GRASP:
A VISIBLE AND MANIPULABLE MODEL FOR PROCEDURAL PROGRAMS

by

Franklyn Albin Turbak

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1986

Signature of Author.....
Department of Electrical Engineering and Computer Science,
May 9,1986

Certified by.....
Andrea A. diSessa
Thesis Supervisor

Certified by.....
D. Austin Henderson, Jr.
Company Supervisor, Xerox Palo Alto Research Center

Accepted by.....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

GRASP:
A VISIBLE AND MANIPULABLE MODEL FOR PROCEDURAL PROGRAMMING

by

Franklyn Albin Turbak

Submitted to the
Department of Electrical Engineering and Computer Science
on May 9, 1986
in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science
in Electrical Engineering and Computer Science.

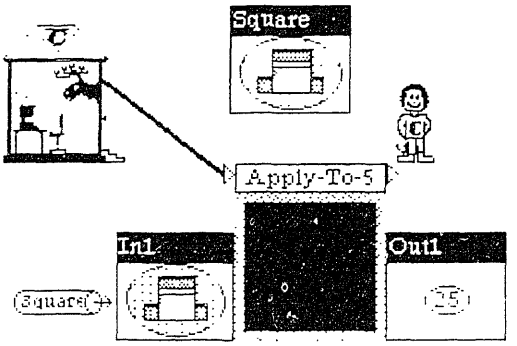
ABSTRACT

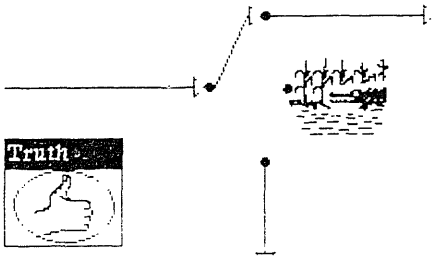
Grasp is a computational model and programming system designed to help novices reason about the structure of procedural programs. Text-based programs in traditional programming languages poorly reflect the rich structure of the underlying model of computation. Grasp makes the elements and processes of the computational model more accessible to the programmer by adhering to two principles: *visibility* - the programmer should see graphical representations of a program's structure; and *manipulability* - the programmer should interact with the elements of a program as if they were physical objects. This report motivates these two principles and describes how they are applied to the development of a computational model and programming system for procedural programs.

Thesis Supervisors:

Dr. Andrea A. diSessa (MIT)
Senior Research Scientist

Dr. D. Austin Henderson (Xerox Palo Alto Research Center)
Research Staff Member

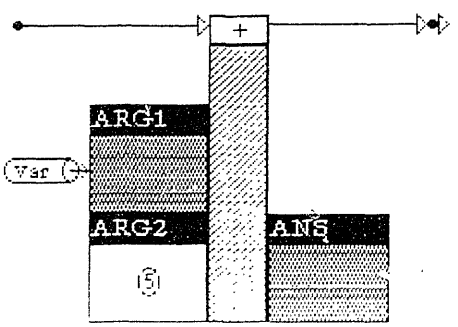


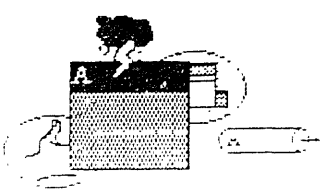


GRASP:

A VISIBLE AND MANIPULABLE MODEL FOR PROCEDURAL PROGRAMS

Franklyn Turbak





ACKNOWLEDGMENTS

If brevity is the soul of wit, then I must be witless. Despite the best efforts of my teachers since childhood, I continue to be plagued by verbosity. The length of this document is evidence of this affliction, as is the length of these acknowledgments. Whereas most people can get by with a few sentences, I require three pages! I hope that the length of this section does not detract from the significance of the individual contributions made by those mentioned herein.

A thesis is an interesting test of friendships. Only friends will put up with the singlemindedness of purpose that grows in the thesis writer and overtakes him as the due date looms ominously nearer. The best of friends will even help him out in his plight.

This experience has taught me that I have many friends. To express my gratitude in this small section of the thesis hardly absolves me of my debt to them. I can probably never repay them for their contributions and kindness, but I hope that I can at least spread the good will they have bestowed upon me by similarly helping out others in the future.

I begin with my advisors, Andy diSessa and Austin Henderson. I am lucky to have chosen two advisors who, in the end, wanted me to graduate even more than I did. Andy put up with an amazing amount of lameness on my part over the past few years and was kind enough to continue as my advisor when he skedaddled off to Berkeley. I thank him for letting me explore structural models even when he knew in his heart that functional models were more appropriate. I would not have appreciated the limitations of structural models if I didn't discover them on my own.

Austin is my hero. It is hard to imagine a more helpful and dedicated advisor. He has been a constant source of ideas, advice, and support over the four years I have known him. His enthusiasm and encouragement lifted my spirits on several occassions when I seemed to be experiencing the nadir of my life. For this, I am forever grateful.

I thank Hal Abelson and Gerry Sussman for developing a wonderful introductory course to computer science. Without their course, my interest in computation and education might never have developed.

Mike Eisenberg, Ellie Long, and Mitch Resnick were dear companions and valuable proof that interest in education still exists at MIT. I am endebted to Mike and Ellie for undertaking the ordeal of a 6.003 TA-ship with me during this past term. Mike had the knack of making me laugh during times when a sane person would otherwise be going crazy. He prepared me for all the *Leave it to Beaver* questions I could expect on my orals (though unfortunately my orals committee decided to concentrate on the *Addams family* instead).

The LENS group at the MIT Sloan School provided me with tremendous support when the chips were down. Tom Malone, the nicest professor imaginable, gave me a summer job. Ken Grant, Dave Rosenblitt, Ramana Rao, and Kum-Yew Lai all helped to improve the quality of this report and the orals talk based on it. Ken's formatting, Dave's proofreading, and Ramana's suggestions helped keep this document off the "endagered theses" list (Dave's joke). Ken was especially kind enough to let me commandeer his carrel and workstation for over a month straight during the final prepartion of this document. He also kept me going by continually repeating that I was a "lean, mean writing machine."

Now for the "clowns out West": Dave Chiang, William Lee, and Andy Litman. Not only were they the best of friends at MIT, but they were wonderful housemates in California. They provided me with megahours of great conversation and carted me around everywhere in their cars. Special thanks go to Dave, my alter ego, who helped me get through some awfully rough nights at Xerox PARC. Dave is the creator of many of the icons in the Grasp system. The moosehead in the controller house is a stroke of his brilliance. And let us not forget his advice on Lisp syntax: "Parentheses are like small bananas!"

The "clown out East", Robert Kwon, was essential to my survival during my past year at MIT. Where would I be now without his continual advice for me to "reach out with my feelings" and his constant reminders that life is but a "dream within a dream"? Who would have thought when we met at an eighth grade spelling bee that ten years later he would stay up late at night to correct spelling mistakes in my thesis? Cheers, R.K!

Many other people deserve special mention. I am forever endebted to Ram Sundaram, who took up the slack when I fell behind in my 6.003 TA duties. I especially want to thank him and Keith Nabors for grading Problem Set 6 for me. David Levy not only selflessly helped me prepare my PARC talk, but he would later provide me with dental floss. I thank him. My teeth and gums thank him. Kathy Takayama, Mike Dawson, Chee-Seng Chow, Judy Litman, Dan Frost, and Maya Paczuski helped me get through difficult times with their friendship. And who could forget Chab, Weed, Slim, and Stearn? Quid?

I owe my biggest debt to my family. Mom and Dad not only have provided me with moral and monetary support for the past six years, but they have showered me with love and affection since birth. I love them. I hope that someday

I can be as good a parent for my children as they have been for me; my greatest fear is that I will be punished by having children who will be as unappreciative of me as I have been of my parents. I also give my love to my brother, Stephen, and his wife, Michelle. They continue to invite me to their home even though I burn their potholders and dust their kitchen with flour.

". . . a speck of water on a desert cactus no longer covered by shade."

- Robert Osong Kwon -

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Grasp is a computational model and programming system designed to help novices reason about the structure of procedural programs. Text-based programs in traditional programming languages poorly reflect the rich structure of the underlying model of computation. Grasp makes the elements and processes of the computational model more accessible to the programmer by adhering to two principles: *visibility* - the programmer should see graphical representations of a program's structure; and *manipulability* - the programmer should interact with the elements of a program as if they were physical objects. This report motivates these two principles and describes how they are applied to the development of a computational model and programming system for procedural programs.

Whereas most programming languages try to exploit the user's familiarity with natural language, Grasp attempts to appeal to his physical intuitions. People generally have well-developed skills for inspecting and manipulating interconnected sets of objects. To take advantage of these skills, Grasp embraces what diSessa has called a *device programming style* [diSessa 86b]. Programming in Grasp consists of "wiring" together computational devices with control and data paths; these devices then serve as the run time structures for actually executing the program. When reified computational elements are made accessible in this manner, programmers can use their familiarity with physical systems to reason about the structure of programs. The name *Grasp*, in fact, is intended to reflect this approach; it suggests a relationship between physical manipulation (grasping an object) and understanding (grasping a concept).

The Grasp project consists of three distinct parts:

1. Probing the problems that novices have in understanding the structure of procedural programs.

2. Developing a model for procedural programs which addresses these problems by including features for supporting structural understanding.
3. Building a programming system based on this model.

The first two parts are reported here. A prototype system is being built, but its implementation details are not relevant to this report.

In its current state of development, the rudimentary Grasp programming system serves mainly as a pedagogical tool for illustrating the Grasp model. Much more work is necesary to extend the system to the point where it can actually be used for nontrivial programming tasks. The model, however, is the real crux of the project. Applying the principles of visibility and manipulability not only to the interface but to the model itself allows procedural programs to be viewed as collections of interconnected physical devices. This report contends that such a view offers a powerful alternative to traditional expression-oriented models for reasoning about the structure of programs.

1.1 MOTIVATIONS

My motivations for undertaking the Grasp project are intimately intertwined with the introductory course in computation offered by MIT's Electrical Engineering and Computer Science department.[1] Based on Abelson and Sussman's *Structure and Interpretation of Computer Programs*, the course is an intensive introduction to the major ideas of computer science. The course moves at a rapid pace, covering such topics as procedural abstraction, data abstraction, modularity and state, functional programming, object-oriented programming, logic programming, interpreters, and compilers. Students have five hours of class contact with the course each week - two in a large (about 400 people) lecture, two in a medium-sized (about 30 people) recitation, and one in a small (4 or so people) tutorial. There are weekly programming laboratories, each of which involves understanding and extending a nontrivial software system. Typical laboratories include a computer psychiatrist, a system for creating simple Escher-like pictures, and an adventure game.

The programming language used to illustrate the high-level concepts covered in the course is a dialect of Lisp known as Scheme. Free of the excess baggage of most programming languages, Scheme derives an extraordinary

power and elegance from a small kernel of tightly interwoven features. The most important of these - first-class procedure objects, lexical scoping, and tail recursion - will be discussed in Chapter 2. A fair number of small Scheme programs are used as examples throughout this report. (The reader unfamiliar with Scheme is encouraged to look at Abelson and Sussman's text [Ableson & Sussman 85a] for an excellent introduction to the language.) The version of Scheme used in this report is the same as that used in the MIT course; it differs in a few respects from the standard Scheme as described in [Clinger 85].

Students often refer to Abelson and Sussman's course as "the Scheme course," but this does not reflect the spirit of what they teach. The focus of the course is not on a particular programming language, but on controlling the complexity of large systems. In this way, computer science is introduced as a field in which abstract engineering principles are examined in their pure form. The Scheme language is not an end in itself, but a vehicle for exploring the kinds of high-level issues raised in the course. For this reason, few hours of class time are devoted to details about the language or the system on which it is implemented. Students are expected to become familiar with these on their own as the course progresses.

Though the instructors like to think that the course has little to do with the Scheme language, the students tend to disagree. For them, the language and its supporting programming environment are a very important part of the course. Certainly it is difficult to grasp the high-level ideas if one cannot understand the examples that illustrate them. Moreover, students typically spend far more time reading and writing Scheme code and sitting in front of a computer then they do attending class or reading the text. There is little wonder that students associate the course with Scheme rather than "controlling the complexity of large systems." Keeping the student's point of view in mind, I will refer to the course as "the Scheme course" in the remainder of this report.

The mechanics of the Scheme language and system pose a potential source of confusion and frustration for students. There are, of course, those students who experience few difficulties. On the other hand, there are those who, at the end of the course, still do not feel comfortable with the language or the system. Most common, perhaps, are the students who begin the course with an array of misconceptions that are gradually cleared up during the term. This is not to say that Scheme is an exceptionally difficult language to acquire. Undoubtedly, a

main problem is a lack of time on the part of the students; as in any university setting, time is a scarce commodity at MIT, and students simply cannot afford the time to absorb all of the material in which they are immersed. However, the Scheme course itself is also a culprit in the case of the confused student. The superficial attention paid to the details of Scheme in the course can be frustrating for many. As explored later in Chapter 4, despite Scheme's elegance and simplicity, there are a host of pitfalls that novices can encounter when learning the language. Requiring them to master the language largely on their own only increases the probability that they will fall into some of the common traps.

I have been involved with the Scheme course in various capacities over the past six years: as a student, as a one-on-one tutor, and three times as a teaching assistant (two terms at MIT and a special course at MIT's Lincoln Laboratories). The duties of a teaching assistant include teaching tutorials and occasional recitations, grading problem sets, and providing students with extra help during office hours. Through these experiences, I have gained considerable insight into the problems students have with the Scheme language. It helps to have seen the course from the perspectives of both student and teacher. All too often instructors understand material so well that they forget what it was like *not* to understand it. I have not forgotten. I still vividly recall many of the confusions and misconceptions I had when I took the course. Many of these were not cleared up until I actually taught the course. I'm sure there are a few more undiscovered misconceptions waiting to surprise me someday.

My high-level goal in undertaking the Grasp project was to use my experiences with the Scheme course as a basis for developing new ways to present procedural programming to novices. Through my teaching endeavors, it has become obvious to me how important presentation is in conveying ideas. As I will argue in Chapter 4, many of the problems novices have with Scheme are rooted in the way it is presented to them. Scheme code in itself evokes few images of the state and behavior of the process it describes. These can only be understood via various explicit models of evaluation taught in class. Yet, in many instances, these models are too incomplete, confusing, or low-level to help the student gain a robust understanding of procedural programs. I believe that it is possible to develop more effective models for presenting computational elements and processes.

In this project I chose to focus in particular on the presentation of *procedures*. Procedures as first-class objects[2] and the environment and control issues related to procedures represent fertile ground for misconceptions in the Scheme course, especially during the first few weeks. Part of the problem is that the abstract nature of procedures makes them particularly hard to visualize. I once had the desire to go to a Halloween party dressed as a procedure object, but I didn't have the faintest idea of what I should wear. What does a first-class procedure object look like?

I developed the Grasp model and system largely in an attempt to answer this question. Its key feature is the way it reifies many of the implicit structures associated with procedures and presents them to the programmer as visible, manipulable objects. Although Grasp is *not* Scheme, it supports similar kinds of structures and is a useful tool for reasoning about many of the concepts in Scheme. In fact, I have already successfully integrated some of the ideas and notations from the Grasp model into my tutorial material for the Scheme course.

1.2 WHERE DOES GRASP FIT IN?

The Grasp project has ties to three major fields of study: mental models, novice-oriented programming, and user interfaces. In this section I attempt to put Grasp in perspective by describing its association with the three fields and discussing some of the related work in each area.

1.2.1 Mental Models

In cognitive science, the phrase *mental model* is used to refer to the internal mental representation people presumably use to think about a subject area. The notion of mental models is particularly slippery to formalize or even talk about. Much of the problem is due to the fact that it is impossible to directly access the mental representations appearing in an individual's mind. We must settle for indirect methods, such as listening to people verbalize their thought processes or observing subjects while they perform some task. However, people are not always conscious of their models, so the way they *say* they think does not necessarily describe the way they *actually* think. Furthermore, an observer's own models concerning the subject and the task interfere with an objective appraisal of the

subject's models. On top of all this, mental representations are not likely to be monolithic, consistent, and well-structured entities; more likely, they are fragmented into many loosely connected pieces, many of which may be ill-defined or inconsistent. This is a basic tenet of diSessa's idea of "Knowledge in Pieces" [diSessa 85b].

Despite the issues raised above, mental models are still a useful concept for explaining the kinds of misunderstandings people encounter in a subject area. For example, Brown, Burton, and Van Lehn have shown in several studies that a large number of errors in subtraction made by elementary school students are attributable to a small set of "bugs" in the steps of the subtraction algorithm they use [Brown & VanLehn 80] [Burton 81]. Such a finding has important consequences for education. Children having difficulty with subtraction might well be following a logical and well-defined sequence of steps, albeit incorrect in some minor way. Rather than labelling such children as having "poor math ability," a more fruitful approach is to diagnose the bugs they have and investigate ways to clear up their misunderstandings. This approach - analyzing poor mental models and then helping to diagnose and fix them - is broadly applicable to many areas of learning.

Recent research has explored people's mental models in many domains. A line of studies known as *naive physics* or *intuitive physics* has probed the way people understand such topics as motion, gravity, electricity, and heat flow.[3] Another area of exploration is human understanding of physical devices. User models of calculators are explored in a number of works, including [Young 81], [Young 83], [Norman 83] and [Halasz 84]. Dekleer and Brown look at ways people might build models of devices like buzzers [deKleer & Brown 83]. Xerox's Operability Project [Henderson 84] focuses on the mental models users form of a machine based on the interface through which they interact with it.

The Grasp project is related to a category of mental models research which might best be called *naive computation*. The goal of naive computation is to investigate the way novices think about computers, especially by studying the kinds of problems they encounter when learning a new programming language. Early studies in this area tended to concentrate on the syntactic issues of the programming languages (e.g. semicolon placement, choice of keywords, and specification of conditionals); a good overview of these studies can be found in [duBoulay & O'Shea 81]. Recent research has evolved to focus on cognitive issues

and models of computation. Kahney, for example, explores the problems novices experience with recursion [Kahney 82], while Soloway, Bonar, and Ehrlich examine naive models of looping [Soloway *et al.* 83]. Mayer has done several experiments to observe the effect of teaching explicit models of computation to novice programmers [Mayer 81]. Strassman's efforts in cataloguing the problems students experience in the Scheme course is particularly relevant to this project [Strassman 84].

An important part of the Grasp project is to explore the way novices think about procedures. Through my experiences as a teaching assistant, I have extensively observed novice programmers in tutorials and in the computer laboratory. These observations are the basis for my theories on student models of procedures in Scheme. The details of these theories are given in Chapter 4. The evidence upon which the theories are based is anecdotal; no formal experiments were performed to derive them. However, I don't believe that the lack of formality is a severe drawback in this business. The intuitions one develops as a teacher are fairly strong and are often corroborated by the intuitions of other teachers. Although the intuitions may not always be on the mark, failure to harness them because of a lack of formal evidence is debilitating to the venture of improving education. Intuitions about novices' misconceptions have played a crucial role in this project; it is difficult to design a system for helping novices unless one first has a good feel for the source of their confusions.

1.2.2 Novice-Oriented Programming

Given that current programming systems present a great barrier for novices, a natural path to explore is the creation of systems which simplify programming for novices in some way. These efforts, which I group under the heading *novice-oriented programming*, follow two main directions:

1. Making programming more *accessible* to a wider audience by insulating the novice from the complexities of conventional programming languages.
2. Making models of computation more *visible* to the novice by exposing the traditionally hidden state and behavior of a computation.

I shall discuss these directions in turn.

1.2.2.1 Accessible Programming

One approach to novice-oriented programming is to invent new programming styles which increase the accessibility of programming to nonprogrammers and novice programmers. A common method is to provide some type of intermediary between the programmer and the raw model of computation. The most frequently employed methodology is called *programming by demonstration*, in which the user somehow demonstrates (usually through an interactive, graphical interface) what the desired program is supposed to do. Gould and Finzer's Programming by Rehearsal is a wonderful example of such a system; it embodies a theater metaphor to help curriculum designers build their own graphics-oriented applications. This system automatically writes programs by "watching" the designers "rehearse" various "performers" which are represented by graphical objects in the Rehearsal World [Gould & Finzer 84].

