1986) visual programming categories, he also presents an analytical approach to qualitatively assessing the visual aspects of programming languages. This idea is not as precise as one would like, but it does provide a way to compare one visual language or visual system to another. Three domains for measuring a language are defined. The first is the *level of a language* where the level indicates the amount of detail the user must stipulate to instruct the computer on how to achieve the desired results. For example, Pascal language is considered higher level than an assembly language. Generally, it takes fewer commands to define a task in Pascal than in assembly code.

The second aspect is the *scope of the language*. The scope ranges from general purpose to special purpose. General purpose is applicable to a wide range of problems while special purpose is applicable to a narrow set of problems. For example, if a multitasking problem is presented as a programming task, then assembly language is a better implementation language than Fortran. Therefore, the assembly language is considered to have a broader scope than the Fortran language.

The third aspect is the *visual extent* of the language. The visual extent measures how much or how little the visual properties are incorporated into the language. The visual features include icons, graphs, diagrams, multiple windows, or pictures.

The three domains for defining a language can be represented in a three dimensional graph as shown in Figure 2.1 (Shu, 1987). Domains that are measured closer to the intersecting axes are less flexible then those domains measured farther out. An example of a general purpose language with medium visual extent, is shown with dotted lines.



Three Dimensions of Programming Languages
Figure 2.1

Shu (1986) points out that there are many questions to be answered in associating languages and pictorial representation. First we must design and define the properties of the visual features and encompass these features within a language. Choosing ill-conceived symbols or over-detailed pictures may be more bewildering than informative to a user. In addition, cluttered icons on the display may produce a "spaghetti" effect. It is believed, and earlier studies have given credibility to this conclusion, that the programmer perception of the software can be enhanced with the use of pictorial representations for

simple tasks, but there is no evidence that similar payoff exists for tasks containing higher complexity. Yet the power of visual representation is overwhelming. The improved technologies contributing to computing will result in more effort being placed on incorporating vision within the programming environment.

Languages and Systems Incorporating Graphics

A review of literature dealing with visual programming systems indicates that the visual programming languages can be grouped by the design principles used in their development. These systems fall into three categories. At one extreme is the total use of graphics defining the language data and control structures. These systems are referred to as executable graphics systems. At the other extreme are software systems that have been extended with graphical features. Between these two extremes are those software systems which are designed with graphics as an integral part of the system, but with limited general purpose programming capabilities. The following literature review classifies the existing languages and systems into these three categories. In studying these languages and systems based on these categories, an overlap of these categories exists, depending on the features found in the language or system (Shu, 1986).

Executable Graphics Systems

The most extreme systems consist of visual languages where the constructs of the language, both data and action, are visually represented. Lakin (Lakin, 1986) refers to these type of systems as "executable graphics". These languages use icons to develop and execute an algorithm interactively. Systems belonging to this group include Xerox's Star (Purvy et al., 1983; Smith et al., 1982), Pict (Glinert & Tanimoto, 1984), Programming by Rehearsal (Finzer & Gould, 1984), PIP (Reader, 1984), PLAY (Tanimoto & Runyan, 1986),

State Transition Diagrams (Jacob, 1985a), PegaSys (Moriconi & Hare, 1985; Moriconi & Hare, 1986), Omega (Powell & Linton, 1983), VennLisp (Lakin, 1986), SIBTRAN (Lakin, 1980a; Lakin, 1980b), VICON (Clarisse & Chang, 1986), HI-VISUAL (Monden et al., 1984), IPL (Chang et al. 1985: Chang et al., 1983), and Pygmalion (Smith, 1975). A brief discussion of these systems follows. Table 2.2 summarizes the characteristics of the systems by language level, scope, and visual content.

Xerox's Star system (Purvy et al., 83; Smith et al., 1982) is one of the earliest icon systems developed. The main emphasis projected by this system is the day to day functions carried on in an office environment. Through the use of icons and a mouse, the user can open documents, folders, file drawers, in-baskets, out-baskets, and a waste basket for managing paper work. The user can open documents, read them and then file them away, discard them, or send them elsewhere using a mail feature. The Star system also has a calculator that can perform simple functions with limited programming capability. The system was designed to include the features and functions that are found in an office environment.

Pict (Glinert & Tanimoto, 1984) uses a graphical programming environment where the user can construct programs using a joystick and icons. He can draw and connect the icons together. The system supplies the user with a set of icons for drawing and erasing as well as a bookshelf of icons representing a library of routines. Basically all programs take on a flowchart metaphor. The system is limited to simple numeric calculations and is considered a suitable system for the novice programmer. The profile of the Pict system is slightly higher than the Star in visual extent and approximately the same in scope. The language level is comparable to Basic or simple Pascal.

Programming by Rehearsal (Finzer & Gould, 1984) is a visual system written in Smalltalk-80 on the Xerox Lisp Machine. The Rehearsal program allows the user to set up

predefined troupes or objects. These objects are positioned in different areas of the screen. A production or an algorithm is defined through the use of objects passing messages among each other. Each troupe is programmed to perform on cue to a response. A troupe can initiate another troupe (place a troupe on stage) or block other troupes based on the cues it receives. The language is very high in visual extent since the troupes and the observation of the execution are visually displayed. The language is limited to manipulating only the troupes displayed on the screen. The system's ease of use allows the development of a limited program quickly.

The Programming in Pictures (PIP) system (Reader, 1984) allows the user to construct a program by drawing pictures. The drawing is managed by an editor. The system is targeted for the casual programmer who creates fairly simple programs. The motivation for the design is based on representing the operations of a program by a similar method that an individual uses as an informal communication of ideas between others. First, a simple data structure is represented. Then, with simple terms, the meaning or flow of the program is expressed. The PIP system allows a user to draw pictures using the picture editor and to associate a type with the images using a type editor. The function editor allows objects to be input and output between types using simple operations, such as arithmetic operations and object transformation or conversions. These functions are performed using pointing and line drawing facilities. The computational model used to execute these programs is based on Backus's (1978) form of functional programming. The system makes programming more interesting to the programmer because it shows meaningful, simple data structures which can be viewed and manipulated through the connected line drawings. The metaphorical power the system offers gives the programmer a sense of the algorithm implementation, but it can mislead the user. For example, large symbols on the display may or may not represent large values within the program or large amounts of processing.

PLAY (Tanimoto & Runyan, 1986), Pictorial Language for Animation by Youngsters, is an iconic programming system developed for the very young who have not yet mastered the textual language. The motivation for the PLAY system is to introduce the computer and its capability to the pre-school aged individuals. It enables its users to experience the functions and capabilities of a computer equivalent to that which is found in the limited domain that game playing offers. Each play or skit within the PLAY system looks like a comic strip consisting of a sequence of iconic sentences. The system allows youngsters to take on different roles in developing a play. The child can be a playgoer who watches a pre-programmed script, or a director who can adjust performances of play's objects, such as the stage or background. Finally, the individual can take the roll of director and compose and create objects for an entire script. The language level and the scope of PLAY are very low. The visual extent is high due to the use of predefined icons and specific semantics actions for certain icon combinations.

Another system which uses limited visual representation is the State Transition Diagram system (Jacob, 1985a). The system uses a finite state automata to show the abstract computation of a program. The system is a graphical representation based on Bakus Naur Form including nested constructs (Aho et al., 1986). A diagram consists of a set of nodes or states connected with links representing transitions, all based on token passing. The tokens can consist of input or output actions or they can represent procedures or function calls used to transfer from one state to another. The user can design, in an interactive mode, a finite state automata using nodes, arcs, and labeling transitions. The system can construct a textual program from icons displayed on the screen. The icons are associated with a set of predefined textual routines set up in a library. With the predefined icons and the interactive connecting capability, the user can define a computational model and then execute it. The State Transition Diagram system does not have any visual representations for data structures. The system has been successful in specifying and

directly implementing user interfaces for several prototype systems (Jacob, 1985b; Jacob, 1985c; Cornwell & Jacob, 1984). The visual extent of the language is considered to be low, but the language level and the scope are high, due to the system support of textually constructed library routines.

The PegaSys system (Moriconi & Hare, 1985; Moriconi & Hare, 1986), developed at Stanford Research Institute, is used for program design and documentation. PegaSys is an icon-based system in which the user can represent the program's functional components as a hierarchy of precise and meaningful pictures. The system does not allow for any recursive or overlapping program components, thereby limiting the language level. The user establishes the functional components of the program and their connection to other components by representing them as input/output or sender/receiver icons. The system uses a form calculus to check the composition of the picture elements for consistency while the user is constructing the program. Once the algorithm is pictorially constructed, it is then translated into Ada code. The system is a one-way translation from the hierarchical pictorial form to textual form. Any further refinement made to the program must be done at the textual level.

The Omega system (Powell & Linton, 1983) is similar to the PegaSys system in that predefined text code representing a program's functional components can be arranged through the use of interactive graphics. The user has the freedom to associate data objects with predefined icons stored in the system. The Omega system separates the object types from their pictorial representation, allowing the user to choose among several ways to display them. The user can choose to display data within the code or outside the code and can design simple icons to represent these data items. The icon design feature gives the user immense flexibility, but lacks the soundness needed in program design. Icons with fixed image and fixed meaning add to the user's ability to function and manipulate them

in problem solving. The Omega system is a textual based system using graphics in its development environment.

The VennLisp system (Lakin, 1986) allows a user to view a Lisp program as a nested set of closed objects rather than a nested set of parenthesis like those found in the Lisp language. The objects are data structures and functions. The system uses different shapes for objects based on the type of function being referenced. A unique feature found in VennLisp is the mechanism for spatial parsing. Using the PAM, PAttern Manipulation system, the parser can parse a spatial arrangement of visual objects or icons. There are a number of systems built on top of the spatial parsing system. VennLisp is one example. The VennLisp system can interpret the spatial objects and produce different outputs, giving the system flexibility. The interpretation can generate higher-level objects, trigger the manipulation or action on other objects, or generate a textual program representation.

Another language that uses the spatial parser is VIC, VIsual Communication (Gardner et al., 1976; Steele et al., 1985). VIC is used to teach and educate aphasics, people who have lost the ability to speak, to communicate again through the use of pictures. The sentences are strings of icons. A feature used in the computer version of VIC is a spatial parser used to parse graphical structures previously mentioned. The spatial parser is used to parse the icon sentence into meaningful structures and then to transform them into a textual sentence. Another use for the spatial parser is for parsing a finite state automata. Lakin (1986) gives an example of a finite state automata translated into a Lisp program.

A system developed by David Sibbet, called SIBTRAN (Lakin, 1980a; Lakin, 1980b) is used as a conversational graphics tool to help people think and understand information in a more coherent way. The SIBTRAN mimics the way an individual uses a chalk board in defining a system and its interconnection. The conversational graphics consist of some graphics, some text, and grouping of these entities into meaningful objects. Arcs are used

between objects to depict dependencies between them. SIBTRAN uses the spatial parser to parse both graphical pictures and limited texts into organized sentences. As the conversation continues and objects are shuffled, the relationship of each object with other objects is maintained and represented on the display. The SIBTRAN system is a manager of objects and cannot actually be classified as a visual language; but the objects and the algorithmic management of those objects, along with their dependencies, qualify SIBTRAN as a visual system.

VICON system (Clarisse & Chang, 1986) is a generalized icon system, implementing objects with two types of attributes. These two attributes are aspect and relation. VICON, running under a LISP environment, allows the user to design his own icon entities and associate with each of them aspect attributes. The aspect attributes include icon name, bitmap name, window name, menu, and built-in functions. The relation attributes are pointers from one icon to another and represent a parent, sibling, and child icon arrangement. The icons are structured hierarchically, allowing them to inherit attributes from their parents and also to define relations between them. The icons are associated with LISP functions which are invoked when the icon is activated. The VICON system is a visual language approach to functional programming. It can be used in circuit design, architectural design, naval architectural design, CAD/CAM, image databases, and geographical database systems. The system is highly visual and is used for designing visual objects which require limited semantic support. The semantics are limited to simple calculations such as those found in spread-sheet software. VICON system has limited programming capability and is implemented in a textual based language.

The HI-VISUAL system (Monden et al., 1984) is very similar to the VICON system with respect to the hierarchical relationships. The hierarchical relations between icons are more explicit in HI-VISUAL than in VICON. HI-VISUAL is based on a hierarchical

multiple window model using user defined icons to operate on data icons (Yoshino, 1984). Although pictures are used for both data and action, the system is implemented in a text based language. There is one drawback to programs written in HI-VISUAL. Each program runs independently of other programs. No data can be communicated between separate routines or programs. This limitation prevents a user from building new programs based on existing programs. HI-VISUAL has led to an international project to design a prototype, general-purpose iconic visual information processing language called IPL (Chang et al., 1985: Chang et al., 1983).

System Name	Level	Scope	Visual	Developer
STAR	Low	Low	High	Xerox
Pict	Low	Low	High	Gilnert, Tanimoto
Programming by Rehearsal	Low	Low	High	Finzer, Gould
PIP	Low	Medium	High	Reader
PLAY	Low	Low	High	Tanimoto, Runyan
State Transition Diagrams	High	General	Low	Jacob
PegaSys	Med-High	General	Low	Moriconi, Hare
Omega	Med-High	General	Low	Powell, Linton
VennLisp	Medium	Medium	Med-High	Lakin
VIC	Medium	Low	High	Gardner, Steele
SIBTRAN	na	na	Medium	Lakin
VICON	Medium	Specific	Medium	Clarisse, Chang
HI-VISUAL	Medium	Medium	High	Moden

Executable Graphics
Table 2.2

Graphics-Enhanced Software Systems

The other extreme for graphical languages and systems is those systems which incorporate graphics into the programming environment as an extension of the language. The graphics enrich the programming environment for the programmer, but the programs are able to run without the use of the graphical interfaces. Some of the systems represented in this group are BALSA (Brown & Sedgewick, 1985; Brown, 1988), Visualization of Independence and Dependence for Program Concurrence (Belady & Hosokawa, 1984), Pigs (Pong & Ng, 1983), SDMS (Herot, 1980; Kramlich, 1984), PV (Brown et al., 1985), PECAN (Reiss, 1984; Reiss, 1985), and VIPS (Isoda et al., 1987). The primary functions of these systems are described individually in the following discussion and summarized in Table 2.3.

BALSA (Brown & Sedgewick, 1985; Brown, 1988) was developed at Brown University as an external tool to be used with a programming language as an instructional tool. The Brown University Algorithm Simulator Animate (BALSA) is a simulation, animation program used for demonstrating program execution and changing data structures. The student can see the effects that certain code or data structures have on the algorithm. The procedure for defining an animation session is as follows. First, the instructor develops a program or algorithm for classroom discussion and presentation. Then, using the algorithm designer, the instructor develops a script that defines the animation elements of the presentation. Once the script is defined, it is then merged with the program code. The program can be executed and the students can see the concept in animated and textual form repeatedly. The script can be stopped and discussed during lecture and minor run time adjustments, such as speed, break points and data values can be made. As the program executes, the student can observe the algorithm's actions. It gives the student an increased understanding of the code supporting the algorithm.

A limitation to BALSA system is the time required to set up a presentation. The initial system required about two hours of development time and about 15 to 25 hours of programming time for each 15 minutes of script. Balsa-II (Brown, 1988) has improved the user interface for the script developer and has reduced the development time. Brown states that it still takes a modest amount of time to set up a script. The system is an excellent tool for teaching and for testing new algorithms and data structures, but it lacks the features necessary for a program development environment. In a program development environment, the programmer must design, code, and test a newly developed program. Once the program has migrated to the stage of correct execution, then the program can be enhanced with Balsa routine calls that demonstrates the program's features. This scenario also carries over to the program maintenance environment. With its high visual content, the BALSA system has been used successfully in the education environment and in the research of new algorithms and data structures.

A system proposed by Belady and Hosokawa (1984) incorporates graphical extensions to explicitly indicate which program segments can be executed in parallel. The proposed system, "Visualization of Independence and Dependence for Program Concurrency", incorporates these extensions into an existing language. The graphics is limited to annotating the right side of the source code with a vertical notation. A two-dimensional textual representation is used to display sequential sections and potential parallel sections of code. The vertical dimension lists the set of statements to be executed and their order with respect to the other statements. The sequencing dependencies are represented along the horizontal or time axis within the vertical axis. The system has very little vision in that the display is textual in nature. The proposed system has not been developed to date.

The Pigs system (Pong & Ng, 1983) is an experimental system for Programming with Interactive Graphical Support. The system is primarily used for interactive debugging

and testing tools to support program development of Pascal programs. The user can incorporate graphical extensions into a Pascal program which, when executed, can be viewed on a display device. The graphical routines are a given set of fixed library routines supporting the graphical interface. The format of the model uses Nassi-Shneiderman Diagrams (NSD) (Nassi & Shneiderman, 1973) to show the execution sequences of a program. The system has a very low visual extent and the language level and scope are that of a simple version of Pascal . The system can only handle integer types and one dimensional arrays in its NSD displays. These limitations reduce the language level.

The Spatial Data Management System (SDMS) (Herot, 1980; Kramlich, 1984) uses icons to pictorially represent database objects and to control the navigation process. SDMS, developed by Computer Corporation of America, is a database system which uses icons as an interactive medium and in turn displays the results graphically. The programmer can use an icon-class description language to create SDMS icon statements. The description language uses system command features such as TEMPLATE, SCALE, and COLOR to create system icons. A text form is used to create the icons, then the SDMS command system uses these icons as its interactive medium. Along with icons, a joystick is used to create views of the static database by zooming in and out of repetitive nodes found in the hierarchical structured database. The SDMS system format is a graphical view of the database and is primarily used as a direct data manipulation in an information retrieval system.

The Program Visualization (PV) system (Brown et al., 1985) was developed based on the results of SDMS. The PV system supports static and dynamic images of a program in a multidimensional informational data space. As the programs executes, the program's control structures and data structures are graphically displayed. The user can monitor

control structures and alter data values during program execution. The programmer can develop graphical pictures and associate code with these images. The executing program can be displayed in a graphical and textual format. The graphics is supported by a library of routines that the programmer must develop and link to within his program. As stated earlier, defining icons and associated text gives the programmer significant flexibility, but it can easily result in ambiguous representations. The PV system is as visual as the user would like, depending on the amount of graphical support code and time the user puts into developing the program. The level and scope of the language are dependent on the actual programming language in which the system is implemented.

The PECAN system (Reiss, 1984; Reiss, 1985) consists of a family of programs used in program development. The PECAN system supports multiple views of a user's program. The user can view his program as a pretty print text, Nassi-Shneiderman interconnecting diagram module (Nassi & Shneiderman, 1973), an abstract syntax tree, and an input-output dialogue. This system allows the user to view the program and monitor it as it runs. The PECAN system uses many formats for displaying a program such as flow chart, displaying variables and their values, and highlighting executing text statement. It has the ability to abstract a syntax tree which represents the underlying system. Another system similar to PECAN is VIPS, Visual and Interactive Programming Support (Isoda et al., 1987). Both of these systems are very text oriented, making use of large high resolution display screens and multiple windows for displaying a program in different formats.

The languages and systems just presented contain graphics as an extension. Some of the systems can be classified under *visual programming* and others under *program environments*. A few systems overlap both classifications. In the following section we present systems that contain graphics as an integral part. These differ from the systems just presented in that graphics is an essential part of the system's operation.

System Name	Level	Scope	Visual	Developer
BALSA	High	Specific	High	Brown, Sedgewick
Visual of Indep. and Dep. for Pgm.Concurrence	High	General	Low	Belady, Hosokawa
Pigs	Medium	General	Low	Pong, Ng
SDMS	Low	Specific	Medium	Herot, Kramlich
PV	High	General	Low	Brown, Carling, Herot, Kramlich
PECAN	High	General	Low	Reiss
VIPS	High	General	Low	Isoda, Shimomura, Ono

Graphics-Enhanced Software Systems
Table 2.3

Software Systems With Essential Graphics

The middle group of systems is designed with graphics as an integral part of the language or system. These systems do not have the powerful icon feature for developing expressions discussed in the first group. However, they do require graphics for execution whereas the second group can run without the graphical interface. The systems that fall into this group are basically tables and forms oriented programs such as those used in office automation and information systems. Representatives of these systems include GRAIN (Chang et al., 1978), PSQL (Roussopoulos & Leifker, 1984), ISQL (Assmann et al., 1986), QBE (Zloof, 1981), OBE (Zloof, 1982), FOBE (Luo & Yao, 1981), and FORMAL (Shu, 1985). Table 2.4 contains a summary of these systems.

GRAIN (Chang et al., 1978) is one of the earliest works in which a language and graphics were incorporated. The user specifies an image query in the GRAIN language

and the system provides the results as a displayed picture. The framework for storing and retrieving information is based on the relational database concepts. A similar example is PSQL (Pictorial Structured Query Language) (Roussopoulos & Liefker, 1984) which introduces a pictorial language for manipulating pictorial and alphanumeric databases. It is an extension of the System R's SQL language. Another database system developed for use in medicine is ISQL (Assmann et al., 1986). Its design is based on a relational conventional DBMS using SQL database language. These systems use images for displaying results of queries, but the languages themselves are textual based.

Query-By-Example (QBE) (Zloof, 1981) is a system that allows users to query a database through the use of a user defined skeleton table. The user sets up a skeleton with appropriate labels representing data base fields. Selection fields with attributes are defined within the table for filtering desired data from the data base. A similar graphical system using tables in the office environment is Forms Operation by Example (FOBE) (Luo & Yao, 1981) and Office by Example (OBE) (Zloof, 1982). OBE allows tables to be nested within each other by defining major headings and subheadings. It is used for processing data and displaying results. FOBE is another office information processing system. It is a user-friendly model using high-level algebraic operations for manipulating nested tables. The office systems require predefined calculating routines that are matched with fields defined in the table format. Most of these systems are considered to be nonprocedural languages which implies their language level is quite high. The visual extents of these systems are not as high as STAR or Pict but are higher than Pigs and the system Belady-Hosakawa proposed. The scope of these systems is narrow since they perform operations with simple flat tables and interface with organized fixed data structures.

Another system with graphics as an integral part is FORMAL, developed at IBM's Los Angeles Scientific Center (Shu, 1985). The design of FORMAL addresses the needs of the nonprogramming office worker by allowing the user to computerize many relatively complex data processing applications. This system prevents the user from having to learn the intricacies of a programming language. The skeleton of a FORMAL program consists of a "form heading" with a list of properties, or commands. These commands specify such things as the SOURCE for the data origination, MATCH using parameters for selecting certain fields, CONDITION for filtering, and ORDER for depicting output sequencing of the instances. The system allows headings and subheadings, contributing to the dimensions of the forms. The language level of FORMAL is comparable to QBE. The forms are compiled entities thus giving the compiler responsibility for generating the correct algorithms needed to produce the desired inquired results. FORMAL is considered an excellent tool for the nonprogrammer in an office environment. It is powerful enough to perform many of the tasks found in office operation systems and it is easy to use. The system is still considered an experimental one and is currently being defined and refined.

System Name	Level	Scope	Visual	Developer
Grain	Low	Specific	Medium	Chang, Reuss, McCormick
PSQL	Low	Specific	Medium	Roussopoulos, Leifker
ISQL	Low	Specific	Medium	Assmann, Venema, Hohne
QEB	Low	Specific	Medium	Zoolf
FOBE	Low	Specific	Medium	Luo, Yao
OBE	Low	Specific	Medium	Zoolf
FORMAL	Low	Specific	Medium	Shu

Software Systems With Essential Graphics
Table 2.4

Desirable Graphical Features

The goal of graphics use in any software system is to help programmers and users achieve a deeper and more accurate understanding of the system's behavior and to enhance the effectiveness of system interaction. Many of the systems presented are primarily designed for data base and office automation systems. The user of these systems interacts with the software systems without having to know a formal programming language. These systems were developed for a specific use and thus they lack general programming capabilities. Many other systems, such as Pict and PLAY, have a high visual context but a low programming level.

The systems that are of particular interest in this research are found in the "Graphics-Enhanced Software" category. These systems are based on the standard programming languages used in developing software systems. The systems with a medium to low level

of programming are not adequate for program development. PECAN and VIPS are high in programming level but are low in visual content because most of their displays are based on textual representation. BALSA is high in all metrics but, as stated by the developers, it is an environment for training and demonstrating rather than for program development (Brown, 1988).

We identify three desirable features of a "Graphics-Enhanced" software system. The first desirable feature is that it reflect the implementation of the program based on the program's text. The current systems that use a higher level language either have a low visual content or the graphical representation is very abstract. The abstract graphical representation requires the user to first interpret the representation and then interpret the code to determine where modifications are to be made. For the programmer to understand the graphical representation, he must abstract for the elements of the display the meaning of each element and how they interact with other elements. If the programmer needs to modify the code, he must extract from the textual representation those sections of code that support the particular elements which need to be modified. Thus, the graphical representation should represent the implementation of the coded algorithm. A program that is coded well or a program that is coded inefficiently should generate equivalent representations respectively. To improve or modify a program, the programmer will ultimately alter the code and not the graphical elements, therefore a graphical representation should have a close relationship to the text it represents. The systems that were just reviewed, which support the flowchart or Nassi-Shneiderman formats reflect this feature.

The second desirable feature of a graphics-enhanced system is that it model the program as defined by the coded algorithm. The programmer needs information about the operational flow of the program's tasks (Soloway, 1988). The graphical medium should

provide a representation of the program that will assist the understanding of the program's execution order while simultaneously relating these executing tasks to their textual representation.

A third desirable feature of a graphical system is that it provide knowledge of the program's identifiers. The order in which the programmed tasks execute is one vital aspect of program understanding. A second aspect is the set of identifiers affected by these tasks. The programmer must be concerned with the identifiers, their attributes and their relationships to the control structures. The current graphically supported systems fall short of supporting all of these features. Many of the systems represent programs as a static entity while others are able to simulate the program's execution.

Summary

The groups of systems described in this chapter are not exhaustive. They were selected due to their connection with a programming language or with functional programming capabilities. There do not exist clear classifications or metrics to group or measure systems and languages containing a visual property. We have presented a categorization of these systems along with additional classification information used by other authors. We have also discussed studies involving icons. These studies indicate the relationships icons should have with their representation and the limitations that humans have in using icons.

In Chapter III, we propose a graphical representation of a program based on its textual definition to assist the programmer in strengthening the control-flow understanding process. The system generates a graphical representation of the program's control flow based on its code, supplies scope and structure information about the identifiers, and addresses the problem of code delocalization. The proposed system is classified as a

Graphics-Enhanced software system. The language selected is Pascal, which is classified as a general-purpose, high-level language.

CHAPTER III

A GRAPHICAL PROGRAMMING ENVIRONMENT

Introduction

Computer programs are used to implement algorithms in a step by step manner. The user must understand the program's construction before he can debug or perform program maintenance. The objective of this study is to graphically represent a programmed algorithm for the purpose of enhancing the program understanding process. To clarify certain areas of the study, an introduction of terms and concepts is provided. Finally, the research problem and methodologies are defined.

There are two basic aspects a programmer must understand when implementing an algorithm for processing by a computer system. One is the program's textual representation and the other is the program's execution flow. The textual format or the language constructs for defining a program is referred to as the programming language. An algorithm is implemented based on these constructs. A program's textual structure can be considered as a linear string of symbols depicting the algorithm. Some authors (London & Duisberg, 1985; Shu, 1986) consider the textual form as a one-dimensional format of an algorithm. These concepts are derived from the program's format as it exists in memory. If we abstract the program's textual structure to a slightly higher level, we can envision a program as a two-dimensional configuration. The first dimension can be expressed as a left-to-right string of tokens for a given line of code. The second dimension is represented by the lines of code tracking from top to bottom. For the purpose of this study, a program is considered to be two-dimensional in form unless otherwise stated. The two-

dimensional configuration is related to the textual form of the program represented in a high level language.