A variant on programming by demonstration is *programming by example*, in which a system automatically constructs a program based on a user's examples. The first system embodying this approach was PYGMALION [Smith 75], in which the user steps through concrete examples of computations by manipulating icons. If the system is in "remembering mode", it automatically constructs a program based on the user's example. Halbert's SmallStar similarly creates a program based on concrete examples [Halbert 84]. Curry's Programming by Abstract Demonstration is a different methodology in which the user demonstrates operations on abstract classes of objects rather than concrete ones [Curry 78]. Yet another approach is taken in TINKER [Lieberman 82], where programmers use concrete examples to aid them in visualizing the Lisp programs they are defining. This style is more programming *with* examples than programming *by* example.

Another way to achieve accessibility is to build systems which take advantage of people's experiences in the world. A major reason for the popularity of spreadsheet systems is that they bear great similarity to the accounting ledgers commonplace in the business world. The LENS system uses people's familiarity with filling out forms to help them specify filtering mechanisms for their electronic mail [Malone *et al.* 86]. Xerox's Star Information System embraces a physical-office metaphor to represent the objects and functions of a document preparation system. The system is designed so that users' familiarity with objects

in the office will help them in their interactions with Star's "desktop." [Smith *et al.* 82]

Our wealth of experiences with physical systems is a motivation for building environments in which programs can be created by interconnecting graphically represented objects. DiSessa refers to this style of programming as *device programming* [diSessa 86b]. The data-flow programming system developed by Sutherland was an early example of device programming. In Sutherland's system, the programmer wires together functional units with state-maintaining data paths to specify a data-flow computation [Sutherland 66] More recently, Robot Odyssey has employed device programming techniques to embed fundamental ideas of logic and computation in an exciting educational game [Dewdney 85].

1.2.2.2 Visible Programming

A second approach to novice-oriented programming is to keep traditional computational models but to supply users with a more revealing interface. A major problem with traditional programming languages is that programs and data are represented as textual expressions which provide little insight into the state and behavior of the underlying computational model. A desirable goal, as suggested in [duBoulay *et al.* 81], is to transform the traditional "black box" models of the computers into "glass box" ones which allow the programmer to see the normally hidden state and behavior of a high-level computational model. This notion is captured in a principle they call *visibility.*

The idea of visibility has been incorporated into programming systems in several different ways. Some systems, which Myers refers to as *visual programming systems* [Myers 86], have used it to make the *specification* of a program more graphical. In AMBIT/G, for example, data structure operations are expressed by pictures which show how to manipulate the data structures [Rovner & Henderson 69]. Other systems adhere to visibility by dynamically illustrating program execution or data structure manipulation; Myers uses the term *program visualization* for this approach. The Brown algorithm simulator (BALSA), for instance, graphically displays how algorithms manipulate data [Brown & Reiss 82].

Some visible systems provide for the integration of program definition and execution within one environment. Reiss's PECAN gives the programmer multiple syntactic and semantic views of programs and their execution [Reiss 84]. Both Boxer [diSessa 85a] and Eisenberg's extension, BOCHSER [Eisenberg 85], combine the functionalities of editor, interpreter, and file system into a single integrated system. Users are free to inspect and directly modify the state of this environment, which is uniformly represented in a spatial hierarchy of boxes on the screen.

Many of the systems discussed so far have been dubbed *graphical programming languages* or *visual programming languages* because of the graphical component they incorporate to make the state and behavior of programs visible to the programmer. The computer science community has developed a growing interest in such systems - witness that *I.E.E.E. Computer* recently devoted an entire issue to visual programming [Grafton & Ichikawa 85]. In many cases, however, the graphics are limited to only one part of these systems. Thus, while AMBIT/G allows graphical specification of programs, it does not show graphical execution; the BALSA system has the inverse property. Though Boxer and BOCHSER maintain state spatially in boxes, they are fundamentally text-based; boxes, in fact, are treated simply as large characters. Furthermore, the dynamic behavior of programs is hidden in these two systems - only the results of program execution are displayed.[4]

"Graphical programming" or "visual programming" are terms which better befit systems in which the creation, execution, and results of a program are all made visible through an interactive, graphical interface. Sutherland's data flow system satisfies these criteria. Not only does it allow the creation of programs by wiring together functional units with data lines, but when the resulting program is executed, the values on the data lines may be inspected [Sutherland 66]. In the Rehearsal World introduced above, the user "auditions" and "rehearses" performers via interactions with graphical representations of them. When a "production" is run, the behavior of any performer can be directly observed [Gould & Finzer 84]. Curry's prototype system supports the creation, review, and execution of a program within a single framework [Curry 78]. In a more recent effort, Glinert and Tanimoto's Pict/D allows programmers to build and run their programs in a wholly graphical environment - icons rather than text are used everywhere, even for variable names [Glinert & Tanimoto 84]. In all of these

systems, the graphical nature of programming facilitiates the creation and debugging of programs since the state and behavior of programs are visible to the user during all stages of the programmming task.

Grasp is a new entry into the class of graphical programming systems. It does *not* introduce a new methodology to isolate the nonprogrammer from the details and complexity of traditional programming languages. Rather, Grasp tries to make many of the implicit structures of traditional programs more visible to the novice programmer. To aid in accessiblity, Grasp *does* embrace a device programming style. Structures such as procedure activations and variables have graphical representations in Grasp which can be moved and interconnected in ways suggestive of interactions with physical objects. Textual names are allowed in Grasp, but they act purely as comments and have no semantic import. All semantics are determined by the structural interconnection of the graphically displayed objects.

Grasp is most closely related to graphical systems like Sutherland's data flow environment and Glinert and Tanimoto's PICT. It shares with these systems a total dependence on interactions with graphically represented objects as a means of specifying and observing programs. This is in stark contrast to systems like Boxer and BOCHSER, in which almost all information is made visible in the more traditional textual form. Whereas these text-based environments take advantage of people's linguistic intuitions, graphical interfaces and device programming styles exploit people's physical intuitions. I make no claim that Grasp's use of graphics and device programming make it superior to text-based systems as a programming environment. Text-based languages are an extremely powerful tool; they allow complex ideas to be expressed with a conciseness and readability which in many cases is unmatchable by a graphical system. However, the power of these languages is often inaccessible to novices because their semantics are not at all immediately obvious. I *do* claim that Grasp is a useful tool for understanding the computational elements and processes implied by procedural languages.

Grasp is set apart from other graphical programming systems in two important ways. First, the focus of the Grasp project is not on the use of graphics but on the development of models to aid novices in reasoning about the structure of programs. Fancy graphics alone are no panacea for clarifying the misconceptions of novices. The graphics are helpful only to the extent that they

can make the features of a clear model visible to the programmer. For example, the Grasp interface illustrates the distinction between the oft-confused notions of procedure, procedure activation, and interpreter, but the important point is that the Grasp model explicitly disinguishes them in the first place.

The second important property of Grasp is that it supports the full power of procedures as first-class objects. As with Scheme procedures, Grasp procedures can be stored in variables, passed as arguments, and returned as results. By emphasizing a procedure's properties as a data object through a graphical interface, Grasp strives to reduce the confusion about first-class objects which are fostered by Scheme's expression-oriented interface. Of the novice-oriented systems I have studied, only BOCHSER [Eisenberg] supports first-class procedure objects, and it does this through an interface based mainly on text. I believe that Grasp is the first system to incorporate the powerful notion of first-class procedures in a truly graphical manner.

1.2.3 User Interfaces

Although the notion of visibility is an important one for novice-oriented programming, the actual implementation of visible systems falls into the domain of *user interface design*. This field focuses on developing general principles for human interaction with machines and realizing these principles in particular systems. The extent to which the principles can be realized is strongly dependent on existing technology. Visibility, for instance, is difficult to achieve in a teletype-style interface. In recent years, the availability of high resolution display devices and the development of the software to drive them has made it possible to explore graphical environments aimed at visibility. New technology has also opened the door for other exciting interface ideas; interfaces which take advantage of touch, speech, gesture, and eye motion are areas of current research [Bolt 84].

The mere utilization of advanced technology does not guarantee a good user interface. Designers often construct systems in an ad hoc way, following their whims rather than a firm set of principles. Principled designs are hard to come by. One of the best examples of a principled design is the Star user interface. Following such general principles as visibility of information, consistency of interactions, modelessness, and simplicity, the Star designers carefully embodied

the physical-office metaphor into their system. The result is a system in which users can apply their familiarity with the office environment to reduce the barriers typically associated with computer interaction [Smith *et al.* 84].

The two main principles incorporated into the Grasp user interface are visibility and manipulability. Visibility means that information is represented in a recognizable visual form accessible to the user. Like Star and many other graphically oriented user interfaces, Grasp uses icons to represent objects of interest. Unlike Star, however, familiar graphical representations for Grasp's objects are hard to come by. Documents, filing cabinets, and printers are concrete objects in the physical-office metaphor which are amenable to easily recognizable icons. Such abstract objects as procedures, variables, and control make the task of choosing representations in Grasp anything but straightforward. An important part of the Grasp project is forming models in which these objects can be viewed as physical objects and devices. Achieving visibility then reduces to the problem of designing representations for the objects and devices.

The principle of manipulability is an extension to the notion of *direct manipulation* interfaces. Direct manipulation is a style of interface characterized by direct interaction with representations of the manipulated objects rather than indirect interactions with them through a separate command language [Shneiderman 83]. I use the term *manipulability* here to mean that the user can interact with representation of the objects in ways reminiscent of his interaction with physical objects. Many text editors have direct manipulation interfaces, but they do not embrace a view that characters are physical objects; characters are usually not the kind of stuff we expect to pick up and hold in our hands. Even systems that embody a physical metaphor choose to depart from the metaphor in some ways. Although physical objects can be moved and positioned continuously in space, icons in Star are constrained to appear in one slot of an array of 154 squares on the display screen. Manipulability implies that such interactions should more closely resemble their physical counterparts.

Together, visibility and manipulability make Grasp what Ciccarelli calls a *presentation system.* A presentation system is one in which the user's interactions with the representations on the screen are interpreted as manipulations on an underlying data base [Ciccarelli 84]. If the presentation system is designed appropriately, the user need not be conscious of the distinction between the graphical representations and the objects they represent. This idea, which

diSessa calls *naive realism*, is a fundamental principle of the Boxer project [diSessa 85a]. Naive realism allows the users to believe that what is displayed on the screen *is* the state of the system. Since users will tend to make such an assumption anyway, it is fruitful to explicitly maintain such an illusion in a presentation system. Through its use of visibility and manipulability, Grasp supports naive realism.

Grasp differs from such interfaces as Star and Boxer in the high degree to which it adheres to the physical metaphor. The goals of the Star and Boxer systems require them to pay a considerable amount of attention to handling text. In Boxer, the user is always interacting with the text editor. In Grasp, on the other hand, text plays a minor role; it is the *structure* of the graphical configurations which is important. This is a strong motivation for incorporating into the interface features which allow users to manipulate the graphical configurations in a physical way.

1.3 Roadmap to this Report

In this document, I have striven not only to explain the details of my project, but also to clarify the motivations, goals, principles, and justifications behind the project as well. As a result, this thesis contains a fair amount of background material. The reader may have neither the time nor the desire to wade through it all. The following overview of the thesis is provided to help the reader find what he or she considers to be the "meat" of this document:

Chapter 2: REASONING ABOUT THE STRUCTURE OF PROCEDURAL PROGRAMS - This chapter describes the general goal of the Grasp project - to help novices build more robust structural models of programs. It contains discussions of models of computation, comparisons of structural and functional models, some notes on the procedural paradigm (especially as embodied in Scheme), and a discussion of what I mean by "helping novices"

Chapter 3: VISIBILITY AND MANIPULABILITY - Two important design rules that are at the foundation of the project - visibility and manipulability - are motivated in this chapter. The principles are motivated by a consideration of how people build mental models of programming languages. I argue that people build models by assimilating explicitly taught models or by inducing models based on their interactions with the programming language. Induction is prone to pitfalls,

but explicit models can have problems of their own. In both cases, the presentation of structural information is crucial for forming robust models. Visibility and manipulability take advantage of people's physical intuitions to help them build more robust models.

Chapter 4: PERILS IN THE PROCEDURAL PARADIGM - Some of the problems that students have in understanding Scheme procedures are discussed in this chapter. These trouble spots were extremely influential on my design of the model underlying the Grasp system.

Chapter 5: THE GRASP MODEL OF COMPUTATION - In this chapter I present a model of computation for the Grasp system. I argue that a well-designed graphical interface is not the only key to a visible and manipulable system - models must first contain objects which are amenable to such interfaces. A principle of reification is developed to derive a set of computational elements used in a model of the procedural paradigm. The chief attributes of the model are its treatment of procedures as first-class objects and its clear separation of procedures from procedure activations.

Chapter 6: A VISIBLE AND MANIPULABLE INTERFACE TO THE GRASP MODEL - This chapter describes how visibility and manipulability are achieved in the interface to the Grasp system. Visibility is achieved through direct mappings and naive realism, which are intended to aid the structural reasoning of programmers. Manipulability is maintained by applying the principle of direct manipulation rigorously. Mouse sensitivity, occlusion, and smoothly moving objects are all important aspects of manipulability.

Chapter 7: DISCUSSION - The advantages and disadvantages of the Grasp model and interface are discussed, including comparisons to other novice-oriented systems.

Chapter 8: CURRENT STATUS AND FUTURE DIRECTIONS - The implementation status of the prototype Grasp system is briefly described, followed by a discussion of future avenues of exploration for the Grasp project.

Chapter 9: SUMMARY - A summary of the key ideas presented in this document.

Appendix: EXAMPLE GRASP PROGRAMS: A collection of sample Grasp programs.

CHAPTER 2

REASONING ABOUT THE STRUCTURE OF PROCEDURAL PROGRAMS

My main goal in this project is to help novices reason about the structure of programs. Since this is a broad and open-ended goal, it is necessary to clarify my intent in the scope of this thesis. In this chapter, I will define some terms, clarify the scope of the work, and justify some of the decisions I made in choosing this area of inquiry.

2.1 MODELS OF COMPUTATION

For the purposes of this report, a *programmer* is any individual who constructs static descriptions intended to bring about a desired dynamic behavior in a computer. When I refer to a programmer in this report, I often do not specify his or her level of skill. Since the research described herein focuses on the non-robust mental models of programmers, I am usually referring to novices rather than more experienced programmers. A programmer uses a *programming language* to specify descriptions, and the resulting structure is a *program.* Programming languages are traditionally textual, though they may have a graphical component as well. Programs are *executed* by the computer in order to engender a dynamic behavior on the machine; the evolution of a computation during the execution of a program is a *process.* The tools provided by the computer for the construction and execution of programs form the *programming environment* or *programming system.* The programmer interacts with the programming environment through a *user interface*, or simply *interface.*

To be able to read or write a program in a particular programming language, a programmer must have some notion of how the program will be executed by the computer. After all, a program is just a static structure with no intrinsic meaning; the only reason that the programmer can say that a particular

program *does* anything at all is that he or she somehow acknowledges the existence of an active entity which uses the program to determine its behavior. This entity is often called the *interpreter*.

A programming language expression only has meaning when paired with an interpreter. For example, the expression `(+ 1 1)` is intrinsically meaningless. When evaluated by a Scheme interpreter, it denotes the numerical value 2. We can, however, imagine an alternate interpreter for which a pair of parentheses denotes an operator for counting the non-blank characters between the parentheses. For this second interpreter, the expression `(+ 1 1)` denotes the numerical value 3.

An interpreter embodies a *model of computation*, a description of the abstract world in which the computations performed by the interpreter take place. A model of computation consists of three major parts:

1. The set of *computational elements* which exist in the abstract world.
2. A means by which the elements may be configured to form a *computation state*.
3. *Interpretation rules* followed by the interpreter for determining how one computation state leads to the next.

In real systems, the interpreter for a programming language exposes many of the details of the system in which it is running - e.g. limits on the representations of numbers, storage capacity of the machine, and so on. To isolate the model from the details of any particular machine or operating system, it is beneficial to consider an *abstract machine* which supplies the interpreter with an unrestricted number of the ideal computational elements of the model. The idea of an abstract machine is similar to the *notional machine* described in [duBoulay *et al.* 81] - an "idealized, conceptual computer whose properties are implied by the constructs of the programming language employed."

The type of elements found in the abstract machine depend on the programming language and the desired level of detail. Abstract machines for microcode, for instance, typically consist of low-level hardware components like ALUs, registers, and buses. For a high level programming language, the abstract machine usually includes such entities as variables, procedures, and data structures, although it is also possible to view the language at a lower level as well.

Programmers must have a good understanding of the model of computation underlying a programming language in order to program effectively. In order to improve this understanding, programming courses, texts, and manuals often provide explicit models for a programming language. The Scheme course, for example, introduces four distinct explicit models of evaluation for Scheme expressions:

1. The *substitution model* gives symbolic manipulation rules for reducing complex expressions to simpler ones. Although it is useful for explaining procedure applications early in the course, this model must be abandoned once side effects are introduced.
2. In the *actor model*, procedures are treated as scripts that are followed by actors. This model is not very rigorous, but it is a good vehicle for introducing the notions of time and space complexity.
3. The *environment model* describes the evaluation of expressions in terms of their effect on abstract machine elements known as environment frames. Though this model completely explains how variables and reference work in Scheme, it gives no insight into control flow.
4. Control flow is explained by an explicit control evaluator described in the *register machine model*. The abstract machine for this model includes lower-level entities than the other models - e.g., registers, stacks, and primitive functional units.

As is evident from the above example, explicit models can vary widely in their completeness, emphasis, and usefulness. In some cases, an explicit model can be useful even though it is not phrased in terms of the abstract machine elements usually associated with the programming language. Scheme's substitution and actor models, for instance, are helpful even though they do not refer to the environment structures normally attributed to the language.

2.2 REASONING ABOUT PROGRAMS

2.2.1 Reasoning Tasks

The goal of this thesis is not to help people learn *how* to program. This is a very difficult and not well understood task. Programming in the general scenario

involves mapping problems and specifications from some application domain into the rather constrained abstract world offerred by the computer. In this way, programming is really engineering and thus involves many of the same difficulties as any engineering tasks. A good programmer must have:

* A firm understanding of the problem domain and goal.
* A good grasp of fundamental engineering techniques, such as breaking up problems into subproblem modules with simple interfaces
* Familiarity with modelling and the design process.
* General knowledge and technical skills - e.g. awareness of algorithms and standard programming tricks.
* A thorough understanding of the programming system as a tool.

I do not attempt to deal with the general problem of programming. To a large extent, many of the skills can be learned only through experience or an apprenticeship of sorts. Nothing substitutes for experience in many cases.

My chief interest in the Grasp project is the last point mentioned above: understanding the programming system as a tool. The behavior of a computer is predictable to a degree of exactness not attainable with physical systems. If the programmer knows the details of the model of computation embodied by a programming language, he can explain the execution of any program in that language. Whereas comprehending how programs are executed is a well-defined task, understanding how to write programs is not. Rather then dealing with problem solving and general engineering principles in this project, I chose to focus on the more tractable goal of helping novices understand the details of program execution.

My main interest in this project is in helping people *reason* about programs. Here I am influenced by my experience with the Scheme course. Most introductory programming courses emphasize writing programs from scratch. Furthermore the kinds of programs they deal with are typically implementations of standard algorithms for sorting, searching, data manipulation, numeric computation, and so on. The approach taken by Abelson and Sussman, on the other hand, stresses understanding, modifying, and extending existing systems.

Typical systems which students explore in their challenging weekly problem sets are a computer psychiatrist, an adventure game, and a Scheme interpreter. The kinds of taks they are given might be:

* Add a history mechanism to the computer psychiatrist so that it can refer to previous statements made by the patient.
* Develop a troll character for an adventure game which will eat other characters unless appeased with a pizza.
* Extend a Scheme interpreter to allow typechecking for variables, procedure arguments, and procedure results.

Often the changes which need to be made to the existing system are not difficult, but making them requires a thorough understanding of the given system. Figuring out *what* the program is doing and *where* the changes need to be made are usually more than half the battle.

To be successful in such an environment, programmers need to be skilled at the reasoning abilities presented in [Young 83]:

> *Explanation*: Say why a system has a given behavior.
>
> *Prediction*: Reason what the behavior of a system will be.
>
> *Invention*: Construct or modify a system so that it has a desired behavior.

In the Scheme course, these abilities are necessary for three important programming tasks:

> *Reading*: With natural languages, we cannot hope to become good writers of a language until we are proficient readers. Through reading, we note the basic structure of the language and become familiar with the common constructs, the idioms, and the exceptions. The same is true of programming languages, but to a much greater degree. Here the structure is especially crucial because the computer demands extraordinary precision. A human being can probably interpret stylistically poor writing in a foreign language; a computer is not so forgiving. However, the ruthlessness of a programming language interpreter makes programs an ideal medium for expressing ideas in a precise manner. This crucial property of programming languages is the

motivation for Abelson and Sussman's claim that "Programs are written for people to read, and only incidentally for machines to execute." [Abelson & Sussman 85b].

Explanation and prediction are crucial reasoning abilities for reading code. If we are told that a given program has a particular behavior, we should by reading it be able to explain the mechanism for that behavior. Even if we are not given information about a program's function, we should be able to predict its behavior by reading it. This does not mean that people should be able to read *any* piece of code. In any programming language it is possible to write ridiculously obscure programs which aren't worth deciphering. I assume here that the code is well-written in a standard style, with insightful choices for variable names and helpful comments where appropriate.

Modifying: The Scheme course emphasizes making changes and extensions to an existing system. This does require the ability to write code, but the amount of code written is small compared to the size of the entire system. What is often just as important is finding where in the code to make the change. Maintaining the style and spirit of the original program is also an important goal. Extensions using side effects, for instance, are not appropriate for a program written in a purely functional style.

Explanation, prediction, and invention are all important abilities for modifying programs. Before we can modify code, we must be able to explain what it does and how it does it. We then must invent a method for modifying the existing code to give a desired behavior. Predictive abilities are useful for seeing whether the invented methods will in fact achieve the desired effect.

Debugging: Programmers expend great effort in finding and correcting errors in their programs. In the Scheme course, students are sometimes presented with "buggy" pieces of code and asked to fix them. However, students get the vast majority of their debugging experience when they make incorrect

modifications to a programming system. Explicit debugging skills are never taught in the course, but students must acquire some mastery of them in order to complete their computer laboratories.

Again, explanation, prediction, and invention play a crucial role in the debugging of programs. Explanation is required for localizing the error. Fixing an error is just another form of program modification, so the inventive and predictive abilities that were useful for that task are important for debugging as well.

The reasoning skills and programming tasks described above are what I have in mind when I talk about "reasoning about programming." Certainly there are other skills and tasks which are important for programming - e.g. the ability to design a large system from scratch. However, these other skills extend to a wide range of engineering disciplines. In this report I have chosen to concentrate on the reasoning necessary to understand the specific tools of computer science - in particular, the structural mechanisms of a programming language.

2.2.2 Models as Reasoning Aids

Models are representations which aid a person in reasoning about a system. *Explicit models* are written or spoken descriptions of the structure and function of systems. The four models of evaluation taught in the Scheme course are good examples of explicit models. The internal mental representations which people actually use to reason about a system are *mental models*. For reasons which I will shall discuss in Chapter 3, the mental models which people use to reason about a system may be only loosely related to the explicit models which they have been exposed to.

Models are sometimes divided into two categories: structural and functional. Structural models are those which explain the behavior of a system based upon the interconnections between its component parts and the behavior of those parts. Often these models can be "run" to mechanistically simulate the behavior of a system given its structure. Functional models, on the other hand, concentrate on the functional characteristics of a system. They are appropriate

for explaining certain aspects of a system's behavior, but do not deal with the general mechanism responsible for that behavior.

As an example of the difference between the two kinds of models, consider models of recursion for Scheme. Suppose we have the standard recursive definition of a procedure for computing the factorial function:

```
(DEFINE (FACTORIAL N)
  (COND ((= N 0) 1)
        (ELSE (* N (FACTORIAL (- N 1))))))
```

Structural models for recursion explicitly account for the environment and control information which must be maintained when a procedure calls itself recursively. In the above example, a structural model would explain how the N parameters for different activations are distinguished and how the pending multiplications are remembered.

On the other hand, functional models of recursion explain what recursion does rather than how it is implemented. In the FACTORIAL example, it is possible to understand what FACTORIAL does in terms of mathematical induction or recursively defined functions. The above piece of code resembles the declarative mathematical notation for factorial:

```
FACTORIAL (N) =  1,  N = 0;
                 N * FACTORIAL (N - 1), N > 0
```

Although this resemblance may help people understand what FACTORIAL does, it does help them attain a deep understanding of recursion. In particular, mathematical intuitions are of little help in predicting the behavior of recursive procedures like the one given below:

```
(DEFINE (COUNT2 N)
  (COND ((= N 0) 0)
        (ELSE (COUNT2 (- N 1))
              (PRINT N))))
```

Here the behavior of the pending PRINT expression is difficult to understand unless one has a good structural understanding of recursion.

Although I have tried to stress the difference between structural and functional models, there is not always a clear dichotomy between the two. Just about any model has both structural and functional aspects, and these may not always easy to separate. However, for the purposes of the following discussion, I will continue to talk about the two types of models as if they are distinct.

Each of the two types of models has its advantages and disadvantages. Structural models seem to be good aids for explanation and prediction tasks. Because they are based on mechanism, they give the model-holder a means of "running" the model to mentally simulate the system of interest. Allowing people to envision the behavior of a system, this process of simulation is at the very heart of explanatory and predictive abilities. Structural models can also help with invention tasks by providing the model holder with an appropriate space in which to look for operations for achieving a desired goal. Halasz, for instance, has demonstrated that explicit structural models of a stack-based calculator aid the user of the calculator in approaching non-routine calculations [Halasz 84].

However, structural models have several undesirable properties. Models of behavior based on mechanism generally must be fairly complex in order to cover all the cases of possible behavior at some level. The level of detail stressed by functional models leads to the following problems (many of these are due to [diSessa 86a]):

* The complexity of the models makes them difficult to apply.
* Incomplete structural models are not very useful. For this reason they usually cannot be learned incrementally.
* The details of a structural model bear little relation to a high-level functional specification for a system. Knowing what a part *does* does not imply knowing how it can be *used*.
* When applying structural models, it is easy to lose sight of the forest because of the trees. The high-level behavior of a system can be difficult to discern from the plethora of details generated by applying a structural model.

Functional models fix many of the problems raised above. By stressing the use of a device, they are much closer to the functional terms in which tasks are phrased. Further, they are at a high level, so the model holder does not become bogged down in a quagmire of low-level details when reasoning with them. Another advantage of functional models is that partial models can still be useful reasoning aids. This implies that functional models can be learned incrementally.

The major disadvantage of functional models is that they do not explain the full range of a system's behavior. They may be useful for describing the most common situations in which the system is used, but they tend to break down in the

less common situations. A classic example of this effect is the algebraic calculator model described in [Young 81]. One simple functional model of an algebraic calculator is that the user enters an algebraic expression and the calculator evaluates it when the user presses the = key. This model covers the legal expressions that the user can enter (such as `1 + 2` and `(3 + 4) / (5 - 1)`), but does not explain how an ill-formed expression such as `1 + +` will be treated by the calculator. The model has no predictive or explanatory power for reasoning about expressions outside the class of legal expressions. In the calculator domain, such reasoning is probably not very important anyway. In a domain such as programming, however, explaining and predicting errors is crucial for the debugging process.

In the Grasp project, I focus on structural models rather than functional models. I have made this design decision for two main reasons:

1. Emphasizing structure fits in well with the goal of helping people to read, modify and debug programs. As discussed above, these activities require reasoning abilities such as explanation, prediction, and invention. Because they stress mechanism, structural models are more supportive of these skills than functional ones.

2. Structural models are far more tractable than functional models for the domain of programming. Programming is far more structural than other disciplines. The functionality of most devices is confined to a small range of behavior. For example, telephones are for communication; calculators are for simple numeric computations. Computers, on the other hand, are such general purpose devices that it is hard to attach any particular functionality to them. The structures supported by a programming language, though, form a small, well-defined kernel. Perhaps it is possible to handle function within a particularly small domain, but handling general functionality seems a hard row to hoe.

My design decision does not indicate that I believe functional models are unimportant. Like structural models, functional models are also useful for reading, modifying, and debugging programs. Recognizing standard patterns, knowing typical ways to use elements, and being familiar with common sources of errors are all important for these activities. However, functional models of this kind tend to be built up from experience and are difficult to teach explicitly.

Structural models can be a good foundation for learning functional ones. If the learner has enough patience, he or she can always determine how a piece of code works using structural models. The next time a similar piece of code is encountered, it is easier to understand, and over time standard functional patterns become apparent. This whole process, though, depends on the ability to understand the mechanism of the code in the first place. It is not always necessary to resort to the structural models - in fact, experts use many functional models when reasoning about the behavior of systems. However, the structural models can always be used as stepping stones for learning the higher-level functional models. The converse statement is *not* true in general. Knowing how to use certain elements in a particular way does not mean we will necessarily understand a new use of those elements.

2.3 HELPING NOVICES

I claim in my goal that I want to help novices reason about the structure of programs. But what defines a "novice"? What do I mean by "helping"? The purpose of this section is to explain these terms more clearly.

2.3.1 Novices

In the context of this thesis, a novice is someone who is new to the procedural paradigm of programming. The term not only designates people who have never programmed before, but it also refers to those familiar with other models of computation but not the procedural paradigm. Note that the procedural paradigm here implies the full powers of procedures as used in Scheme. Although other programming languages, such as Pascal, ADA, and CLU share many of the features of this paradigm, they do not support first-class procedure objects. Thus, even people with backgrounds in such languages would be considered partial novices in this thesis. Of course, experience in other languages might help, but it can often be detrimental as well.

I assume in this project that novices desire to be good at the reasoning tasks described in the previous section - namely reading, writing, and debugging programs. I assume that they are not casual programmers trying out a new language, but that they actually plan to do a fair amount of work on nontrivial programming projects. For the casual programmer, high-level functional models

are probably sufficient reasoning aids in most cases. For the other group of novices, however, more structural models are necessary. The main reason for focusing on this latter group is that students in the Scheme course fall into this category. These are the kind of people with whom I have had the most contact and with whom I am the most familiar.

Through my experiences as a teaching assistant, I have observed that novices have *poor* structural models of programs. The models are poor in the sense that they are inadequate to support the appropriate reasoning tasks for reading, writing, and debugging programs. Poor models tend to be:

* Incomplete - the models are missing pieces.
* Ill-defined - the models have incorrect pieces.
* Unintegrated - many of the pieces of the model are present but they have not been synthesized into a coherent whole.
* Inconsistent - the models support differing interpretations to a given problem.
* Superstitious - the models contain irrational pieces based on bad experiences.

The poorness of students' structural reasoning is especially surprising in light of the fact that a considerable amount of emphasis in the course is placed on teaching explicit models of evaluation. Many examples of poor reasoning in Scheme are considered in Chapter 4.

A desirable property for structural models is that they be *robust*. Here I use the term "robust" in a similiar way to [deKleer & Brown 83] - a robust model is one which continues to support proper reasoning even on problems which are new and unexpected. Presumably *expert programmers*, those whose reasoning about programs is clear and accurate, hold such robust models. This is probably not the only advantage experts have - as mentioned before, they also have a well-developed spectrum of models ranging from structural ones to functional ones. Further, they have the ability to discern situations in which application of a model will be profitable, and those in which it will not. Robust structural models alone do not make an expert. However, they are a crucial foundation on which many of the other models may be built.

2.3.2 Helping

The goal of this thesis is to help novices build more robust structural models of programs. The term *help*, however, is a fuzzy one, and it deserves an explanation.

There are a wide range of ways to help learners understand a subject area. In methods of *active help*, an independent agent is available to give lessons, answer questions, and provide comments as the learner progresses. Traditionally, this agent is another person in the capacity of a teacher, employer, or colleague. However, in some Intelligent Computer Assisted Instruction (ICAI) systems, this agent can be a computer. Anderson's Lisp Tutor [Anderson *et al.* 84] and Burton and Brown's West [Burton and Brown 79] are examples of programs which provide the learner with feedback.

Learning environments in which no active agent can respond to an individual learner employ methods of *passive help*. Textbooks and lectures are prime examples of methods of learning where there is no feedback. An example of passive help on a computer are computer-based microworlds which give the learner a domain to explore. The classic example of a microworld is turtle geometry [1], in which the learner can explore properties of geometry via a pen-carrying "turtle" which "lives" on the display screen.

A key property of passive help systems is that they depend largely on presentation to help the student learn. Since they cannot dynamically reconfigure themselves to a particular person, they must aim to be beneficial for a large percentage of the audience for which they are targeted by approriate presentation of the subject matter at hand. Order of material, number and content of examples, and rigor of reasoning, for example, are all important considerations for textbooks and lectures; for microworlds, important decisions include the design of building blocks, how they can be manipulated, and the interface the user has to them.

In this project, I have opted for the passive help approach. The Grasp project described in this report is essentially a microworld for exploring some of the important structures of the procedural paradigm. In particular, it is not an active system which tries to diagnose and correct student programming errors. This is not to say that active help is of no use with a system like Grasp. To the contrary, the educational effect of any passive help environment is immeasurably

enhanced if it is used as a tool by teachers who can answer students' questions and suggest further directions of exploration. I firmly believe that human tutors are the most effective means of teaching programming. I envision Grasp as a tool to supplement the traditional teaching of programming.

The value of Grasp lies in the way it presents the pieces and relationships of procedural programs to the novice programmer. Granted, this particular presentation may not be very helpful to some learners. A major drawback of any passive help system is that it will be helpful to different people in differing degrees. However, when used as a tool, it provides an alternate point of view which may help some people build better structural models of programs.

I would like to make a comment here about making models explicit. I have assumed throughout this project that developing and teaching explicit models is a good thing. This is not a universally accepted belief. There are those people who believe that such explicit models can actually be a hindrance because they "force" a certain viewpoint on the learner. To some extent, this point has validity - people learn not by being told, but by developing models on their own. Although it is possible that too many explicit models may make it difficult for learners to discover their own, I doubt it. I see the explicit models only as suggested ways of looking at the subject matter. The learner is free to accept some of their parts and disregard others Ultimately, however, learners build all models by themselves; if they are not satisfied with the models they have seen, they will create new ones.

2.4 THE PROCEDURAL PARADIGM

There are an infinite number of computational models, differing from one another in their elements, allowable computation states, and interpretation rules. Although many of these models may be equivalent in terms of the kinds of processes they can specify, from the programmer's point of view they are still different in terms of the *way* they describe a process.

Despite an infinite variety of computational models, many of the models can be classified according to certain programming *styles* or *paradigms*. Some of the more popular programming paradigms are:

1. *Procedural Programming (Imperative Style)* - The cornerstone of this paradigm is the *procedure*, which describes how to manipulate data objects and variable bindings.

2. *Functional Programming (Applicative Style)* - This style is similar to the procedural one, except that no side effects (mutations of data or variable bindings) are allowed.
3. *Object-Oriented Programming (Message-Passing Style)* - In this paradigm, the world consists of state-maintaining objects which communicate with each other via messages. A *method* for an object determines how it handles a message.
4. *Logic Programming (Rule-Based Programming, Declarative Style)* - Here the basic model consists of a database of facts and rules for deducing new facts from old ones. Computations are initiated by queries to the database.

All the above styles are covered in the Scheme course, though the Scheme language itself falls under the heading of the procedural paradigm.

In the Grasp project, I have chosen to focus on the procedural paradigm as an arena for exploring ways to help novices build structural models. There were two major reasons for this decision. First, procedural programming is the style with which I have become the most familiar through classwork, teaching, and research. Second, my teaching experiences have made it increasingly clear that the procedural paradigm is capable of fostering a great many misconceptions in novice programmers (see Chapter 4 for details). I believe that many of these misconceptions can be avoided with the help of interactive environments which better present the structural information contained in programs.

2.4.1 Basic Elements of the Procedural Paradigm

Despite the differences between languages supporting the procedural paradigm, the elements of their computational models can be classified into three broad categories:

1. *Data Structures* - explicit elements which are manipulated during the execution of the program.
2. *Procedures* - elements which describe how to manipulate the data.
3. *Interpretation Structures* - implicit elements which are used by the interpreter to execute programs.

I will describe how these elements are embodied in Scheme. Before I do this, though, it will be fruitful to deal a little more with them in the context of procedural languages in general.

Data structures can be analyzed along several dimensions:

Composition - Data which can be decomposed into smaller units of data are *compound*, while those for which no further decomposition is possible are *primitive*. Number and booleans are examples of primitive data, while arrays, records, and lists are typical compound data structures.

Mutability - If a data object maintains state which a program can change, it is said to be *mutable*; otherwise it is unchangeable or *immutable*. For example, the programming language CLU makes a clear distinction between data based on mutability - it has both mutable and immutable versions of arrays, records, and discriminated unions [Liskov *et al.* 79].

Level of Abstraction - A programming language directly supports only a small set of data structures. Using techniques of *data abstraction*, the user can conceptually extend this set by defining procedures to create, decompose, and manipulate new kinds of data. These operations form an *abstraction barrier* which separates the specification of the *user-defined* data from its implementation. Languages offer varying amounts of support for data abstraction. CLU, for example, ensures that abstraction barriers are not violated, while Scheme programmers must maintain the barrier by convention.

Like the data structures discussed above, procedures come in both system-defined and user-defined forms. *Primitive* procedures are those which are predefined in the language; typically their definition cannot be changed. Usually the primitive procedures are for creating and operating on the primitive data elements. Arithmetic operations and procedures for composing and decomposing compound data structures fall into this category. In order to avoid continually repeating common patterns of operations, it is useful to be able to capture such a common pattern into a new procedure. This technique, known as *procedural abstraction*, allows the programmer to define new procedures. As in data

abstraction, there is an abstraction barrier which separates the specification of the procedure's behavior from the implementation realizing that behavior.

Although procedures vary widely between languages, they share some common characteristics. Typically they consist of a specification of inputs, called *formal parameters*, and a pattern of operations, called the *body* of the procedure. The body describes how to produce outputs based on inputs; it may also describe side-effects to be performed by the procedure. A procedure is enacted by *calling* or *invoking* it on pieces of data known as *actual parameters* or *arguments*. The resulting invocation is often termed an *activation* of the procedure. The way in which the actual parameters are associated with the formal parameters is determined by the *parameter-passing mechanism* of the language - this is an important distinguishing feature for different procedural languages.

Interpretation structures are the entities used by the interpreter at run-time to carry out computations. They are typically necessary for handling the flow of data and control during the execution of a program. One common interpretation structure is the variable, which allows the program to name a piece of data and refer to it subsequently by this name. Because they allow data computed at one point in a program to be used at another point, variables are critical elements for implementing data flow in a program. To aid in control flow, interpreters usually have a representation of the current expression they are executing along with a control stack of pending operations to be executed once the current one has completed. These kinds of structures are the ones that the interpreter manipulates when handling conditional branches, iteration loops, and procedure calls.

Interpretation structures differ from data structures mainly in the degree of explicitness with which the user is able to manipulate them. Data structures are under the direct control of the programmer. Programs explicitly construct data structures, take them apart, and operate on them. Interpretation structures are much less under the direct control of the user. For example, stack-based languages generally do not include user-level operations for pushing and popping the stack - these operations are available to the interpreter only. However, since the contents of the stack change in a way dependent on the program, the program can be said to manipulate the stack indirectly.

In some cases the line between interpretation structures and data may not be well-defined. For example, one normally thinks of the arguments to a procedure as being data structures. In the call-by-reference parameter-passing mechanism, a variable rather than the piece of data bound to that variable is passed into a procedure as an argument. In this situation, the variable has a property we normally associate with data structures. Why not consider the variable itself to be a data structure rather than an interpretation structure? The problem is that a variable does not share many of the other properties held by data structures. In particular, we cannot make a variable the value of another variable, nor can we make it a component of a compound data structure. For such reasons it is difficult to consider variables in the same light as data structures. Interpretation structures include all those structures which do not fit neatly into the category of data structures.

2.4.2 The Scheme Approach to the Procedural Paradigm

The approach Scheme takes to the procedural paradigm is distinguished by its simplicity, expressive power, and conceptual clarity. Unlike many other procedural models, the model underlying Scheme is unfettered by a morass of complex details. Instead, a small number of carefully crafted elements interact together in predictable and powerful ways. Since these elements play a crucial role in the evolution of the Grasp project, it is worthwhile to discuss them in some detail.

2.4.2.1 First-Class Objects

One important idea in Scheme is the notion of *first-class objects*. A first-class object in the Scheme world is a piece of data which is characterized by an important set of properties. Any such object:

* May be named by a variable
* May be passed to a procedure as an argument.
* May be returned by a procedure as a result.
* May be used as a component in a compound data structure.
* Has a printed representation.