The second aspect of program comprehension is the arrangement of the control structures. This arrangement defines the execution flow that the program follows to satisfy the desired goal. This concept is viewed as the program's dynamics. The dynamics are the execution flow of the control statements from one construct to another or from one construct into another, if nesting occurs. The textual structure of an algorithm does not completely reveal the dynamics associated with a program. The programmer discovers the dynamics by first browsing the text and then transforming the text into mental images representing the program's execution flow. The programmer must develop the program's dynamics to understand the functions of the program. Once the programmer develops this understanding, he then uses this information to perform alterations needed for debugging, maintenance, or in new program construction based on old program segments.

Problem Definition and Motivation

A program's textual form yields inadequate semantics about the program. This research develops a graphical framework that assists the programmer in the code understanding process. The main goal of this research is to develop a model which reflects the dynamics of a program and in turn supports the programmer in developing the program's semantics. The design of the system will also reflect the natural processing method a programmer uses to develop the program semantics.

First, we define the basic elements needed to represent the program's dynamics. Since the dynamics of a program are based on the ordering of its control structures, we will treat the program's control structure as significant entities of the model. We will refer to

such entities as *tiers*. A tier in this context refers to the entire construct defining the control structure. For example, an iterative tier consists of a FOR statement and its body or a WHILE statement and its body. With this definition, a program is described as a set of tiers where the tiers are defined by the textual specifications of the programming language.

The program control structures are grouped into seven tier types. These types are the iterative, alternate, procedure body, function body, main program body, and procedure reference and function reference tiers. We select these elements to describe the dynamics of a program based on the practices a programmer uses in defining the dynamics (Basili & Mills, 1982; Soloway et al., 1983). The programmer browses the text and assigns meaning to these sections or tiers of code. The tiers are the control structures and thus are the objects we use to define the program's dynamics.

Second, we expand the program's two-dimensional perception into a three-dimensional space. A group of control structures or tiers can be viewed as either sequential or nested structures with respect to the local structures. We identify the program's basic control structures, as described above, along with their relative positioning to each other. One structure following another defines a sequential ordering while one structure within another defines a nested ordering. The first two dimensions of our three-dimensional perception are used in defining sequential flow of the program. The third dimension defines the nesting of tiers. This third dimension is referred to as the nesting axis. A three-dimensional perception of the program is obtained by positioning the nested tiers along the nesting axis. A nested tier appears deeper on the nesting axis than the tier which contains it. We refer to the positioning of the tiers along this axis as the nesting levels. For example, the program's main body is placed at the highest level and

nested structures are placed at the next deeper levels depending on the tier's relative positioning.

Given the control structures as the objects and a three-dimensional space as defined, we represent the dynamics of a program graphically. By placing the tiers in the three-dimensional space, a programmer can see the order in which the control structures are executed. Sequential execution of tiers indicates control flows through one tier and into the next. This is graphically represented by arranging the tiers, one under another and on the same nesting level. Nested execution indicates control flows from an outer tier into the inner tier, then back. Nested control structures are represented as tiers placed along the nesting axis and are contained within the outer tier. The program's control flow is understood to move from the top tiers to the bottom tiers and from the outer tiers to the inner tiers within the three-dimensional space. The three-dimensional representation provides a way to graphically show the program's flow of control.

The program's semantics are not solely based on its dynamics. The programmer must be aware of the identifiers and their use within the control structures. Control structures control the flow of execution and the order the variables are altered. Therefore, understanding the program's semantics also involves understanding the relationship between the program's dynamics and the identifiers. To understand identifiers, we need to know their attributes. The attributes of the identifiers are required in order to synthesize the program's semantics. These attributes consist of the the identifier's name, type and scope. When a programmer utilizes the textual form of the program, he browses the text looking for the identifiers and their attributes. This is a localization of information problem in which the information we need is not local to the point of interest. If the programmer has the set of identifiers available at the point of interest, he can save time and also maintain a coherence in his understanding process. This feature assists the

programmer in developing an accurate understanding of the relationships among the control structures and among the control structures and the associated identifiers.

Based on the description of the process a programmer uses to develop program semantics, a graphical model has been described that reflects a program's flow structure and methodology used by the programmer in developing program understanding. Using these concepts as our foundation, we now describe the goals of the research.

Research Objective

The goal of this research is to provide an environment that supports the program understanding process. A problem the programmer must deal with is to recover the program's intentions as defined by the code. The programmer needs analysis techniques that make correct program facts easily available (Letovsky & Soloway, 1986). Our goal is to investigate a graphical technique that provides easy access to the local and non-local information at a specific point of interest. This information must be accessible in a time-effective manner. We investigate methods to represent the code, local and non-local, and scope and type information about identifiers.

We propose the utilization of a visual environment to represent localized information with interactive features allowing the user to achieve the level of detail or level of abstraction required for developing program understanding. The environment we developed is the Peec System, Program's Execution Environment Configuration. It provides a three-dimensional graphical representation of a program based on its control structures. The environment supports the textual comprehension process but augments it with visually represented structures and with identification of live identifiers. It provides interactive features and gives the user the power to view the program's control flow from different perspectives.

The development of the support environment consists of the following five phases:

Phase 1: Definition of the Building Blocks

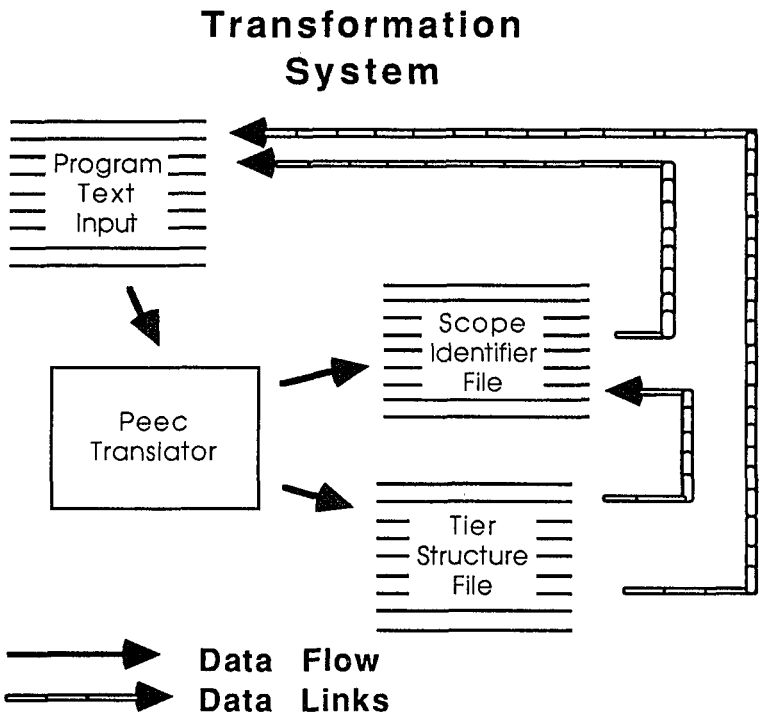
The Peec system creates a graphical representation of the program's dynamics based on the program's textual structures. In order to create a graphical representation, we must understand the natural procedure a programmer uses in developing program understanding. From this procedure, we obtain the control structures as the building elements the programmer uses in constructing his internal execution model. The programmer organizes these structures, assigns meaning to them, and then treats the structure and its meaning as a signal entity.

Phase 2: Definition of Graphical Format for Model

Once the building elements are defined, we create a graphical format for displaying the program's execution flow. The human's imagery system perceives objects naturally as three-dimensional structures, therefore the execution flow model is based on a three-dimensional representation showing a spatial organization of the control structures. The objects are the control structures arranged in the order of execution within a three-dimensional space.

Phase 3: Program Transformation

The transformation phase is a primary phase of the Peec system. A textual formatted program is transformed into tier information, based on the program's control structures. This phase is a translation process where the input is a textual program and output is a tier structure file and a scope identifier file. It is illustrated in Figure 3.1.



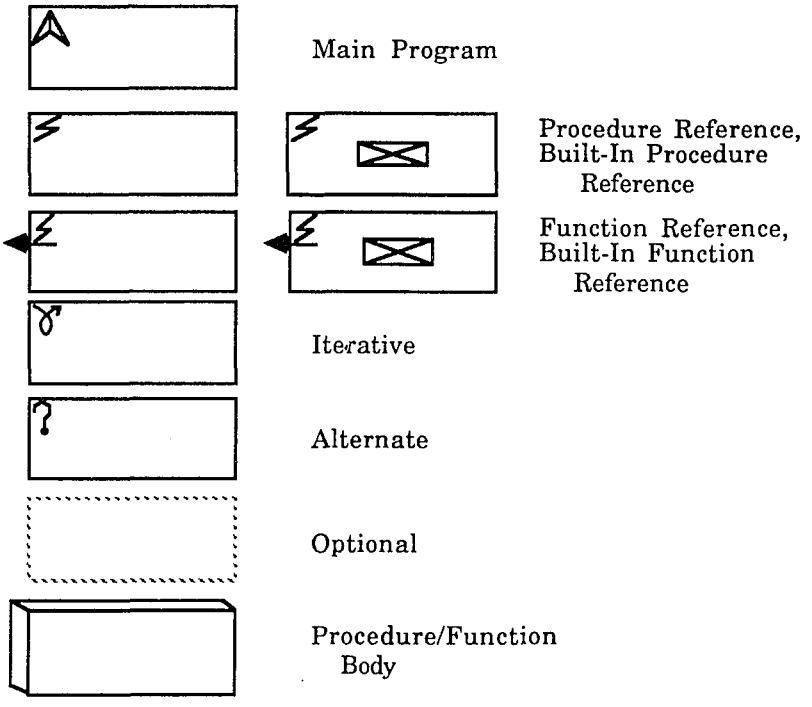
Transformation Process
Figure 3.1

The tiers are the objects that the Peec system uses in developing the program's flow model. The translation phase is separated from the display system to allow for increased language independence. An identifier scope file is also generated in the transformation process. The scope file contains the attributes of each identifier and their relationship to each tier. Both the tier and scope files are linked with each other and with the program's text file. These three files are the inputs to the image generation phase.

Phase 4: Image Generation

The image generation phase creates a graphical representation of the program's execution environment. The generated image is based on the information supplied by the tier file. The tiers are represented as two-dimensional planes positioned in a three-dimensional space. The tier's types are identified by either tier shape or the icon associated with each tier. The tiers and their icon definitions are shown in Figure 3.2 and are described in detail in Chapter IV.

Tier and Icon Definitions



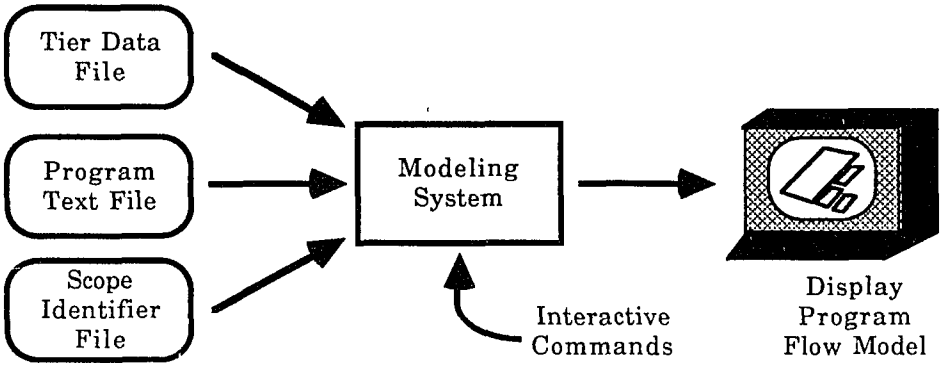
Tier Definitions
Figure 3.2

Phase 5: System Interaction

The interaction phase allows the programmer to interact with the three-dimensional representation. The interactive features allow the programmer to browse through the program's control flow model from tier to tier. He can view the relationship of structures to other structures based on their execution order. The programmer can decompose the tier image to finer details when needed. For example, he can display the control structure's text and study in detail the particular code defining the structure. The programmer can also display the identifiers or a subset of identifiers, based on the scoping rules and their type. This feature supplies the programmer information local to the point of interest. For example, the programmer can display local integer and real identifiers for the current tier. The control structure's text and the identifiers are displayed on demand allowing the programmer to decompose the tiers down to their lowest level, the textual representation.

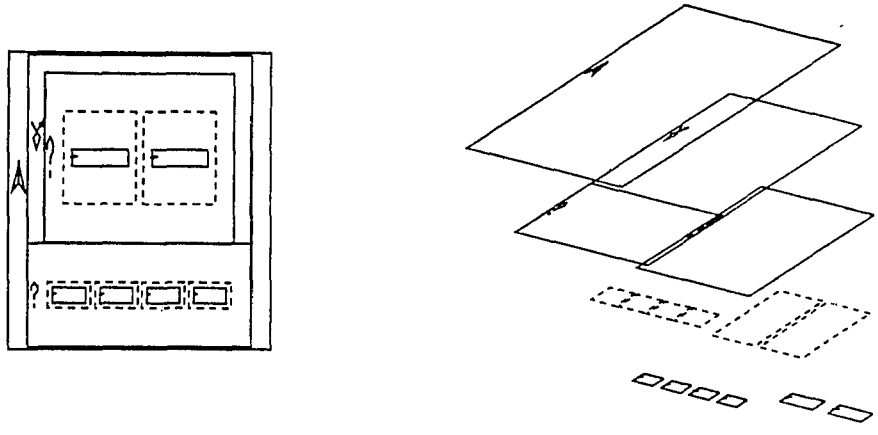
A set of interactive commands control either the browse point or the image as a whole. The browse point is represented as a modified tier image. The programmer controls the movement of the browse point allowing him to move through the three-dimensional image. The programmer can also issue commands that alter the three-dimensional model's view. Since the model is a three-dimensional image, there is a set of commands to rotate, scale, and translate the image. This flexibility gives the programmer a different perspective of the program's execution flow.

An overview of the Peec system's major segments and the information flow are shown in Figure 3.3.



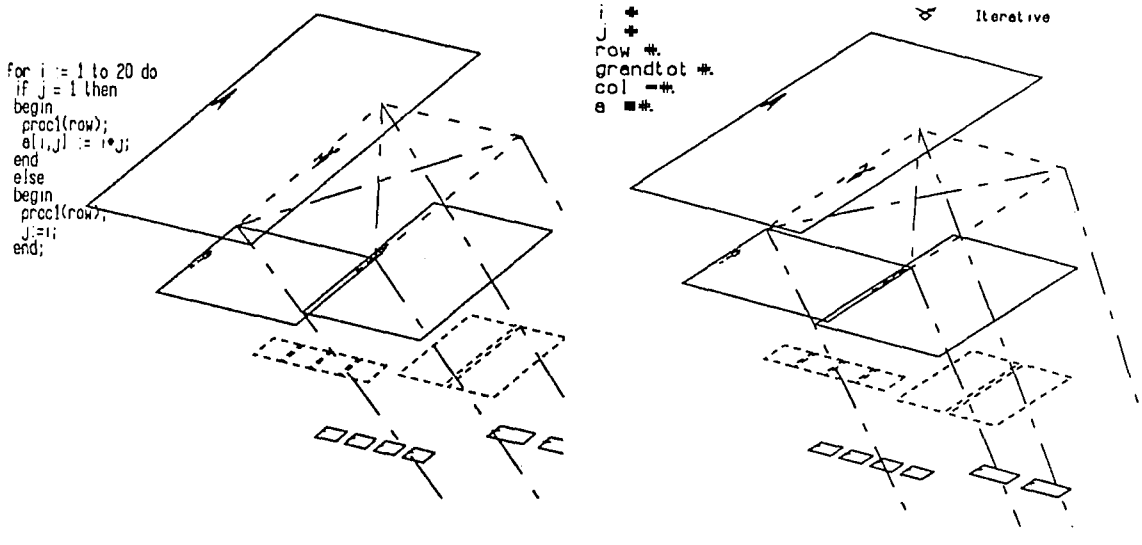
Peec System
Figure 3.3

An example has been supplied to assist with clarification of the work. A Pascal program and its execution flow model are displayed in several figures. Program A in the Appendix A is the example Pascal program used in the model. Figures 3.4(a) and 3.4(b) show the program's execution flow from two perspectives. Figure 3.4(a) is sighted down the nesting axis and Figure 3.4(b) is sighted off center of the nesting axis. The tier structure's text is shown in Figure 3.5(a) and its identifiers in Figure 3.5(b).



(a) (b)

View of Model
Figure 3.4



(a) (b)

Text and Identifier Display
Figure 3.5

Hardware and Software Support

The features required in the Peec system help define the appropriate software needed in the developmental stages. The first phase of development is a translation task. The Unix operating system supports compiler construction tools used in constructing translators. The Lex and YACC tools are used in constructing the Peec translator. The second phase requires the use of three-dimensional graphics. A three-dimensional graphics package, Graphic MetaFile Resource (GMR), is used to construct the program's control flow image. The GMR package requires the C language as its interface language. With Lex, YACC and GMR graphics packages, the Peec's developmental language is C. C is also chosen for its portability across different systems within the Unix environment.

The implementation of the Peec system, based on interactive graphical images, requires specific hardware capabilities. To display three-dimensional images with an acceptable level of detail requires a high resolution monitor. Also, the system must allow interaction with the image. These functions require updates and alterations to be made to the flow model. The updates to the model must be fast enough to maintain continuity between the programmer and the abstraction process. Therefore, fast hardware or fast hardware/software combination is required for development. The selected hardware is the Apollo DN3000 workstation. The Apollo's processor is a 68020 with a 68881 math-coprocessor. The associated monitor is a 1280 by 1024 pixel, monochrome device. The Apollo system runs the native Aegis operating system with Unix implemented on top.

Summary

The goal is to investigate a method which represents the code and the identifiers local to the point of study in a time-effective method. The concepts of the Peec system are to

display a block structured language in a three-dimensional form and to allow the user to browse through the program structures in order to develop an understanding of these structures and their relationships. The Peec system supports the procedure similar to one that the programmer uses in reviewing a program listing, that is, it synthesizes from it the control structures and their operational flow patterns. Within the Peec system, the user can shift from one control structure to another and view these structures in the context of their execution environment instead of their textual definition. The graphical model constructed by Peec depicts control structures local to their execution point which aids in the understanding of the program's operational sequence.

The Peec system provides the user with icons for labeling the control structures. For convenient and quick reference, icons are used to identify the type of structures, such as loops, conditional statements or procedure references. If the user needs to study the contents of a particular structure in greater detail, he can display the actual text defining the structure.

Peec also provides the user the ability to view identifiers, or a subset of identifiers, within the scope of the current tier or control structure. An individual who is studying a program can browse through the model, viewing the local control structures, popping up text for more detail and displaying identifiers. The Peec system is designed to assist the programmer in understanding the program by suppling information on the program's control flow, access to the control flow definition, and access to the active identifiers. The Peec environment allows the programmer to interact with the model to develop a stronger understanding of the program's goals and how these goals are met.

The system has many viable uses in the programming environment. The programmer can be taught the fundamentals of program execution through the Peec visual system. He can visualize how the program is designed to execute through the execution

flow model. The Peec system also provides the programmer a tool for developing program understanding for use in debugging and program maintenance. This system allows the programmer to view the dependencies or lack of dependencies between control structures and variables. Finally, the human imagery system is enhanced with the three-dimensional representation of the program's execution flow model which allows him to study a program, from diverse perspectives, in terms of its execution order.

CHAPTER IV

GRAPHICAL REPRESENTATION OF PROGRAM COMPREHENSION

Introduction

In the initial phases of our research effort we defined the processing method a programmer used in program comprehension and then identified the basic elements of this method. The programmer is assisted in developing understanding and meaning of a program if some type of image is used for support. The image can be internal, external, real or abstract. Hence, part of the first stage consists of designing a model which supports the internal image a programmer can utilize for program understanding. We have presented the framework of such a model, its elements, its concepts, and the research goals in Chapter III. In the next section, we describe the building elements and the model, and provide arguments to support these entities.

Program Comprehension Process

The relationship between a language statement and its intended operation is referred to as the semantics of the statement. If we abstract the semantic concept to a higher level, we can associate semantics with a group of statements or segments of the program. The programmer derives meaning from these segments by comprehending the execution flow and the variables affected by the code arrangement.

Traditionally, a programmer develops program understanding by abstracting its operational flow from a program's listing. The flow is based on the ordering of the control structures, the relationship among these control structures, and the identifiers that are affected by these structures. The programmer also develops understanding by studying

small sections of code in a local area of the program. The user then assigns meaning to these sections representing the task the code is designed to perform. For example, he may determine that a loop computes a summation or it may traverse a link list. We refer to these sections of code as tiers. Once the meaning is established for the individual tiers, the programmer continues the abstraction process by systematically grouping these tiers and their meaning, and abstracting understanding to higher levels. This process continues until the programmer has enough understanding to allow him to carry out appropriate programming functions such as updating, modifying, or migrating code. This process is supported by Basili and Mills (1982) and Soloway (1988).

Over the past 30 years, the refinement of programming language's textual structures has improved significantly. The improved structures have increased the programmer's ability to manage and design data structures and control structures. As a result, the programmer has been able to improve both the quality and quantity of programs generated. However, there still exist a large gap between the program's textual structure, or the algorithm's symbolic representation, and the program's semantics. For example, suppose a procedure is defined within a program where the procedure is a generic type used by other segments of the program. The procedure can take on a different implication depending on where it is referenced and its set of input parameters. If the procedure is referenced from the main body of code, it can have a different intent than a reference made from a function. For the programmer to understand the purpose of the procedure, he must know the context in which it is referenced. The programmer develops this understanding by browsing the code and internally developing the program's dynamics, placing the procedure at the point of reference, that is, localizing code.

There is another method used to develop program understanding. The programmer can explicitly determine the semantics by executing the program multiple times with

different sets of input data. By designating certain input data and placing output statements or trace statements within the code, the programmer can observe the results, follow the program flow through the debug output, and then develop an understanding of the program's functions. If the program is very complex, it may take several runs with different inputs before the user can obtain this understanding. This methodology is used frequently for developing understanding for small segments of a program.

Another technique used in discovering the program's semantics is found with the use of debug and trace software support tools. With these tools, the programmer controls the execution of the code by stepping through the code by instruction or by sections and observing the order the program tasks are executed. By tracing the steps of the program, the user develops a perception of the program's dynamics. As with the previous case, the programmer is limited to a particular set of data for a specific run. Depending on the complexity of the code, the programmer may repeat this process numerous times before developing the understanding he needs.

We have described different methods a programmer uses to determine the program's semantics. These procedures allow the programmer to develop an abstract view of the program's execution sequence. The elements used in establishing the program's semantics are its dynamics, the identifiers and the context in which they are used. The program's dynamics is based on the execution order of its control structures where the control structures and their particular arrangement in the text define the program's operational flow. In the programming environment, more efficient methods are needed for the understanding process. This need inspired this research.

There are two facets of the understanding process which we address. One facet is the control structure denoted as a tier and the other facet is the model representing the program's operational flow. The tiers are used to define the operational flow. To justify

tiers as the building elements of the program's operational flow model, we draw from four sources. One is personal experience. I have designed, written, debugged and maintained programs for a number of years. I have used this approach to programming by abstracting from the text the program's dynamics and associated meaning.

A second source is the studies on cognitive strategies of understanding a program's constructs which suggest that program understanding is not typically done on a line-by-line basis (Soloway et al., 1983; Shneiderman, 1979). The process consists of reducing the program to smaller understandable components then combining them in a step-by-step process until an understanding of the program is achieved (Basili & Mills, 1982).

The third source comes from interacting with other programmers. The interaction involves tracking down program bugs or determining if a segment of code handles a problem a particular way. In these cases, the programmers recognize certain sections of code which handle particular areas of the problem. These sections of code are either a control structure or a group of control structures such as procedures. The programmers uses these entities and their associated meaning in developing and discussing aspects of the program.

The last source comes from text books examples where program code is used to demonstrate and support concepts of programmed algorithms. If the code has comments, the comments are used to described the control structure or structures in terms of its function and purpose. The program comments are placed at the top of loops, top of procedures, and the true and false sections of IF statements (Dale & Weems, 1987). The reader associates meaning with the control structures through these comments in the same fashion described earlier. From these sources, we see that meaning is attached to control structures and the programmer associates meaning with these structures. Therefore, we

define the control structures as tiers and use them as the building blocks for the program's execution model.

The second facet of program understanding that we address is the configuration of the program's operational flow model. Generally, the programmer abstracts from the text the execution order of the control structures. The programmer generates an internal representation of the program's execution flow. He used this internal model to support his developmental process in understanding program functions. A graphical representation of the program's control flow enhances the programmer's abilities by supporting his internal model with a visual image.

In the human mind, information and relationships are represented in holistic or spatial arrangement (Rosch, 1978). Since the human mind has a three-dimensional aspect, or a picture oriented mechanism, the model representing the program's operational flow is similar in order to conveniently support the programmer in his understanding methodology. A three-dimension image relates to the programmer's internal representation, allowing him to conveniently and easily interface with such a model.

Model Configuration Definition

The program's dynamics represent the order in which the control structures execute. The Peec system uses a three-dimensional flow model to reflect the program's dynamics rather than its textual organization. To clarify this difference, we present two examples. The first example shows little advantage gained in a three-dimensional representation within a narrow area of text. The second example gives a better flavor for a three-dimensional model configuration.

We are familiar with a loop nested within a loop. An example of a program's textual representation of a nested structure can been shown as a FOR loop, coded inside another. The programmer typically uses indentaion to emphasize the nesting of structures. In the three-dimensional model, the structures are represented as two iterative tiers, one placed behind the other along the nesting axis. The programmer can determine that one tier is nested within another by their positioning along the nesting axis. In this example, the text and the three-dimensional model are very similar in that the programmer has two different forms of visual support for abstracting the structure's relationships to one another. The programmer can abstract the control flow from either the indented text or the positioning of the tiers. An individual can reason the same conclusion for multiple levels of nested loops structures.

In the second example we define the dynamics associated with a procedure reference. In the textual form, a procedure reference is shown by using the name of the routine in a reference statement. The procedure's declaration is defined within the declaration sections of the program. The program syntax places the procedure reference and procedure declaration at two different points within the text. The programmer realizes control moves from the reference point to the body of the procedure at the time the procedure is called. To develop an understanding, the programmer's internal model places the procedure code at the point of reference. The idea that localization of information contributes to program understanding is supported by Solway (Soloway et al., 1983; Letovsky & Soloway, 1986). With this feature, the programmer associates meaning with the procedure relative to its reference point. The programmer discovers a clearer meaning of the procedure based on the context of its use and based on the internal dynamic model he has developed from the code. We can continue the same reasoning process for procedures referencing other procedures or for procedures referenced recursively. With the recursion, the user

perceives a successive ordering of procedure references which define the operational flow of the program.

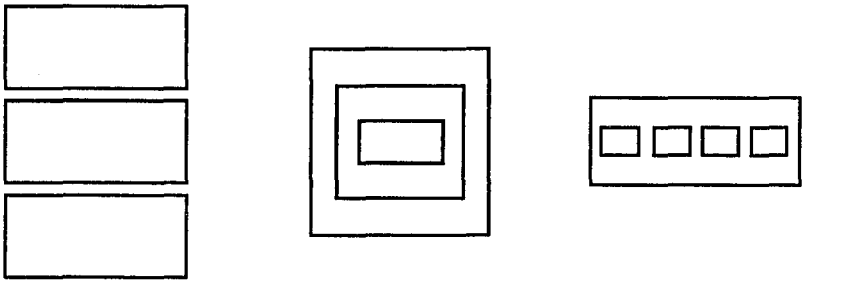
In a three-dimensional model, the dynamics of a procedure reference is represented in a similar way. The procedure reference tier is positioned on the nesting axis at the level the call is made. The procedure's body, with all of its control structures, is shown as nested tiers with respect to the reference tier. The arrangement shows the program's flow will move from the higher levels down to the lower ones. The procedure's tiers represent a deeper nesting of control structures which results in these tiers being placed further back on the nesting axis. If the procedure is recursively referenced, then the procedure's body is nested in each reference tier and is shown as the user browses into the reference tier. This representation symbolizes the order the flow of execution takes when the program executes. The three-dimensional configuration of the procedure references gives the user a sense of the execution flow defined, the order the procedures are referenced, and the depth of the nesting.

Thus far, we have defined the operational flow model as a three-dimensional representation and the elements in which to construct this model. The third attribute to resolve is the arrangement of these objects in the three-dimensional space. There are three relationships that control structures have with one another. One relationship is the sequential control flow. In a sequential flow, one control structure follows another, indicating that control is passed from one structure to the next when the first structure has completed execution. Figure 4.1(a) depicts this relationship.

Another relationship is nesting of control structures. A nested relationship shows that one structure is contained within another. Control is passed from the outer control structure to the inner one. The inner structure executes and then relinquishes control to the outer one. Another way to view this process is that control moves from the outer structure

down into the nested structure or next lower level of control. The outer level regains control only after the inner structure has completed execution.

We can extend the nesting concept to procedure references as illustrated in the previous example. Control flows from one level to the next lower level where the next lower level is the body of the procedure. If one procedure references another, then the referenced procedure will be at a lower nesting level than the procedure initiating the reference. All nested structures must execute before control flows back to the outer levels. Figure 4.1(b) shows how nested tiers are represented.



(a) Sequential (b) Nested (c) Alternate & Optional

Relationships of Tiers
Figure 4.1

The final relationship is the alternate flow, such as the body of an IF or CASE statement. The alternate flow signifies that one of several control paths is to be taken. In an IF statement, there can be a true or false segment of code to select, whereas in the CASE statement, there are several possible choices of code to select. Control flows through one of the optional segments to the next sequential structure following the IF or CASE construct.

The textual representation of an alternate statement shows the optional segments coded sequentially or one optional segment following another within the listing. The programmer abstracts that only one segment will execute. In a flow chart representation, the optional segments are shown horizontally, indicating control flows through only one of the segments. This gives an accurate representation of the control flow for the optional statements nested within the alternative statements. The horizontal placement of optional segments nested within the alternate constructs is the representation adopted in the operational flow model. Figure 4.1(c) shows a CASE statement with four optional segments.

The Nassi-Shneiderman charts have some similarities to this relationships of tiers. They show a static arrangements of the control structures as defined by the textual representation of the program. The Nassi-Shneiderman diagrams convey more information about program-component break-down than about the program control flow. Also the pictures used are uninteresting for the programmer (Reader, 1985). The Nassi-Shneiderman charts do not show procedures calls or data types.

We explicitly define the tier types used in the model. The program control structures are grouped into the following types of tiers: iterative, alternate, optional, procedure/function reference, procedure/function/main body tiers. The iterative tier is used for looping control structures. Such structures are the FOR and WHILE statements. The alternate tier defines the conditional control structures. An example of an alternate tier is an IF or CASE statement. Since conditional constructs alter the execution flow, we distinguish between the optional statements in the body of these constructs and define them as optional tiers. An IF-THEN-ELSE construct is defined as an alternate tier with two nested optional tiers representing the TRUE and FALSE segments of the IF statement.

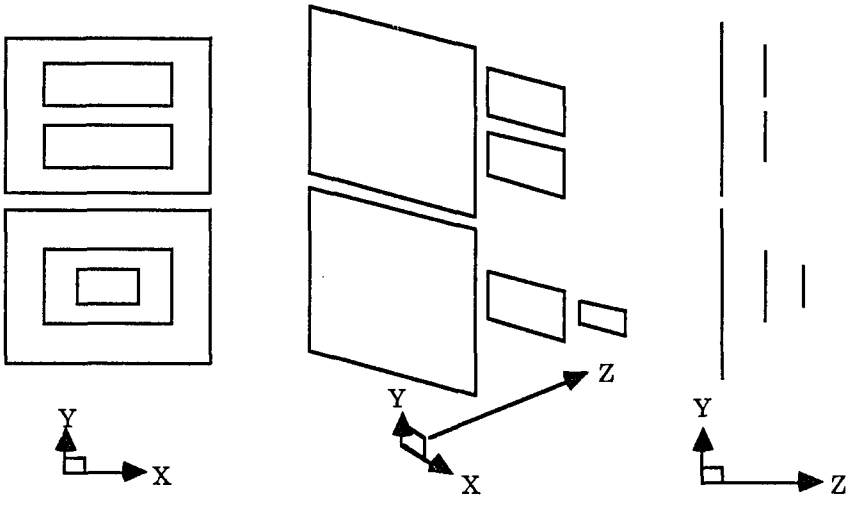
The procedure and function references define the procedure reference tier and the function reference tier. At the point a procedure or function is referenced, it is considered as a segment to which the programmer associates meaning. It also alters the execution flow of the program, therefore we identify the reference statement as a tier. Finally, the body of a procedure, function, and main line code are treated as single control structures and are identified as tiers.

Based on the definition of the tiers and the relationships among them, we formally define the arrangement of these tiers within the three-dimensional model. We define the three-dimensional space as having X, Y, and Z axes. We will refer to the Z axis as the nesting axis where the negative direction is into the display and the positive direction is outward. A tier's image is defined as a two-dimensional wire frame image positioned in the three-dimensional space. The placement of the tier in three-dimensional space is based on the tier's nesting level and the relationship to surrounding tiers. Tiers that represent sequential processing are placed one above the other in the same XY plane.

Nested tiers are placed along the nesting axis at incremental distances in the negative direction. For example, the main program is defined at nesting level zero and is positioned in the XY plane at z=0 nesting level. Tiers nested in the main body are placed along the Z axis in the negative direction. Nested tier structures are placed at $z' = z - \sigma$ distance further back along the nesting axis.

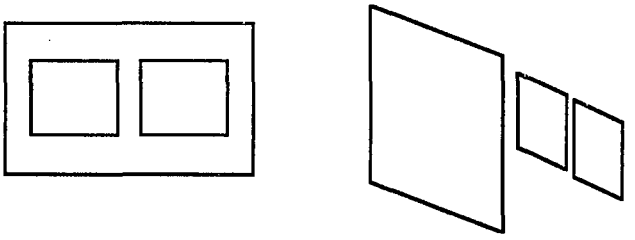
Figure 4.2 gives examples of the tiers in three-dimensional space. The figure shows two sequential tiers at the highest nesting level. The first sequential tiers contains two nested tiers, which are themselves sequential and are placed in the same XY plane. The second top level sequential tier shows two levels of nesting defined within it. Each nested tier is positioned on different XY planes along the nesting axis. The figure shows the configuration of these tiers. Since the image is in three-dimensional space, three views of

the same configuration are shown from different points of perspective. The programmer obtains a sense of the control flow by the sequential and nesting relationships of the control structures and by viewing such relationships within a model.



Three Views of Nested and Sequential Tiers
Figure 4.2

The optional tiers are the nested entities of an alternate statement such as the IF or CASE. Optional tiers are represented as nested tiers positioned in the same XY plane but arranged horizontally. This arrangement reflects control flows through one optional tier down to the next sequential tier. Figure 4.3 shows an IF statement in a three-dimensional space with two views. The three-dimensional space and the configuration of nested and sequential tiers presented are the format used to define the operational flow of the program.



IF-THEN-ELSE Statement Tiers
Figure 4.3

Summary

In this chapter, we have described a methodology that programmers use to develop program understanding. Based on this methodology, we have defined the building blocks used in constructing a graphical model of the program's execution flow. We have also defined the format of the flow model, portrayed in a three-dimensional space using two-dimensional tiers positioned in XY planes along the nesting axis. The tiers and the three-dimensional configuration of these tiers form the basis for graphically representing the program's execution environment. In Chapter V, we define the abstraction process used to collect the tier and identifier information needed to construct the program's execution flow model.

CHAPTER V

PROGRAM TRANSFORMATION PROCESS

Introduction

To generate a program's execution model, we must abstract from the program's textual form the attributes describing each control structure. These attributes consist of the control structure's type, relationship to the surrounding control structures and its associated identifiers. The following sections describe the transformation process that provides the information needed by the Peec system to define the program's execution model.

Data Acquisition

The transformation process requires input of a textual formatted algorithm, written in the Pascal language. The transformation process abstracts two types of information from the text. One type consists of tier information and the other type consists of identifier information. Two files are produced from the transformation process, the tier structure file and the scope identifier file.

The information collected by the transformation process is organized into four categories. The first category is a set of parameters describing each tier. The tier information includes the number of lines within a tier, the number of nested tiers, and the tier's nesting level. The line count information is used to calibrate the height of the tiers images represented within the model. The tier's height is proportional to the number of lines defining the structure. The control structure's nesting value is based on its declaration or definition within the program's text. For example, a loop structure in the main program is defined at one level and a nested loop is defined at the next lower level.

The nesting value determines the relative position of the tier on the Z axis. The relative nesting position refers to the tiers position in the execution environment and not to the program's textual definition.

The second category of information consists of linkage between the tier's record description and its text definition as defined in the source file. Indices link each tier to the actual source statement defining the structure. Tiers that define procedure and function declarations have two links to the text file. One link defines the starting point of the procedure's heading or declaration section and the other link identifies the first source statement in the body of the procedure or function. These attributes are used to display source statements during the browsing process.

The third category defines the set of identifiers within the scope of the given tier. The scope identifier file includes information about the scope of identifiers using the standard scoping rules of the Pascal language. Each tier defined is linked to the set of identifiers that are within its scope. A complete definition and explanation of the scope identifier file is presented later in this chapter.

The last information category deals with the relationship the control structures have with each other. This relationship is defined through appropriately linked records within the tier file. One such link connects each tier to its immediate outer tier. The outer tier link establishes the hierarchy between these structures relative to their position in the program text. The tier record for a procedure or function has an inward link with respect to the nesting level. The procedure or function body and its declarations are treated as a single entity and are referred to as a declaration tier. The definition includes the routine's heading, local declarations, including other procedures and functions, and the routine's body. The inward link links the declaration tier to the first tier defined in the body of the procedure or function. The Peec system displays only the procedure or

function's body as the execution portion of the routine and not its declaration. The image generated by Peec system shows the procedure or function's reference point positioned on one plane and its body on the next nested plane. The source statements for the procedure's body or the procedure's declaration are accessible upon request while the user browses the model.

The last link defined binds a procedure or function reference tier to the routine's declaration tier. This connection is defined only for references to user-defined procedures and functions. Built-in routines cannot be browsed; therefore, the link for these reference tiers are not defined. The Peec system distinguishes between user-defined and built-in procedure and functions within the model. The user perceives these differences within the model based on the type of icon associated with the reference tier.

The information presented above defines the attributes required for each control structure. These attributes are used in sizing, identifying and representing the tiers within the three-dimensional model. Links within each tier connect the structure to source statements and a list of accessible identifiers. Finally, the relationships between structures are defined through links and are used to determine relative positioning of tiers within the flow model. These attributes form the information base needed by the Peec system's display and browse phases.

The following sections describe the major data structures and algorithms necessary for abstracting the tier and scope information used in the Peec environment.

Abstraction Process

The transformation system was developed using LEX and YACC compiler tools. With these tools, we constructed a Peec-Pascal compiler which accepts a Pascal program as input and outputs a tier structure file and a scope identifier file. A set of regular expressions

specifying the basic lexemes of Pascal was defined for the lexical phase of the compiler. The tokens defined by LEX are the standard tokens found in most languages and will not be enumerated here. Included in the BNF rules were semantics actions or segments of C code. With the LEX and YACC outputs and a number of supporting routines, the Peec-Pascal compiler was constructed. The major features of the transformation procedure are described in the following sections.

Symbol Table

The transformation process maintains a symbol table of the source program's identifiers and their associated attributes. The information included in the symbol table is the identifier's lexeme, token value, nesting level, data type, index to source code, and an internal index which defines the identifier's scope. The lexeme, which is a string of characters defining the symbol, is stored in a linear structure indexed from the symbol table. An extension to the identifier token is included in the Pascal grammar. The extension classifies identifiers as either program name, procedure name, function name, a type identifier, constant, record identifier, or input/output identifiers. By default, an identifier is classified as a variable type identifier. The extension assists in the parsing phase and in managing the identifier's data type. This is reflected in the records produced in the scope identifier file.

The symbol table is initialized with the language's reserved words including the set of built-in procedure and function names. The Peec system discriminates between user-defined routines and routines supplied by the system as mentioned earlier. Associated with each identifier is its predefined token value used by the parser for syntax analysis. Also, the identifier's type is used in forming a data type descriptor.

The descriptive structure describing the identifier's data type is made up of a four part integer value. The first field describes the identifier's basic type. The basic type for variables is the type of value that can be assigned to the variable. A variable declared as INTEGER allows only integer values to be assigned, therefore, its basic type is integer. Other identifiers have a basic type in which the identifier is not a variable such as PROCEDURE and TYPE identifiers. The basic type for a procedure identifier is a procedure basic type. The identifiers defined as TYPEs have negative values to differentiate TYPE identifiers from other identifiers. A list of the basic types and their associated values are shown in Figure 5.1.

Basic Type	Value	Basic Type	Value
Type	negative	Pointer	7
Constant	1	Set	8
Integer	2	Procedure	11
Real	3	User_Define	12
Character	4	Function	13
Boolean	5	File	14
Record	6		

Basic Data Types
Figure 5.1

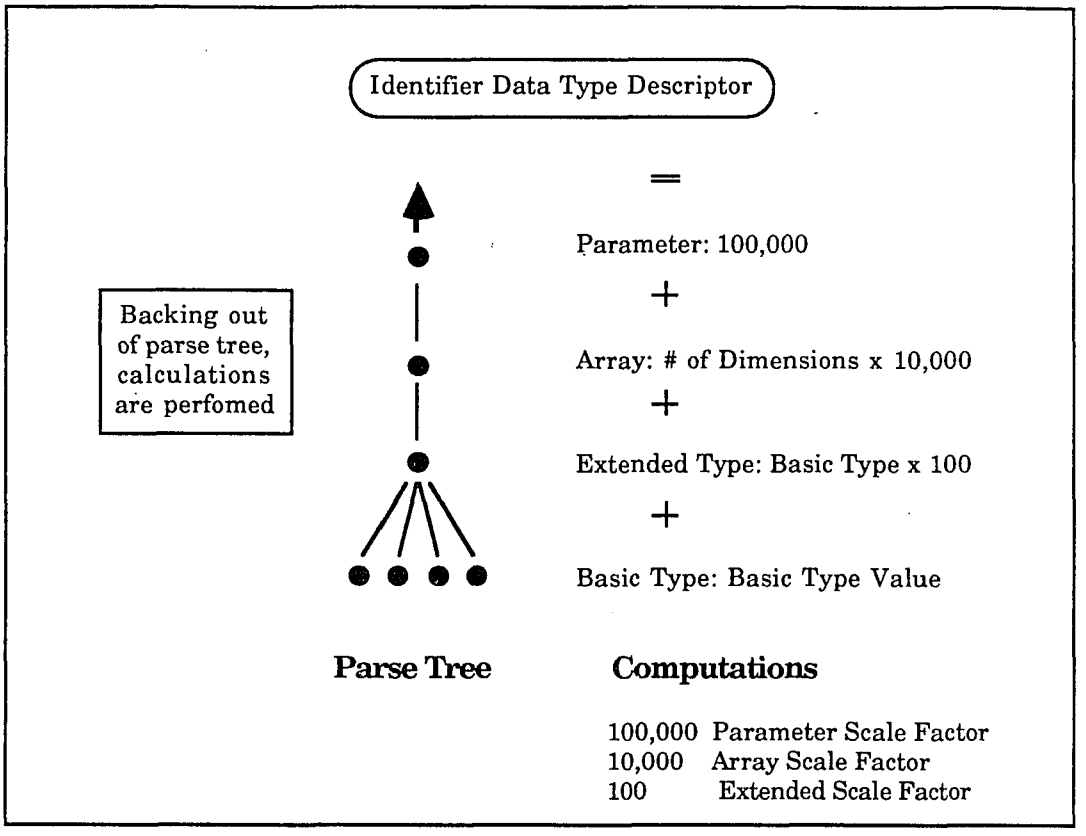
The second part of the data type descriptor is the extended type field. The extended part is used to describe composite types for identifiers. A simple example is an identifier defined as a set of integers. The basic type is *integer* and the extended type is a *set*. Another example is a real constant identifier declared in the CONST section. The

identifier is described by its basic type *real* and its extended type *constant*. The extended value is calculated by scaling the extended type value, and adding it to the identifier's basic type. In the example of a real constant, the *constant* type is scaled by a factor and then added to the *real* basic type value, yielding the identifier's descriptor.

The third part of the descriptor field describes array identifiers. This field signifies identifiers that are arrays and includes the number of dimensions associated with the array. The array can be considered as another extended type, but in the Peec system the array is an exceptional case and it is handled in a separate field. By separating the array type, we can improve the description of an identifier. A scale factor is used to define the array attribute. Multiples of this scale factor indicate the number of dimensions defined for the array. The resulting array value is added to the data type descriptor.

The last field indicates whether the identifier is a parameter to a procedure or function. If the parser determines that an identifier is a parameter, then the parameter scale factor is added to the data type descriptor. The data type descriptor is a fixed point field and its values which are calculated during intermediate phases of the parsing process.

Figure 5.2 depicts the algorithm used in calculating identifier type attributes. In Figure 5.3, an example of code and resulting data type descriptors is given.



Data Type Algorithm
Figure 5.2

```
Const   max 99.9;
Type    array20      = array [1..20] of char;
        EmployeRec   = record   name: array20; age: integer; end;
        Biggy        = array [ char, 1..200, boolean ] of real;
        PtrAry       = array [1..20 ] of ^integer;
        UD           = ( one, two, three );
Var     UDAry        : array [ char, 1..20 ] of set of UD;
        a            : integer;
        b            : Biggy;
        c            : array [ 1..50 ] of Biggy;
        d            : ^EmployeRec;
        e            : PtrAry;

Function    F1 : real;
Procedure   P1;                 { Sample Identifiers }
```

Identifier	Base	Extended	Array	Descriptor
max	3	1x100		103
array20	-4		1x10000	-10004
EmployeRec	-6			-600
name	10004			10004
age	2			2
Biggy	-3		3x10000	-30003
PtrAry	-2	7x100	1x10000	-10702
UD	-12			-12
UDAry	12	8x100	2x10000	20812
a	2			2
b	30003			30003
c	30003		1x10000	40003
d	6	7x100		706
e	10702			10702
F1	13	13x100		1303
P1	11			11

Example Data Type Calculations
Figure 5.3

In addition to the type information, the identifier's source index and the identifier's nesting level are recorded. The source index links each identifier to its declaration point

in the program text file and is used to reference the identifier's definition. A nesting level is recorded for each identifier based on the identifier's declaration position. The main program identifiers or the program's global identifiers are defined at nesting level zero. All the identifiers in the main program declared in the CONST, TYPE, and VAR sections are defined at this level. The procedure and function identifier names declared in the main program are also defined at level zero. The identifiers declared within these routines are defined at nesting level one. If nested procedures and functions are defined within these routines, then the identifiers declared within them are defined at the next greater nesting level.

The last part of identifier information designates the scope. The symbol table maintains a list of identifiers that are in the scope of the current line of text being parsed. Each identifier is linked to a previous identifier that is within the same scope definition. As the symbols are output to the scope identifier file, each scope record is linked to the previous identifier record within the same scope.

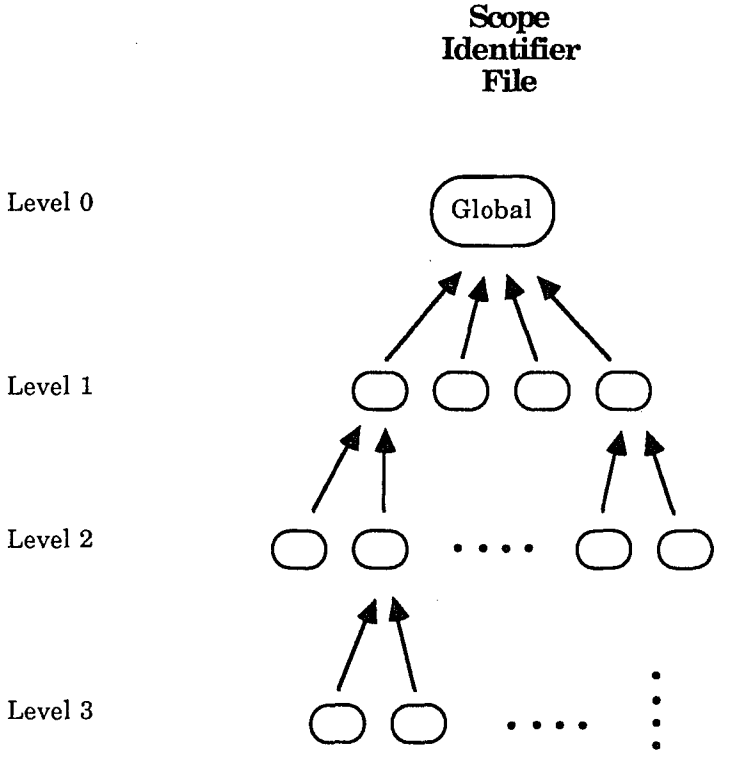
The symbol table management has a dynamic feature. During the transformation process, the number of identifiers within the symbol table increases and decreases based on the nesting level. As the parser moves in and out of nesting levels, the symbol table continues to grow adding new identifiers and attributes. When the nesting level reaches zero, the identifiers and their associated attributes are output to the scope identifier file. These identifiers are then deleted from the symbol table. The resulting file represents intermediate snap shots of the symbol table based on the scoping rules of the language.

Scope Identifier File

The scope identifier file's underlying data structure is organized as a multi-node tree defined by the scoping rules of the language. A node in the tree is composed of several scope

identifier records. The information specified in each identifier record was given earlier in the symbol table definition. Each identifier record is backward linked to a previous equivalent nesting level identifier or to the next next outer level identifier within the same scope. The first set of identifiers listed in the scope file are the global identifiers and are considered the root of the scope tree. The exception to this specification is the procedure and function names which are not included at the beginning of the file. The procedure and function identifier records are scattered throughout the scope file. They are however, included in the global set through links defined within the file.

Figure 5.4 shows the underlying abstract data structure of the scope identifier file. The root node represents the global identifiers and are shown at level zero. Level one represents the identifiers declared within the procedures and functions of the main program. The level two nodes represent identifiers nested within level one procedures and functions and so forth. Each node of the scope tree is linked to a node higher in the tree. This organization allows a tier record to be linked to a scope tree node which defines the tier's set of local identifiers. Other identifiers within scope can be determined by traversing upward through the tree. From this organization, Peec can determine which identifiers are local, nonlocal and global to a given tier definition.



Underlying Structure of the Scope Identifier File
Figure 5.4

Tier File

The second output file generated during the transformation process identifies each control structure within the program. First, we describe the format of the tier structure file. We then describe the tier abstraction process.

The attributes describing a tier are its construct type, number of lines, number of nested tiers, nesting level, immediate outer tier, and various indexes. Many of these

attributes and their purposes have been described earlier in this section. The transformation process generates a tier file where each record of the file describes a control structure defined in the program text file. Also, each tier record has indexes where the indexes are links within the tier file itself, links to the scope identifier file, and links to the source code file. First, we will present the tier file format and then present the algorithm used to construct it. The description of the tier file format clarifies the algorithm and the data structure format that are used to support the abstraction process.

The tier attributes are recorded in fixed binary records. Each control structure is described by a tier record. The records are arranged in the same order in which the tiers are specified in the program's text file. For example, if sequential control structures are defined in the text, then the tier records defining each structure are recorded one after another. If structures are nested, then the outer tier is recorded first followed by the nested tiers. The position of each tier record in the tier structure file correlates with the first line of code defining each control structure in the program.

There is one exception to this format. The main program's tier is positioned as the last record in the file. The placement of the main program's tier gives the Peec system a known starting point for generating the program's flow model. A link in the main program tier connects it to the first nested tier defined in the body of the main program. The first tier in the program's main body is located up in the file and is defined relative to the other tiers. The resulting tier structure file defines the tiers of the program's as a static form of the program's control structures.

Tier File Construction Algorithms

There are two aspects to defining a tier and recording its attributes in the tier structure file. The first is identifying points in the parsing process where control structure attributes

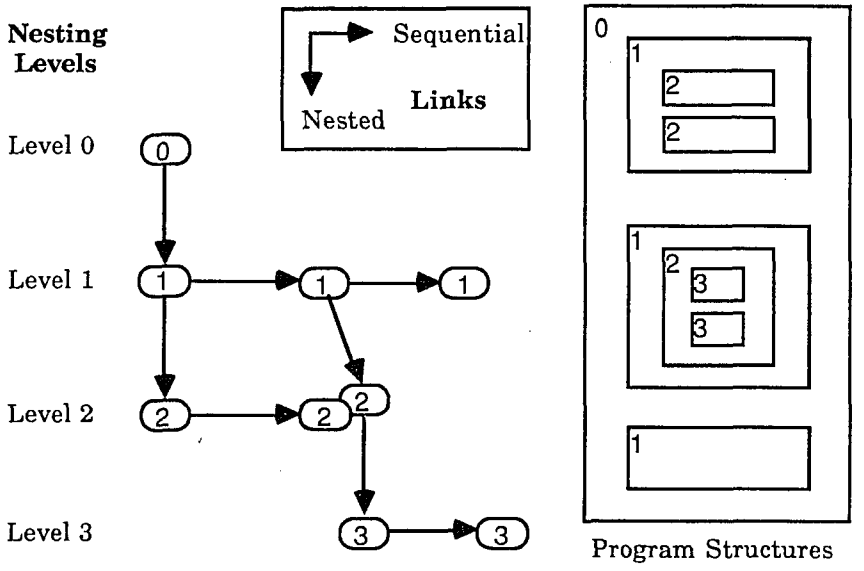
need to be collected. The second aspect is the management of sequential and nested structures information for recording the tier's sequential and nested attributes correctly. The identification and collection of the basic tier information is prompted at two locations relative to the control structure's parsing points. These two locations are the beginning and ending of a control structure definition, referred to as the opening and closing points. When the parser detects the beginning of a new control structure, a tier record is opened and initial tier information is recorded. The initial information is its nesting level, indexes into source file by byte address and by line number, and an index to its immediate outer tier. When the end of the tier is encountered, additional attributes information is recorded and the tier is closed. These attributes are the number of lines, number of nested tiers, and an index into the scope identifier file.

The management of the tier records requires that all nested tiers be known and that the output of the tier records be placed in the same corresponding order as listed in the text file. Some tier attributes are collected only when the tier is closed, such as the number of nested tiers or the number of coded lines that define the structure. Also, none of the nested tiers can be written to the file until the outermost tier is closed. The organization of the tier structure file requires recording the tiers in the order they appear in the text with respect to the tier's first line of code.

A dynamic binary tree structure is utilized to manage the tier records. The structure can maintain all nested tiers until the outermost tier is closed while managing the sequential and nested characteristics among the tiers. The structure manages the outermost tier and all of its nested tiers during the parsing process. When the outer tier is closed, the tier and all of its nested tiers are written to the tier structure file in the correct order.

The initial data structure to manage nested and sequential tiers was defined as a multinode tree. Each level of the tree represented the tier's nesting level where the children of a given node represented all the structures nested within the tier. The multinode tree was transformed to a binary tree for easier implementation and manipulation. The nested links of the multi-link structure were converted to the binary tree's left link and the levels of the multi-link structure were converted to a right link. For example, if several tiers are sequential, they are linked via the right link in the order defined by the program. If a parent tier has nested tiers, then the first nested tier is linked to the left of the given parent tier. All remaining nested tiers are linked to the first nested tier, either as sequential or nested. The subtree defined by the left link of a node contains all the nested tiers of that node. The subtree defined to the right contains all the tiers that sequentially follow the parent tier.