* Has a conceptually infinite lifetime.

Every piece of data used in Scheme is a first-class object. This includes traditional data structures such as numbers, symbols, and lists, as well as not-so-traditional data as procedures, environments, and continuations.[2]

2.4.2.2 Procedures in Scheme

A second critical aspect of the Scheme model is its view of procedures. The combination of the following three properties is mainly responsible for their great power and versatility. Procedures are:

1. First-class objects.
2. Lexically scoped.
3. Tail-recursive.

The most important characteristic of Scheme procedures is that they are first-class objects. In most other procedural languages, procedures fall into the category of interpretation structures because they possess only a limited subset of the properties of a first-class data structure. Usually they can be named, but the binding between name and procedure cannot be changed once it is made. In a few languages (such as Algol, Pascal, and CLU), procedures can be passed as arguments to other procedures. However, it is rare to find languages which allow procedures to be returned as results from a procedure, to be components in a compound data structure, or to have an infinite lifetime. There are probably two main reasons for treating procedures as interpretation structures rather than data structures:

1. At the conceptual level, procedures may seem quite distinct from data. A common view is that *data is stuff to manipulate* while procedures are *stuff which describes manipulations.* However, they are certainly both *stuff*, and this is the basis for treating the two in a similar fashion.
2. At the implementation level, it may be considered difficult or inefficient to implement procedures as first-class objects.

Scheme grants procedures the full rights of every first-class object. The utility of some of these rights may not be obvious at first, but [Abelson & Sussman

85a] and the Scheme course provide many convincing examples of the advantages of this approach. Many of the examples deal with *higher-order procedures* - that is, procedures passed as arguments to, or returned as results from, other procedures. Such examples not only make clear the power inherent in this approach, but they also demonstrate how this view of procedures greatly simplifies the model of computation in terms of uniformity. A programmer need not remember special-case rules for what can and cannot be done with procedures. Instead, one set of liberal properties applies to all first-class data objects, procedures included.

Scheme's support of lexical scoping is closely tied to its view of procedures as first-class objects. *Scoping* is the method of determining what non-local names mean within a procedure. *Lexical scoping* dictates that a free variable name within a procedure refers to the first accessible variable with that name in the text surrounding the expression which *creates* the procedure. A more common type of scoping for Lisp dialects is *dynamic scoping*, in which a free variable name within a procedure refers to the first accessible variable with that name in the text surrounding the expression which *calls* the procedure. Dynamic scoping and higher-order procedures don't mix well, however; name confusions can occur with procedures passed as arguments (the *funarg* problem), and state can be lost with procedures returned as results (the *funval* problem). Lexical scoping not only avoids these problems, but permits the use of Algol-like block structure within Scheme procedures.

Although it is not as intrinsic to procedures as lexical scoping, *tail recursion* is an important property of Scheme procedures. Tail recursion is poorly named; it really has little to do with recursion. If the last expression to be evaluated in a procedure body is a procedure call, then tail recursion allows the state of the outer activation to be discarded at the point when the inner procedure is about to be applied to its arguments. This subtle rule allows certain syntactically recursive procedures to run iteratively. Iterative looping in Scheme does not require special DO or WHILE constructs; because of tail recursion, iteration can be expressed simply through procedure calls. Thus, the procedure call is Scheme's main control structure.

2.4.2.3 Environments and Continuations in Scheme

Environments and continuations are structures in Scheme which might best be classified as interpretation structures. Environments are the structures which maintain bindings between names and objects. They are used for implementing lexical scoping. Continuations are representations of the "rest of the program" from a particular point in the program. Nonstandard control constructs in Scheme are expressible via continuations.

The status of environments and continuations in Scheme is not clear. In standard Scheme (as described in [Clinger 85]), environments and continutations are *not* first-class objects. In the Scheme course taught at MIT, environments are first-class objects; in fact, they must be explicitly passed as a second argument to the Scheme's EVAL procedure.[3] Continuations are never introduced in the Scheme course. On the other hand, MIT Scheme supports both environments and continuations as first-class objects.

It is not clear that environments and continuations make good semantic sense as first-class objects at the user level. In the approach taken by 3-Lisp [Smith 84], environments and continuations are objects maintained by the interpreter which can only be accessed through a process called *reflection*. During reflection, the focus of a program conceptually jumps from the current level of code to the level of a metacircular interpreter [4] running the current level of code. The environment and continuation which were implicit at the previous level become explicitly accessible during reflection. Certain versions of Scheme, on the other hand, treat environments and continuations as explicit objects at the same level of all user-defined objects in a flat semantic space. Since the "appropriate" status for environments and continuations is an open question, I will take a conservative approach in this report by refering to environments and continuations as interpretation structures.

2.5 Summary

This chapter clarifies the goal of the Grasp project, namely to help novices reason about the structure of procedural programs. The structure of a program is determined by the model of computation or abstract machine that defines the semantics to the program. Reasoning about the structure of programs requires applying such skills as explanation, prediction, and invention to the tasks of

reading, modifying, and debugging programs. Although both structural and functional models are useful as reasoning aids, the Grasp project focuses primarily on structural models. Designed as a passive help system, Grasp is a microworld for exploring the structural aspects of procedural programs. Grasp's approach to the procedural paradigm is largely influenced by the Scheme programming language, especially Scheme's view of procedures as first-class objects.

CHAPTER 3

VISIBILITY AND MANIPULABILITY

As noted in the previous chapter, passive help environments depend chiefly on presentation as a means of helping the learner. In this chapter, I will motivate two important design principles - visibility and manipulability - which are at the heart of the way Grasp presents a model of computation to the programmer.

My approach will be as follows: I will argue that programmers build models based both on explicit models they are taught and on models they acquire from experience. Models based on induction are subject to many pitfalls, yet explicit models often do not help because people do not learn them or neglect to use them. Visibility and manipulability are proposed as principles for helping programmers build more robust models of computational elements and processes.

3.1 BUILDING MODELS OF PROGRAMS

3.1.1 Assimilation and Induction

I claim that programmers build structural models in two distinct ways. The first is by *assimilating explicit models* presented in textbooks, lectures, or other educational settings. In the Scheme course, for example, four explicit models of evaluation are taught to the students. By "assimilation", I suggest that the programmer's internal mental model is related to the external explicit model in a relatively straightforward manner. Certainly it may be modified along the way - some pieces may be lost or altered - but the mapping between the internal and external pieces is fairly direct. If most of the model survived the assimilation process intact, we would expect that the learner would be able to simulate the model in a predictable fashion.

A second method for building models is *induction based on interaction with the programming language.* Here the programmer generates hypotheses about the model of computation and tests them out with sample programs encountered

in the text, in class, or in lab. The models which learners generate depend on the corpus of knowledge they bring with them to the programming task. Note that induction is not independent of assimilation - knowledge of explicit models will certainly influence the kinds of models generated to explain particular examples. The key point here is that people seem to use inductive techniques all the time, regardless of whether or not they assimilate explicit models. For example, children are not explicitly taught about many aspects of the physical world - they learn through their experiences. A child need not be taught that moving objects tend to come to rest in a world full of friction. He or she, though, will probably have a very different model than a physicist as to *why* this is the case.

As an example of models based on induction, I clearly remember constructing and testing a model for side effects during my first days as a student in the Scheme course.[1] We were presented with an example that was something like:

```
=> (DEFINE X 2)
2

=> (DEFINE Y X)
Y

=> X
2

=> Y
2
```

Upon seeing the above example, I constructed two plausible models for variable binding. In the first, bindings created by the two DEFINE expressions might be represented as follows:

```
X: 2
Y: 2
```

In this model, DEFINE associates the name of the variable with the value of the last subexpression. Evaluating a variable name means returning the value associated with that variable. The second model might be represented as:

```
X: 2
Y: X
```

In this model , DEFINE associates the variable with an unevaluated form of the last subexpression. The rule for evaluation of a variable is also different: if the value of a variable is a number, return it; if the value is the name of another variable, return the value of *that* variable.

When I learned that the first model was correct, I felt a little silly for not concluding this on my own. I did not realize until years later that my "incorrect" model actually makes sense for certain interpreters - in particular, those which use normal order evaluation. In fact, in a purely functional subset of Scheme, the two models of variables presented above will, for programs which terminate, always describe the same behavior. If side effects are considered, though, the models are different. After evaluating `(SET! X 3)` in the first model, `Y` will still evaluate to `2`; in the second model, its new value will be `3`. Finding counterexamples such as this one are crucial to model formation because they allow us to disqualify some models and strengthen our belief in others.

In the above example, I consciously generated and compared different models. In general, however, the formulation and testing of models can occur at a subconscious level as well. This fact has some important consequences for the validity and consistency of our models. If we consciously generated and tested all of our models, then it seems that the resulting models would be fairly robust - we would cast aside models that did not work (or at least be aware of their limits), while we would maintain all those that still seemed useful. In reality, though, we probably generate many models but do not rigorously compare them or weed out the bad ones. Rather they become part of a *distributed model* we have for a domain. (Here I use the term "distributed model" after diSessa, who uses it to describe the collection of partial, perhaps inconsistent, ways in which we understand a domain [diSessa 85a]).

I noted above that people use their corpus of knowledge as a basis for generating their models. Some of the knowledge particularly relevant to the domain of programming includes:

Familiarity with other programming languages or systems - Since many programming languages have similar pieces, it is often possible to transfer knowledge from one to another.

Knowledge of natural language - Most programming languages are in a textual form which to some degree resembles the written form of natural language. The Scheme expression `(SQUARE 5)`, for instance, can be viewed as a command to the computer, where `SQUARE` is an imperative form of a verb, and `5` is its direct object. LOGO was designed to take advantage of exactly such linguistic intuitions [diSessa 86a].

Experience interacting with the physical world - Our interactions in everyday life can be a basis for generating models of the structures of programming languages. For example we can understand a last-in first-out stack in terms of stacks of plates or trays in a cafeteria or in terms of a deck of cards.

Understanding of humans and the way they interact - It is often easy to view the computer or some of its structures as independent agents much like humans. It is then possible to understand the behavior of the computer or its structures in terms of analogies with individuals or societies. For example, the message-passing paradigm of object-oriented programming languages makes sense in terms of the way people communicate with each other.

Knowledge of design or engineering principles - Many people come to the programming task with a rich set of principles from other disciplines. These can affect the plausibility they are willing to assign to particular models. Notions of simplicity, elegance, uniformity, and efficiency will play a role in determining the kinds of models they will generate. For example, someone might disregard the second of my variable models because it implies extra complexity in the implementation (the second model requires the ability to delay expressions, whereas the first does not).

Both assimilation and induction have advantages. Explicit models have the clear advantage that they are generally correct. After all, they have been crafted by experts who are familiar with the domain. Induced models have the advantage that the learner is likely to feel more at home with them. Rather than blindly assimilating models that are given to them, the learner of induced models is more likely to be in touch with the reasoning behind the model. This is especially true if the model-building process is carried on at a conscious level - then the learner is aware of where certain hypotheses fail and what the crucial components are.

3.1.2 Induction is Prone to Pitfalls

As to be expected, induced models are prone to many pitfalls. Good models are nontrivial to build. Since learners do not have much depth, scope, or

experience in a domain, they are at a distinct disadvantage in building models. There are two major problems for learners to contend with when forming models:

1. Misleading intuitions - The intuitions people have based on their corpus of knowledge can often lead them in the wrong direction when they build models.
2. Paucity of information - Learners must generate and test their hypotheses based on the scarce amount of information they encounter in their interactions with the programming language.

I shall consider both of these problems in more depth in the following sections.

3.1.2.1 Misleading Intuitions

The corpus of knowledge which serves as a basis for generating and testing hypotheses can often lead people astray in their efforts to understand a domain. It will be fruitful to consider some of the ways in which the types of knowledge described above can lead novice programmers into inducing faulty models.

Familiarity with other systems - Although analogies between systems often serve as a useful way of understanding a new system, the structure and function of the old system often do not carry over to the new. For example, one of my students wondered why the expression `(= A 5)` did not change the value of `A` to `5`. The `=` procedure in Scheme is purely functional and *tests* if its two arguments have the same numerical value. This student was appartently familiar with a language such as BASIC, in which `=` is an assignment operator. In another instance, a student attributed the slowness of his personal computer to a "high load average," a term which is intended to apply to a time-sharing environment. Transfering knowledge from a familiar programming system to a new one can clearly result in poorly formed models for the new system.

Knowledge of natural language - Transfering knowledge about natural language to the programming domain can lead to non-robust reasoning about programs. The fact that English is read from left to right is a motivation for Scheme's prefix form. However, reading from left to right

can confuse learners about the order in which nested operations are performed. In the expression

```
(SQRT (+ (SQUARE 3) (SQUARE 4)))
```

linguistic intuitions might say that the operations are done in the order SQRT, +, and SQUARE. A little thought shows that they are performed in exactly the opposite order that one might expect based on knowledge of English. As we will see in Chapter 4, problems in determining the order of evaluation of expressions can lead to confusion with recursion.

Intuitions about the physical world - Understanding elements of a programming system in terms of physical objects can be a powerful metaphor. However, like most metaphors it has limitations, and these limitations can lead to confusion. For example, I was once working on a spelling correction program in the programming language CLU. A dictionary object had to be passed as an argument to several procedures. Curious to see what words the dictionary contained, I printed it out and discovered it filled over twenty pages. Based on this experience, I began to think of the dictionary as a large and unwieldy object. I worried that passing such a massive object around to other procedures would be grossly inefficient in terms of time and space. My conceptual problem was that I was trying to understand parameter passing in terms of physical movement. In fact, every object in CLU can be considered to be passed as a pointer, so all objects take the same amount of time and space to move around. My intuitions about the physical world clearly led me astray in this case.

Understanding of human interaction - Understanding programming languages or systems in terms of human beings and their interactions is particularly conducive to the formation of faulty models. People are so familiar with communicating with other sentient beings that they tend to anthropomorphize the machine. Indeed, to a novice, the behavior exhibited by a computer can seem so magical and mystical that it is easy to believe that there is something very human-like about it.

Understanding programming languages as a means of communicating with the computer can lead to poor reasoning about computation. Humans are very powerful interpreters of languages in the

sense that they can read much more into a sentence than is actually said, such as implications, the emotions of the speaker or writer, etc. Programming language interpreters differ in this respect. They demand an extraordinary amount of precision and are not forgiving of errors. This is not necessarily bad; it is unlikely that we would want them to attribute any more meaning to our code than what was actually there. In fact, perhaps the best properties of computers is their ruthless interpretation of code. Unlike with natural language, there can be no debate about what a piece of code means. This makes it a perfect medium for describing computations.

As an example of faulty reasoning based on familiarity with human interaction, suppose we have defined a SQUARE procedure as follows:

```
(DEFINE (SQUARE X) (* X X))
```

If we now evaluate (SQUARE (= 2 3)), in which the squaring procedure is applied to the boolean false value, we will get an error. It is very easy for a programmer to believe that the error occurs because numbers, but not booleans, can be squared. Here the programmer is assuming that his or her knowledge of the constraints governing the ways in which SQUARE can be used are explicitly represented inside the machine. In Scheme, this is simply not the case. The error occurs not at the level of SQUARE but at the level of the primitive multiplication procedure, which cannot be applied to booleans. This may seem to be a minor difference, but such structural knowledge is important for debugging a program. An error message for the above expression would say nothing about SQUARE - it would only mention the multiplication procedure.

Knowledge of design or engineering principles - Sometimes knowledge of design or engineering principles can lead to the generation of incorrect models. I held a faulty model of Lisp's QUOTE for over three years, even as a teaching assistant. I developed the following three rules of evaluation to explain the behavior of QUOTE:

```
(QUOTE <symbol>) --> <symbol>

(QUOTE <number>) --> <number>

(QUOTE (<elt1> <elt2> . . . <eltn>))
       --> (LIST (QUOTE <elt1>) (QUOTE <elt2>) . . . (QUOTE <eltn>))
```

This model handles about 99% of all cases normally encountered in Lisp, and, if I'm not mistaken, all examples in the Abelson and Sussman text. It was also easy to believe this model based on principles of elegance and simplicity, since it has the kind of nice recursive decomposition so commonly associated with Scheme. Unfortunately, as I later discovered, the model is wrong and shows a very poor understanding of what all the fuss over QUOTE is about in Lisp.[2] This is a case where arguments based on design principles made me feel smug with an incorrect model.

3.1.2.2 Paucity of Information

Above we saw how a person's corpus of knowledge can be a stumbling block for correct model formation by induction. It is not the only source of trouble. Another important reason why induced models may be faulty is that the evidence used for generating and testing these models is not very good. Learners must induce models based on the paucity of information they encounter in their interactions with a programming language.

3.1.2.2.1 The Importance of Examples

The evidence serving as a basis for model formation comes mainly from examples of code presented in the text, class, or labs. These examples are supplemented by the programmer's interactions with the computer. Such interactions are crucial for generating and testing models. Theoretically, it is possible to learn programming without ever touching a computer. In reality, though, it is difficult to replace interactions with the computer as a learning experience. We can reason about code in any way we want to, but there is always room for doubt. The computer is the final authority as to what the code actually does.

The examples that learners encounter in their interactions with a programming language represent only a small portion of the range of the possible uses of the langauge. Even though lectures, text, and laboratories may provide some examples of language use, learners must generate and test hypotheses based on a relative scarcity of information. The fewer the number of examples,

the more likely it is that they will build incorrect or incomplete models of the abstract machine they are programming.

Examples are important because they drive model formation. Learners will build models to explain the behavior of the examples they encounter in their interactions with the programming language. These models won't necessarily explain all the *features* of the examples encountered - it is important to note that the learner will often be focusing only on a particular aspect of the examples. Counterexamples which expose the limitations of previously held models are especially important to the process of induction - they force the learner to modify old models in order to explain the new behavior. An example with side effects, for instance, makes it possible to distinguish the two models for variables considered at the beginning of this chapter.

If learners do not encounter a wide enough range of examples, the models they form will tend to lack robustness. Failure to consider the right counterexamples can give people an unwarranted strength of belief in their faulty models. As noted above, I held an incorrect model of Scheme's QUOTE for several years before a suitable counterexample exposed my folly. Examples also serve to reinforce the ties between different pieces of knowledge the learner may have. A student with a passing understanding of list structures and of procedures as data objects may still not feel comfortable thinking about lists of procedures. Examples which explicitly use lists of procedures will help to solidify the possibly hazy connection between lists and procedures by providing a suitable context for exercising the pieces of knowledge the student already has.

Unfortunately, novices are unlikely to experiment with a wide variety of examples in their interactions with a programming langauge. Often they are satisified with the examples they see in lectures, books, and laboratories; if they do generate examples on their own, these are usually restricted to simple examples which do not adequately test their understanding. Consider examples illustrating the behavior of Scheme's MAPCAR.[3] Both instructors and students alike will often limit themselves to examples in which MAPCAR is applied to a list of numbers. To square each member of a list of the first four integers, we would write:

```
=> (MAPCAR (LAMBDA (X) (* X X)) (LIST 1 2 3 4))
(1 4 9 16)
```

There are many other interesting applications of MAPCAR, however. MAPCAR can be used with a list of procedures, as illustrated in the following example:

```
=> (MAPCAR (LAMBDA (PROC) (PROC 5))
           (LIST (LAMBDA (X) (* X X))
                 (LAMBDA (X) (> X 0))
                 (LAMBDA (X) (LAMBDA (Y) (+ X Y)))

(25 ! #TRUE [COMPOUND-PROCEDURE 1589763])
```

Here MAPCAR applies each member of a list of procedures to the number 5. The resulting list contains a number, a boolean, and a procedure of one argument which adds 5 to that argument.

Not only does this example illustrate the behavior of MAPCAR, but it also demonstrates the utility of having lists which contain such objects as procedures and booleans. This kind of example offers a context in which a student can unify possibly unconnected pieces of knowledge to form more complete and consistent models. Models based only on simpler examples might not exhibit the same robustness because they would not necessarily be extendible to more complex situations.

3.1.2.2.2 Opaque Interfaces

The major reason why a wide range of programming examples is needed to form robust structural models of a programming language is that traditional command language interfaces provide the programmer with only a limited amount of information on a particular example. The interfaces are *opaque* in the sense that they hide important details about how the interpreter evaluates the expressions which are given to it.