Figure 5.5 illustrates the data structure used by the parse module to manage the tiers. The tree's left link represents the tiers nested in the father node and the tier's right link represents the next sequential tier defined at the same nesting level. The binary structure allows each tier to be defined as either an immediate nested or sequential tier relative to its parent tier. The binary tree also defines the nesting levels in which each tier is defined within the program's text. Figure 5.5 depicts a program with its control structures represented as blocks and its corresponding tree representation. The numbers within the nodes and the blocks indicate the nesting level of each structure.



Tier Management Data Structure
Figure 5.5

As indicated earlier, a tier cannot be written out until it is closed. Relating this to the tree structure, the tier nodes can only be written when the root is closed. The root represents the outermost control structure being parsed. Therefore, we must define what constitutes a root node. The main program's highest control structure, with respect to nesting, is defined at nesting level zero. The global procedures and functions are also defined at nesting level zero. When the parser detects a new tier, a new node is added to the tier tree structure either as a nested or as a sequential tier. If the tier is defined at nesting level zero, then the root of the tier tree is established. When the parser encounters the end of a tier, that tier is closed. If the closed tier is defined at level zero or the root node, then the parser outputs the nodes to the tier structure file in the proper order. The binary tree is traversed in preorder, correlating to the order the control structures are defined by the

program text. Once the tiers are recorded, the tree is deleted and a new root is defined at nesting level zero.

This approach to managing the tiers prevents the tree from getting any larger than the largest procedure, function or the largest top level control structure in the main body of the program. The main program tier is a special tier and is managed unlike the other control structures. The information pertaining to the main program tier is collected throughout the transformation process. Once the last statement of the main program is parsed, the main body tier is written to the tier structure file as the last tier in the file.

The steps for defining a new tier and for closing a tier are shown in Figures 5.6 and 5.7. The steps that reference links are interpreted as addresses of the records within the related file.

New Tier Steps

- Parser detects beginning of new tier.
- Increment nesting level count.
  If new tier type is procedure/function,
  Increment scope level.
- Add new tier node to tree as either nested or sequential
- Record: nesting level, indexes to source file, outer tier link.
  If new tier is procedure/function reference tier,
  Establish link to procedure/function declaration tier.
  If new tier is procedure/function body,
  Establish link from procedure/function declaration tier to new tier.
- Stack pointer to node for tier closing routine. (prevents traversing tree)

New-Tier Algorithm
Figure 5.6

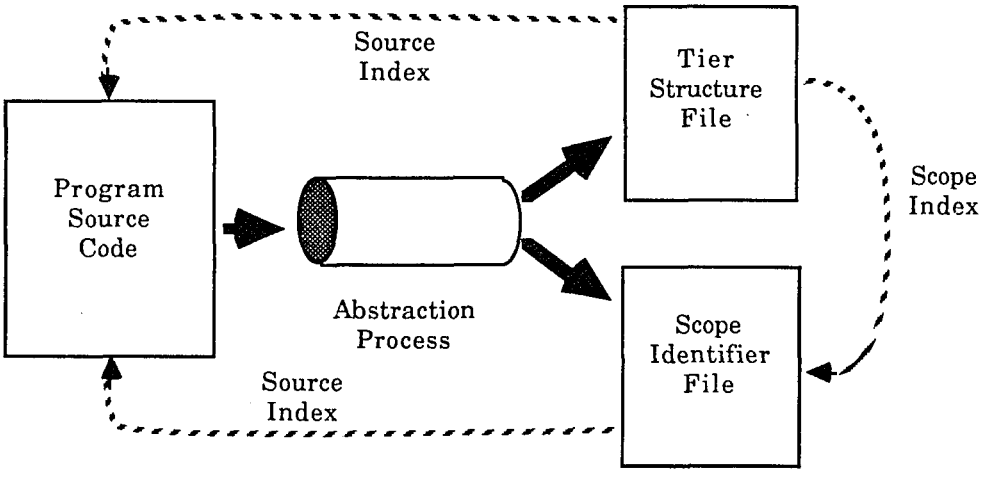
Close Tier Steps

- Parser detects end of tier
- Find tier to close in tree through stack containing tree indexes.
- Record: number of lines in tier, number of nested tiers (traverse up tree), index to scope identifier file, close tier.
- Decrement nesting level counter.
  If tier type is procedure/function,
  Decrement scope level.
- If tier is root of tree,
  Print out tiers in tree in preorder.
- If tier closed in main body,
  Update main body tier attributes, output to tier structure file.

Closing-Tier Algorithm
Figure 5.7

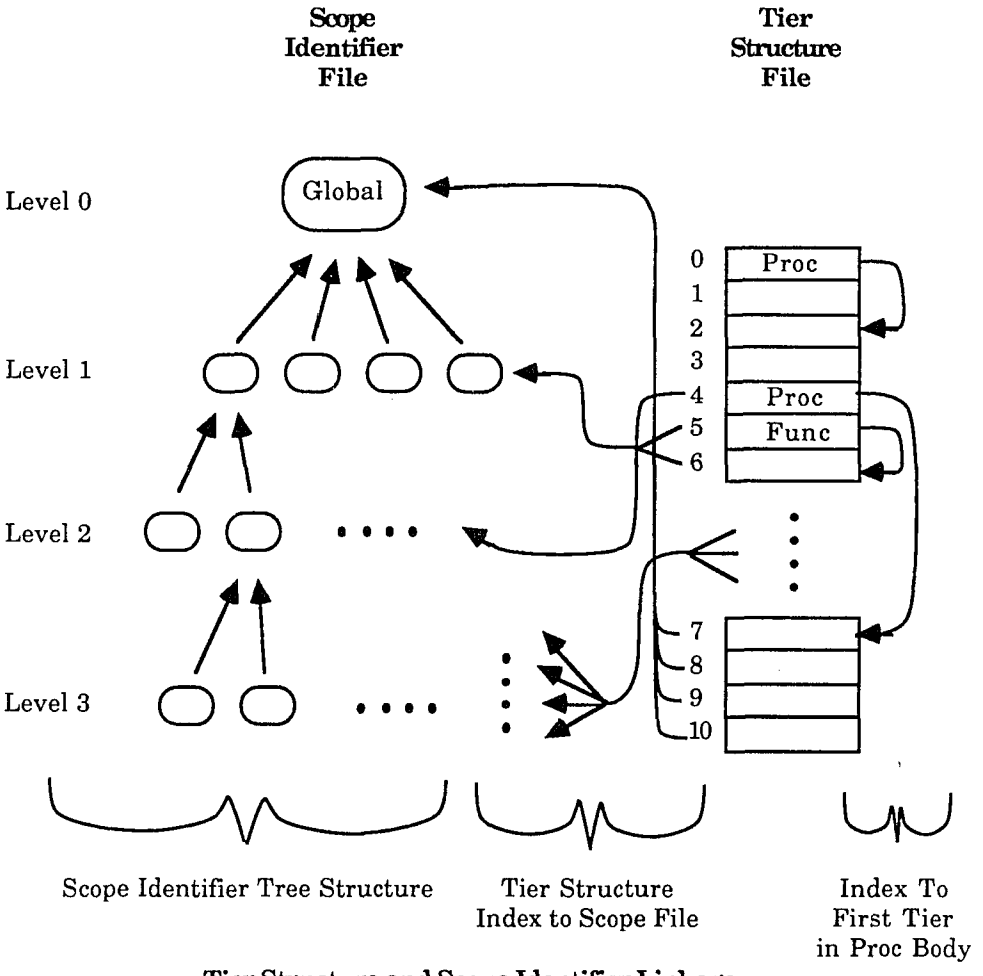
Overview of the Abstraction System

An overview of the abstraction process is shown in Figure 5.8. The transformation system inputs a program in text form and produces as output a tier structure file and a scope identifier file. Indices are used in the tier file to connect each tier record with the actual source statements and with the scoping information.



Transformation System
Figure 5.8

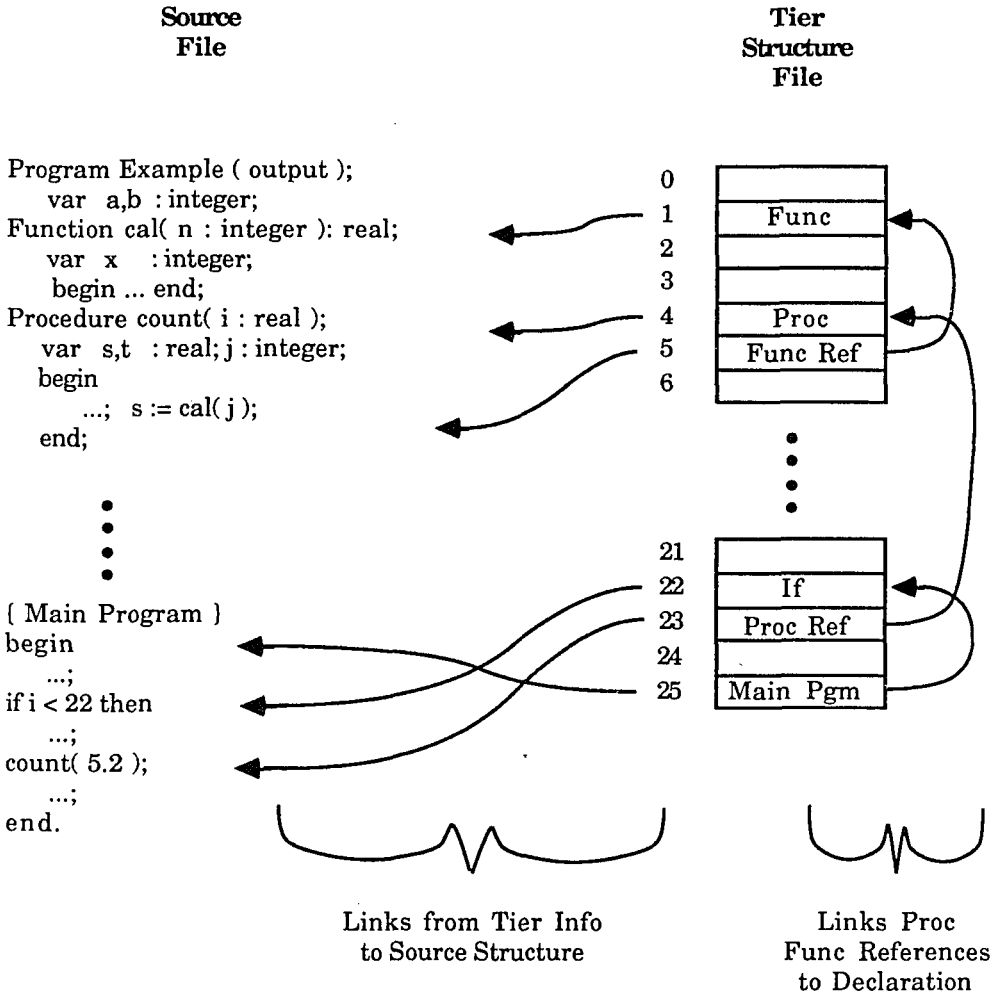
Figure 5.9 shows the scope identifier file and the tier structure file with their appropriate indexes. The scope identifier file's underlying data structure is an upward linked tree. The records in the tier file are linked to the scope identifier records, shown as nodes, establishing the tier's accessible identifiers. The records in the tier file identified as "Proc" or "Func" represent procedure and function declaration tiers. The internal link from these tiers connects the declaration tier to the first tier of the routine's body. There may be numerous intermediate tiers between the declaration tier and the routine's body definition.



Tier Structure and Scope Identifier Linkage
Figure 5.9

Figure 5.10 shows additional internal links and linkage to the source file for the tier structure file. The source index links each tier in the tier file to its textual definition in the source file. Also shown is the linkage between the procedure and function reference tiers and the procedure and function declaration tiers. Finally, the main program tier, placed at the end of the file, is linked to the first nested tier defined in the main program's body.

In the example, the main program tier is linked to an alternate tier represented as an IF statement.



Tier Structure File with Source Links and Proc/Func Links
Figure 5.10

An example Pascal program and the abstracted information are presented in Figures 5.11 through 5.13. Figure 5.11 is the source code for a Pascal program which inputs a list of

numbers and builds a binary tree. The "BuildTree" routine is stubbed out. The tree is traversed and a total is accumulated and printed out. The number on the left side of the Pascal statements identifies the control structures or tiers recognized by the transformation process. This number is associated with the record number in the tier file. The tier structure file generated is shown in table form in Figure 5.12. Each row in the table represents one tier and its associated attributes. The attributes and the block type descriptions are abbreviated.

Figure 5.13 shows, in table format, the records in the scope identifier file. Each row represents an identifier record with its symbol, type, nesting level, source index, and scope index. The scope index is an internal link used to define the underlying scope tree within the scope identifier file.

```
Program TreeTraversal( input, output);
const   Limit = 100;
type    dummy          =    real;
        ElementType    =    integer;
        NodePtr        =    ^Node;
        Node           =    record
                            LeftLink:     NodePtr;
                            Value:        ElementType;
                            RightLink:    NodePtr;
                            end;
var     element,Accumlate:  ElememtType;
        i,Cnt:              integer;
        TreeHead:           NodePtr;

0       procedure BuildTree( vaule : ElementType);
            begin { code to build tree } end;
1       procedure InOrder( Node : Nodeptr);
        var  temp : integer;
2           function Visit ( Nodevalue : ElementType ) : integer;
                begin
3                   writeln( Nodevalue );
                    Visit := Nodevalue;
                end;
4,5         begin   if Node^.LeftLink <> nil then
6               InOrder( Node^.LeftLink );
7               Accumlate := Accumlate + Visit( Node^.Value );
8,9             if Node^.RightLink <> nil then
10                  InOrder( Node^.RightLink );
            end; { end of Inorder }
11      function Average( Val : ElementType; Cnt : integer ) : real;
        var  hold : real;
            begin   hold := Val / Cnt;
                    Average := hold;
            end;
21      { Main Program }
        begin
12          writeln(' Enter data values to build a tree data structure');
13          for i := 1 to Limit do
                begin
14                  readln( element );
15                  BuildTree( element );
                end;
16          writeln(' Traversing tree in order');
17          InOrder( TreeHead );
18          writeln(' Total of all values in tree', Accumlate );
19,20       writeln(' Average of values in tree',Average(Accumlate,Cnt));
        end.
```

Example Pascal Program
Figure 5.11

Blk#	Nest Level	Type	Dcl Line#	Body Line#	Line Count	Nested Tiers	Scope Index	Outer Blk Ndx	Refer Index
0	1	Proc	16	17	2	0	728	-	-
1	1	Proc	18	25	13	7	936	-	-
2	2	Func	20	21	5	1	988	1	-
3	3	BPrf	22	-	1	0	988	2	-
4	2	Alt	25	-	2	2	936	1	-
5	3	Opt	25	-	2	1	936	4	-
6	4	Prf	26	-	1	0	936	5	1
7	2	Frf	27	-	1	0	936	1	2
8	2	Alt	28	-	2	2	936	1	-
9	3	Opt	28	-	2	1	936	8	-
10	4	Prf	29	-	1	0	936	9	1
11	1	Func	31	33	5	0	1196	-	-
12	1	BPrf	38	-	1	0	1040	-	-
13	1	Loop	39	-	5	2	1040	-	-
14	2	BPrf	41	-	1	0	1040	13	-
15	2	Prf	42	-	1	0	1040	13	0
16	1	BPrf	44	-	1	0	1040	-	-
17	1	Prf	45	-	1	0	1040	-	1
18	1	BPrf	46	-	1	0	1040	-	-
19	1	BPrf	47	-	1	1	1040	-	-
20	2	Frf	47	-	1	0	1040	19	-
21	0	Pgm	37	37	12	9	1040	-	-

Tier Structure File
Figure 5.12

Rec#	Symbol Name	Base Type	Nesting Level	Source Index	Scope Index
0	Limit	integer	0	3	0
1	dummy	real	0	5	0
2	ElementTre	integer	0	6	1
3	NodePtr	pointer	0	7	2
4	Node	record	0	8	3
5	LeftLink	pointer	0	8	4
6	Value	integer	0	9	5
7	RightLink	pointer	0	10	6
8	element	integer	0	12	7
9	Accumlate	integer	0	12	8
10	i	integer	0	13	9
11	Cnt	integer	0	13	10
12	TreeHead	pointer	0	14	11
13	BuildTree	Proc	0	16	12
14	value	integer	1	16	13
15	InOrder	Proc	0	18	1
16	Node	pointer	1	18	15
17	temp	integer	1	19	16
18	Visit	Func(I)	1	20	17
19	Nodevalue	integer	2	20	18
20	Average	Func(R)	0	31	15
21	Val	integer	1	31	20
22	Cnt	integer	1	31	21
23	hold	real	1	32	22

Scope Identifier Records
Figure 5.13

Summary

The Peec-Pascal compiler is used to abstract from a source file the tiers, tier attributes, the identifiers, and the identifier attributes. The tier records are linked with the scope identifier file and the source file. They are used as inputs into the Peec system. With this information, the Peec system generates a graphical image of the program's operational

flow and provides access to text definition and to appropriate identifiers. In Chapter VI we describe the display generator for the Peec environment.

CHAPTER VI

MODEL GENERATION

Introduction

In the fourth phase of this research we generate a graphical representation of the program's operational flow. Two aspects of the model design are of the utmost importance. The first is the design of a graphical representation of an abstract concept, the program's execution model. We have defined such a three-dimensional graphical model in Chapter IV as control structures arranged in three-dimensional space.

The second aspect of the design is the graphical representation that relates to the elements or building blocks used to construct the flow model. The elements should be designed to enhance the human imagery system. These images should represent a clear and unambiguous representation; therefore, the design of these images is important. Studies (Rohr, 1986; Korfhage & Korfhage, 1986; Lodding, 1982; Montalvo, 1986; Carroll & Thomas, 1982) indicate that limits exist on the amount of information that can be represented within a single image. If an image is too complex, the user is not as productive. Rohr's (1986) studies indicate that if an image is too complex, the image should be decomposable to its smallest elements in terms of known meanings. The user can then recompose these elements and regain the original meaning of the complex image. This concept is the central idea behind the operational flow model and the interactive environment supported by the Peec system.

Flow Model Elements

In this section, we discuss the general features the Peec system provides the user. A description of the model and the modeling elements are present first, followed by the functionality features of the Peec environment. The data structures and algorithms that support these features are then presented.

The Peec system represents the program's execution flow as a three-dimensional image. There are two reasons for choosing a three-dimensional image in representing an abstract concept of the program's operational flow. First, a three-dimensional image gives the user more information in a shorter amount of time, plus, the computer generated image can manage more details than may be possible with the human's imagery system (Rohr, 1986; Weber &Kosslyn, 1986). The second reason is the three-dimensional image supports the human's imagery system in the program understanding process (Glinert & Tanimoto, 1984; Cooper & Shepard, 1984).

Tier Image Definition

The control structures are the building blocks used to construct the flow model. Earlier, we defined a tier as a single control structure. We extend this definition to define the physical representation of a tier. The physical representation associated with a tier is a wire frame box, positioned in the three-dimensional space. The image representing a control structure conveys to the programmer its type, relative size, and its relationship with its surrounding structures. The arrangement of the tiers within the model defines the tier's relationship with other structures. An icon, which identifies the control structure's type, is associated with each tier. The icon denotes the tier type as either an iterative, alternate, optional, procedure, function, main program, or a reference control structure. The programmer associates with the tier image the type of control structure being represented based on the tier's icon.

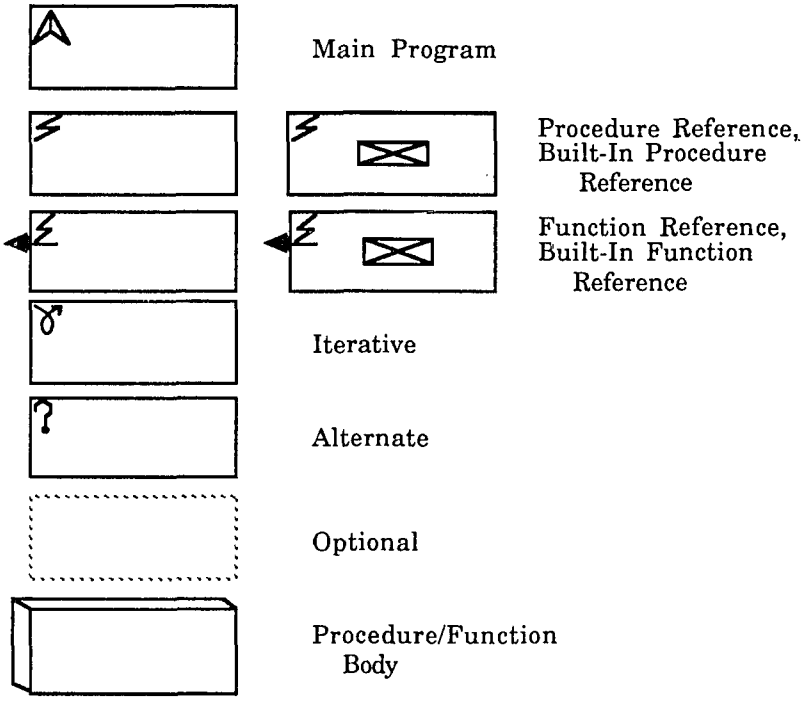
Figure 6.1 depicts the tiers, denoted as boxes, and the identifying icons. An explanation and restriction on the design of the icons are given. The restriction on the icon design requires it to be simple and represent one entity (Korfhage & Korfhage, 1986; Rohr, 1986). A simple icon allows the user to have an unambiguous understanding of its meaning. The icon is described graphically by relatively few vectors in order to reduce the time required to generate the image on the display. The first icon, an arrow head pointing upward, identifies the main program. It represents the main program body defined at the top level or the peak of the program's operational flow model.

The next icon is used for procedure and function references, for both user-defined and built-in routines. The icon used for a procedure or function reference is a communication link symbol. We think of a referenced statement as a call to a procedure or function and that control is transferred from the reference point to the procedure or function body. The procedure or function reference statement calls a routine and transfers control to its body. The left arrow is added to the function reference, indicating a value is returned through the function name. The arrow is the distinction made between a procedure and a function reference. The boxes with an X inside represent a reference to a built-in procedure or function. The X-boxed icon indicates that either the procedure or function cannot be viewed from the Peec environment or that the code is not available for inspection.

The icon for a loop is readily understood. Its design suggest its implications. The alternate icon is used for such control structures as the IF and CASE statements. The question mark icon indicates that one of the possible optional blocks is selected for execution. The optional block icon, nested in alternate tiers, is a modified box represented as dashed lines. The incomplete box indicates its execution is based on the results of the alternate statement at execution time and therefore may not be executed in some cases. The last icon represents the body of a procedure or function. The tier is modified to

represent a box with depth. This icon represents the body of a routine which has a broader meaning than the simpler structures, with respect to program understanding.

Tier and Icon Definitions



Tier and Associated Icons
Figure 6.1

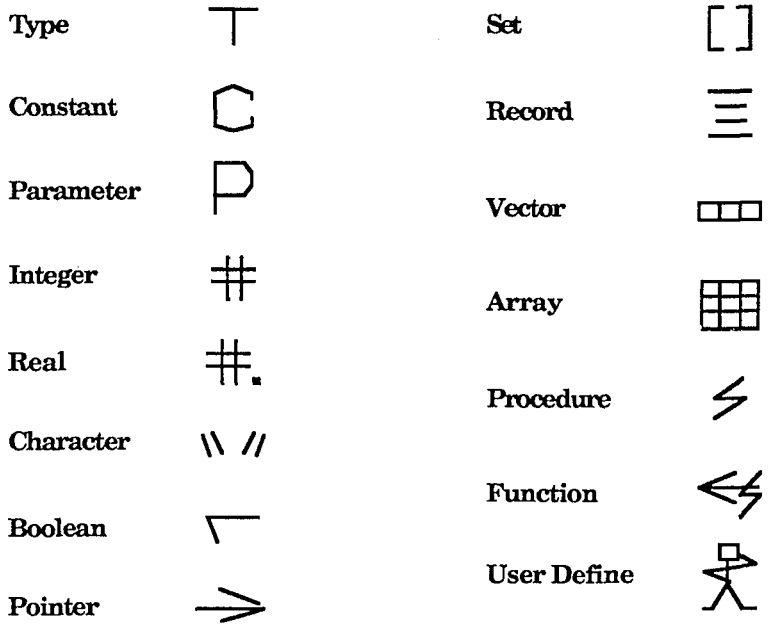
Another characteristic of the tier image is its height. The tier's height is relative to the number of text lines defining the control structure. Also, the spacing above and below each tier within the model is relative to the number of text lines defined outside each control structure. In one case, the height is out of proportion from one nesting level to the next.

This case occurs in a reference tier. The procedure or function body is represented as a nested tier, nested within the referenced tier. The body of the tier is scaled to fit as a nested structure within the referenced tier. The tiers within the body of the routine are scaled proportionally to the tier's body as defined above.

Text Display

There are two additional features that are useful to the user. One is the option to display the source statements that define a tier. If the user needs to study a control structure in more detail, he can display the text that defines the tier image. The second option is the displaying of the set of identifiers that are accessible from the current browse point. The user can optionally display the identifiers as he browses from point to point. The list represents the set of identifiers in scope of the current tier. Each identifier's name and type is displayed. Data type icons describing the identifier basic type and structure are displayed beside each identifier name. While the programmer browses the tiers, the identifier list is updated. As the user's browsing carries him into procedures or functions, the scope of the identifiers can change. These changes are reflected automatically in the identifier list.

The data type icons are shown in Figure 6.2. Each icon or a combination of these icons, defines the identifier's basic type and a description of its data structure. First, we discuss the icon representation and its intended meaning. The description of an identifier's structure is show in Chapter VII. The first three icons, representing identifiers declared as TYPE, CONST, or Parameters, are denoted with the symbols T, C, and P, respectively. The next two icons are for identifiers with base type INTEGER or REAL. The number sign is used for the integer icon and a number sign with a decimal point is used for a real number.



Data Type Icons
Figure 6.2

The character icon is a set of double quotes indicating a character type identifier. The boolean icon is a NOT operator, found in the set of boolean operators. The pointer type uses a right arrow. It indicates the identifiers indirect addressing and points to its value through the address in the identifier. An identifier's base type value is the type of value it points to. Therefore, the description of a pointer identifier is represented by two icons, a pointer followed by its base type icon. An example of this representation is shown in Chapter VII. The set icon consists of open and close brackets. An identifier declared as a set also uses two icons in its description; one icon indicating it is a set followed by another icon indicating the type of values within the set.

A record icon uses multiple line segments of different lengths. It relates to the multiple fields defined in a record structure. The vector icon represents a sequence of contiguous location denoted by the ordered squares. The vector is an array with one dimension. The array icon represents an array with multiple dimensions. Both the vector and array icons use an additional icon to describe the basic element type or the type of values stored in the structure.

A procedure identifier icon is defined even though the procedure identifier can neither be assigned a value nor represent any data structure. The user has the capability to display the list of defined procedures if he wishes, or he can verify if a particular identifier is a procedure or a function. The icon used to represent procedure identifiers is the communication symbol. This symbol was selected because the same symbol is used in a reference tier and it matches well with its representation. The function icon, a communication symbol with an arrow, was selected based on the same argument.

The last symbol defined is for identifiers of a user-defined type. Since the programmer defines new constants and is responsible for their manipulation, we define a stick man icon representation.

The identifier display feature is useful during the browse mode. The identifiers can be displayed to give the user two types of information, the set of variables within scope and a quick reference to the identifier's type and structure. As the user continues to browse from one tier to another, the identifier list is updated when the scope changes.

In the following section, we discuss the concepts and supporting data structures used to generate and manage the execution flow model within the Peec environment.

Data Structures

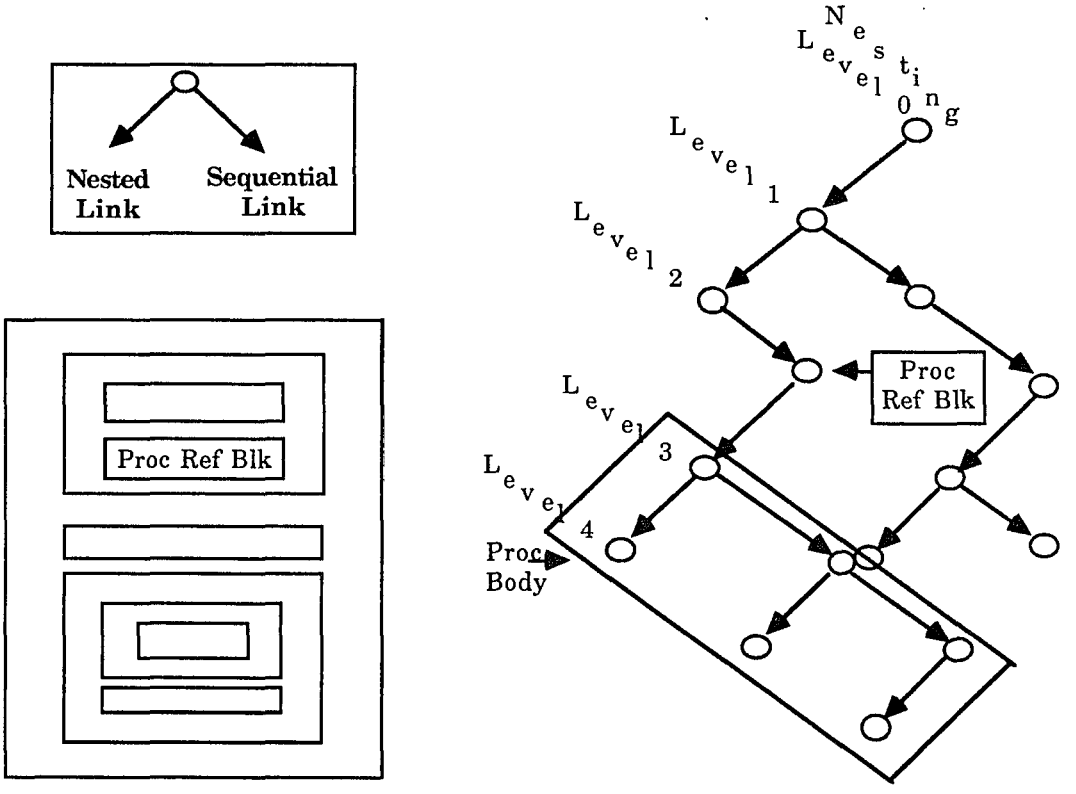
The program's three-dimensional execution model reflects the program's execution order. The model initially displayed represents the main program. As the user browses to the outer limits of the flow model, Peec adds additional tier images local to the browse point. For example, if the programmer browses into a nested tier, Peec will generate the additional tiers. Additional nested tiers are generated and displayed as the browse point moves farther down the nesting axis. As the programmer moves out of nested tiers, they are deleted from the display. The generation and deletion of tiers allows the programmer to browse only the area that interests him and the deletion of the tiers prevents the model from becoming too cluttered. In the following sections, we present the data structures and algorithms used to describe and manage the flow model.

Tier Display Structure

The Peec system uses a binary tree as its supporting data structure for managing the program's control flow model. It is similar to the data structure used to manage the tiers in the transformation process. This dynamic structure allows addition and deletion of tiers based on the browsing requirements. Each node in the structure represents a control structure within the program and a tier image within the flow model. The image generator uses the information in each node to construct a tier image for the flow model. We will refer to this data structure as the display tree since each node contains the attributes describing each tier of the model.

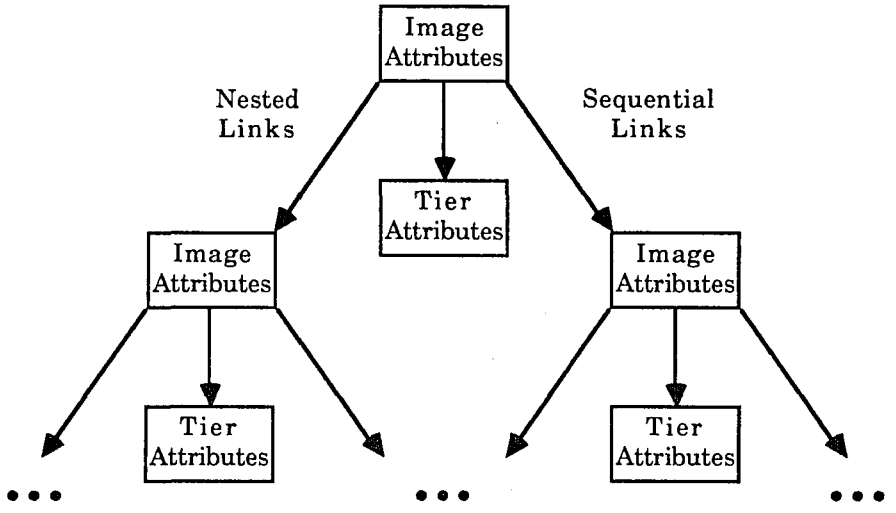
The nodes are arranged as either nested or sequential. The left child represents a nested tiers where all nodes in the left subtree are interpreted as nested tiers within the parent node. The right child represents sequential structures following the parent node. All nodes along a given sequential link are control structures defined at the same nesting level and in the same order as defined in the program. A reduced version of a display tree

and the associated program, illustrated in block format, are shown in Figure 6.3. The root node represents the main program tier defined at nesting level zero. The procedure reference node has been expanded by the procedure's body as it would look when the programmer browses into the reference tier. The procedure's tiers are shown in the box outlining a subset of the nodes defining the procedure's body. The procedure's declaration is not shown in the block represented format, only the main program.



Display Tree with Procedure Reference
Figure 6.3

Each node in the display tree has two sets of attributes associated with it. The first set is the transformation attributes discussed in Chapter V. The second set is defined at the time the tier node is created in the display tree. These attributes describe the graphical representation of the tier and additional information used in managing the browse feature. We will refer to the first set of attributes as the tier attributes and the second set as the image attributes. Figure 6.4 shows the data structure organization representing a display tree node. The tier and image attributes are separate structures for design and management convenience.



Display Tree Node
Figure 6.4

The values of the image attribute are used by the display system to define the tier's image. These attributes define the tier's size, position, solid or dotted line, and visibility. A modeling matrix is used to define the tier size and position. The image attributes,

including the modeling matrix attribute, are defined at the time a tier is added to the display tree. The modeling matrix is discussed later in this section. The display tree is the major data structure used to define the flow model and to manage the model during the interactive phase.

Display Tree Construction

When the Peec system is initiated, it builds a display tree for the main program tiers only. These tiers include the main program and all the nested tiers which define its body. This process uses the information from the tier structure file produced by the transformation phase. The last record in this file contains the description of the main program tier and a link to its nested tiers. The display tree is constructed by reading in the main program tier first, which becomes the root of the display tree, then reading all remaining tiers defined within the main program's body. The resulting initialized display tree defines the flow organization of the main program. Figure 6.5 depicts the algorithm used in initializing the display tree with the main program. A dummy root node is created above the main program node within the display tree. This dummy node was added as a convenience for implementing the different algorithms which manage the display tree. Many of the algorithms require information from the previous node and the dummy root node allows streamlining of the implementation code.

```
• Open Tier Structure Data File
• Initialize dummy root in display tree  { image attributes }
        identity modeling matrix, scale factor (1.0), nest level (-1)
• Position file pointer to last tier structure record
• Read Main Pgm Tier
• Link Main Pgm Tier to dummy root
• Position tier file pointer to first main program tier in structure file
• Allocate display tree node
• Read tier record into tier node
• LOOP:  While tier != Main Pgm Tier
•               If tier nested
•                       Increment nesting level {image attribute}
•                       Carry down previous tier scale factor
•                       Update parents nest link and new node's father link
•                       Save pointer address of leaf at current nesting level
•               If tier sequential
•                       Find last leaf at same tier level as current input node
•                       Carry down previous tier scale factor and nesting level
•                       Update parents sequential link and new node's father link
•                       Save pointer address of leaf at current nesting level
•               If tier = Optional Tier
•                       Update optional tier counts with number of optional tiers
                                in current Alternate Structure {image attribute}
•               Allocate tree node and read next tier structure record
• End LOOP:

        { Note: The LOOP is referenced by Update Display Tree algorithm below }
```

Build Display Tree for Main Program
Figure 6.5

Once the display tree is constructed, the remaining tier images attributes are calculated. Each tier's image attribute is dependent on the previous tier's image attribute calculations. Once a display tree node is defined and the image attributes evaluated, then a tier image can be generated when needed.

The model is expanded when the programmer browses into a procedure or function reference tier. If the programmer browses inward, the Peec environment will generate, as needed, the additional tiers defining the body of the routine. Nodes in the display tree are

added when the tiers, defining the body of a procedure or function, do not exist within the tree. The information for the new nodes is obtained from the tier structure file. Often these tiers exist in the display tree but are not visible, therefore the tier images need only be displayed using the information that exists in the display tree. The display tree update algorithm is shown in Figure 6.6.

- Allocate and initialize a display tree node
- Position file pointer in tier structure file to Proc/Func declaration tier record
- Read declaration tier record
  { Proc/Func is always a nested tier so it is added to nest link }
- Increment nesting level { image attribute }
- Reduce scale factor to produce nested tiers in reference tier
- If Proc/Func have nested tiers in body
- Position file pointer to first nested tier
- Execute LOOP -> END LOOP in Main Display Tree Algorithm { Fig 6.5 }
- else { no nested tiers in body of routine, return }

Proc/Func Display Tree Update
Figure 6.6

The updating of the display tree is control indirectly by the programmer's movements while browsing. As the user browses through a procedure reference, the tree's branch is expanded. The tier structure file is a static representation of the program which supplies the data for the display tree. The display tree represents the program's operational flow. Therefore, a tier record from the structure file may be instantiated several times within the flow model depending on the number of references made and whether the user browses into these reference tiers. For example, a recursive procedure is defined once in the tier file, but in the display tree, the routine can be represented numerous times, depending on how deep

the browse point moves through the recursive references. The displayed flow model shows the recursive routine as a nested tier within the reference statement.

Tier Image Generation

Another image attribute is the modeling matrix used to define a tier's image for displaying. The modeling matrix is an image attribute that instantiates a tier image within the flow model. These attributes are initially calculated when the users browses to an edge of the flow model. The number of nested and sequential tiers to be displayed is controlled by the maximum nesting and maximum sequential parameters. Figure 6.7 depicts the algorithm for generating tier images. It traverses through the display tree generating and displaying tiers.

The algorithm is a recursive procedure which traverses the display tree, generates each display tier image and updates image modeling attributes. The input parameters consist of a display tree node, maximum nesting depth, and a maximum sequential depth. If the display node is within these limits, then a modeling matrix is generated in order to display the tier image. A tier block can be defined but not be visible if it is outside the nesting and sequential limits. As the user browses through the model, the tiers come into view.

```
Parameters -  (Display tree node,  Nesting level, Sequential level Counts)
• If node is valid & within nesting and sequential limits & not instantiated
•       If node not instantiated
•               Carry previous modeling matrix down to current node
•               Define modeling matrix
•       Instantiate new tier with appropriate icon identification
• Reduce nesting level count by 1
        Generate Display Tier( Nest link node, Nest-Cnt, Seq-Cnt )
• Reduce sequential level count by 1
•       Generate Display Tier( Sequential link node, Nest-Cnt, Seq-Cnt )
                { Post order traverse of display tree }
```

Generate Tier Algorithm
Figure 6.7

The elements of the operational flow model are made up of tier images or wire frame blocks, positioned in three-dimensional space. The tier's image is defined by a rectangle template and a modeling matrix which define the size of a tier and its position within the model. The rectangle template is a generic definition of a tier image which is instantiated numerous times during the construction of the flow model. There are five modeling parameters used to describe a tier's graphical image. These parameters are the X and Y scale factor and the X, Y, and Z translation parameters.

The rectangle is scaled in the Y direction based on the number of text lines in the source code and on the local scale factor. The local scale factor controls the representational height of a line of code relative to its parent node. The initial local scale factor represents the height for one line of text at the top nesting levels of the flow model. The scale factor is reduced when the body of a procedure or function is nested within the reference tier. The local scale factor and the number of lines defining the tier are used to calculate the height or the Y scale factor for each tier.

The X scale factor is based on nesting depth of the tier's control structure. A nested tier must fit in the bounds of the parent tier. If the relationship between two tiers is sequential, then the width of each tier is the same. To compute the width of a nested tier, we must know the nesting level of the tier and the width of its outer tier defined by the previous node in the display tree. First, we must define what is meant by the nesting level of the tier. There are two different nesting levels defined within the Peec environment. The structure's nesting level, defined by the code, differs from the nesting level of the tier in the model. The program's textual organization defines the nesting levels for each control structure within the text. This level is a static value. The tier's nesting level image attribute defines the nesting level of control structures being represented within the execution flow model.

An example of the level concept is given for clarification. Assume a function is defined in the program at static nesting level zero. In the execution flow model, the function's body is shown at the point of reference, nested within the referenced tier. If the referenced tier is defined at nesting level three, then the body of the function is positioned at nesting level four within the model. If the function is recursively called, each instantiated function is shown on succeeding XY planes along the nesting axis. The tier's nesting level is used to distinguish between the different instantiated functions at the different nesting levels. It is also used to calculate the X scale factor for each tier.

The X scale factor is based on the tier's nesting level and the X scale factor of its previous tier. Each nested tier has a smaller width than the previous. This gives the effect that the nested tiers are behind or farther back on the nesting axis.

The translation parameters position the tier in the three-dimensional space. The Z translation parameter positions the tier in a XY plane along the nesting axis. The only value needed to calculate the Z position is the tier's image nesting level. The greater the nesting level, the further back the tier is placed on the nesting axis.

The X translation parameter is fixed for all tiers except the optional tiers. Most of the tiers generated are centered with respect to the parent tier. The exception is the optional tiers. The optional tiers are positioned in the same XY plane but are positioned horizontally across the screen. Each optional tier is translated to the left or right based on the number of optional tiers to be displayed horizontally. The optional tiers are equally spaced. The X translation parameter is computed from the width of the optional tiers and the spacing between them.

The distance a tier is translated in the Y direction is in relation to the center of the previous tier. The center of the previous tier is the relative origin for translation in the Y direction. The combination of the previous tier's center, the new tier's center and the height of the current line representation determine the Y translation parameter.

The Peec system uses a modeling matrix to define two aspects of a tier block. The first aspect is the X and Y scale factors which give the tier block its height and its width. The second aspect is the three-dimensional translation which positions the tier block in the XYZ space. The matrix is depicted in Figure 6.8. The parameters in the four by four matrix control the scaling and translating characteristics of each tier. The Z scale is not used in the Peec environment, but it is shown in the figure for completeness.

Sx			
	Sy		
		Sz	
Tx	Ty	Tz	

4 x 4 Block Modeling Matrix

Sx - X Scale Tx - X Translate
Sy - Y Scale Ty - Y Translate
Sz - Z Scale Tz - Z Translate

Modeling Matrix Definition
Figure 6.8

The tier's modeling matrix parameters are determined in relation to its parent tier or parent node in the display tree. The current display tree node obtains a copy of its parent node's modeling matrix. Adjustments are made to the appropriate parameters based on the current tier's relative position. Certain parameters are altered and certain parameters are ignored depending on whether the current tier is sequential or nested to its parent. If a scale or translation value is not updated, then the value remains the same as its parent. The algorithm in Figure 6.9 shows the procedure used to compute the values in the modeling matrix.

```
   { Xscale - Sx}
•  If tier is nested
•    Case on Tier-Type
•       Optional Tier : {for first opt. tier only, propagate to others }
•                       Sx <- Sx • Xreduction factor / Number-Optional-Blks
•       Proc/Func Tier : Sx <- 10% reduction
•       Otherwise:       Sx <- Sx • Xreduction factor
•  If sequential tier  No change in Sx

   { Yscale - Sy }          { Scale both x and y }
•  Case on Tier-Type
•       Proc/Func Tier :          Sy <- Nbr-Lines-In-Body • Height Scale factor
•       Otherwise :               Sy <- Nbr-Lines-In-Tier • Height Scale factor

   { Zscale - none }

   { Xtranslate - Xt }      { for optional tiers only }
•  If Optional Tier & first Optional Tier
•       Xt <- position opt tier to left side of Alternate tier
•  else
•       Xt <- Xt + incremental position along x axis

   { Ytranslate - Yt }
•  Case on Tier-Type
•       Proc/Func Tier, Proc/Func Ref Tier, Optional Tier :  { no change }
•       Otherwise :
•        Yt <- Half Nbr of lines in tier • Scale factor • Height Scale factor

   { Ztranslate - Zt }
        { Distance along z axis is proportional to nesting level }
•  Zt <- Zt - Zfactor x Nesting Level
```

Modeling Matrix Algorithm
Figure 6.9

Scope Identifier Structure

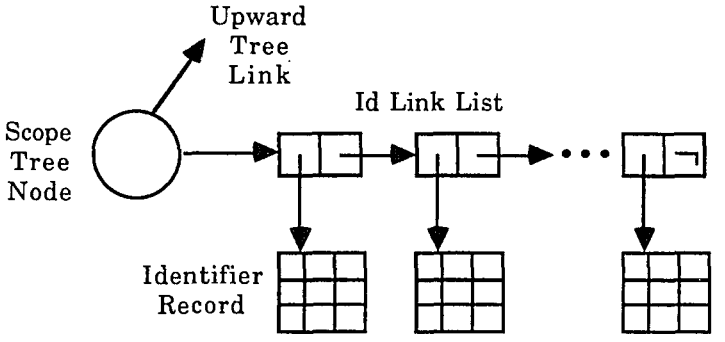
One feature in the Peec system allows the programmer to display live identifiers associated with the current structure. The programmer can optionally select a set of identifiers based on their type and display them as he browses through the model. Not only

can he select the identifiers by type, but he can also select them based on their scope. For example, the user may optionally select to display *integer* and *char* type identifiers which are *local* to the current structure. As the user browses the model, only the local integer and character identifiers which are live or accessible at the current browse point are displayed.

The data structure used to represent the identifier information is a general upward linked multinode tree structure. We will refer to this structure as the scope tree. The scope identifier file supplies the data to construct the scope tree. The scope identifier file is one of two output files generated by the transformation process discussed in Chapter IV. The attributes that describe each identifier are its name, basic type, nesting level definition, index to text file, and an index to previous identifier within the same scope.

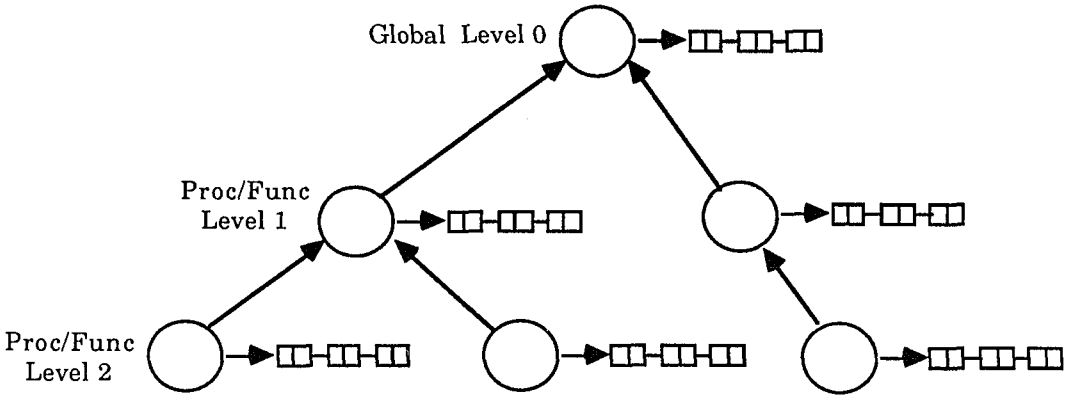
There are two reasons for using the upward linked data structure. First, it gives fast access for displaying the set of identifiers and second, it orders the identifiers in a way that naturally matches the scoping rules of the language. Each tier is indirectly linked to a scope tree node, that is, the tier contains an address to an identifier record in the scope identifier file. When the identifiers are displayed, the link address in the tier attribute is mapped to a node in the scope tree. The path from a given scope tree node to the root defines the variables that are in scope for a given structure.

Each node in the scope tree represents either a procedure, function or the main program. Associated with each node or routine is a list of its local identifiers. The list of identifiers and their attributes are maintained in a link list. The link list allows for varying number of identifiers defined within a routine. Figure 6.10 shows the organization of a specific scope tree node.



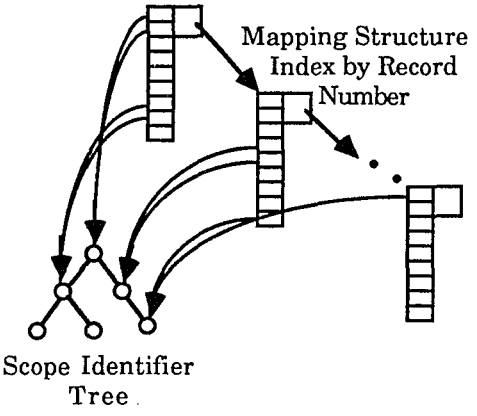
Scope Tree Node Definition
Figure 6.10

The individual nodes representing the functions and procedures are arranged in a hierarchical order based on their textual definition. The root of the tree defines the global identifiers or the set of level zero identifiers. Procedures and functions at level one are the globally defined routines. The nodes at level two are procedures and functions nested within level one routines. Each node in the tree is linked to its parent routine, or each routine is linked to the routine it is nested within. Figure 6.11 shows the scope tree and its levels.



Scope Identifier Structure
Figure 6.11

The last structure associated with the scope identifier tree is its mapping structure. Each tier node in the display tree is linked to a record in the scope identifier file. The link is a record address within the scope file. The record address is mapped to a node in the scope tree through an address-to-node mapping structure. The mapping data structure is a link list of arrays where each array index corresponds to a record in the scope identifier file. The value within the array is a pointer to a scope tree node. The tier's scope address is mapped to the mapping data structure. The node pointer is retrieved and the link to the scope tree is established. The set of identifiers in scope of the current node is accessible. Figure 6.12 represents the mapping structure and some represented links to the scope tree.



Scope Mapping Structure
Figure 6.12

The algorithm in Figure 6.13 describes the procedure used to compute the link between the tier's scope attribute and the node in the scope identifier tree.

{ Compute the node or array in the mapping structure }

- ST <- Divide the record address of the scope id file with the array size of the node
- Traverse to the ST^{th} node in the mapping structure
- AI <- Mod the record address with array size
- ScopeNodePtr <- Array[AI] of mapping structure

Mapping Tier's Scope Address to Scope Tree Node
Figure 6.13

The scope tree and the mapping structure are constructed from the data in the scope identifier file. Both structures are defined at the time the user requests a display of identifiers. They are created only once for a given session with the Peec environment and only when the user requests the identifiers. The scope input file has most of the identifiers organized or grouped correctly from the transformation process. The one exception is identifiers which are procedure and function names as discussed in Chapter IV. They are not physically grouped in the identifier file based on the scoping rules, instead are scattered throughout the file and linked appropriately.

When the scope identifier tree is constructed, the procedure and function identifiers are added to the correct scope tree node. The algorithm to build the scope tree is shown in Figure 6.14. The process looks for changes in the scope level value. If the scope level decreases, then a search up the scope tree to the correct node is performed. The new identifier is added to the identifier link list of the current node. If the scope level increases, then a new node at the next level of nesting is defined and an identifier link list is started.

```
• Allocate Scope Head, Identifier Node, Mapping Node
• Update linkage between new nodes
• Read first identifier record
• Update tree path { used to move back up scope tree }
• Update mapping node { record one is mapped to current scope node }
  LOOP:   While identifier records to read
•     Case  Compare current scope level to previous
•           Equal :
                Link new id rec to id link list on current scope node
•           Less Than :       { new deeper scope level }
•               Allocate new scope tree node
•               Link to higher level scope tree node { back link }
•               Update tree path
•           Greater Than :    { back up in scope tree to correct level }
•               Move up scope tree until equal scoping level
•               Add new node to end of identifier link list
•     Update mapping node
•     If mapping node array full
•           Allocate new mapping node
•           Link to previous mapping node
•           Set array index to zero
  END LOOP
```

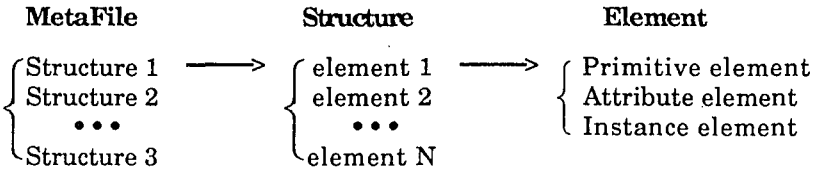
Scope Identifier Tree Construction
Figure 6.14

Graphics Package Structures

The Peec system interfaces with the Apollo's 3-D Graphic MetaFile Resource, GMR. The GMR system is based on the Core graphics standards. The GMR library is made up of a set of functions where the user interfaces with the library through the C language. The graphics system uses the metafile concept, that is, the graphics programmer does not manipulate images on the screen directly, but indirectly. The GMR system creates the image on the display by reading the graphical descriptions from a metafile. The metafile contains the data that describes the images produced on the screen. The graphics programmer generates images by writing a description of the structures to the graphics

metafile. A structure is defined by the data points, its modeling matrix, and associated graphics attributes. Some of these graphics attributes are line type, intensity, and whether or not the image is viewable.

The GMR's display generator processes the metafile by accessing each structure and its attributes to produce images on the display device. As the metafile is altered, the changes are reflected on the screen. The metafile hierarchical organization is shown in Figure 6.15. A MetaFile contains structure descriptions where each structure defines an image on the screen. Each structure is defined by a set of elements which describes the individual parts of the image. The elements can be graphic attributes or the instantiation of other structures (Apollo, 1985a & Apollo, 1985b). Each structure and element can be accessed and modified, thereby affecting the displayed image.



GMR Metafile Data Structure
Figure 6.15

The graphical image can be modified by first referencing the structure identification, indexed by element identification, then updated appropriately. The programmer can alter individual elements, groups of elements, or the entire display window in this fashion. Additional attributes are available which are associated with the entire image or the

display window. With these attributes, the graphics programmer can define and modify the view of the image, giving the user various views of the displayed images.

The GMR package has an automatic viewing feature available. The package supplies the graphics programmer a set of viewing parameters which when altered generates different perspectives of the image. In the design of the Peec system, we elected to control the image through a global modeling matrix managed by the Peec environment. We elected to manipulate the object rather than the viewpoint. The global modeling matrix is controlled by the Peec system which causes the flow model to be scaled, rotated, and translated as needed.

Another reason for this design decision was due to GMR's handling of text in three-dimensional space. Peec's processing and displaying of text within the displayed image did not mesh well with the automatic viewing feature. The scale and position of the text had to be adjusted for each view of the model. The global modeling matrix is maintained in the root node of the display tree. Any changes made to the global matrix is reflected in the displayed flow model.

The display tree, scope tree and GMR's metafile are the major data structures which support the Peec environment. The scope tree is a static data structure initially defined by the scoping information at transformation time. The set of identifiers accessed from the scope tree vary depending on the position of the browse point. The display tree and metafile are dynamic structures that are constantly changing based on the interaction between the programmer and the Peec environment. In Chapter VII, we present the Peec system operations which describe the data structures and interactive algorithms.