The standard view of programming is *linguistic* in nature. Programmers communicate their intentions to an interpreting agent via specifications called *programs* expressed in some language of description. Although this language is usually textual, it need not be; languages based on pictures, sounds, or touch are possible as well. It is important to note that elements of the language have no intrinsic meaning; all meaning is determined by the details of the interpreting agent to which they are given. It is perfectly possible to consider a different interpreting agent for the same program. For example, the semantics of Scheme programs would change if the properties of the Scheme interpreter were modified in any of the following ways:

* Changing applicative order evaluation of arguments to normal order.
* Using dynamic rather than lexical scoping.
* Removing tail recursion.

It is exactly this "feature" which makes structural reasoning about programs difficult tc carry out. The syntactic elements of the language contain *no information* about their meaning; only the rules of interpretation bestow meaning to them. Yet, in their interactions with the computer, people become most familiar with the representational properties of the language itself. The interpreter is largely invisible; programmers can observe it only indirectly through its behavior in response to sample expressions from the language.

Conventional command language interfaces to programming languages show the user representations of programs and their results. Programs, though, are only static descriptions of dynamic processes. To fully understand how the process described by a program evolves, the programmer must be able to reason about the run-time structures which are manipulated during the execution of a program. Yet, command language interfaces generally do allow the user to view run-time structures or intermediate points of a computation. In Scheme's READ-EVAL-PRINT loop, for example, the programmer sees a character string input to READ and a character string output to PRINT, but the all-important action of EVAL remains hidden.

For experienced programmers the fact that much of the evaluation process remains hidden is a blessing; they do not want to be swamped by the substantial amount of information associated with EVAL every time they evaluate an expresssion. For novices, on the other hand, this approach robs them of information which is crucial for building robust structural models. Many different abstract machines can exhibit the same input/output behavior for certain ranges of examples. For instance, we saw above that two different models of variables were consistent with any examples which did not involve side effects. Novices encountering only a small range of examples cannot be expected to form complete and consistent models under these conditions.

To be fair, many programming environments offer debugging tools which allow the programmer to inspect the kinds of state and dynamic behavior which are invisible in the normal interface. The Scheme system at MIT, for instance,

supports facilities for stepping through a computation, tracing the calls and returns of procedures, examining the stack of pending operations, and inspecting the environment information available at a particular point in a computation. Although such tools help experienced programmers access more information about a computation, their utility to novices is limited by three factors:

1. Proper use of the tools often requires the programmer to be familiar with an explicit model of the language. Accessing information in Scheme's environment inspector is difficult unless the user has command of the environment model. Most novices, however, do not have a firm enough grasp on the explicit models to effectively use the debugging tools.
2. Programmers must be aware of the tools in order to use them. Since little emphasis is placed on them in the Scheme course, many students complete the course without ever knowing the tools exist.
3. The tools must be explicitly invoked. Using the debugging tools takes time and requires special incantations which are easy to forget. Even novices aware of the existence of the tools would often rather simply read through their buggy code than go through the bother of invoking the debugging tools.

For these reasons, traditional debugging tools are not satisfactory methods of providing novices with more information about the model of computation underlying the programming language.

It is interesting to note that debugging tools are almost universally employed only in situations where an error occurs. Rarely are they used to illustrate the evolution of a process in an example which exhibits the proper input/output behavior. Yet, using debugging tools in this manner would be extremely valuable to novices, since it would provide them with important additional information for forming and testing their models of the abstract machine.

3.1.2.2.3 The Limits of Textual Representations

The textual representations employed by most programming language interfaces are a poor medium for conveying the structural information of programs. Linear sequences of characters are not well-suited for describing interconnections among computational elements. To illustrate this point, we will consider alternate representations of the list data structure for Lisp.

A *list* in Lisp is conceptually a sequence of other data objects. It is composed of a chain of smaller data structures known as *pairs* or *cons-cells*, each of which can maintain a pointer to two other data objects (see Figure 3.1).

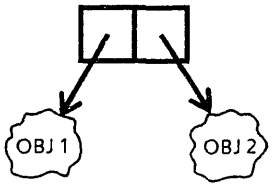
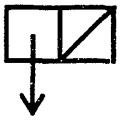
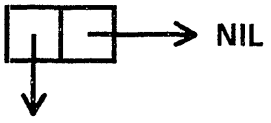


Figure 3.1: Representation of a Lisp pair.

Box-and-pointer diagrams use a collection of the graphical representations for pairs shown above to show how pairs are chained together to form a list. For instance, Figure 3.2 shows a list of the numbers 1, 2, and 3 in box-and-pointer form. (The representation



is a shorthand for



where NIL serves as an end-of-list marker.)

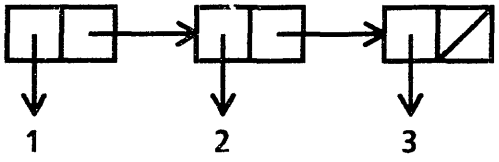


Figure 3.2: A list of three numbers.

This style of representation is especially well-suited to illustrating the phenomena of shared structure. Thus, in Figure 3.3, the top-level list is a sequence of two elements, each of which is the *exact same* (i.e. shared) list of 1 and 2.

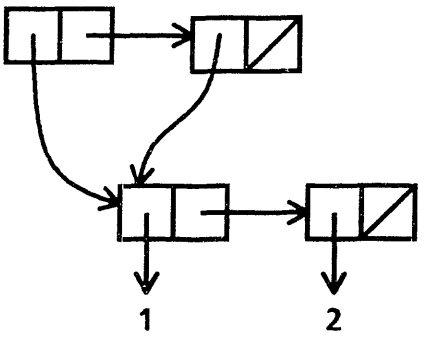


Figure 3.3: An example of shared structure.

The standard representation for list structures in most Lisp systems is *parenthesis notation.* In this notation, a list is represented as a pair of parentheses enclosing a sequence of textual representations of its parts. A list of 1, 2, and 3 is represented as `(1 2 3)`. Individual pairs are represented by the *dotted-pair* notation - here a period separates the textual representations of the pair components between a set of parentheses. For example, the pair consisting of 1 and 2 has the representation `(1 . 2)`.

Box-and-pointer diagrams are related to the list structures they represent by a one-to-one mapping. First, they are *unique* - only one box-and-pointer diagram can exist for a particular configuration of pairs (though it can be formatted in many different ways). Second, they are *unambiguous* - only one list structure corresponds to a given box-and-pointer diagram. Box-and-pointer representations support structural reasoning about list structures because they explicitly show all relevant structural details of the list.

The mapping between parenthesis notation and list structure is much less direct. First, because the list notation is really just an abbreviation for the dotted-pair notation, it is not unique. The list of 1, 2, and 3, for instance, can be represented both as `(1 2 3)` and `(1 . (2 . (3 . ())))`. Second, because it cannot adequately represent shared structure, parenthesis notation is also ambiguous. The textual notation `((1 2) (1 2))` is the representation of both the shared list structure illustrated in Figure 3.3 and the unshared in Figure 3.4:

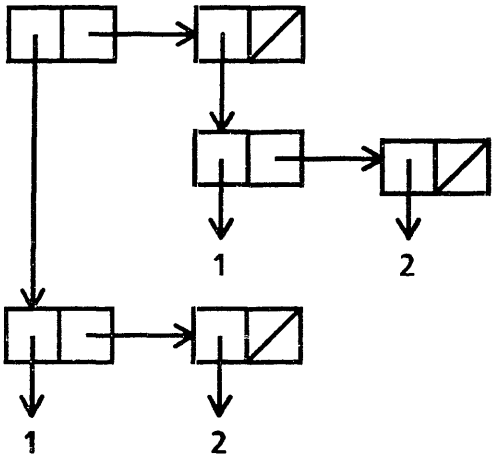


Figure 3.4: Unshared version of ((1 2) (1 2))

Because parenthesis notation does not have a one-to-one correspondence with list structure, it does not support structural reasoning as well as box-and-pointer diagrams. In fact, it can be quite misleading, since mutation of a list can result in unexpected results if the programmer is unaware of sharing.

A second problem with textual representations in command language interfaces is that the input language commonly differs from the output language. Although certain objects in Scheme, such as numbers and booleans, have the same input and output representations, other objects do not (se Figure 3.5).

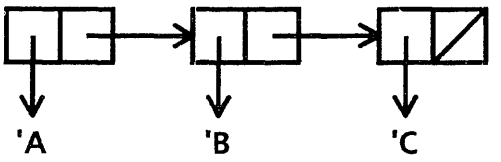
OBJECT	INPUT REPRESENTATION	OUTPUT REPRESENTATION	DIFFERENT BECAUSE . . .
A list of the numbers 1, 2 and 3	(QUOTE (1 2 3))	(1 2 3)	(1 2 3) would be interpreted as applying 1 to 2 and 3.
The symbol A	(QUOTE A)	A	A would be interpreted as evaluating the variable A.

Figure 3.5: Input and ouput representations differ for some Scheme objects.

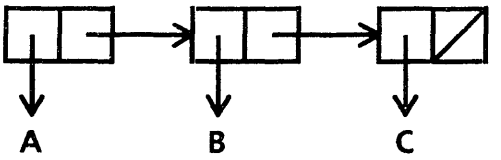
Symbols and lists must be quoted as inputs to distinguish them from variables and combinations. The output representation of a Scheme procedure cannot be used in an input expression in any form.

Having input and output expressions that differ in the above manner is a source of confusion for novices. It is not easy for students in the Scheme course to develop consistent canonical representations for Scheme objects. This problem most clearly manifests itself in box-and-pointer diagrams, where many students

will add a quote mark before a symbol. Figure 3.6 illustrates this bug (and also a correct version) for a list of the symbols A, B, and C.



(a) Representation with quoting bug.



(b) Correct representation.

Figure 3.6: Incorrect use of quotation marks.

The quote mark is really a macro that expands '*<stuff>* into (QUOTE *<stuff>*), so Figure 3.6a is really just a shorthand for Figure 3.7:

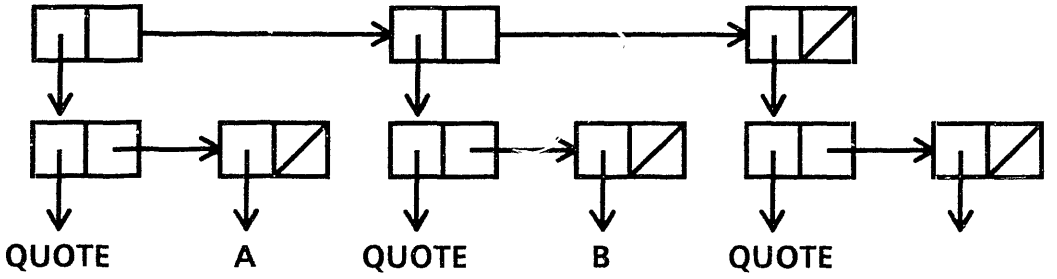


Figure 3.7: Expanded version of Figure 3.6a

Given that the only way students can refer to a symbol in their interactions with the Scheme interpreter is to quote it, this confusion naturally arises from the difference between input and output language Scheme.

Yet another drawback of textual representations is that the actions the user takes to construct a programs (namely typing characters at a keyboard) and the form of the final program (a linear string of text) are very far removed from the conceptual elements of the abstract machine. Experienced programmers often visualize the running of programs in terms of environment operations, flow of control, stack manipulations, and so on. Little that novices see or do in their interactions with text-based interfaces can give them a good feeling for the run-time structures and activities of the underlying model of computation.

Traditional interfaces, especially text based ones, block the novice's access to information which is important for the formation of robust models.

3.1.3 Assimilation Should Be Better But...

Given that induction is subject to the pitfalls discussed above, assimilation of explicit models would seem a much better route toward the development of a robust understanding of the structure of programs. Although this sounds nice in theory, I claim from my experience that in practice this approach leaves something to be desired.

Based on my experience in the Scheme course, I believe that there are three major reasons why explicit models do not always lead to robust structural models. The first is that many novices never really learn them. In the Scheme course this results from many factors:

* Though a fair amount of time is spent on explicit models in class, they are not properly motivated. Many students never quite appreciate their value as reasoning tools.
* Explicit detailed rules for the models are rarely presented to the student. Instead, rules covering some common cases are given. When students try to apply the models, they reach impasses which they don't know how to handle properly.
* Students get little feedback as to whether they are actually using the models appropriately. They may, in fact, not be using the right model.
* Some students may have the belief that the models are "toys" and do not represent the ways experts think about programs. Consequently, they see no great advantage to learning the models.
* Students are more than learning machines. Struggling to juggle heavy courseloads, jobs, a social life, and sleep, they drop some things along the wayside. In the Scheme course, students are more likely to ignore the classes and reading material than the heavily emphasized laboratories. Since models are covered in lectures and the text but are not stressed in labs, many students have only minimal contact with them.

A second reason is that even those students who learn the models don't necessarily use them. Sometimes the models are so unwieldy to apply that students do not see the benefit of using them. One only has to teach the Scheme course once to become acquainted with the weeping and gnashing of teeth evoked by environment diagram exercises on problem sets. In addition, observation of students in the computer laboratory leads to the inescapable conclusion that they would rather do *anything* within the world of the computer than to step back and think about the problems they are encountering. Instead of pulling out a pencil and some paper to apply a model for assessing their situation, students tend to debug their programs by trying different combinations of parenthesis placement, quotation mark insertion, expression reordering, and so on. Using Halasz's terminology, the students seem more comfortable working in the methods space rather than the models space [Halasz 84]. A main reason for this is that the interface provides little feedback for reminding students about the models or helping them to apply the models. The bottom line is that students tend to work in a space which does not allow for robust structural reasoning.

The third reason for the possible failure of explicit models is that the models themselves may be misleading. In Chapter 4, for example, we will examine how a poor choice of representation in Scheme's substitution model can lead to a confusion between the names of procedures and the actual procedure objects. This is one of many possible instances in which the intended model is not the one which a student actually assimilates. The modified model may still work in many cases but will not exhibit the robustness we expect from a structural model.

The factors outlined above indicate that assimilating explicit models is not always a viable path towards the formation of good structural models. As a result, few learners will depend solely on assimilated models for reasoning about programs. The majority are likely to employ some hybrid of assimilated and induced models. However, the inherent unreliability of models induced from the paucity of information provided by most programming language interfaces is a major source for the poor structural models of novices.

3.2 The Principles of Visibility and Manipulability

In the previous section, we explored reasons for the poor quality of novices' structural models. When inducing models on their own, novices are misled by their intuitions and hampered by the lack of structural information provided by traditional programming language interfaces. Yet, explicit models do not necessarily improve the situation. Programming systems rarely help programmers apply an explicit model or even remind them of its existence. Such an environment is not conducive to learning or using the explicit model. Worse yet, the representations chosen in the explicit model may be misleading as well. The upshot of all these factors is that novices' models can remain non-robust despite the existence of explicit models.

What can be done to improve this state of affairs? One approach is to make the structural information of an explicit model inspectable through the interface to the programming environment. This is the fundamental idea behind the principle of *visibility* as espoused by du Boulay, O'Shea, and Monk. Instead of presenting the novice with a "black box" system which computes outputs from inputs, these authors suggest that novices be able to see workings of the abstract machine through a "glass box" [duBoulay *et al.* 81].

Visibility makes a great amount of sense. We saw above that poor structural models are partially attributable to the opaqueness of traditional programming language interfaces. With more transparent interfaces, the programmer has greater access to the structural information inherent in programs. This additional information aids the process of generating and testing hypothetical models and should thus aid the formation of more robust structural models.

The advantages of visibility are even greater if an environment presents the structural information in the form of an explicit structural model taught in classes or texts. In this case, the programming environment continually reminds the programmer about the explicit model. Even if the programmer has never really learned the model, its embodiment in a visible interface should help him or her to induce its important properties. This process of induction should be less error-prone than normal because the availability of structural information makes it less likely that the programmer will be led astray by misleading intuitions.

Visibility means that programmers are able to *see* the abstract machine which is running their programs. In our everyday life, however, we are not only familiar with seeing objects but we are used to interacting with them as well. A principle of *manipulability* suggests that programmers should be able to interact with the elements of an abstract machine as if they were physical objects. This principle is intended to exploit people's well-developed intuitions about interacting with the physical world for constructing and modifying programs.

Manipulability is closely related to the *direct manipulation* approach to user interfaces. Shneiderman defines a direct manipulation interface as one characterized by rapid, incremental interactions with continuous representations of the object of interest [Shneiderman 83]. Manipulability is an extension of direct manipulation in which interaction with the objects of interest closely resembles interactions with physical objects. For example, many modern text editors have direct manipulation interfaces, but they do not support manipulability. Operations like "cut" and "paste" evoke images of moving text from one place to another, but editors show only the result of the movement and not the process of motion itself. I am not suggesting that showing characters flying across the screen would actually be a good idea. Text editing is simply not a domain well-suited to a manipulable interface. Manipulability makes the most sense for representations attempting to show structural connections between objects. Box-and-pointer notation for list structure is an example of a representation for which a manipulable interface is well-suited.

The emphasis of manipulability on physical rather than linguistic intuitions is related to a style of progamming which diSessa calls *device programming* [diSessa 86b]. In device programming, the programmer pieces together representations of physical devices to achieve some desired behavior. For example, a computer microworld for studying fluid flow might contain such devices as reservoirs, pipes, pumps, and meters. People could explore properties of fluid flow systems by connecting these elements together in interesting configurations and observing the simulated behavior. Device programs need not have so direct a physical interpretation as the above example might indicate. Instead, we might consider systems similar to the above which allowed the exploration of momentum flow or vector flow.

The key advantage of the device programming style is its physical nature. When people build device programs, they can pretend that they are manipulating

physical objects analogous to the ones that they would use in constructing mechanical, electrical, or chemical systems. Further, the meaning of device programs appears to be centered in the devices and their connections rather than in a separate interpreting agent. Of course, in any implementation of a device programming system there must be a central interpreter which is actually responsible for simulating the behavior of all the devices. In the model of computation presented to the programmer, however, activity is not located in a central agent but is distributed over all the devices. This corresponds to the independence of objects which people are familiar with from their experiences in the world.

Applying the ideas of device programming to a general computational environment implies viewing the elements of an abstract machine as physical devices. The structure of the program can then be represented as explicit interconnections among the computational devices. Programs can be constructed and modified by interacting with graphical representations of the pieces of the program through a manipulable interface.

When combined with visibility, manipulability allows for the interesting possibility of *programming in the model*. Normally, explicit models are presented as an aid to help programmers reason about the abstract machine running their programs. The model is outside the domain of the computer in that the interface generally does not offer any support for helping the programmer apply the model. However, visibility allows the user to *see* the representations of the model in the interface and manipulability allows the user to *interact* with them. In such a situation, it becomes possible to construct programs by manipulating the elements of the model rather than by the more traditional method of creating programs in a text editor. This approach greatly decreases the distance between the programmer and the explicit model. The constant contact the programmer has with the model space should facilitate the building of more robust structural models.

3.3 SUMMARY

This chapter proposes a theory of why novices have poor structural models and suggests two principles for improving their models. Novices build models by two processes: inducing models based on their interactions with a programming

language and assimilating explicitly taught models. Induction tends to foster non-robust models because opaque interfaces hide structural information which is crucial to the model building process. Without this information, novices can be easily misled by their intuitions into forming inappropriate models. Assimilaing explicit models *should* be a better way to form models, but for many reasons novices never learn or use the models.

Visibility and manipulability are intended to help alleviate the problems associated with inducing and assimilating structural models. Visibile interfaces make the structural information inherent in a program readily available to the programmer. Manipulable interfaces allow the programmer to use physical intuitions for interacting with the structure of a program. Together, visibility and manipulability not only support the use of an explicit model but also allow the programmer to program *in* the model.

CHAPTER 4

PERILS IN THE PROCEDURAL PARADIGM

One need only spend a small amount of time observing novice programmers in classroom or laboratory situations in order to witness the extent and variety of their misconceptions. Problems crop up at many different levels:

* Getting used to the the physical environment (keyboard, display, floppy disks)
* Using the support system (editor, file-system, interpreter)
* Understanding the model of computation
* Becoming familiar with the language constructs
* Learning the higher-level concepts (procedural abstraction, data abstraction, modularity, etc.)

Teaching assistants in close contact with students can easily amass a wealth of information concerning student misunderstandings at these levels. In this chapter, I will probe those misunderstandings that were most influential on the direction of my project and the design of the Grasp system.