CHAPTER VII

PEEC INTERACTIVE FEATURES

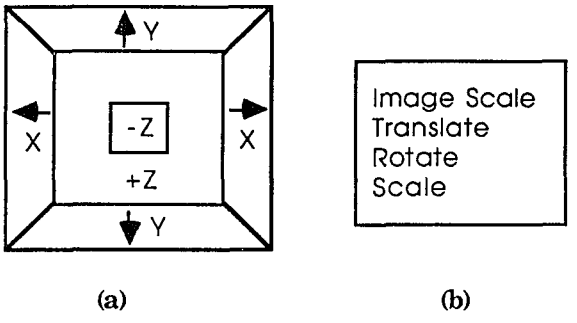
Introduction

In this Chapter, we describe the interactive features of the Peec environment. The interaction between the programmer and the Peec system is controlled through predefined icons and a pointer. Peec interprets the input and updates the model immediately. In the following section, we describe these icons, their interpretation and the functionality within the Peec environment. We provide examples to show the implications of each icon's action. The three Pascal programs supporting the example flow models are shown in Appendix A. The code for Program A represents a threaded binary tree with two traverse routines. One traverse routine is recursive and one is nonrecursive (Tenenbaum & Augenstein, 1986). Programs B and C are nonfunctional code for representing basic concepts of the Peec system. The illustrated operational flow models reference the appropriate program.

Flow Model Commands

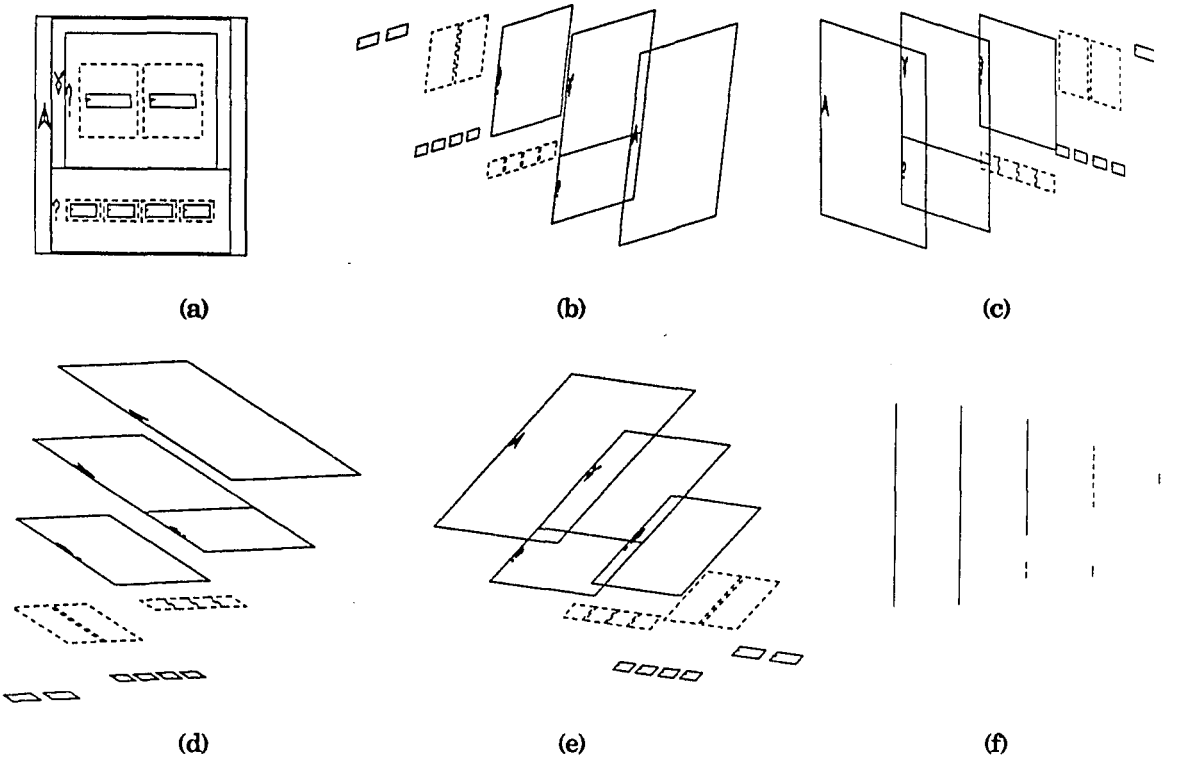
The first set of interactive commands is used to manipulate the flow model's image. Figure 7.1(b) shows the commands the user can select using the pointer device. When a command is selected, it activates the icon shown in Figure 7.1(a). The program's flow model is defined as a three-dimensional image with length, width and depth. The model can be rotated around any of the three axes giving the user different perspectives of the program's structures and their relationships. The user can rotate the image in the positive or negative direction in either the X, Y, or Z axes. The icon in Figure 7.1(a) is activated when the rotate command is selected. This icon is a multiple function icon used with a

number of other Peec commands. We will refer to the icon as the three-directional icon because it manipulates the model in the three-dimensional space. For the rotation command, the multiple sectioned box represents the six different directions of rotation. The positive and negative direction for X are defined on the left and right sides of the icon. The top and bottom sections represent the positive and negative directions for Y. The two center boxes are used for Z movement. The outer box is for rotation in the positive Z direction and the smaller box is for negative Z direction. The smaller box represents smaller Z values along the nesting axis or farther back into the screen.



Three-Directional Icon and Command Selections
Figure 7.1

In Figure 7.2, Program A's operational flow model is shown from different perspectives. Each of the examples represent the results of rotating the model on one or more axes. Figure 7.2(a) represents the program sighted down the nesting axis. The other figures represent additional views. Figure 7.2(f) is a view from the side of the flow model.



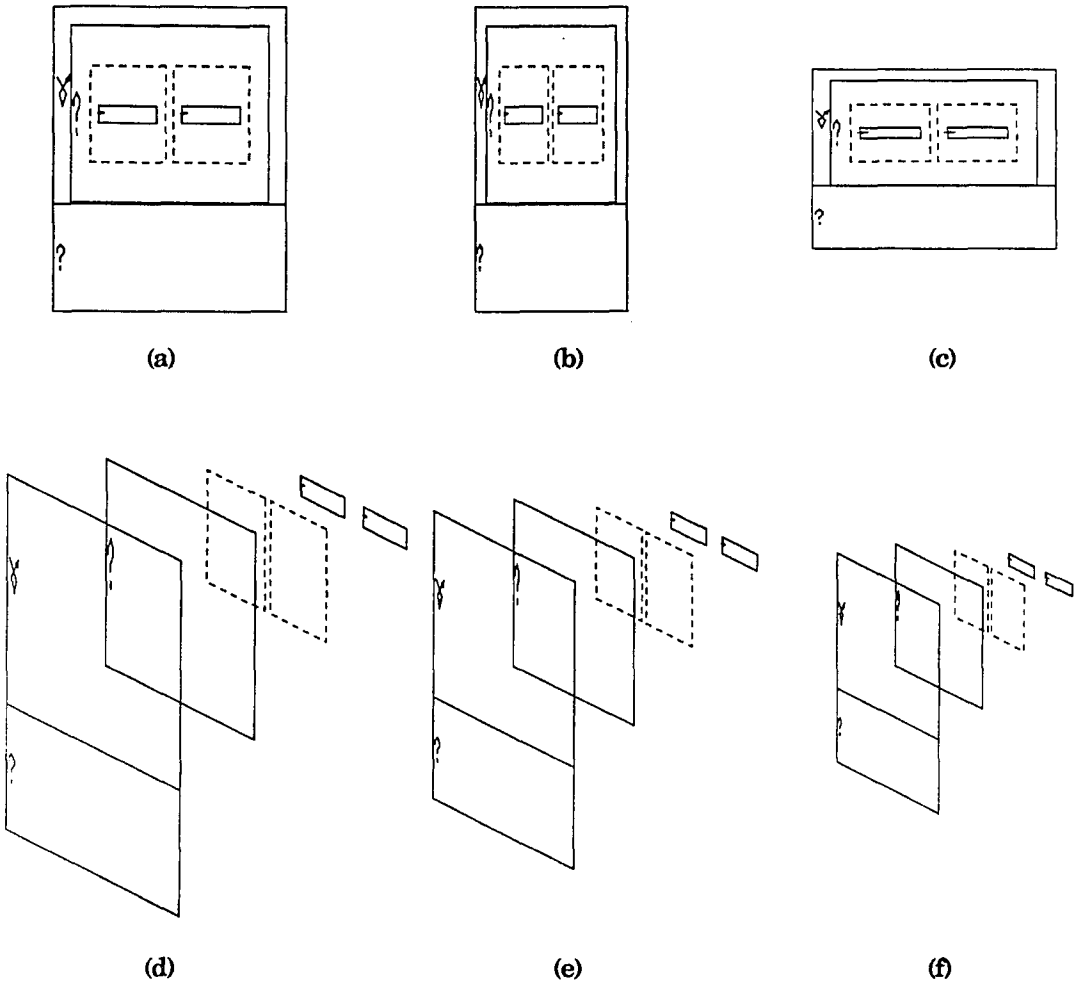
Rotational Views
Figure 7. 2

Another model command allows the user to translate the image in any of the three directions. This enables the user to make adjustments to the flow model if needed. The translation command, shown in Figure 7.1(b), activates the three-directional icon. The programmer then controls the translation by pointing to the appropriate field in the three-directional icon which translate the image in one of six directions.

The programmer can also scale the model by selecting the scale command shown in Figure 7.1(b). The user can scale the image in the X or Y direction by pointing to the

appropriate field in the three-directional icon. The scale affects the width and height of the tiers.

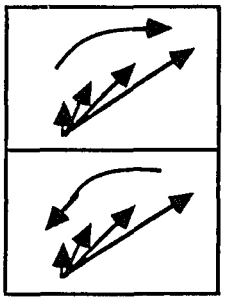
The "Image Scale" command is used to scale the model in all three axes. Pointing to any positive or negative section of the three-directional icon scales the model appropriately. Examples of scaling on one axis and all three axis are shown in Figure 7.3. Figure 7.3(a) is the normal view of the model. Figure 7.3(b) and 7.3(c) are models scaled in the X and Y direction respectively. Figure 7.3(d) thru 7.3(f) show the model scaled on all three axes. Any combination of scaling is allowed in the Peec environment.



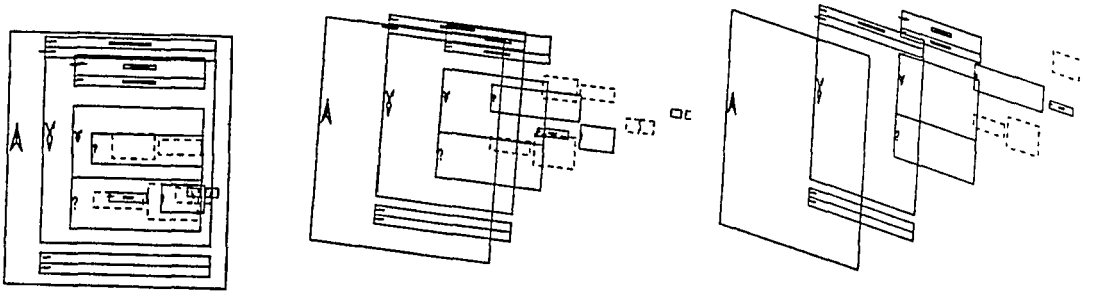
Views of Scaling Model
Figure 7.3

A convenient view of a flow model is with the tiers rotated around three axes. Peec provides an interactive feature which creates such a view where the tiers are facing downward, to the left and out of the screen, and where the nesting axis moves to the right, slightly upward, and into the screen. A special XYZ ROLL icon is provided as a

convenience for the programmer to roll the model around all three axes at once to obtain this view. The roll icon is shown in Figure 7.4. The user selects the direction of the icon to roll the image. The image is incrementally rotated in the positive x and negative y and z direction. The directional signs are reversed for rolling the image back. An example of the rolled image for Program B is depicted in Figure 7.5.



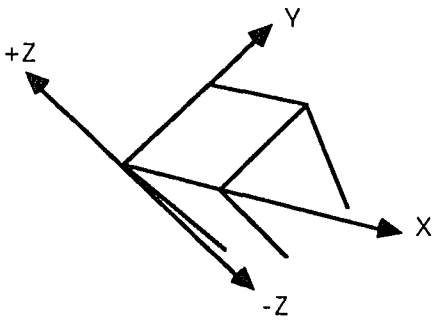
Roll Icon
Figure 7.4



(a) (b) (c)

Incremental Rolls
Figure 7.5

Since the model is made up of two-dimensional wire framed tiers positioned in three-dimensional space, it is not always easy to maintain the orientation of the flow model. An orientation key is supplied to assist the user with the model's orientation. Figure 7.6 shows the "key" image used for orientation. The key reflects only the rotational aspects of the model. The orientation image does not show the scale or translation effects on the flow model. The key representation is made up of a labeled XYZ axes, showing the positive and negative Z axis directions and a tier positioned at nesting level zero.



Orientation Key
Figure 7.6

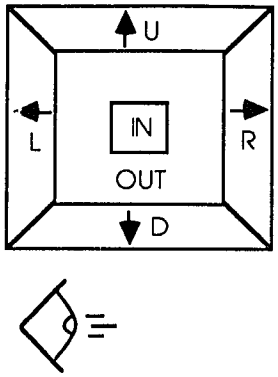
Browsing Feature

One of the most useful features within the Peec system is the ability to browse the image. The user can move from one structure to another studying the organization of the program, the set of accessible identifiers, and the relationship the control structures have with one another. As the programmer moves from point to point within the model, the Peec

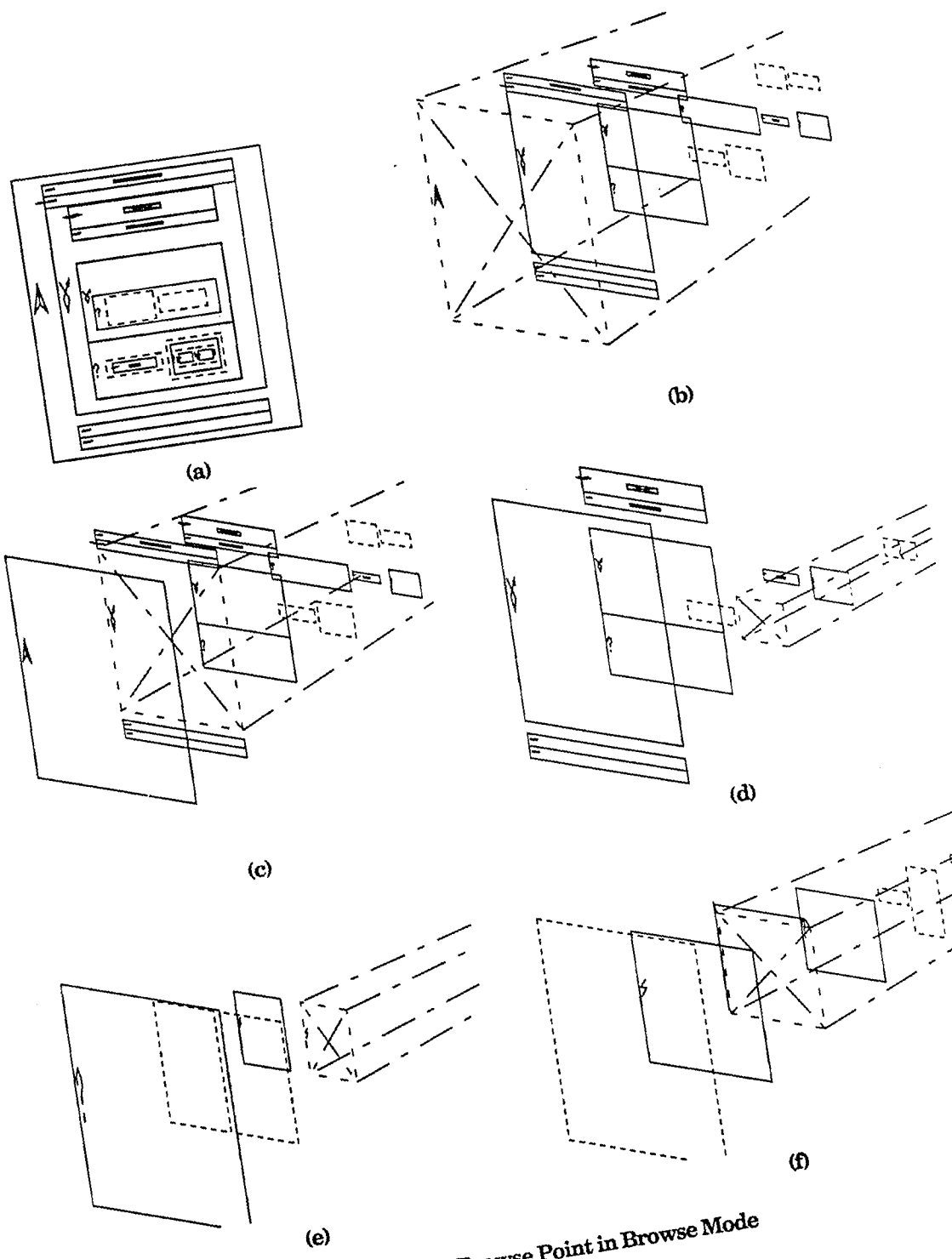
system will highlight the current tier being browsed to give the programmer a reference point. The highlighted tier is referred to as the browse point.

The programmer moves from one structure relative to another, that is from one sequential tier to either the next or previous sequential tier. The same movement is true for nested tiers, that is from one nested tier to either the next or previous nested tier. The programmer can browse into a procedure through the reference tier, study the procedure, then back out. The same process is defined for functions. This gives the programmer the ability to view procedures and functions at the point of reference, resulting in code localization. Of course, the user can ignore browsing the procedure or function if he chooses.

The browse point is represented as a highlighted tier. The programmer controls the movement of the browse point with the pointer device and the three-directional icon. The directions on the three-directional icon are labeled differently from the scale and rotate commands. The labels reflect the browse command movements relative to the control structure's organization. The three-directional icon and its browse switch icon are shown in Figure 7.7. The browse switch places the user in browse mode and activates the three-directional icon. The browse point can move six directions. To move form one sequential tier to the next, movement is either up or down denoted by U or D. The movement form left and right, denoted by L and R, are for movements between optional tiers. Movement from one nested tier to another is either IN or OUT. These commands allow the programmer to move about the three-dimensional flow model, study, and display information relative to the browse point. Figure 7.8 depicts the model in browse mode. The browse point is shown at different points within the flow model.



Three-Directional Icon and Browse Switch
Figure 7.7



Current Browse Point in Browse Mode
Figure 7.8

The model represents the nested tiers as smaller blocks positioned along the nesting axis. These tier images can become quite small if nesting is very deep. The Peec environment assists with this problem. As the user browses the smaller nested blocks, the Peec environment allows the image to be scaled up to give the user a better view of the nested structures. When the user moves out of a nested structure, the image is scaled down for the same reasons. The user can explicitly control the scaling if adjustments are required.

Figure 7.8 illustrates the browse point positioned at different nesting levels. Figure 7.8(a) thru 7.8(d) show the browse point at successive nesting levels. As the browse point moves deeper, the tiers are scaled up in size and translated up the nesting axis. This allows deeper nesting tiers to come into view. The Peec system adjusts the image as the browse point moves through the model, positioning the browse point in the same relative position on the display.

There are three additional features associated with the browse movements. Their primary functions are to reduce the number of tiers displayed within the flow model. Two features delete tiers automatically. As the user browses into deeper nested tiers, the model is translated up the nesting axis. The model's top level tiers are deleted as the nested tiers come into view. Figure 7.8(d-f) shows the top level tiers deleted as the user browses into nested tiers. The user can set the number of tiers that are viewable above the browse point. This feature is described in the following section.

The second automatic tier deletion is associated with browsing procedures and functions. As the user browses into a reference tier, the procedure's body and its nested tiers are added to the flow model. Figure 7.8(e) and 7.8(f) show a procedure's tiers before and after the user browses through the reference tier. When the user moves back to the reference tier, the procedure's tiers are deleted from the model. If the user browses through the reference tier again, the procedure is added to the flow model. The user can optionally

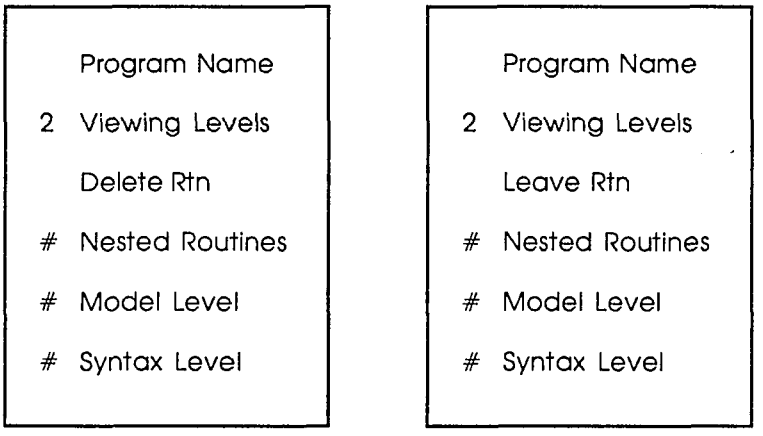
switch on and off the delete routine feature allowing procedure and function routine's tiers to remain visible. This feature is discussed in the next section.

The last tier management feature is directly controlled by the user. A complex flow model may have several nested structures represented. For example, a program with three sequential loops and each loop structure with several sequential and nested structures within them represents a complex flow model. The user may find it difficult to distinguish which parent tier a set of nested tier belongs to. The Peec system resolves this problem by blinking on or off a parent's nested tiers. To study one area of the flow model, the user can blink off all other structures not under study. Figure 7.8(d) shows the tiers for the loop blinked off. If the user browses into any nested structures not visible, the Peec environment will reestablish them automatically. The blink feature allows the user to manage the number of tiers viewable at one time. The user has limited control over the first two automatic tier control features. These features are discussed in the next section.

Information Window

The Peec system maintains an information or status window of parameters. The user may select the parameter fields and enter new values. Figure 7.9 depicts the fields of the information window. The last two fields represents the text's current nesting level and the model's current nesting level. The Peec environment informs the user how deep he has browsed into the flow model as well as the static nesting level. The next field shows the current number of reference procedures or functions through which the browse point has moved. The value is incremented for each procedure the user browses into and decremented for each one he browses out of. These three fields are controlled by the Peec environment and cannot be altered by the user.

The DELETE RTN is a toggle switch command. If the switch set to DELETE RTN and the user is in browse mode, then all procedures and functions tiers are deleted when the user moves out of routines or when the user moves from the body of a procedure to the reference tier. If the switch is set to LEAVE RTN, then the procedure or functions tiers remain viewable within the model. During browsing, the switch setting may be turned on and off allowing some routines to be left while others are deleted. Figure 7.9(a) shows the DELETE RTN set and Figure 7.9(b) shows it toggle to LEAVE RTN.



(a) (b)

Information Window
Figure 7.9

The VIEW LEVELS field indicates the number of levels above the browse point that are visible while browsing. As the user browses into deeper levels, the higher tiers are deleted from the viewing model. As the user moves up the nesting levels, the higher level tiers are added to the model image. In Figure 7.9, the viewing level is set to two so that two levels above the current browse point are viewable. The user can select the VIEW LEVELS field and change the number of levels viewable above the browse point. The last field identifies

the program being modeled by its file name. It is the name of the current program the user is viewing. The user may change from one program to another by selecting the name field. The system will prompt the user for a file name in the text window. The user supplies a new file name, the system is reset, the new file is loaded, and the main program tiers are displayed. The user is now ready to explore a new program.

Displaying Text and Identifiers

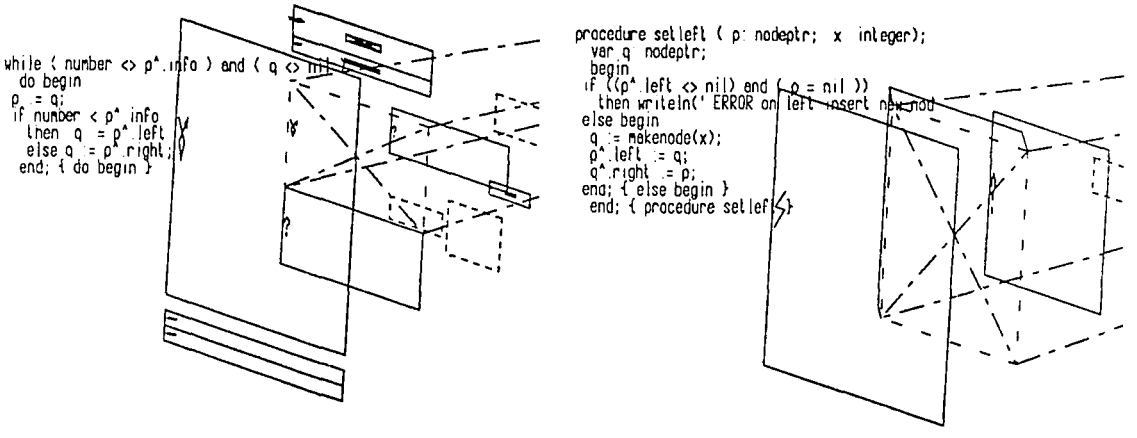
As stated earlier, the programmer develops understanding by localizing code. He browses the structures and studies their text and associated variables. The Peec system supplies the programmer with the ability to study the text which defines a given tier or to view the set of identifiers accessible by the tier. The text is displayed on demand. The programmer can study a tier's definition at its lowest level, the code. The user then associates meaning with the tier's image in relation to local structures. At a later time, if the user needs to update his understanding of the structure, he can again display the text. This action is similar to the process a programmer uses by briefly scanning text to refresh his memory about a section of code. Another way to interpret this feature is that Peec allows the programmer to decompose any tier down to its lowest definition, the text itself. The algorithm for displaying a control structure's text is shown in Figure 7.10. Figure 7.11 illustrates the text displayed for Program A.

- Define window for displaying text
- Position file pointer on first line of tier in Text file
- If Proc/Func & Print Declaration switch on
- Nbr of Lines <- Total of Proc/Func declaration and body
- else
- Nbr of Lines <- Number of lines in tier { Rtn's body only }
- Compute text display parameters { height, position, etc }

  LOOP:
- For Nbr of Lines
- Read Text line
- Display text line in window
- Adjust display parameters

  END LOOP
- Wait for user input to clear
- Delete text displaying window

Text Display Algorithm
Figure 7.10

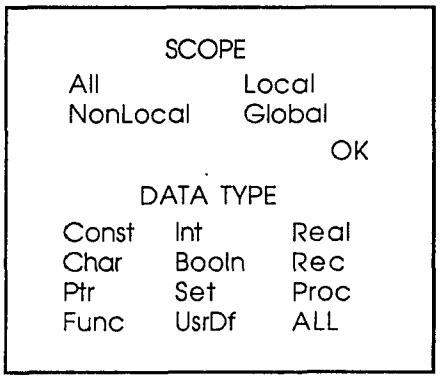


Text Display Example
Figure 7.11

In some cases, the programmer may feel more comfortable if the text for the current structure is continuously available. The Peec environment gives the programmer the

option of continuously update a text listing window as he moves from one tier to the next. While in browse mode, the programmer selects the TEXT WINDOW to switch on and off this feature. The text window is updated as the browse point moves through the flow model.

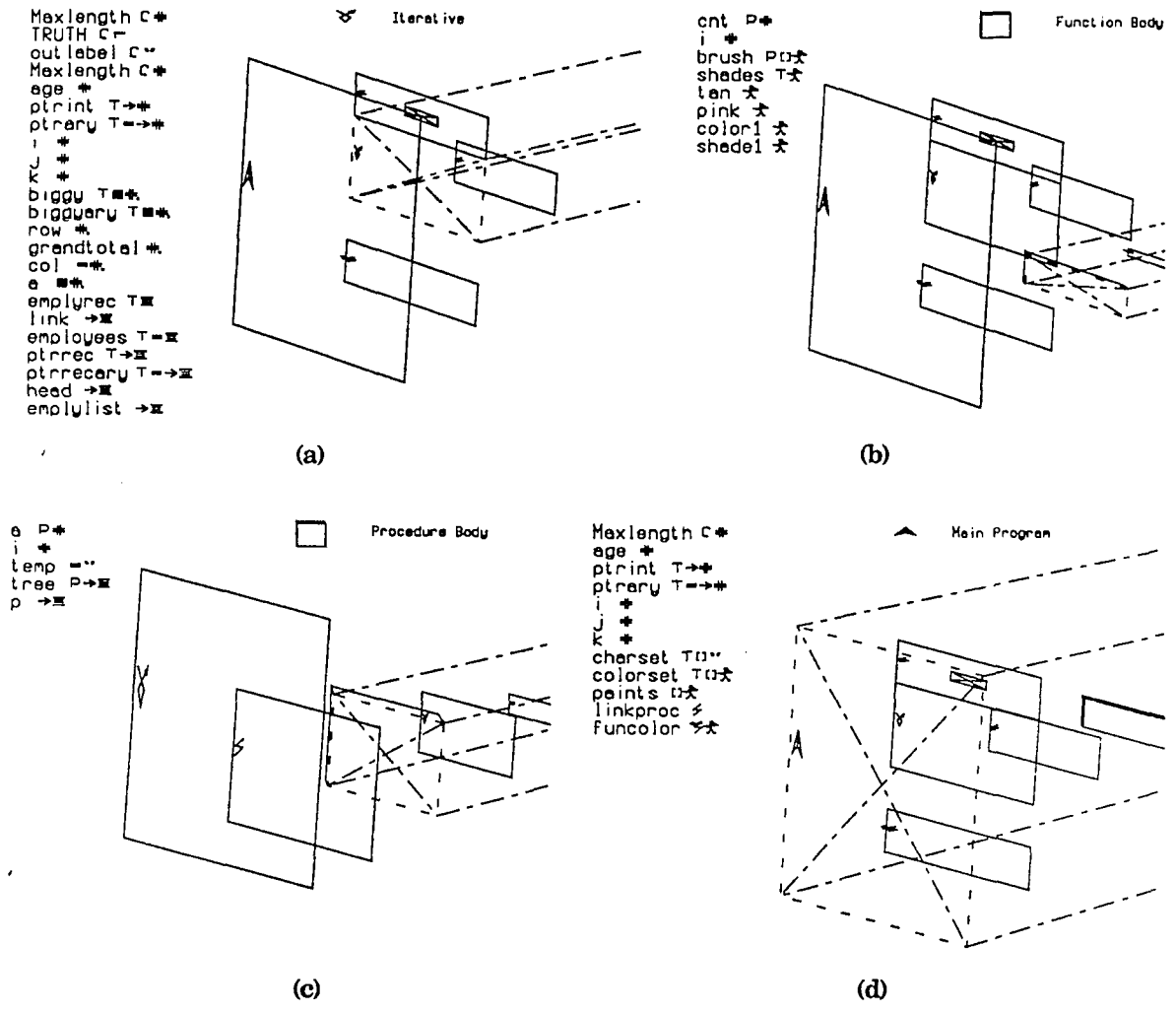
Another feature stated earlier is the ability to display identifiers and their type for the current browse point. The programmer can optionally display a set or a subset of identifiers for any tier in the flow model. The user selects the identifiers to be displayed based on their scope definition and basic type. Figure 7.12 shows the menu the programmer uses for selecting identifiers. The programmer can select one of the scoping options as either *local*, *nonlocal*, *global*, or *all*. He can also select the type of identifiers to be displayed by selecting the data types. If the programmer wishes to display identifiers, he must select both a scope rule and a data type attribute. More than one data type can be selected. The selected options are displayed in italics within the scope menu. The identifiers are displayed while the programmer is browsing the model. As he browses from tier to tier, a list of identifiers and their attributes are displayed. The list is automatically updated as the user browses. This is more apparent when browsing in and out of procedures and functions.



Scope Menu
Figure 7.12

Figure 7.13 gives some examples of displaying identifiers for Program C. In Chapter VI, we defined the type icons which describe the identifier. The icons associated with each identifier describe its basic type and its data structure. Figure 7.13(a) shows integer, real, constant, and record identifiers. The "C" and "T" symbols represent constant and type identifiers. The identifier "biggy" is interpreted as a TYPE defined as an array of reals where the array is greater than one dimension. The identifier "col" is a vector of reals. The identifier "ptrrecary" is interpreted as a TYPE defined as a vector of pointers, pointing to records.

Figure 7.13(b) shows the current browse point as the body of a function. The identifiers are local to the routine. "P" denotes parameter identifiers and the *man* icon denotes user-defined types. Figure 7.13(c) displays local identifiers for a procedure. In Figure 7.13(d), the identifier "paints" denotes a set of user-defined type. The identifiers "linkproc" and "funcolor" denote a procedure and function names, respectively.



Displaying Identifiers and Descriptors
Figure 7.13

Figure 7.14 provides the algorithm used to generate the list of identifiers. If the scope tree is not defined, then it is built before the variables are referenced and displayed. The scope tree was described in Chapter VI.

```
• If Scope tree does not exist
•       Build Scope Tree { Figure 6.14 }
• Traverse down mapping link list
•       Find Address of Scope tree node for current tier
  LOOP1 :
•       While Scope Node in Scope Limits { global, local, nonlocal, or all }
        LOOP2 :
•               Traverse down identifier link list
•               If identifier base type in set of Selected Base Type
•                       Display identifier
        END LOOP2
•       Next parent node in scope tree
  END LOOP1
```

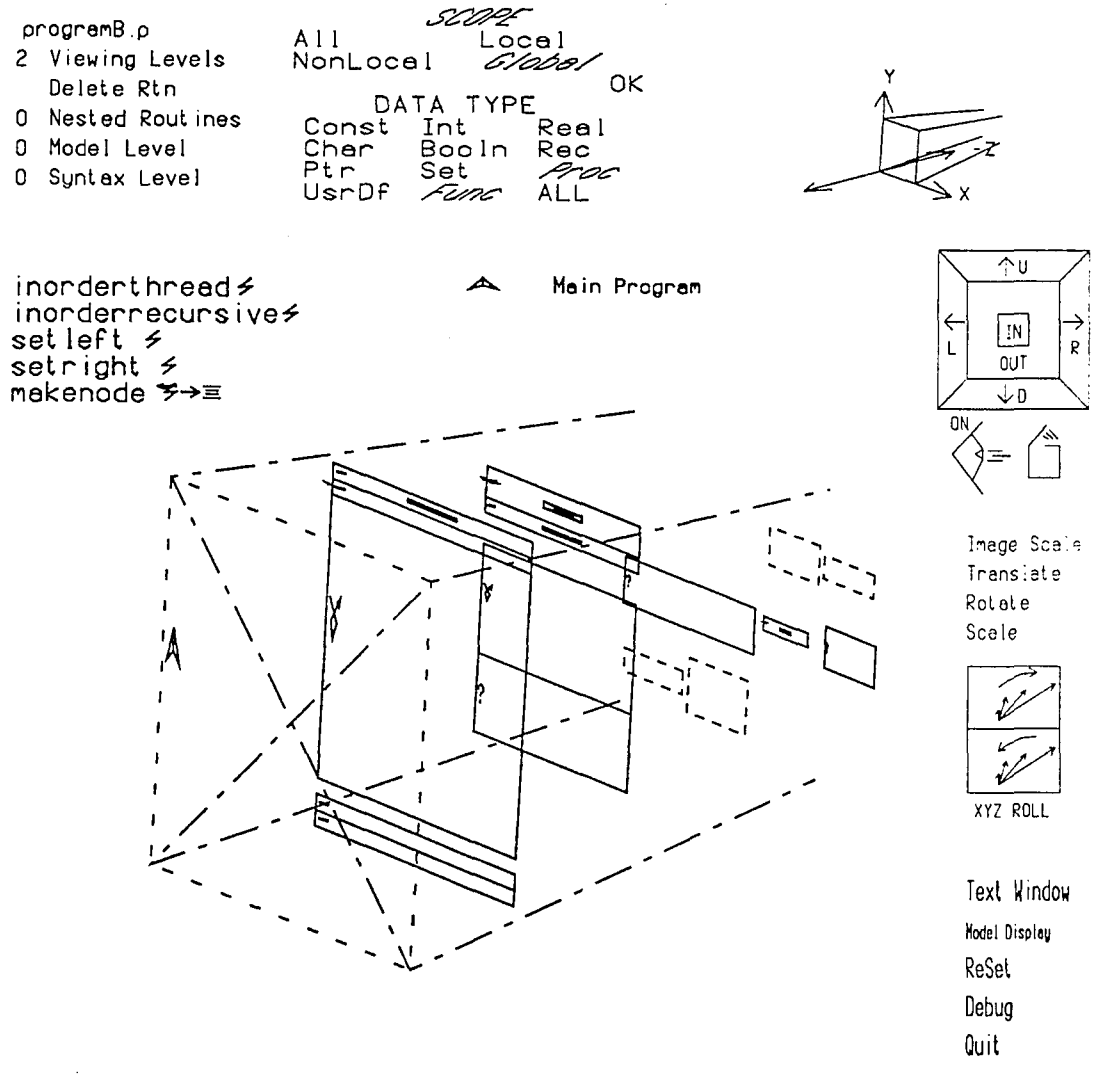
Identifier Display Algorithm
Figure 7.14

Summary

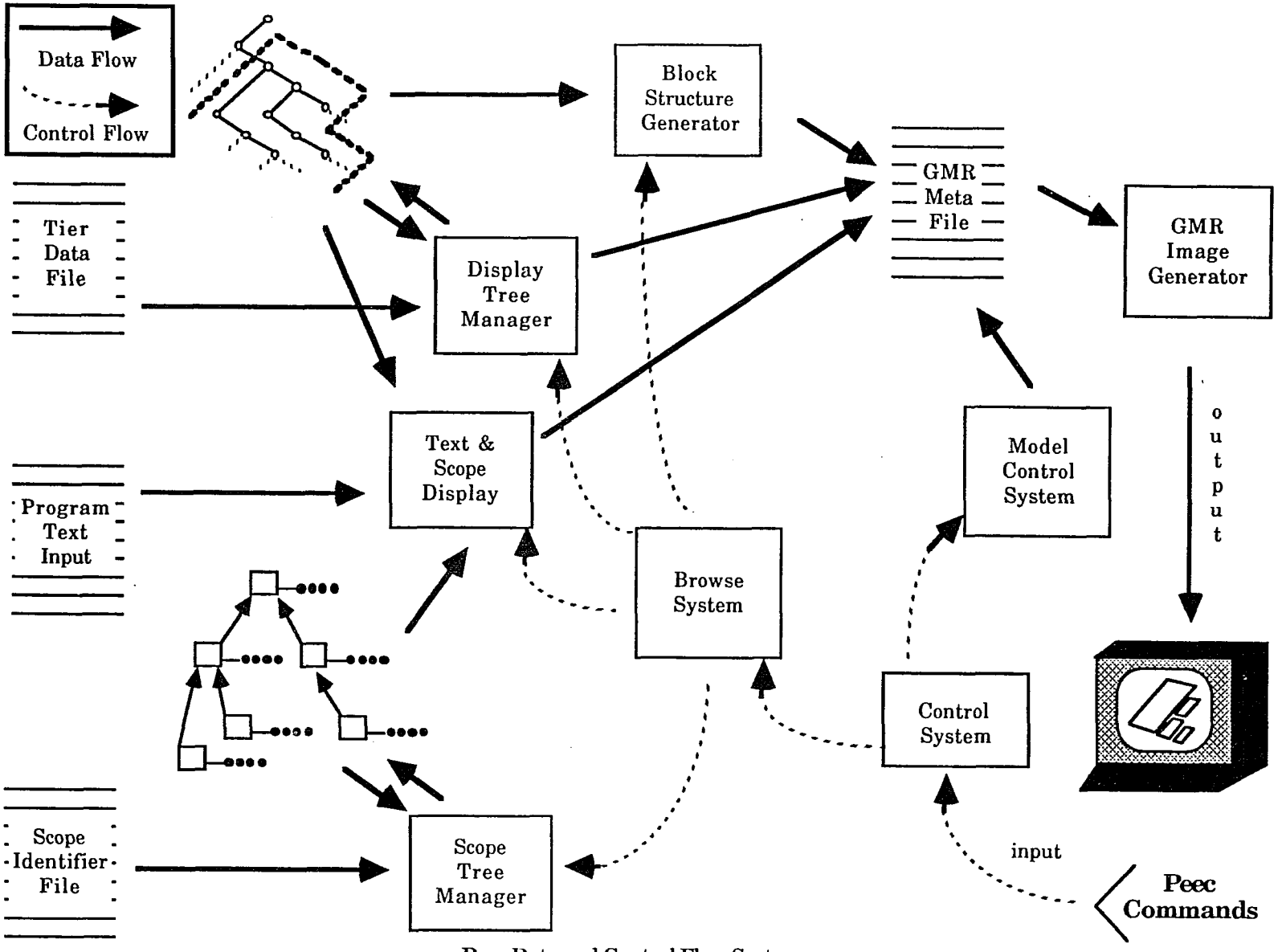
The Peec interactive features allow the user to browse the program in a manner that emulates the way he browses a source code listing. The user has access to all the code defining each tier as well as the identifiers. As he browses through the model, the identifier list is automatically updated showing the live identifiers and a description of their structure. The interactive environment supplies the user many features to browse and control the flow model as well as status information on the program under study and on the Peec environment. The format of the interactive screen with the menus and icons described in this chapter is shown in Figure 7.15.

We have presented the data structures, algorithms and interactive functions found in the Peec environment. An overview of the Peec model is shown in Figure 7.16. The figure shows the flow of data from left to right and the flow of control from bottom up. The two

primary data structures, the display tree and the scope identifier tree, are shown along with their management routines.



Peec Interactive Screen Format
Figure 7.15



Peec Data and Control Flow System
Figure 7.16

CHAPTER VIII

SYSTEM EVALUATION

In order to evaluate the effectiveness of the Peec system, we conducted an empirical evaluation. This evaluation is intended to indicate whether the subjects perceived the system as a system that enhances their ability to understand unfamiliar code.

Subjects

Nineteen subjects were used in this study. The subjects were graduate students, working on a Master's in Computer Science, and senior undergraduate Computer Science majors at the University of Southern Mississippi. The students were unpaid volunteers who had taken several computer science courses in which programming was a major aspect. Each student had written one or more Pascal programs of lengths greater than 500 lines of code. The number of students in each classification along with a breakdown based on sex are given in Table 8.1.

	Males	Females
Senior	10	4
Graduates	3	2

Subject Groups
Table 8.1

The graduate and undergraduate students were separated because the graduates had had more programming experience. The distinction between males and females was based on results from previous studies where images were part of the testing system (Glinert & Tanimoto, 1984; Malone, 1980). Glinert and Tanimoto's testing of the Pict system indicated a more favorable response from females than from males. Malone encountered similar results in his study of video games.

Procedure

Each subject was given an explanation and a demonstration of Peec's features. An overview of the system was presented and an example program was used to demonstrate the functions of Peec. The introduction session took approximately thirty minutes. After the introduction, the students were encouraged to continue to use the system and become familiar and comfortable with the interactive capabilities. They were also allowed to ask additional questions and to discuss any aspect of the system. The students were supplied with several programs for use in the learning session. The time ranged from thirty to sixty minutes for this intermediate phase. During this time, very few questions were asked which would indicate that they understood the functions of the system. The students spent much of the extra time browsing and testing the limits of the system.

The students were asked to return the next day for the final phase of the test. Each subject was given a program to study. The student was asked to describe the functions for a given program. The time ranged from thirty to forty-five minutes for this phase. Next, the subjects were asked to complete a questionnaire in which the responses, ranging from one to five, addressed their evaluation toward the usefulness and responsiveness of the system. The students were asked to provide any additional comments. The complete questionnaire is shown in Appendix B.

Results

Tables 8.2 through 8.7 summarize the results of the evaluations. Tables 8.2 and 8.3 give the number of responses and the percentage of answers greater than three for the senior and graduate students respectively. Tables 8.4 and 8.5 indicate the responses for seniors and graduates by sex.

Questions	1	2	3	4	5	Percentage Answering >=3
Peec easy to use?	0	0	0	8	6	100
Menus and icons easy to use?	0	0	0	9	5	100
Like menus set up for communicating?	0	0	0	4	10	100
Like using the mouse to control model?	0	0	0	3	11	100
Easy to learn control structure icons?	0	0	3	4	7	100
Easy to learn identifier icons?	0	0	4	8	2	100
Scope of variables helpful?	0	0	2	8	4	100
Like three-dimensional model?	0	0	0	1	13	100
Tier helpful in understanding nesting?	0	0	1	3	10	100
Flow model help understand execution order?	0	0	1	6	7	100
Peec assist more than just program listing?	0	1	0	6	7	92.9
Peec helpful in first programming courses?	0	1	1	2	10	92.9
Peec helpful in teaching programming concepts?	0	0	1	2	11	100
Peec useful in studying larger programs?	0	0	2	1	11	100
Peec to be useful to experienced programmer?	0	0	1	7	6	100
How useful in developing program understanding?	0	0	0	4	10	100
Useful in studying algorithm implementation?	0	0	1	5	8	100
Would you use a system like Peec?	0	0	0	2	12	100
Overall evaluation of the Peec system?	0	0	0	4	10	100

Undergraduate Responses
Table 8.2

Questions	1	2	3	4	5	Percentage Answering >=3
Peec easy to use?	0	0	0	0	5	100
Menus and icons easy to use?	0	0	0	1	4	100
Like menus set up for communicating?	0	0	0	1	4	100
Like using the mouse to control model?	0	0	0	0	5	100
Easy to learn control structure icons?	0	0	1	1	3	100
Easy to learn identifier icons?	0	0	1	2	2	100
Scope of variables helpful?	0	0	0	2	3	100
Like three-dimensional model?	0	0	0	0	5	100
Tier helpful in understanding nesting?	0	0	0	0	5	100
Flow model help understand execution order?	0	0	0	1	4	100
Peec assist more than just program listing?	0	0	0	1	4	100
Peec helpful in first programming courses?	0	0	0	3	2	100
Peec helpful in teaching programming concepts?	0	0	0	1	4	100
Peec useful in studying larger programs?	0	0	0	0	5	100
Peec to be useful to experienced programmer?	0	0	2	0	3	100
How useful in developing program understanding?	0	0	0	1	4	100
Useful in studying algorithm implementation?	0	0	0	2	3	100
Would you use a system like Peec?	0	0	1	0	4	100
Overall evaluation of the Peec system?	0	0	0	1	4	100

Graduate Responses
Table 8.3

Questions	Male	Female
Peec easy to use?	4.3	4.8
Menus and icons easy to use?	4.5	4.0
Like menus set up for communicating?	4.6	5.0
Like using the mouse to control model?	4.9	4.5
Easy to learn control structure icons?	4.4	4.0
Easy to learn identifier icons?	3.7	4.3
Scope of variables helpful?	3.9	4.8
Like three-dimensional model?	4.9	5.0
Tier helpful in understanding nesting?	4.5	5.0
Flow model help understand execution order?	4.3	4.8
Peec assist more than just program listing?	4.1	5.0
Peec helpful in first programming courses?	4.3	5.0
Peec helpful in teaching programming concepts?	4.6	5.0
Peec useful in studying larger programs?	4.5	5.0
Peec to be useful to experienced programmer?	4.2	4.8
How useful for developing program understanding ?	4.6	5.0
Peec useful in studying algorithm implementation?	4.3	5.0
Would you use a system like Peec?	4.8	5.0
Overall evaluation of the Peec system?	4.6	5.0

Seniors Response
Table 8.4

Questions	Male	Female
Peec easy to use?	5.0	5.0
Menus and icons easy to use?	4.7	5.0
Like menus set up for communicating?	4.7	5.0
Like using the mouse to control model?	5.0	5.0
Easy to learn control structure icons?	4.0	5.0
Easy to learn identifier icons?	3.7	5.0
Scope of variables helpful?	4.3	5.0
Like three-dimensional model?	5.0	5.0
Tier helpful in understanding nesting?	5.0	5.0
Flow model help understand execution order?	4.7	5.0
Peec assist more than just program listing?	4.7	5.0
Peec helpful in first programming courses?	4.3	4.5
Peec helpful in teaching programming concepts?	4.7	5.0
Peec useful in studying larger programs?	5.0	5.0
Peec to be useful to experienced programmer?	4.3	4.0
How useful for developing program understanding ?	4.7	5.0
Peec useful in studying algorithm implementation?	4.7	4.5
Would you use a system like Peec?	4.3	5.0
Overall evaluation of the Peec system?	4.7	5.0

Graduates Response
Table 8.5

Tables 8.6 and 8.7 give the average responses by sex, classification and overall responses.

Questions	Male	Female	Total
Peec easy to use?	4.5	4.8	4.6
Menus and icons easy to use?	4.5	4.3	4.5
Like menus set up for communicating?	4.6	5.0	4.7
Like using the mouse to control model?	4.9	4.7	4.8
Easy to learn control structure icons?	4.3	4.3	4.3
Easy to learn identifier icons?	3.7	4.5	3.9
Scope of variables helpful?	4.0	4.8	4.3
Like three-dimensional model?	4.9	5.0	4.9
Tier helpful in understanding nesting?	4.6	5.0	4.7
Flow model help understand execution order?	4.4	4.8	4.5
Peec assist more than just program listing?	4.2	5.0	4.5
Peec helpful in first programming courses?	4.3	4.8	4.5
Peec helpful in teaching programming concepts?	4.6	5.0	4.7
Peec useful in studying larger programs?	4.6	5.0	4.7
Peec to be useful to experienced programmer?	4.2	4.5	4.3
How useful for developing program understanding ?	4.6	5.0	4.7
Peec useful in studying algorithm implementation?	4.4	4.8	4.5
Would you use a system like Peec?	4.7	5.0	4.8
Overall evaluation of the Peec system?	4.6	5.0	4.7

Response Averages by Sex
Table 8.6

Questions	Senior	Grad	Total
Peec easy to use?	.4	5.0	4.6
Menus and icons easy to use?	4.4	4.8	4.5
Like menus set up for communicating?	4.7	4.8	4.7
Like using the mouse to control model?	4.8	5.0	4.8
Easy to learn control structure icons?	4.3	4.4	4.3
Easy to learn identifier icons?	3.9	4.2	3.9
Scope of variables helpful?	4.1	4.6	4.3
Like three-dimensional model?	4.9	5.0	4.9
Tier helpful in understanding nesting?	4.6	5.0	4.7
Flow model help understand execution order?	4.4	4.8	4.5
Peec assist more than just program listing?	4.4	4.8	4.5
Peec helpful in first programming courses?	4.5	4.4	4.5
Peec helpful in teaching programming concepts?	4.7	4.8	4.7
Peec useful in studying larger programs?	4.6	5.0	4.7
Peec to be useful to experienced programmer?	4.4	4.2	4.3
How useful for developing program understanding ?	4.7	4.8	4.7
Peec useful in studying algorithm implementation?	4.5	4.6	4.5
Would you use a system like Peec?	4.9	4.6	4.8
Overall evaluation of the Peec system?	4.7	4.8	4.7

Response Averages by Classification
Table 8.7

The students made numerous comments about the system. A few of these comments are cited below:

- Peec is useful in conjunction with a source listing ... with large programs you don't have to flip through listings to find variables and procedure code.
- Fantastic graphics and user interface. I would like to see editing feature while you are browsing ...
- ... I like the 3-D break down ... and how it puts procedure and functions in the order they execute. ... The commands were very easy to learn. I became fast with the system in a matter of 10 to 15 minutes.
- Very useful in programming courses ... as it gets you going with the "feel" for where you are in a program.
- Would like to see assignment statements represented (explicitly) rather than gaps in and between control structures tiers.
- I enjoyed using the Peec system. It's easy to learn and puts you at a level where you can actually "walk around" in your program. This in itself will help anyone who is not sure of what his or her program is doing.
- By displaying the dynamic structures of a program in a static manner, beginning programming students should be able to understand more about abstract concepts better such as recursion.
- When a program gets really large, it might be helpful to look at two (or more) modules at the same time (two display screens).
- ...could be very useful especially in explaining to beginning programmers how nesting and levels occur within a program. ... gives them a better concept of how the computer executes the source code.

The scope was useful when determining what names could be referenced within a procedure or function.

- ... everything you needed to know could be found on the screen (ie. functions of icons was easy to determine).
- (Reference to question on teaching programming concepts). The best part about this system is that the majority of what can be learned can be generalized through analogy to any (or almost any) other high level programming language, not just Pascal.
- Excellent program representation in 3-D and very easy to communicate with. I like it when I can learn a useful tool in an hour or two.

As seen from the tables, the students viewed the Peec system in a positive sense. We can categorize the questions as either "responsiveness to the system" or "the effectiveness of the system". The first six questions measured the students' responsiveness to Peec's functions. The graduate students scores were consistently higher than the seniors. Comparison by sex shows that females had three responses higher, one equal and two lower than males. In general, the students' responses were very favorable. They indicated that the system was easy to learn and easy to use. They also liked controlling the model using a pointer device rather than textual commands.

The responsiveness of the system was measured in the next twelve questions. Eleven responses for the females were higher than males and nine of the graduate students responses were higher than seniors. From the results, the graduates were more favorable than the seniors. There were three questions with the highest results that dealt with "mouse interface", the "three-dimensional model", and the likelihood of using the Peec system. The students expressed a positive response to the visualization of the control structures and specifically the representation of the recursion.

In response to Peec's effectiveness in teaching, both the questionnaire and the comments made indicate that the system would be very helpful in teaching programming concepts. Also, the effectiveness in developing program understanding obtained a high rating. The graduate students' comments stated they were accustomed to using a listing only and they found it difficult to alter this method in the program understanding process. They did feel the Peec environment would help them once they were accustomed to thinking of a program in terms of the flow model format.

In summary, the response of the subjects to Peec indicates that the Peec system complements the program understanding process in a responsive manner.

CHAPTER IX

CONCLUSION

Summary

The goal of this research was to provide a graphical system that supports the program understanding process by representing the program control flow, the code and the identifiers local to a specific portion of the code. By having more information local to the point of interest, the programmer can maintain continuity in developing program understanding (Letovsky & Soloway, 1986). We have defined and implemented such a system. The Peec system supplies a graphical representation of a control flow model in which the control structures are represented as tiers. The programmer associates meaning with the tiers in a local area or browse point. If the user needs to decompose the tier, he can pop up the text and list the identifiers without having to look elsewhere. When the programmer browses a procedure or function reference tier, the routine's tiers are nested within the reference tier so that the programmer views the routine local to its reference point. The advantage is that the routine is local to the area of study and the programmer does not have to look elsewhere for the procedure or function text.

Each element or tier of the image is decomposable down to its textual representation. This allows the programmer to maintain a relationship between the image and the text. It also gives the programmer the ability to decompose the image down to its definition. The Peec environment gives the programmer many interactive capabilities. All of these features are aimed at assisting the programmer in developing understanding in an effective manner.

The image is created from the tier information file generated by the transformation phase. Peec arranges the control structures, represented as two-dimensional rectangles, in a three-dimensional space. The program's control flow is viewed as sequential, optional, and nested control structures where each control structure is identified by an associated icon or by the block's shape. The programmer can see loops, procedure calls, and other structures with respect to their execution order and can view them in the environment or context in which they will execute. The size and position of the tier is equated to the lines of code and nesting definition within the text file.