I will focus primarily on the problems novices have in understanding Scheme procedures. Although students' bugs with list structures, language constructs, the editor, and other topics are all fruitful and interesting areas of exploration, they are simply outside the scope of this thesis. I will only refer to these areas insofar as they relate to the topic of procedures. Those readers who are interested in the other areas mentioned above are referred to Steve Strassman's *"Learning Lisp: The Barriers to Novice Programmers at MIT."* [Strassman 84]. Based on his experiences as a teaching assistant for the Scheme course, Strassman covers a much wider range of students' problems than are documented in this report.

It is important to discuss some points about the nature of my analysis before I begin. First of all, the evidence I present is mainly anecdotal - that is to say that I did not set forth with a predetermined set of experiments to test people's understanding of procedures. Rather the evidence I have collected is based on my informal observations of the bugs people exhibit in tutorials and the computer laboratory. Second, my interpretation of the problems students have is to some degree influenced by my memories of my own misunderstandings of the material. It is much easier to understand the confusion of others in terms of the confusions one has personally experienced. There is a danger here, of course, of not faithfully representing what problem a student actually has, but I have tried to be as objective as possible in my analyses. Third, the misunderstandings reported here are of varying degrees. Some misconceptions are cleared up quite easily, while others remain entrenched long after the course is over. Finally, it is important to realize that there is no such thing as the "typical student." Each individual student has his or her own set of difficulties with the material. Certainly no one student experiences all the kinds of difficulties reported here. The examples to be discussed are meant to be representative of the wide range of problems people have in understanding procedures.

The difficulties novices have with Scheme procedures can be grouped into three broad areas. They have trouble with:

1. Procedures as first-class objects.
2. Naming issues associated with procedures.
3. Control issues associated with procedures.

The following sections consider these problems in depth.

4.1 PROCEDURES AS FIRST-CLASS OBJECTS

As discussed in Chapter 2, a major advantage of Scheme is the way it treats procedures as first-class objects. Students can encounter many problems when trying to understand the full implications of this idea. Generally these problems appear to stem from the view that procedures are "different" from data and therefore do not fit into the category of first-class objects. This notion is fostered not only by the syntax of the language but even by the explicit models used for reasoning about the underlying model of computation. Both of these cause

confusion by what they fail to present to the students as well as what they present in a misleading fashion. In the case of procedures, the syntax and explicit models support at least three non-robust views of procedures:

1. Procedures as Patterns
2. Procedures as Doers
3. Procedures as Names

I will explore each of these views in turn.

4.1.1 Procedures as Patterns

Scheme's syntax for procedure definition and execution promotes the understanding of procedures in terms of *patterns*. Since this inappropriate model seems to be rooted in Scheme's special syntax for procedure definition, it will be advantageous to briefly discuss the meaning of DEFINE. The only job of DEFINE in Scheme is to bind a name to an object. The expression

```
(DEFINE A (+ 2 3))
```

binds the unevaluated name A to the result of evaluating the expression (+ 2 3) - i.e. 5. Names can be given to procedures in a simlar fashion;

```
(DEFINE SQUARE (LAMBDA (X) (* X X))
```

binds the name SQUARE to the procedure created by evaluating the LAMBDA expression (LAMBDA is Scheme's construct for creating a procedure object). The procedure created above can be called with arguments as in

```
(SQUARE A),
```

which computes the square of 5.

Scheme provides an alternate syntax for creating and naming a procedure. The procedure definition above is more commonly expressed in Scheme as

```
(DEFINE (SQUARE X) (* X X)).
```

Note that the (SQUARE X) subexpression in the DEFINE expression looks very much like a *call* of a procedure on a dummy argument. In fact, this syntax was especially chosen to provide a syntactic link between the procedure and the call [Abelson 86].[1] It is not unreasonable for novices to form the model that what is being defined is a *pattern* for a procedure call. That is, just as

```
(DEFINE A 5)
```

associates the variable A with the value 5, the definition

```
(DEFINE (SQUARE X) (* X X))
```

can be interpreted as associating the procedure call pattern (SQUARE X) with the procedure body (* X X). In such a model, a particular call is an instantiation of the pattern, and the meaning of the call is determined by using the substitution model to evaluate the procedure body with actual values substituted in for the dummy arguments.

Although this pattern-based view of procedures is a suitable functional model for understanding the evaluation of simple expressions, it seriously undermines a deep structural understanding of procedures. First of all, the special DEFINE syntax obscures the two fundamentally distinct operations of *creating* a procedure object and *naming* the resulting object. The expression

```
(DEFINE SQUARE (LAMBDA (X) (* X X)))
```

separates the two operations much more clearly: the LAMBDA expression is evaluated to yield a squaring procedure; DEFINE associates the name SQUARE with this procedure object in the same way it associates a name with any object. The special syntax for DEFINE hides the uniformity and clarity of Scheme by making it look like special cases exist in places where they really don't.

Awareness of the syntactic transformation mentioned above, though, is not necessarily inconsistent with a pattern-based view of procedures. Students are explicitly informed about the transformation, but this does not mean that they incorporate this information into their models. Once introduced to the transformation, students may develop the notion that there are only *two* patterns for defining a procedure:

1. (DEFINE (*<name>* *<args>*) *<body>*)
2. (DEFINE *<name>* (LAMBDA *<args>* *<body>*))

In fact, procedures can be named with DEFINE in many more ways - consider such expressions as

```
(DEFINE FIRST CAR)
(DEFINE COS (DERIVATIVE SIN))
```

The fact that students often greet these expressions with puzzlement is an indication that such examples do not fit neatly into their schema for defining procedures.

The worst aspect of the pattern-based view is that it does not provide a robust understanding of procedures as first-class objects. The special DEFINE syntax may make it easier to understand the standard invocation in which parentheses encapsulate a procedure name followed by argument expressions. However, this typical pattern gives no clue how to handle expressions in which the name occurs by itself, e.g.

```
SQUARE

(DEFINE NEW-SQUARE SQUARE)

(APPLY-TO-FIVE SQUARE)
```

Novices are very confused by such examples. When confronted with the evaluation of SQUARE, most will say that it gives an error because it has not been supplied with any arguments. This is a strong indication of the pattern-based view - the procedure has meaning when part of a standard pattern with arguments, but alone it is meaningless. Even though students know that the other two examples above do not give errors, a typical reaction is "But how does SQUARE get its arguments?." Again, the procedure seems incomplete without the arguments to fill out the pattern.

Similar confusions arise when the first subexpression of a procedure call is a more complex expression than a name. In the pattern-based view of procedures, a procedure call begins with the name of the procedure. Most examples students encounter in the course do in fact fit this pattern. However, the procedure object can be designated by any arbitrary expression, of which a single name is only an example. For instance, even early in the course students will stumble across examples such as

```
((LAMBDA (X) (* X X)) 5)
```

or

```
(DEFINE (REPEATED F N)
  (IF (= N 1)
      F
      (LAMBDA (X) ((REPEATED F (- N 1)) (F X)))))
```

in which expressions other than names occur in the first position of a procedure application. A pattern-based model of procedures does not help novices to reason

about procedures in these examples. A model emphasizing procedures a first-class objects is necessary to expose the common thread of uniformity and simplicity underlying all the examples discussed in this section.

4.1.2 Procedures as Doers

Novices are much more comfortable in their understanding of what procedures *do* than what procedures *are*. They find it difficult to decouple the description of a process from the process specified by that description. The result is that novices have a great tendency to attribute activity to procedures rather than to the processes generated from them. This gives rise to a notion of "procedure as doer" which can be detrimental to a novice's understanding of procedures as first-class objects.

A full appreciation of procedures as first-class objects requires a clear understanding of three concepts: instruction, agent, and action. An instruction is a description of what an agent is supposed to do. When an agent carries out an instruction, the action is done. In computation, instructions are embodied in procedures, the agent carrying them out is the interpreter, and the action generated is a computational process. In particular, procedures don't *do* anything.[2] Rather, they are specifications of how some active agent should act. That active agent is the interpreter; in some sense, it is the only element of the computational model which has the capability of *doing* things.

These ideas can be clarified by an analogy with cooking. Such an analogy is sometimes explicitly presented in programming courses. In cooking, recipes are the analog of procedures and the cook plays the role of interpreter. We rarely think of recipes as "doing" anything. We may sometimes figuratively associate action with them, as in "This recipe bakes a mean pecan pie" or "This recipe serves twelve," but we know that in reality recipes don't bake anything or serve anyone. Recipes don't bake cakes; cooks do. The active agent (the cook) uses a passive description of a process (the recipe) to carry out an instance of the process so described. And so it is with the interpreter and a procedure.

The naive notion of procedures as doers is fairly common among students in the first weeks of the course. When a recent questionnaire [Turbak] asked students in the third week of the course "What is a procedure?", over a third gave responses which indicated the procedure as an agent or a location of action. A

response which exemplified this group was "A procedure is something that does something."

Students were also asked to describe analogies they used to understand procedures. Several viewed procedures as machines which took in inputs and produced outputs. A large number did view procedures more in terms of descriptions, such as instructions, recipes, and maps, but even in these cases there was some confusion about the separation of instruction, agent, and action. A particularly interesting response along these lines was:

> "A procedure is a recipe that tells you how to take ingredients and put them together to get a final product. In addition, (1) the procedure is ALSO the cook - it makes the product according to its rules, and (2) the ingredients can be other recipes (procedures), in which case the procedure would tell you how to use OTHER recipes together."

Here the student understands the descriptive aspect of procedures but clearly attributes agency to them as well.

Why is the notion of "procedure as doer" a potentially harmful one? Certainly many expert programmers have functional models based on this notion. It frees the programmer from continually having to view action as resulting from the interpreter's evaluation of expressions. The programmer can then concentrate on the behavior of the program without getting mired in the details of its structure. However, the expert programmer is fundamentally aware of the structural models as well. When necessary, he or she can deftly switch to more appropriate models depending on the circumstances. When dealing with the properties of procedures as first-class objects, such a programmer will readily adopt a structural view rather than a functional one.

What is unfortunate for novices is that their models are generally not well-developed enough to support the kind of model-switching that experts have. Since they are not aware of the extent and limitations of their models, they can apply them in instances where they don't make sense. The notion of "procedure as doer" is not an intrinsically harmful model, but there is the danger that novices will misapply it. In the following subsections, I will consider how novices develop this notion and will give examples of how it interferes with reasoning about the properties of procedures.

4.1.2.1 Whence Comes This Notion?

It is not difficult to understand how students develop the view of procedures as doers. First of all, the notion is propagated by the figurative way in which teachers talk about procedures. Instructors, myself included, are guilty of generally referring to procedures as doing things. Such statements as "The SUM-OF-SQUARES procedure squares its arguments and adds them" or "PRINT evaluates its arguments" are common in classroom situations. Referring to procedures as agents is indicative of the functional models experts use in thinking about programs. However, as noted above, the expert is aware of the limitiations of such models whereas the novice may not be. The teacher really knows, for example, that the interpreter is responsible for evaluating the arguments to PRINT, not the PRINT procedure itself. For a novice who cannot switch so easily between models, however, these figurative statements can be confusing. For example, when an instructor says "Scheme blurs the line between procedures and data," what is the student supposed to think except that there is a difference between them to begin with? It might be clearer to say "Procedures *are* data" and leave it at that.

Regardless of the ways in which procedures are described to them, students have many other avenues for developing bad intuitions about procedures. Linguistic intuitions can lead novices to view procedures as verbs and their arguments as direct objects. The Scheme text explicitly translates the definition

```
(DEFINE (SQUARE X) (* X X))
```

as "To square something, multiply it by itself" [Abelson & Sussman 85a]. The LOGO language was specifically designed to take advantage of the linguistic intuitions children have; it explicitly uses the word TO to introduce procedure definitions [diSessa 86a]. The same type of intuitions can be fostered by Scheme. In fact, one respondent to the aforementioned questionnaire about procedures directly compared procedures to verbs. Since verbs are the pieces of language associated with action, it is easy to see how linguistic intuitions could lead to the association of action with procedures.

People's intuitions about physical devices are also partially responsible for their assignment of agency to procedures. As noted before, procedures are sometimes thought of as machines with a certain input/output behavior. It is natural to view many physical devices, such as food processors, vending machines,

or circuit elements, as active agents operating on input objects or signals. However, analogies comparing procedures to such devices suffer a serious flaw. In the physical world, the devices themselves are responsible for activity. In computation, procedures abstractly describe a *class* of activities; it is the procedure activation, an instantiation of a procedure, which is directly associated with activity. When viewing the procedure itself as an active device, it is possible to overlook the descriptive properties of procedures.

Finally, spatial intuitions about Scheme expressions can also be instrumental in giving rise to the assignment of agency to procedures. A procedure application in Scheme is denoted by a sequence of subexpressions wrapped in a pair of parenthesis; the first subexpression denotes the procedure while the rest designate its arguments. This form for an application naturally represents a localized unit of action. In the substitution model, the local spatial transformation which takes place when the application of a primitive procedure yields a value makes it seem as if action is associated with the expression itself.

This notion of localized action is supported even more strongly by interactions with the Scheme interpreter. Although the heart of the READ-EVAL-PRINT loop is in EVAL, the programmer sees only the textual inputs to READ and outputs from PRINT. When the user indicates the end of the input expression, the only change which takes place on the screen is the printing of the result directly below the evaluated expression. This type of interaction promotes the view that the expression itself is somehow the location of action for the computation. Since most expressions are procedure applications, it is natural to associate procedures and action. This association can be even stronger is systems where evaluation is initiated by the closing the final parenthesis of an expression; it seems as if the final parenthesis is an indication to the procedure that it can go ahead and "do its thing".

4.1.2.2 Confusions Cause by "Procedure as Doer"

There are at least three area of confusion which are associated with the "procedure as doer" model.

1. Confusion about how action can be bundled up into an object.
2. Problems with recursion - how can something call itself?

3. The tendency to expect action anywhere in the vicinity of procedures.

I will discuss each of these in turn

Novices who too closely associate procedures and action can find it difficult to meaningfully interpret their properties as first-class objects. The confusion seems to stem from an inability to reify the action in such a way that the properties make sense. How can activity be bundled up in order that it can be named? How does one take a piece of computational energy and pass it as an argument or store it in a data structure? To such people, procedures are totally understood in terms of what they do rather than what they are; in this context it is difficult to determine what higher-order procedures could possibly mean. One just as well might be asking how to package up the "loves" in "John loves Mary" to send it off to a friend.

Problems with recursion can arise from focusing on procedures as the locus of action in computations. A robust understanding of recursion requires a familiarity with the idea that several invocations of the same procedure may be in progress at the same time. In models where the procedure itself is considered a doer, whether a human-like agent or input/output machine, explaining how the state and control information of different activations could be maintained at the same time can be difficult.

This is a possible source of the "loop model" which Kahney describes in his work on novices' understanding of recursion [Kahney 82]. In the loop model of recursion, the recursive call of a procedure indicates that the body of the procedure should be executed from the beginning, but any state or control information associated with the current activation of the procedure is lost. It is interesting to note that such a model is accurate in the case where the recursive call occurs in the tail recursive position. However, it fails in the case of recursive procedures which require the state and control information of each recursive call to be maintained. The loop model seems closely akin to the Fortran view of subroutines - there the storage for a subroutine call is in the subroutine itself rather than in some activation record associated with the subroutine. For this reason, Fortran cannot support general recursion. If novices have an agent-centered view of procedures, they may have similar troubles in seeing how recursion could work in the general case.

This is not to say that consistent agent-centered views of procedures are not possible. Certainly the actor model uses agents to explain procedures. However, the actor model is a multiple-agent model where several actors may be reading the same script. This corresponds to several procedure activations for the same procedure - that is, the actor is an activation and not a procedure. This is a perfectly good way of understanding procedures. However, consistent single-agent models for procedures are harder to come by. In such models, an agent is associated with a procedure rather than an activation. To accurately explain general recursion, such a model must explain how the single agent keeps track of the state and control information of multiple activations - e.g. by writing them down on sheets of paper maintained in stack-like order. There is nothing wrong with such a model, but it is unlikely that a novice will be able to discover it on his or her own.

Perhaps the most prevalent side-effect of the notion of procedure as doer is the naive tendency to associate action with procedures wherever they occur. Procedures can be viewed as active entities waiting to operate on the arguments which follow them. Given this point of view, cases in which procedures are not followed by arguments are perceived as being a cause for error. Examples like

```
SQUARE

(APPLY-TO-FIVE SQUARE)
```

which were introduced in the "procedures as patterns" section are also relevant here too. There the confusion occurred because SQUARE by itself did not match a pattern which the students knew. Here, there is a different notion of incompleteness. With the "procedure as doer" model, the SQUARE procedure is waiting to act on an argument but it is nowhere to be found. Students often suggest that an error will be signalled in this case.

Another symptom of action-centered procedure bugs is a confusion between procedure creation and invocation. Consider what happens when we evaluate

```
(DEFINE (SQUARE X) (* X X))
```

In the first few weeks of class, there are inevitably a few students who say something like, "But we can't do that because we don't know what X is." Even worse, if X has already been defined as a global variable, say with a value of 5, some will say the above expression gives 25. Clearly there are some problems here in terms of understanding naming and the evaluation of special forms,[3] but there

is also a confusion as to when the activity of a procedure occurs. If the notion of a procedure as an object is understood, then it is clear there must be a way to *create* the object (via LAMBDA or the special DEFINE syntax) as well as a way to *use* the object (via procedure application). When the distinction is not understood, however, there can be an overzealousness to expect action wherever procedures are involved.

This problem is compounded even further when dealing with procedures which return other procedures as objects. Consider

```
(DEFINE (MAKE-EXPONENENTIATION EXPONENT)
  (LAMBDA (NUM) (EXPT NUM EXPONENT))
```

When we evaluate

```
(MAKE-EXPONENENTIATION 3),
```

for example, a cubing procedure is returned. The above definition can be quite disturbing for the novice, because return of a procedure is hardly an "action" in the normal sense. It is clear that there is an EXPT buried in the body of MAKE-EXPONENENTIATION, but since only one number is available, the EXPT clearly can't do anything yet - a piece of information is missing. Some are inclined to say that there is an error because NUM isn't defined for EXPT. These are people showing the bug of trying to evaluate an expression in a procedure body at procedure creation time. Another group of people know that there will be no error, but still feel uneasy about what is going on. In particular, they wonder where the other number is going to come from and when the exponentiation is going to happen. A better structural model of procedures would help clear up such confusions.

The problem of forgetting to initiate an internal loop may also be related to the model of procedure as doer. Iterative procedures in Scheme are commonly defined with an internal procedure to maintain the extra state of an iteration. For example, an iterative version of factorial might look like:

```
(DEFINE (FACT N)
  (DEFINE (ITER M ANS)
    (IF (= M 0)
        ANS
        (ITER (- M 1) (* M ANS))))
    (ITER N 1))
```

In this example the internal ITER procedure maintains an extra argument ANS for storing the accumulated state of the answer. Note that ITER must not only be *defined*, but it must be *called* on initial values to get the computation going. A

common error in defining such procedures is neglecting the initial call of the internal procedure. Part of the reason may be that procedures are so closely associated with action that merely declaring what the internal procedure is supposed to do is seen as providing the body of the external procedure with sufficient "action" for doing its job.

Two other trouble spots with procedures for novices may be related to the notion of procedure as doer. Novices have an especially hard time understanding two special cases of procedures [Sussman 85]:

1. Procedures which take no arguments, e.g.

```
(DEFINE (THREE) (+ 1 2)).
```

2. Procedures which have primitive expressions in their bodies, e.g.

```
(DEFINE (IDENTITY X) X).
```

Sometimes these special cases occur together, as in

```
(DEFINE (FOUR) 4).
```

Certainly a major problem here is that these cases are unfamiliar - few examples covering these cases occur in the text or problem sets. Yet, some of the difficulties may be related to an expectation of action. In the case of parameterless procedures, the view of procedures as input/output machines can be misleading. One might consider the input as starting the machine (e.g. coins in a vending machine), so that the lack of input would make it hard to tell when to start. Furthermore, an interpretation of procedures as manipulators of input data raises the question of what the procedures would be manipulating in this case. For procedures with primitive expresssions as bodies, the confusion may be that such an expression does not fit in with the expectation of action for a procedure. Having another procedure call, such as `(+ 2 3)`, as a body meets the expectation of action, but having just the number `5` does not. Although I have never studied these two phenomena in students, I would not be surprised if the notion of agency was to some degree intertwined with these problems.