Peec is fully implemented. Our empirical evidence gives strong support to the validity of the system as a support to the program understanding process. Applications of the Peec system have been discussed throughout the paper. Program maintenance is one predominate area. The programmer who needs to alter code must first have an understanding of the program's functions and organization before such tasks can be performed. Except for the simplest program modification, the programmer spends time abstracting from the text the program's operational flow in order to understand what changes need to be made and the impact these changes have on the program. Program development and debugging require the same basic methodology.

Education is another area in which the Peec environment can be used. Certain programming concepts can be taught through the use of Peec. The beginning programmer can be shown how the program's execution flow is based on the textual organization of the code. First the Peec system supplies the programmer a visual representation for modeling a program's operational flow. Second, he can see the impact that control structures have on the program's execution flow model. Therefore, the system can assist the programmer in developing a representation of a program's execution order and it can display the code organization of the executing program.

Another aspect of the system that is useful in the education area is the capability to view recursive routines and the identifiers. The programmer can view the representation of the recursive routines execution within the flow model. As the user browses through the references, the flow model depicts each instantiated routine as a new set of tiers. Also, the programmer can develop an understanding of the scoping rules defined by the language. He can see which identifiers are active while browsing through each routine. The Peec environment can give the beginning programmer a stronger feel for the programmed algorithm through the visual representation of its operational flow model.

The more advanced programmer can use the Peec system to compare different implementations of the same algorithm. He can see the different flow models for each coded algorithm. The operational flow model shows the arrangement of nested and sequential control structures, allowing the programmer to study the differences between the algorithm implementations. The system supplies the programmer with a visual feature which adds to the algorithm analysis techniques.

The motivation behind the design of the three-dimensional model was to develop a representation that expressed the program's control flow based on and in relation to the program's textual structure. The flow model representation contains relational aspects between it and the text. The programmer's natural understanding process is to abstract a flow model from the text, then update the model by browsing or rebrowsing the text. If a flow model is based on something other than the text or if the representation is not related to the textual structures, the programmer can find himself abstracting from both the text and the representation before any modifications to the program can be made. After all, the programmer who updates a program will alter the code not its representation, therefore, his understanding of the program must be in relation to the code.

The three-dimensional representation gives the user an image of the programs operational flow, an abstract concept. The graphical representation gives the programmer more information and more detail than that found in the text. With the interactive features supplied by the Peec environment, the programmer has more information readily available which is local to the point of interest.

The Peec system is classified as a "Graphics-Enhanced Software System". The language level is considered high since the input language is Pascal. The scope of the system is consider general for the same reason. The visual content of Peec is classified as medium. The "Graphics-Enhanced Software Systems" summary table, covered in Chapter II, is repeated here with the inclusion of Peec system. The Peec system provides features for all levels of programmers. The novice can learn certain programming concepts while the experienced programmer can use the localizing features to study the code. The BALSA system is rated higher in the visual extent, but BALSA is not designed for the experienced programmer. Its primary use is in the educational environment and in the study of new algorithms (Brown, 1988). Also, the Peec system requires minimal amount of time to setup and view a program. The programmer need not add or alter any of the program's code. He simply compiles the Pascal program, then initiates the Peec system and loads the program. There is little overhead needed, with respect to the programmer's time, in order to run the Peec system.

System Name	Level	Scope	Visual	Developer
BALSA	High	Specific	High	Brown, Sedgewick
Visual of Indep. and Dep. for Pgm.Concurrence	High	General	Low	Belady, Hosokawa
Pigs	Medium	General	Low	Pong,Ng
SDMS	Low	Specific	Medium	Herot, Kramlich
PV	High	General	Low	Brown
PECAN	High	General	Low	Reiss
VIPS	High	General	Low	Isoda
Peec	High	General	Medium	Rimes, Carver

Graphics-Enhanced Software Systems
Table 9.1

Future Work

The Peec environment leads to a number of areas for continued study. The design of the system allows the high level language and the display system to function independently of each other. The block structured language used for the Peec system is Pascal. An immediate extension of the Peec system includes the development of translators for other block structured languages where the translators produce tier and scope identifier files. The language C and Modula are appropriate choices for this extension. The Ada language is also a good choice, but there would be additional design work required to handle and represent Ada's structures.

Another extension to the system is to allow the user to execute the program in the Peec environment. The programmer would actually view the order the program's tasks execute in a three-dimensional format. He would observe the paths the programs takes in addition

to having access to the variables and their values during execution. With this extension, the Peec system would become a language-dependent system.

There are a number of languages that are not "block structured" that are applicable for the Peec system concept. Lisp is such a language. The nature of implementing an algorithm in Lisp is based on functions calling other functions to accomplish its tasks. The programmer is required to abstract from the code the order these functions will execute when designing and debugging the program. The nesting of these functions can be much deeper than what a programmer typically sees in a block structured language. An extended Peec system could be used with Lisp programs to clarify and to maintain information about the execution order of these functions.

Another problem with understanding Lisp programs is the dynamic scoping of variables. Many Lisp environments use dynamic scoping which makes program understanding more difficult. For this reason, some Lisp environments use static scoping. The features found in the Peec system would benefit the programmer greatly in this respect by alleviating much of the burden required to manage the identifiers based on the order the functions are referenced. The programmer could view the identifier within scope of the current function as he browses from one function to anther.

Another extension is to allow editing features within the Peec environment. A program could be altered by two editing concepts. One is the traditional editing concept where the actual code is altered, statement by statement. The flow model would reflect these changes immediately. The second concept would allow the movement of the tiers from one place to another. The tiers could be moved within the three-dimensional space or they could be duplicated and moved to new positions in the model. A duplication of a tier would represent a repeat of the code and would be interpreted as the automatic formation of a procedure or function. The text file would reflect the appropriate modifications.

Another area where extensive research is being conducted is automatic program documentation. The amount of information collected in the translation process can be used in this endeavor. The translator currently collects information about each control structure and its relation to the immediate control structures. It collects information about each identifier and its scoping information in relation to the tier. The Peec system also allows the flow model image to be captured and printed. These images could be used to automatically document the program's control flow. It also could list the live identifiers associated with model. With this information and additional algorithms for documenting the program, the system could contribute to automatic documentation research. The documentation would reflect the current implementation features of the algorithm and the documentation could easily be updated as modifications are made to the program.

Finally, the hardware and software supporting the Peec environment could be advanced. The addition of color to the image model would improve the visual effect and enhance the programmer's ability to understand programs. Color could be used to emphasize certain aspects of the image. The nested tiers along a path could be highlighted to show which tiers are currently active. Including the text within the tiers would be an asset to the understanding process, but the limitation of the hardware does not allow for convenient implementation. Improvements to the speed of handling an image in three-dimensional space would enhance the system significantly. Finally, a more animated transition during browsing would add an aesthetic effect to Peec's interactive features.

Bibliography

Aho, Alfred V., Sethi, Ravi, & Ullman, Jeffrey D. (1986). *Compilers: Principles, techniques, and tools*, Massachusetts: Addison-Wesley Publishing Company.

DOMAINS 3D Graphics Metafile Resources Call Reference. (1985). Apollo Computer Inc., Massachusetts: Chelmsford.

Assmann, K., Venema, R., & Hohne, K. H. (1986). The ISQL language: A software tool for the development of pictorial information systems in medicine. In S. Chang, T. Ichikawa, P. A. Ligomenides (Eds.), *Visual Languages* (pp. 261-284). New York: Plenum Publishing.

Backus, J. W. (1978, August). Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *CACM, 21* (8), 613-641.

Basili, Victor R. & Mills, Harlan (1982). Understanding and documenting Programs. *IEEE Transactions on Software Engineering,* SE-8 (3), 270-283.

Belady, L. A. & Hosokawa, K. (1984). Visualization of independence and dependence for program concurrence. *Proc. of the 1984 IEEE Computer Society Workshop on Visual Languages,* (pp. 59-63). Hiroshima, Japan.

Brooks, Frederick P., Jr., (April,1987). No silver bullet: Essence and accidents of software engineering. *Computer, 20* (4), 10-19.

Brown, Lew (1987, October). Visualization in the lab. *Government Data Systems, 16* (7), 10-12.

Brown, Marc H. (1988, May). Exploring Algorithms Using Balsa-II, *Computer, 21* (5), 14-36.

Brown, M. H. & Sedgewick, R. (1985, January). Techniques for algorithm animation, *IEEE Software, 2* (1), 28-39.

Brown, G. P., Carling, R. T., Herot, C. F., Kramlich, D. A. & Souza, P. (1985). Program visualization: Graphical support for software development. *IEEE Comput. 18* (8), 27-35.

Carroll, J. M. & Thomas, J. C. (1982). Metaphor and the cognitive representation of computing systems. *IEEE Trans. Syst. Man Cybern., 12* (2), 107-116.

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

Chang, Shi-Kuo, Reuss, J. & McCormick, B. H. (1978). Design consideration of a pictorial database system. *Int. J. Policy Anal, Inf. Syst., 1* (2), 49-70.

Chang, S. K., Jungert, E., Levialdi, S., & Tortora, G. (1983, November). IPL - An image processing language and programming environment. *Proceedings of the IEEE Workshop on Languages for Automation,* (pp. 78-84). Chicago, Illinois.

Chang, S. K., Jungert, E., Levialdi, S., Tortora, G., & Ichikawa, T. (1985, August). An image processing language with icon-assisted navigation. *IEEE Trans. Software Eng.* SE-11(8), 811-819.

Clarisse, Olivier & Chang, Shi-Kuo (1986). Vicon. In S. Chang, T. Ichikawa, P. A. Ligomenides (Eds.), *Visual Languages* (pp. 151-190). New York: Plenum Publishing.

Cooper, L. A., Shepard, N. R. (1984, December). Turning something over in the mode, *Sci Am.* (pp. 106-114).

Cornwell, M. R. & Jacob, R. J. K. (1984). Structure of a rapid prototype secure military message system. *Seventh DOD/NBS Computer Security Conference.* (pp. 48-57). Gaithersburg, Maryland.

Dale, Nell & Weems, Chip (1987). *Introduction to Pascal and Structure Design,* Heath Publishing Company.

Finzer, W. & Gould, L. (1984, June). Programming by rehearsal. *Byte 9,* (6).

Gardner, H. E., Zurif, B., Berry, T., & Baker, E. (1976). Visual communication in aphasia. *Neuropsychologia, 14,* 275-292.

Glinert, E. P. & Tanimoto, S. L. (1984, November). Pict: An interactive graphical programming environment. *Computer, 17* (11), 7-25.

Grafton, Robert B. & Ichikawa, Tadao (1985, August). Visual Programming. *Computer, 18* (8), 6-9.

Herot, C. F. (1980). Spatial management of data. *ACM Trans. Database Syst. 5* (4), 493-514.

Isoda, S., Shimomura, T., & Ono, Y., (1987, May). VIPS: A visual debugger. *IEEE Software, 4* (3), 8-19.

Jacob, R. J. K. (1985, August). A state transition diagram language for visual programming. *IEEE Comput., 18* (8), 51-59.

Jacob, R. J. K. (1985). An executable specification technique for describing human-computer interaction. In H. R. Hartson(Ed.), *Advances in Human-Computer Interaction,* (pp. 211-242), Norwood, New Jersey: Ablex Publishing Co.

Jacob, R. J. K. (1985). Designing a human-computer interface with software specification techniques. In J. C. Agrawal and P. Zunde (Eds.), *Empirical Foundations of Information and Software Science,* (pp. 129-156). New York: Plenum Press.

Jones, S. H. (1983). Stereotype in pictograms of abstract concepts. *Ergonomics, 26,* 605-611.

Kitagawa, H., Gotoh, M., Misaki S., & Azuma, M. (1984, June). Form document management system SPECDOQ - Its architecture and implementation. *Proceedings of the Second ACM conference on Office Information Systems*, 132-142.

Korfhage, R. R. & Korfhage, M. A. (1986). Criteria for iconic languages. In S. Chang, T. Ichikawa, P. A. Ligomenides (Eds.), *Visual Languages* (pp. 207-232). New York: Plenum Publishing.

Kramlich, D. (1984). Spatial data management on the USS Carl Vinson. *Database Eng. 7* (3), 10-19.

Lakin, Fred (1986). Spatial parsing for visual languages. In S. Chang, T. Ichikawa, P. A. Ligomenides (Eds.), *Visual Languages* (pp. 35-86). New York: Plenum Publishing.

Lakin, Fred (1980, July). A structure from manipulation for text-graphic objects. *Proceedings of SIGGRAPH '80,* Seattle, Washington.

Lakin, Fred (1980, August). Computing with text-graphic forms. *Proceedings of the LISP Conference at Standford University.*

Letovsky, Stanly & Soloway, Elliot (1986). Delocalized plans and program comprehension. *IEEE Software*, 3 (2), 41-49.

Lodding, K. N. (1982). Iconics - A visual man-machine interface. *Proc. Nat'l Computer Graphics Assoc., 1,* (pp. 221-233). NCGA, Farifax, Va.

London, Ralph L. & Duisberg, Robert A. (1985, August). Animating Programs Using Smalltalk. *Computer, 18* (8), 61-71.

Luo, D. & Yao, S. B. (1981, June). Form operation by example - A language for office information processing. *Proceeding of SIGMOD Conference* (pp. 213- 223).

Malone, T. W. (1980). *What makes things fun to learn? A study of intrinsically motivating computer games.* Phd dissertation, Dept. of Psychology, Stanford University, Stanford, Calif.

Monden, N. Y. Yoshino, Hirakawa, M., Tanake, M., & Ichikawa, T. (1984). HI-VISUAL: A language supporting visual interaction in programming. *Proceedings of the 1984 IEEE Computer Society Workshop on Visual Languages* (pp. 199-205). Hiroshima, Japan.

Montalvo, F. S. (1986, June). Diagram understanding: Associating symbolic descriptions with images. *Proc. of Second IEEE Workshop on Visual Languages*, Dallas.

Moriconi, Mark & Hare, Dwight F. (1985, August). Visualizing program designs through PegaSys. *Computer, 18* (8), 72-85.

Moriconi, Mark & Hare, Dwight F. (1986, October). The PegaSys system: Pictures as formal documentation of large programs. *ACM Transaction on Programming Languages and Sys., 8* (4), 524-546.

Nassi, I. & Shneiderman, B. (1973, August). Flowchart Techniques for Structural Programming. *SIGPLAN Notices, 8* (8), 12-26.

Nicholas, J. M. (Ed.) (1977). *Images, Perception, and Knowledge.* D. Reidel, Dodrecht.

Paivio, A. (1977). Images, Perception, and Knowledge, in Ref. Nicholas (1977)

Pong, M. C. & Ng, N. (1983). PIGS - A system for programming with interactive graphical support. *Software Practice Experience, 3* (9), 847-855.

Powell, M. L. & Linton, M. A. (1983, June). Visual abstraction in an interactive programming environment. *Proc. Sigplan Symp. Programming Language Issues in Software Systems 83,* printed as *Sigplan Notices, 18* (6), 14-21.

Programming with DOMAINS 3D Graphics Metafile Resources. (1985). Apollo Computer Inc., Massachusetts: Chelmsford.

Purvy, R., Farrell J., & Klose, P. (1983). The design of Star's records processing: Data processing for the noncomputer professional. *ACM Trans. Office Inf. Syst., 1* (1) 3-24.

Reader, G. (1984, November). *Programming in pictures* (Tech. Rep. No. TR-84-318). University of Southern California.

Reader, George (1985). A survey of current graphical programming techniques. *IEEE Comput. 18* (8) 11-25.

Reiss, S. P. (1984, May). Graphical program development with PECAN program development systems. *Proc. ACM Sigsoft-Sigplan Software Engineering Symp. Practical Software Development Environments,* April 1984, printed as *Sigplan Notices, 19* (5), 30-41.

Reiss, S. P. (1985). PECAN: Program development systems that support multiple views. *IEEE Trans. Software Eng., 11* (3), 276-285.

Rohr, Gabriele (1986). Using Visual Concepts. In S. Chang, T. Ichikawa, P. A. Ligomenides (Eds.), *Visual Languages* (pp. 325-348). New York: Plenum Publishing.

Rosch, G. & Lloyd, B. B. (Eds.). (1978). *Cognition and Categorization,* Lawrence Earlbaum Associates. New Jersey: Hillsdale.

Roussopoulos, N. & Leifker, D. (1984). An introduction to PSQL: A pictorial structured query language. *Proceedings of the 1984 IEEE Computer Society Workshop on Visual Languages*, (pp. 77-87). Hiroshima, Japan.

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

Shu, Nan C. (1986). Visual programming languages: A perspective and a dimensional analysis. In S. Chang, T. Ichikawa, P. A. Ligomenides (Eds.), *Visual Languages* (pp. 11-34). New York: Plenum Publishing.

Smith, D. C. (1975). *Pygmalion: A creative programming environment. Department of Computer Science* (Tech. Rep. No. STAN-CS-75-499). Stanford University.

Smith, D. C., Irby, C., & Kimball, R. (1982). The star user interface: An overview. *Processings of National Computer Conference*, (pp. 515-528).

Shneiderman, B., & Mayer, R. (1979) Syntactic/semantic interactions in programmer behavior: a model and experimental results. *International Journal of Computer and Information Sciences*, 8, 219-238.

Soloway, Elliot, Bonar, Jeffrey & Ehrlich, Kate (1983, November). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM.* 26 (11), 853-860.

Soloway, Elliot, Pinto, Jeannine, Letovsky, Stan, Littman, David, & Lampert, Robin (1988, November). Designing documentation to compensate for delocalized plans.*Communications of the ACM.* 31 (11), 1259-1267.

Steele, R., Illes, J., Weinrich, M., & Lakin, F. (1985, June). Towards computer-aided visual communication for aphasics: Report of studies. *Rehabilitation Engineering Society of North America 8th Annual Conference.* Memphis, Tenn.

Tanimoto, Steven L. & Runyan, Marcia S. (1986). PLAY: An iconic programming system for children. In S. Chang, T. Ichikawa, P. A. Ligomenides (Eds.), *Visual Languages* (pp. 191-206). New York: Plenum Publishing.

Tenenbaum, Arron M. & Augenstein, Moshe J. (1986). *Data Structures Using Pascal*, Prentice-Hall: Englewoods Cliff Publishing Company.

Weber, Robert J. & Kosslyn, Stephen M. (1986). Computer graphics and mental imagery. In S. Chang, T. Ichikawa, P. A. Ligomenides (Eds.), *Visual Languages* (pp. 286-304). New York: Plenum Publishing.

Webster's New Twentieth Century Dictionary. (1983). Unabridged.

Yao, S. B., Hevner, A. R., Shi, Z. & Luo, D. (1984). FORMANAGER: An office forms management system. *ACM Trans. Office Inf. Syst.,2* (3), 235-262.

Yoshino, Y., Yoshimura, H., Nishi, N., Tanaka, M. & Ichikawa, T. (1985, November). A hierarchical model for multiple window processing. *Proc., IEEE Workshop on Languages for Automation,* (pp. 26-31).

Zloof, M. M. (1981, May). QBE/OBE: A language for office and business automation. *Computer*, 13-22.

Zloof, M. M. (1982). Office-by-Example: A business language that unifies data and word processing and electronic mail. *IBM Systems J., 21* (3), 272-304.

Appendix A

```
program example( input, output );
        var
                i,j                     : integer;
                row, grandtotal         : real;
                col                     : array ( 1..20 ) of real;
                a                       : array ( 1..20, 1..20 ) of real;

procedure  proc1 ( a : integer );
        var  i : integer;
        begin
                for i := 1 to 20 do
                begin
                                        a := a * 2;
                                        if i > j
                                           then  i := j;
                end;
        end;

begin
        for i := 1 to 20 do
                if j = 1 then
                begin
                                        proc1( row );
                                        a(i,j) := i * j;
                end
                else
                begin
                                        proc1( row );
                                        j := 1;
                end;
        case i of
                1: proc1( row );
                2: proc1( grandtotal );
                3: proc1( 99 );
                4: proc1( j );
        end;
end.
```

Program A

```
program treepgm(input,output);
  type
     nodeptr  =  ^nodetype;
     nodetype   = record
                        info: integer;
                        left,right: nodeptr;
                        rthread: boolean;
                        end;
  var
     p,q,tree:  nodeptr;
     number:  integer;

procedure inorderthread(tree:nodeptr);
  var  p,q: nodeptr;
  begin
     p := tree;
     repeat
          q := nil;
          while p<> nil do { travers left }
             begin       q := p;
                         p := p^.left;
             end;  { while }
          if q <> nil
             then  begin
                writeln( q^.info );
                p := q^.right;
                while ( q^.rthread) and
                          ( p<> nil )
                   do begin
                         writeln( p^.info );
                         q := p;
                         p := q^.right;
                   end; { while }
             end; { then }
     until q = nil;  { repeat }
  end; { procedure inorderthread }

procedure inorderrecursive( tree: nodeptr);
  begin
     if tree <> nil
          then begin
             inorderrecusrive( tree^.left );
             writeln( tree^.info );
             inorderrecursive( tree^.right );
          end;  { then begin }
  end; { procedure inorderrecrusive }

function makenode( x: integer ): nodeptr;
  var     p:  nodeptr;
  begin
     new( p );
     p^.info := x;
     p^.left := nil;
     p^.right := nil;
     p^.rthread := true;
     makenode := p;
  end;  { function makenode }

procedure setleft (p : nodeptr; x: integer);
  var     q:  nodeptr;
  begin
     if (( p^.left <> nil ) and ( p= nil ))
          then writeln('ERROR left insert ')
     else  begin
          q := makenode( x );
          p^.left := q;
          q^.right := p;
     end;  { else begin }
  end;  { procedure setleft }

procedure setright(p:nodeptr; x:integer);
  var     q,r:  nodeptr;
  begin
     if (( p=nil ) or ( not p^.rthread ))
          then  wirteln('ERROR right insert')
     else begin
          q := makenode( x );
          r := p^.right;
          p^.right := q;
          p^.rthread := false;
          q^.right := r;
     end;  { else begin }
  end;  { procedure setright }

begin     { Main program }
  readln( number );
  tree := makenode( number );
  while not eof
     do begin
          readln( number );
          p := tree;
          q := tree;
          while ( number <> p^.info)
                and ( q <> nil )
             do begin
                p := q;
                if number < p^.info
                   then       q := p^.left
                   else  q := p^.right;
             end;  { do begin }
          if number = p^.info then
              writeln( number, 'isduplicate')
          else  if number < p^.info
                then      setleft( p,number)
                else      setright(p,number);
     end;  { while do begin }
     inorderthread( tree );
     inorderrecursive( tree );
end. {{program tree }
```

Program B

```
program t1 ( input, output );
    const   Maxlength   = 100; TRUTH = true; outlabel = 'TOP OF HEAP';
    type  nameary       =  array ( 1..20 ) of char;
          emplyrec      =  record
                             name:   nameary;
                             age:    integer;
                             sex:    char;
                             married:        boolean;
                             link:   ^emplyrec;
                           end;
          biggy         =  array ( char, 1..200, boolean ) of real;
          charset       =  set of char;
          color         =  ( red, green, orange );
          colorset      =  set of color;
          employees     =  packed array ( color ) of emplyrec;
          biggyary      =  array ( char ) of biggy;
          ptrint        =  ^integer;
          ptrary        =  array (1..20) of ptrint;
          ptrrec        =  ^emplyrec;
          ptrrecary     =  array ( char ) of ptrrec;

    var   i,j,k                 :  integer;
          row, grandtotal       :   real;
          col                   :  array ( 1..20 ) of real;
          a                     :  array ( 1..20, 1..20 ) of real;
          head                  :  ^emplyrec;
          line                  :  nameary;
          paint,newcolor        :  color;
          paints                :  colorset;
          emplylist             :  ptrrec;


procedure linkproc ( a: integer; tree:ptrrec );
  var     i       :   integer;
          p       :   ptrrec;
          temp    :   nameary;
  begin
        p := head;
        while p <> nil
          do begin
               for i := 1 to 20 do
                  p.name(i) := ' ';
                  p := p.link;
          end;
  end;

function funcolor ( brush   : colorset;
                    cnt     : integer ): color;
  type  shades      =   ( tan, pink );
  var   color1      :   color;
        shade1      :   shades;
        i           :   integer;
  begin
        {....}
        color1 := funcolor( paints );
  end;


begin  { Main Program }
       readln( line );
       for k := 1 to 20 do
             linkproc( k, head );
       paints := [];
       newcolor := funcolor( paints );
end.
```

Program C

Appendix B

Evaluation Questions

Subject Data: Sex: M ____ F ____

Undergraduate Fr___ Soph___ Jr___ Sr___ Graduate ____

Circle one answer for each question. "1" represents the most negative answer and "5" represents the most positive answer.

1. In general, how easy did you find using the Peec system in studying a program?
   1 2 3 4 5 (5 - Very easy)

2. Did you find using the menus and icons within the Peec system easy and conveniently to use?
   1 2 3 4 5 (5 - Very convenient)

3. Did you like the way the menus were set up for communicating with the Peec system?
   1 2 3 4 5 (5 - Very pleased)

4. Did you like using the mouse to control the three-dimensional model?
   1 2 3 4 5 (5 - Very pleased)

5. How easy was it to learn and remember the meanings of the icons identifying the control structures?
   1 2 3 4 5 (5 - Very easy)

6. How easy was it to learn and remember the meanings of the icons identifying the program's identifiers?
   1 2 3 4 5 (5 - Very easy)

7. Did the scope of the variables and their associated icons help in understanding the program?
   1 2 3 4 5 (5 - Very useful)

8. Did you like Peec's representation of the program's flow format using a three-dimensional model?
   1 2 3 4 5 (5 - Very pleased)

9. Were the tier positioning within the model helpful in understanding the nesting levels of the control structures?
   1 2 3 4 5 (5 - Very helpful)

10. Did the image of the program's execution flow model contribute to your understanding of how the program's control structures are to be executed?

11. Would the Peec system assist you in developing an understanding of an algorithm's implementation rather than just using a listing of the program to develop this understanding?
    1 2 3 4 5 (5 - Assist greatly)

12. In the first programming courses you study, would the Peec system have been useful in understanding how nested structures operate and how procedures and functions are executed.
    1 2 3 4 5 (5 - Very helpful)

13. Would this system be helpful in teaching programming concepts?
    1 2 3 4 5 (5 - Very helpful)

14. Would this system be useful in studying larger programs in which the program listing would be several pages long?
    1 2 3 4 5 (5 - Very useful)

15. Would this system be useful to an individual who is experienced in programming?
    1 2 3 4 5 (5 - Very helpful)

16. How useful would this system be in developing understanding of a program?
    1 2 3 4 5 (5 - Very useful)

17. How would you compare using the Peec system in studying a program verses using a source listing of the program in developing an understanding of how an algorithm is implemented?
    1 2 3 4 5 (5 - More useful)

18. Would you use a system like Peec, if it were available, in studying your programs?
    1 2 3 4 5 (5 - Would like to use)

19. What is your overall evaluation of the Peec system?
    1 2 3 4 5 (5 - Very pleased)

Any Comments:

VITA

Brady R. Rimes is an Assistant Professor at the University of Southern Mississippi in Hattiesburg Mississippi. He has been married to Pamela for 12 years and has two children, Toby and Tyson. He received two B.S. Degrees from the University of Southern Mississippi in 1974, one in Computer Science and one in Mathematics, and an M.S. degree in Computer Science in 1978. Mr. Rimes has taught Computer Science for 12 years in the University and industrial environment. His research interests are in compiler design, operating systems, and programming languages. Mr. Rimes' plans are to continue working at the University of Southern Mississippi after graduating from Louisiana State University.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Brady R. Rimes

Major Field: Computer Science

Title of Dissertation: A Program Visualization System That Supports The Program Understanding Process

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

April 28, 1989