4.1.3 Procedures as Names

When Shakespeare penned, "What's in a name," he was underscoring the difference between a name and the object it denotes. This distinction is an important one in the study of semantics for both natural languages and

programming languages. In the basic Scheme model of computation, procedures manipulate objects, not names; a clear separation between the two is built into Scheme environments. The use of a name in a program denotes the object to which the name is bound in the appropriate environment. In short, names aren't important - objects are.

Unfortunately, the inability to fully appreciate the name/object distinction when dealing with procedures is a stumbling block for some novices. A confusion between the name of a procedure and the procedure object is an impediment to the understanding of procedures as first-class objects. This section probes how this confusion may come about and gives examples of the kinds of misunderstandings it can lead to.

Many programming languages do not emphasize a clear distinction between the name of a procedure and the procedure itself. Often this is a consequence of the fact that procedures are not true first-class objects in these languages. When procedures cannot be assigned to variables, inserted into data structures, passed as arguments, or returned as results, there is little conceptual leverage to gain by stressing the difference between the two. Such languages generally treat the names of procedures in a special way. First of all, procedures in these languages are not associated with names by normal variable assignment (as they are in Scheme), but by special declarations which put procedures into a separate namespace. Second, whereas Scheme allows arbitrary expressions to appear in the procedure position of a procedure call, most programming languages require this position to be filled by the *name* of a procedure. Pascal, C, and CLU are examples of languages with these characteristics.

Even many dialects of Lisp treat procedure names in a special way by maintaining separate namespaces for functions and variables. Maclisp uses SETQ for general variable assignment but DEFUN to associate a procedure with a name. In Interlisp, the procedure position for a call can be filled with a procedure name or a lambda expression but not an arbitrary expression. When explicitly applying a procedure to arguments in Interlisp, one passes a *name* to APPLY, not a procedure object. For example, the Scheme expression

```
(APPLY SQUARE '(5))
```

has as its Interlisp counterpart

```
(APPLY 'SQUARE '(5)).
```

A problem with Scheme is that it is easy for new programmers to get the mistaken impression that it also treats procedure names differently from other names. Most of the reasons have to do with the fact that procedures *look as if* they are being treated specially even though they really aren't. I can think of four factors contributing to this confusion:

1. In most introductory examples, the procedure position of a procedure call is filled with the name of a procedure only. Novices can get the impresssion that it *must* be a procedure name, especially if they are already familiar with another language in which this is the case. This explains the confusion that accompanies the first uses of other expressions in this position.

2. The special syntax for DEFINE disussed earlier is also a cause of trouble in this context. The special syntax makes it *look* like procedures are different from other objects when in reality this is not the case. Is there any wonder why novices are almost universally confused when confronted with evaluating

   ```
   => SQUARE
   ```

   for the first time? Asking the corresponding question for Pascal, C, or CLU would be absurd - such an expression is simply not meaningful in those languages. Yet the special DEFINE syntax does not give any indication that Scheme is different from those languages.

3. The substitution model presented in the Scheme course further compounds name/object confusion by handling the evaluation of procedure-designating expresssions in an inadeqate fashion. Consider the following sample evaluation using the substitution model:

   ```
   1. (SQUARE (+ 2 3))
   2. (SQUARE 5)
   3. (* 5 5)
   4. 25
   ```

   Between steps 1 and 2, the expression (+ 2 3) evaluates to the number 5, but the name SQUARE remains the same. Again, procedures seem to be treated differently from other objects. In actuality, SQUARE is not an appropriate notation for the second line - it should really be replaced by a different notation for the procedure object denoted by the name SQUARE.

As it stands, the model makes it seem as if the procedure is integrally tied to its name. In this way, the currently taught substitution model reinforces any name/object confusions a novice may already have.

4. Printed representations for procedures can also contribute to name/object confusions. There is no agreed-upon representation for procedures in standard Scheme [Clinger 85], but MIT Scheme prints compound procedures out in one of the following ways:

```
[COMPOUND-PROCEDURE SQUARE]
```

or

```
[COMPOUND-PROCEDURE 0187328]
```

The first form is for procedures defined with the special define syntax - in this case it is possible to grab hold of the name when creating the procedure.[4] The second is for procedures created with an explicit LAMBDA - in this case, a name is not available when the procedure is created, so a unique ID number is assigned to the procedure instead.

The problem with names of the first form is that they give the impression that the name is intimately bound up with the procedure object itself. Since no other printed representation contains a name, procedures are again singled out as "looking different" from other objects because of their names.

There are several typical situations in which name/object confusion manifests itself. First, new Scheme programmers commonly expect procedure calls to begin with the *name* of a procedure. Such counterexamples as

```
((LAMBDA (X) (* X X)) 5)
```

and

```
((REPEATED F (- N 1)) (F X))
```

expose the folly of this expectation. The expressions of confusion worn by novices when they are first faced with such examples suggests that they do not fully appreciate the name/object distinction.

Second, students sometimes believe that names rather than objects are being passed around in higher-order procedure examples. Upon seeing the sequence of expressions

```
(DEFINE (SQUARE X) (* X X))
```

```
(DEFINE (APPLY-TO-FIVE PROC) (PROC 5))

(APPLY-TO-FIVE SQUARE)
```

some novices voice the opinion that the *name* SQUARE is being handed to the APPLY-TO-FIVE procedure in the final expression. In fact, I have seen one case of a student actually writing the equivalent of

```
(APPLY-TO-FIVE 'SQUARE)
```

in this type of example. Here there is a clear confusion between name and object.

Reasoning about procedures as elements in data structures is a third area in which the name/object confusion leads to trouble. I myself was guilty of such an error in a handout I prepared in my first semester as a teaching assistant. Desiring to express a list of trigonometric functions, I wrote

```
'(SIN COS TAN)
```

rather than the correct

```
(LIST SIN COS TAN).
```

The former is a list of procedure *names*, while the latter is a list of *procedures*.

This type of confusion usually comes to a head for the students in the laboratory on generic arithmetic. To keep track of the operators for different types of mathematical objects (rationals, complex numbers, polynomials, etc.), a two dimensional table of procedures is maintained. Students must not only define the procedures for handling polynomials, but they must also insert them into the table. A common bug often happens at this stage. The student notices that a procedure, say PLUS-POLY, is defined incorrectly, so he or she fixes it it but forgets to reinsert the new procedure into the table. The effects of the edit are not seen because the generic arithmetic package is still using the incorrect version in the table. At first glance, this error might seem to be related to the understanding of side-effects rather than the understanding of procedures. However, when the problem is explained to students who encounter this error, a large number will say something to the effect of "Oh - I thought the *name* PLUS-POLY was being stored in the table." The name/object confusion strikes again!

A fourth manifestation of the confusion between name and object is illustrated in procedure definitions in which a new name is given to an existing procedure - e.g.

```
(DEFINE FIRST CAR)
```

Novices are typically at odds to explain what the above definition means. Some will independently propose the mistaken idea that FIRST is being associated with the name CAR in such a way that subsequent uses of FIRST will really be referring to CAR. Although this predicts the right behavior, it is a non-robust model which does not reflect the Scheme model of computation. Furthermore, the fact that many students must be explicitly informed about the semantics of the above expression is an indication that they do not fundamentally grasp the notion of first-class procedure objects.

A final example of procedure names resulting in faulty reasoning occurs with message-passing.[5] Consider the adventure game laboratory in which characters in the game are represented by message-passing objects defined with the following skeletal form:

```
(DEFINE (MAKE-PERSON NAME)
        .
        .
        .
  (DEFINE (ME MESSAGE)
          .
          .
          . )
  ME)
```

Suppose we now create two people via MAKE-PERSON:

```
(DEFINE HAL (MAKE-PERSON 'HAL))
(DEFINE GERRY (MAKE-PERSON 'GERRY))
```

An environment diagram skeleton for the resulting environment is illustrated in Figure 4.1.

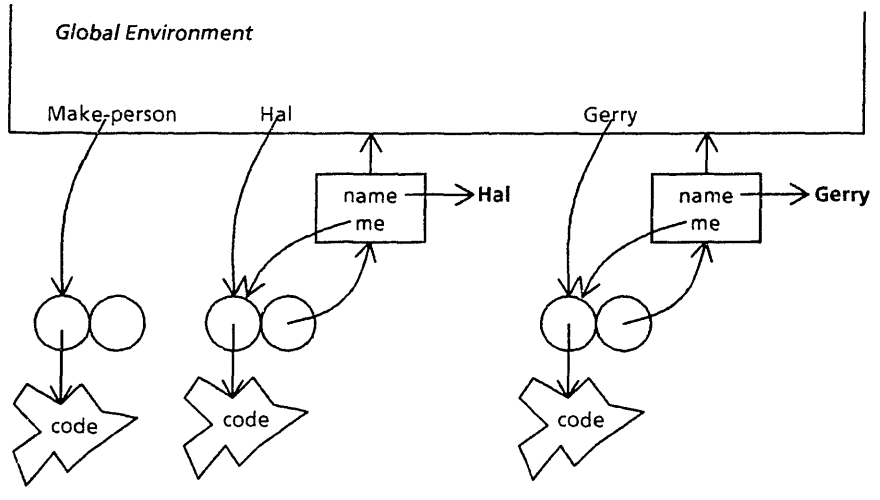


Figure 4.1: Environment diagram for the MAKE-PERSON example.

The environment diagram makes clear that each person object "knows" its name only because that name is explicitly stored in the environment it has access to. In particular, contrary to some naive expectaions, the procedures have no idea what names they are known by in the global environment. A further confusion which hampers reasoning in this example is that the printed representaticns for both person objects is `[COMPOUND-PROCEDURE ME]`, a non-unique representation that gives no clue which procedure is being referred to. In such cases the printed representation showing the unique identifier number would be a much more helpful reasoning aid.

4.2 ENVIRONMENT ISSUES

Understanding the implication of names in Scheme is not an easy task for the new programmer. We have already seen in the previous section how it is possible to blur the distinction between names and objects in the case of procedures. This section explores some of the other difficulties which novices encounter when trying to understand names in Scheme.

4.2.1 Parameter Passing

One area for potential misunderstanding of names is Scheme's parameter-passing mechanism. The Scheme interpreter evaluates a combination (an expression denoting a procedure call) by first evaluating all subexpressions for the combination into objects; it then applies the value of the leftmost subexpresssion (a procedure object) to the other objects. As a part of the application process, the formal parameter names are associated with the corresponding argument objects. This is a *call-by-value* parameter-passing mechanism; it is also known by the name *applicative-order* evaluation. Many alternate mechanisms are possible - for example, in *normal-order evaluation* (also referred to as *call-by-name*), the formal parameter name is associated with the expression itself rather than the object which results from evaluating the expression. In such a scheme, expressions are not evaluated until they appear as arguments to a primitive procedure. A distinction between the two mechanisms is illustrated below by different versions of the substitution model; here `SQUARE` refers to a squaring procedure, and `A` is bound to `3`.

Applicative order

1. (SQUARE A)
2. (SQUARE 3)
3. (* 3 3)
4. 9

Normal order

1. (SQUARE A)
2. (* A A)
3. (* 3 3)
4. 9

When reasoning about procedure applications, it is easy to misunderstand exactly what information is available to the interpreter at a particular point in the computation. Even in the simple example of squaring show above, it is possible to have a misleading model. When asked what (SQUARE A) does, many programmers will respond "It multiplies A by itself." Sometimes this really means "It multiplies the *value* of A by itself," in which case the programmer shows an understanding of applicative-order evaluation. In other cases, however, the programmer really has a model that the interpreter still has access to the name A within the body of SQUARE. Such a model is not necessarily related to normal-order evaluation; it may just be a hazy notion that the interpreter "remembers" that the 3 came from evaluating the name A. After all, the programmer can clearly remember that this is the case, so why can't the interpreter?

The problem with the above kind of model is that it can hinder proper reasoning. First, it unnecessarily clutters the programmer's view of what information is actually available to the interpreter. Second, it can lead to confusion or improper lines of reasoning if the programmer actually tries to make use of the extra information. I cannot pinpoint a particular example of this situation, but I remember having seen cases in which thinking the interpreter knew more than it actually did got novices into trouble.

Another parameter-passing misunderstanding which inevitably crops up with a few students every term is confusion between formal and actual parameters. These students get the mistaken impression that the formal and actual parameters to a procedure must be the same. A sure-fire sign of this

confusion is the unnecessary naming of objects before they are passed as parameters to a procedure. For example, suppose we have a procedure SUM-OF-SQUARES:

```
(DEFINE (SUM-OF-SQUARES A B)
  (+ (SQUARE A) (SQUARE B)))
```

To apply this procedure to 3 and 4, a person exhibiting the formal/actual bug will write

```
(DEFINE A 3)

(DEFINE B 4)

(SUM-OF-SQUARES A B)
```

This certainly works, but it belies a lack of true understanding for the way Scheme handles parameters.

Although the formal/actual bug is usually cleared up within a week, it is worthwhile to consider why it occurs at all. Certainly there are plenty of examples shown in class and in the text where the actual argument is *not* the name of a formal parameter. However, there are some important examples where the distinction is obscured. Consider the set of procedures presented in [Abelson & Sussman 85a] for using Newton's method to approximate the square root of a number:

```
(DEFINE (SQRT X)
  (SQRT-ITER 1 X))

(DEFINE (SQRT-ITER GUESS X)
  (IF (GOOD-ENOUGH? GUESS X)
      GUESS
      (SQRT-ITER (IMPROVE GUESS X) X)))

(DEFINE (GOOD-ENOUGH? GUESS X)
  (< (ABS (- (SQUARE GUESS) X)) .001))

(DEFINE (IMPROVE GUESS X)
  (AVERAGE GUESS (/ X GUESS)))

(DEFINE (AVERAGE X Y)
  (/ (+ X Y) 2))
```

Note that X and GUESS are used to name many different variables. There are really five different Xs and three different GUESSs in the above code. In this example, the actual argument is the same name as the formal in six instances. Because Scheme syntax does not indicate the *structure* of reference (i.e.. which Xs are really related) there is certainly ample room for confusion to arises. When this lack of structural visibility is combined with examples in which the structure

is obscured by the unnecessary reuse of names, problems such as the formal/actual bug are sure to arise.

4.2.2 Scoping

The confusion about the structure of reference in Scheme is most clearly evidenced in examples involving Scheme's use of lexical scoping. Although lexical scoping is explained in depth by the environment model, many students never become facile enough with that model to use it as a reasoning tool. Furthermore, the notion of lexical scoping must be understood in the context of higher order procedures before the environment model is ever introduced in the course. The result is that students inevitably discover non-robust methods for thinking about names.

The way in which higher-order procedures can "remember" information is a possible area for confusion. Consider the following definitions:

```
(DEFINE (MAKE-ADDER N)
  (LAMBDA (X) (+ X N)))

(DEFINE ADD-3 (MAKE-ADDER 3)).
```

Evaluating

```
(ADD-3 7)
```

results in the value 10. The ADD-3 procedure clearly remembers the value of N, but how? Before the environment model is introduced, this behavior can only be explained by the substitution model. The substitution model explains this phenomenon by saying that the value of 3 is substituted for N in the body of the procedure created by MAKE-ADDER. Thus the 3 is hardwired into the ADD-3 procedure, as if it had been created by evaluating

```
(LAMBDA (X) (+ X 3)).
```

However, the way the substitution model is taught, this point often does not get across. Students sometimes seem to use other methods to understand such examples. One method is the attribution of their own knowledge to the interpreter as discussed in the previous section. That is, the student knows that ADD-3 was defined when N was 3, so the interpreter must know that as well. Sometimes this notion will be even fuzzier, perhaps even guided more by the student's understanding of *what* the answer should be rather than *how* it is actually computed. I have witnessed students who predict that (ADD-3 7) should be

10 based simply on the fact that the procedure is named ADD-3 rather than on any deep understanding of *why* ADD-3 adds 3 to its input. Even if ADD-3 were given a less descriptive name, an astute problem solver could note that in the above code, there is only one operation (+) and two numbers (3 and 7) so a likely meaningful action to perform is to apply the operation to the numbers to get 10. This may sound absurd, but I have actually had students who explicitly told me that they were using such a method to approach a more convoluted higher-order procedure example in which other reasoning approaches failed them. The drawbacks of such approaches are obvious - they are simply not a robust way to reason about programs.

Matters grow progressivly worse with the introduction of block structure. Here proper reasoning about names almost requires a detailed model like the environment model. Yet people are creative beings and are bound to develop alternate, albeit wrong, models of naming.

My favorite example of confusion with naming in block structure involves a problem set dealing with a simple software psychiatrist. The program reads inputs from the user and responds to them in some fashion. The basic driver loop of the program consists of the three parts shown below:

```
(DEFINE (START)
  (DRIVER-LOOP))

(DEFINE (DRIVER-LOOP)
  (LET ((USER-RESPONSE (READ)))
    (REPLY USER-RESPONSE)
    (DRIVER-LOOP))

(DEFINE (REPLY RESPONSE)
  <ways to respond>)
```

Through each iteration of the loop, an input is read from the user and the REPLY procedure is called on that input to generate an appropriate response.

Students were asked to extend the system to include a history of the user's responses so that the psychiatrist could sometimes reply

```
(EARLIER YOU SAID THAT <previous user response>).
```

Since this problem-set occurred before side effects were introduced, students were expected to handle this in an applicative fashion. The standard trick in such cases is to add an extra argument to the loop in order to maintain the desired state. This approach leads to the following code;

```
(DEFINE (START)
  (DRIVER-LOOP NIL)) ; start off with an empty list of responses
```

```
(DEFINE (DRIVER-LOOP HISTORY)
  (LET ((USER-RESPONSE (READ)))
    (REPLY USER-RESPONSE HISTORY)
    (DRIVER-LOOP (CONS USER-RESPONSE HISTORY))); Add the new response
                                                ; to the history.

(DEFINE (REPLY RESPONSE PREVIOUS-RESPONSES)
          .
          .
          .
  (PRINT (APPEND '(earlier you said that)
     (PICK-RANDOM PREVIOUS-RESPONSES)) ; pick-random extracts a random
                                       ; element from a list.
          .
          .
          .)
```

Several students were not familiar with this trick and instead attempted an approach where a global history list would be updated each time through the loop. The most interesting of these attempts was the following piece of code:

```
(DEFINE (START)
  (DEFINE HISTORY NIL) ; history #1
  (DRIVER-LOOP HISTORY))

(DEFINE (DRIVER-LOOP HISTORY) ;history #2
  (LET ((USER-RESPONSE (READ)))
    (DEFINE HISTORY (CONS USER-RESPONSE HISTORY)); history #3 followed
                                                 ; by history #2
    (REPLY USER-RESPONSE HISTORY) ; history #3
    (DRIVER-LOOP HISTORY))) ; history #3

(DEFINE (REPLY RESPONSE HISTORY) ; history #4
          .
          .
          .
  (PRINT (APPEND '(earlier you said that)
     (PICK-RANDOM HISTORY)) ;history #4
          .
          .
          .)
```

This code actually works, but for reasons not at all suspected by the student. From the student's explanations, it was clear that he thought his code was mutating a global variable named HISTORY. However, mutations are accomplished using SET!, not DEFINE. Scheme's DEFINE creates a new variable in the current block, so the above code actually contains four different variables named HISTORY (as indicated by the comments). This code does exactly what the standard solution above did, although in a much more obscure manner.

The student's model of naming here shows clear bugs as to what names mean. The above code even shows signs of the formal/actual bug discussed previously. My guess is that the student started off with the side effect model in mind, but that early attempts at implementing it did not work. He then kept making modifications to his code until it finally exhibited the correct behvior, resulting in the program above. The irony here is that the program's behavior

supports the student's fundamentally flawed model. This is a good example of hidden information resulting in a poor structural model.

Introduction of side effects and the full-blown environment model result in innumerable difficulties. The problems, however, aren't so much in understanding the concept of state as in understanding the mechanics of the environment model. The major problem seems to be that students get so stuck in a quagmire of the low-level details of applying the model they they lose sight of the high-level phenomena the model is meant to explain. Blind application of the environment model rules gives little insight into what environments are all about. Furthermore, since it is easy to make mistakes when applying the model, such an approach is not even guaranteed to aid the reasoning process.

As an example, consider one of the most common mistakes associated with using the environment model. To ensure that lexical scoping is enforced, a crucial rule of the model decrees that the new environment frame created by application of a procedure closure must have as its parent environment the environment of the closure. Inevitably, students will instead follow a very strong intuition that the parent environment should be the calling environment. Unfortunately, rather than describing lexical scoping, this results in dynamic scoping, which is what the whole environment model was designed to circumvent.

The sad part about this whole affair is that lexical scoping is one of the few issues where the textual representation of code is helpful in assisting reasoning about its semantics. The meaning of a name in a program is totally determined by its position within the text of the code. In fact, there are straightforward ways (which are unfortunately rarely described in class) of determining what the shape of an environment structure for a procedure must look like based on the textual representations of the program.[6] Experienced programmers tend to use such tricks and higher-level reasoning to build environment diagrams *without* resorting to blind application of the environment model rules.

The problem here is that even though the information is at the user's fingertips, it is not presented in a way that the user becomes aware of it. All names look the same in a standard text editor. If some distinguishing feature - color or font, for example - were used to highlight the names which referred to the same conceptual variable, perhaps the programmer could make more use of this

information. The main point of this section is that until the *structure of reference* is emphasized in some manner, people will think about names in non-robust ways.

4.3 CONTROL ISSUES

The third major area of confusion with procedures is that of control. Understanding how the interpreter evaluates expressions - the order it follows, what it evaluates and doesn't evaluate, how it remembers pending operations, and so on - is a nontrivial task for novices. Again, the troubles are rooted in the fact that the textual representation of programs does little to aid reasoning about these matters. With control, this is an especially acute problem, since a suitable explicit model for handling it, the explicit control evaluator, is not presented until the end of the Scheme course. Students are basically on their own to develop models about control. This section will consider several areas where students encounter problems with control.

4.3.1 Order of Evaluation

Scheme's use of nested expressions can make it difficult for novices to see the order in which expressions are evaluated. Strong linguistic intuitions dictate that expressions are read from left to right, but this belies the fact that they are evaluated "from the inside out." An expression such as

```
(SQRT (+ (SQUARE 3) (SQUARE 4))),
```

translates so cleanly into the English "The square root of the sum of the squares of 3 and 4," that people can understand its meaning without paying attention to the order of computation it implies. Alas, there are many situations where the order is not so obvious.

A favorite example of mine in this area involves the factorial procedure. In the first few weeks of the Scheme course, students become very familiar with simple recursive procedures, such as the factorial procedure given below:

```
(DEFINE (FACTORIAL N)
  (IF (= N 0)
      1
      (* N (FACTORIAL (- N 1)))))
```

Many of them seem to understand the recursive call at a functional level but not necessarily at a structural one. That is, the above definition looks very much like an inductive definition from mathematics:

$$\begin{aligned} FACT(N) &= \quad 1, \; N = 0 \\ N * FACT(N-1), &\; N > 1 \end{aligned}$$

However, the mathematical definition is more *declarative* than *imperative* - it stresses what factorial *is* and only implicitly says *how* it can be calculated.

To an experienced programmer, it is clear that no multiplication can occur until (FACTORIAL 0) is reached. At this point all the pending multiplications are done in the reverse order of that in which they are encountered. Thus, (FACTORIAL 0) is the first call of FACTORIAL to actually return a value, and the first multiplication which a programmer "sees" in simulating the behavior of factorial is actually the last one to be applied.

To test student's understanding of these ideas, I often present the following variant of FACTORIAL:

```
(DEFINE (FACTORIAL N)
  (IF (= N 0)
      1
      (LET ((RESULT (FACT (- N 1))))
        (* N RESULT))))
```

Here the use of the LET expression merely emphasizes that the program rushes toward (FACTORIAL 0) before any multiplication can be done. Yet, about half the students who see this example say it will not work, usually predicting that it will result in an infinite loop. There are many possible sources for confusion here, such as a misunderstanding of the LET construct or a general weakness with recursion, but it seems that a bad model of the order in which expressions are evaluated is the most likely candidate as the cause for these responses.

4.3.2 Pending Operations

A notion related to order of evaluation which also crops up in recursive procedure examples is that of *pending operations*. A pending operation is one which the interpreter has to "remember" to do when it completes the current computation. In the nested expression example of the above section, for example,

the + procedure is a pending operation to be applied to the results of (SQUARE 3) and (SQUARE 4).

Students encounter a great amount of difficulty when dealing with pending operations. Consider the following two procedures, which are often used as examples in the Scheme course:

```
(DEFINE (COUNT1 N)
  (COND ((= N 0) 0)
        (ELSE (PRINT N) (COUNT1 (- N 1)))))

(DEFINE (COUNT2 N)
  (COND ((= N 0) 0)
        (ELSE (COUNT2 (- N 1)) (PRINT N))))
```

The two procedures differ only in the placement of the PRINT expression with respect to the recursive call. COUNT1 prints numbers in decreasing order from the initial number to 0. COUNT2 prints number in increasing order from 1 to the initial number.

Most students have no trouble in predicting the behavior of COUNT1. COUNT2, on the other hand, poses a major hurdle to students. When I presented this problem to a class of thirty students, not a single one was able to predict correctly the behavior of COUNT2. The problem is that they did not know how to deal with the pending PRINT expression after the recursive call. Actually, the pending PRINT in COUNT2 is very similar to the pending multiplication in FACTORIAL, but unlike the FACTORIAL case, COUNT2 is not easily understand in terms of mathematical induction.

It is interesting to note that the recursive call to COUNT1 is tail recursive, whereas the recursive call to COUNT2 is not. Novices in general seem to have less trouble with the tail recursive cases than the non-tail recusive ones. There is a good reason for this - there is no need to remember pending operations in the tail recursive case. In such cases, simple models of recursion (such as the "looping model" discussed in [Kahney 82]) will accurately predict behavior even though they break down in the more general case.

Trying to explain the actual concept of tail recursion itself to novices, though, is extremely difficult. The concept requires a deep understanding of what state must be maintained by an activation of a recursive procedure. Although the idea is presented in the first chapter of [Abelson & Sussman 85a], few students pick it up until the introduction of the explicit control evaluator, and even then many find it too subtle to grasp.

4.3.3 Return of Values

An interesting attribute of Scheme is that evaluation of any expression returns a value. In most cases there is an obvious value to return, but in some instances (e.g. PRINT or DEFINE) the returned value is rather arbitrarily defined. The body of a procedure, the body of a LET expression, and the action sequence of a conditional clause may contain a sequence of expressions to be evaluated; in these cases the value of the last expression in the sequence is the value of the construct in question. Students do not always fully appreciate this notion of expressions returning values.

A standard example where the return of a value causes confusion involves the definition of internal procedures. Consider the following example of a procedure returning a procedure as its result:

```
(DEFINE (ADD-N N)
  (LAMBDA (X) (+ X N)))
```

Here the last expression in the body of ADD-N is the LAMBDA expression. The result of calling ADD-N is the result of the LAMBDA expression - i.e., a procedure. The above definition can also be written in the following form.

```
(DEFINE (ADD-N N)
  (DEFINE (NEW-ADDER X) (+ X N))
  NEW-ADDER)
```

In this case, the body of ADD-N consists of two expressions. The result of calling ADD-N is the result of evaluating its last expression, namely NEW-ADDER. Evaluation of this expression gives the procedure defined by the previous expression.

When novices encounter examples like the second one above, a common question is "What is the final NEW-ADDER for?" In this case, the second expression is necessary because DEFINE returns (by convention) the name being defined rather than the object associated with it. Without the final NEW-ADDER expression, ADD-N would create a new procedure but return the name NEW-ADDER rather than the procedure associated with it. This kind of example sorely tests the novices understanding of the details of returned values.

4.3.4 Special Forms

There exist in Scheme a small number of constructs which are not evaluated as regular procedure calls. These contructs are introduced by keywords called *special forms.* DEFINE, COND, IF, LAMBDA, LET, and SET! are the most common of these. Each introduces an expression which is evaluated in a special way by the interpreter.

Unfortunately, special forms *look* no different than regular procedure calls except for the keyword which introduces them. Furthermore, there are no special indications of *how* the special forms are to be evaluated. Consider the DEFINE construct, which has the form

(DEFINE *<NAME> <EXPRESSION FOR VALUE>*)

Evaluating a DEFINE expression involves two separate steps:

1. Evaluate *<EXPRESSION FOR VALUE>* to get an object
2. Bind the unevaluated *<NAME>* to the object resulting from from the first step.

There is no clue in the syntax that these are the steps to be taken. In fact, some people get the model that the expression gets associated with the name and is only evaluated when the name is referenced. Witness my own confusion on this matter when I was a student in the course, as discussed in Chapter 3. Matters get worse for DEFINE when it is used in the special syntax for defining procedures.

Based on my experience, it seems that students do not develop a firm understanding of the set of rules which guide the evaluation of special forms. Instead, they have functional models for the kinds of *patterns* they have seen in examples. The patterns for DEFINE in procedure definitions were discussed in the section on procedures as patterns. As another example, consider the pattern for a conditional clause. Conditional clauses are of the form

(*<PREDICATE> <ACTION-1> <ACTION-2> . . . <ACTION-N>*)

where if *<PREDICATE>* evaluates to a non-NIL value, the expressions in the action sequence following it are evaluated in order. A sample conditional expression is

```
(COND ((< A 0) (PRINT "A < 0") (- A)
      ((= A 0) (PRINT "A = 0") A)
      (ELSE (PRINT "A > 0") A))
```

The expectation that a clause begins with two parentheses (except in the special case of an ELSE clause) is so strong that novices often do not know how to deal with a case such as:

```
(COND (A 7)
      (ELSE 5))
```

Here the predicate is simply the expression A - if its value is non-NIL the conditional expression will return a 7. The fact that students are confused by such an example is evidence that they view speical forms in terms of patterns rather than in terms of evaluation rules.

4.4 Summary

This chapter documents many of the troubles which students have with procedures in the Scheme course. The main source of the problems is that the textual representations people encounter in the programming language give little insight into the computational elements and processes they are describing. In some cases, the representations give no information from which the programmer can induce good models. In other cases, such as the special DEFINE syntax for procedure creation, the representations are downright misleading and lure the programmer off the path towards a robust model. Even the explicit models intended to explain the evaluation process can confuse the novice - the substitition model, for example, blurs the sharp line between the names of procedures and the objects they denote.

In order to help novices avoid the kinds of poor structural models described in this chapter, it is necessary to provide them with better information about the underlying model of computation in which they are programming. Part of the work is to make explicit in the model the kinds of structures which are implicit in the programming language. Once this is done, the resulting computational elements need to be made easily inspectable and manipulable in the interface to the programming language. The way in which the Grasp system approaches these two steps is the subject of the next two chapters.

CHAPTER 5

THE GRASP MODEL FOR PROCEDURAL PROGRAMS

The previous chapter explored some of the pitfalls novices encounter when trying to understand procedural programs. The problems arise for two basic reasons. First, the interface through which the programmer interacts with the underlying computational model can hide valuable information about its state and dynamic behavior. Scheme's READ-EVAL-PRINT loop is a classic example. The input to READ and the output of PRINT are displayed, but the all-important action of EVAL remains invisible. Presented with a lopsided view of the model in the interface, novices are often misled by their intuitions into building mental models which poorly characterize the model of computation. Second, the explicit models taught to novices can themselves be misleading. Part of the problem is presentational in nature. The representations used in the substitution model, for example, can give rise to a confusion between the names of procedures and the procedures themselves. The rest of the problem is more fundamental. Designed to explain only certain aspects of the full computational model, explicit models can be incomplete in their coverage of important issues. For instance, neither the substitution model nor the environment model attempts to explain flow of control.

The principles of visibility and manipulability address the presentational issues raised above. These principles are aimed at providing the novice with more information about the model *through the interface* to that model. They may not be of much help if the underlying model of computation is incomplete or counterintuitive. Consider a model of computation for a Scheme-like language in which procedures passed as arguments *really are* treated as the substitution model seems to indicate - i.e. they are just names. As an attempt to make this model more visible, we could have the interpreter print out each of the intermediate expressions it forms on its way toward the final result. The intermediate expressions so printed would be the same ones predicted by the

substitution model. This visible interface to such a model is not likely to improve a novice's understanding of procedures as first-class objects. To begin with, the model itself is flawed because of its nonuniform treatment of procedure names. In addition, the pieces of the model (namely, textual expressions) are hardly conducive to visible and manipulable representations.

On the other hand, suppose we modify the model so that procedure objects rather than procedure names are passed as arguments. One possible visible interface to the modified model would be the same one described above. In this case, though, the representations of the interface are ill-chosen, not the elements of the model. It is perfectly possible to choose representations which better distinguish the difference between names and the objects they denote. A visible interface using such improved representations might indeed help novices understand first-class procedures.

As an example, consider a new way to present the substitution model. Two representational formats for the substitution model are compared in Figure 5.1. The traditional representation is shown in Figure 5.1(a). Figure 5.1(b) illustrates a variant which makes a clear distinction between name and object through its choice of representation - names appear as text, but objects appear in boxes. Also note that the second version explicitly distinguishes the EVAL and APPLY phases in the evaluation process (the EVAL stage is shown within parenthesis; the APPLY stage within brackets). The first version shows only the EVAL stage and thus does not faithfully represent the true character of evaluation in Scheme.

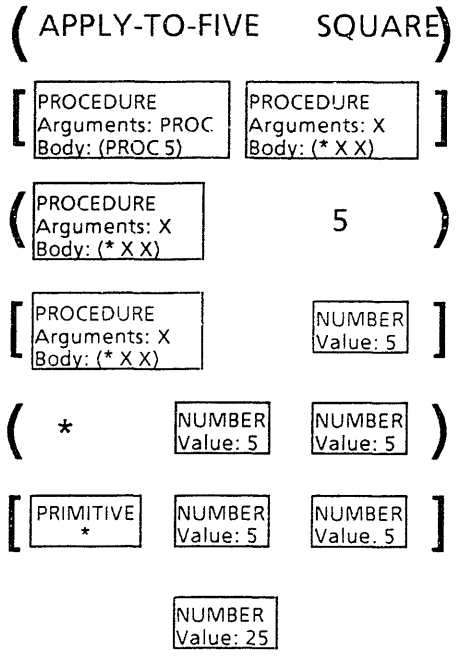
The lesson to learn from the above example is that the model of computation is itself crucial when considering visibility and manipulability. These principles are no panacea; they cannot magically turn an incomprehensible model into a comprehensible one. Their value depends greatly on the nature of the model to which they are applied. The utility of the principles is closely related to the extent to which the model satisfies the following properties:

1. The model completely describes the domain of interest in a simple, straightforward way.
2. The model is consistent with the intuitions and expectations people bring to the task. Of course, this is not always possible, but models which violate too many intuitions should be avoided.

3. The model consists of elements which are amenable to visible and manipulable representations.

```
(APPLY-TO-FIVE SQUARE)
(SQUARE 5)
(* 5 5)
25
```

(a) Traditional form of substitution model



(b) Suggested representation for the substitution model.

FIGURE 5.1: Two representations for the substitution model

The purpose of this chapter is to develop a procedural programming model which is in line with the above properties. The chapter begins with a discussion of why a new model is necessary. A device programming model is suggested as a way to make the structure of a program more explicit. Next I discuss the scope of the Grasp model and the important ways in which it differs from Scheme. The principle of *reification* is introduced as a means of developing a device programming model for procedural programs. Finally, the reified elements of the Grasp model are presented. In Chapter 7, I will argue that these elements should help learners avoid the kinds of misconceptions discussed in Chapter 4.

5.1 WHY A NEW MODEL?

In designing the Grasp system, it was necessary to decide upon a model of computation. I considered developing a new abstract machine for Scheme, but it soon became clear that this was inconsistent with my view of visibility and manipulability. The major problem is that Scheme is inherently an expression-oriented language. The emphasis in Scheme is on the symbolic description of objects and processes. Constructs such as QUOTE and EVAL only make sense in a model where expressions are the primary focus of attention. This does not mean that visibility and manipulability are impossible for Scheme. In fact, Eisenberg's BOCHSER is an integrated, interactive Boxer-like environment which allows a Scheme programmer to inspect and manipulate the elements of the environment model [Eisenberg 85]. However, because it supports Scheme so directly, BOCHSER is of necessity deeply rooted in a linguistic approach to program specification.

In Grasp, my desire was to stress a more physical approach to the specification of programs. I wanted to shift the focus away from expressions to the objects they denote and the run-time structures they imply. I imagined an environment in which a programmer could construct a program in an erector set fashion by interconnecting representations of data structures and interpretation structures. Such a system would allow the structure of a program to be represented in a more direct manner than is possible in an expression-oriented language. For instance, in expression-based environments, the structural relationship between a variable and references to that variable is achieved indirectly through names and environments. In the type of system I envisioned, this structural relationship could be specified directly - by, say, first pointing to a representation of the variable and then to the place where its value is to be used. Not only would the static structure of a program be inspectable and manipulable in this kind of system, but its dynamic behavior could be made visible as well. For example, the programmer could see animated representations of control flow and data flow in a program. Furthermore, such a system would be well-suited for examining, constructing, executing, and debugging programs within a single integrated environment.

This vision is related to the device programming style discussed in Chapter 3. In Grasp, I attempt to apply the device programming style to the domain of programming itself. Developing a model in which computational elements can be

consistently viewed as physical objects is not entirely straightforward. Designing such a model constituted the major effort of this project.

I should note that Grasp is certainly not the first environment which embodies a device programming style for general computation. In fact, over twenty years ago, Wiliam Robert Sutherland developed an innovative system for interactively specifying the structural connections of data flow programs through a graphical interface. Programmers could connect functional units by data lines in order to form a program. The resulting configurations could be executed, and the values on the data lines could be inspected in a special debugging mode [Sutherland 66]. More recently, Glinert and Tanimoto have devised the Pict/D system, in which programmers build simple Pascal-like numerical programs solely through the manipulation of icons. In fact, the keyboard is never used in interactions with the Pict/D system [Glinert & Tanimoto 84].

To my knowledge, however, Grasp is the first device programming system to support procedures as first-class objects. Both Sutherland's system and Pict/D allow the programmer to create procedures, but the only data objects in these systems are numbers and booleans. Grasp also is constrained to a limited set of data structures, but in addition to numbers and booleans, this set contains procedures. Experience with Scheme shows that treating procedures as first-class objects is a source of great power and elegance. By supporting this powerful notion in a device programming system, Grasp aims to help novices build more robust models of procedures.

5.2 SCOPE OF THE MODEL

To avoid getting bogged down in a morass of special features, it was necessary to narrow the scope of the Grasp model to cover a relatively small kernel of computational elements. Since the focus of the project was on procedures, I wanted to include in the model those elements necessary for supporting procedures as first-class objects. My hope was to build a system in which novices could explore simple examples covering various interesting uses of procedures. In particular, I wanted a system which could handle the kinds of examples presented in Chapter 1 and the first half of Chapter 3 in [Abelson & Sussman 85a]. These chapters cover naming, procedural abstraction, recursion, higher-order procedures, lexical scoping, and procedures with local state.

Examples of the kinds of programs I envision demonstrating in Grasp are given in Scheme form below. Even though the examples are small, students with a deep understanding of the concepts they embody would generally be well prepared to handle more complex examples. This kind of situation is characteristic of robust models; they aid reasoning even for new and unexpected kinds of examples.

1.Straightforward numerical procedures, e.g.

```
(DEFINE (PYTHAGORAS A B)
  (SQRT (+ (SQUARE A) (SQUARE B))))
```

2. Recursive procedures, especially those with pending operations, e.g.

```
; No pending operations
(DEFINE (COUNT1 N)
  (COND ((= N 0) 0)
        (ELSE (PRINT N) (COUNT1 (- N 1)))))

; Pending operations
(DEFINE (COUNT2 N)
  (COND ((= N 0) 0)
        (ELSE (COUNT2 (- N 1)) (PRINT N)))))

(DEFINE (FACT N)
  (COND ((= N 0) 1)
        (ELSE (* N (FACT (- N 1)))))))
```

3. Procedures with state, e.g.

```
(DEFINE COUNTER
  (LET ((CURRENT-COUNT 0))
    (LAMBDA ()
      (SET! CURRENT-COUNT (+ CURRENT-COUNT 1))
      CURRENT-COUNT)))
```

4. Procedures taking procedures as arguments, e.g.

```
(DEFINE (APPLY-TO-FIVE PROC)
  (PROC 5))
```

5. Procedures returning procedures as results, e.g.

```
(DEFINE (MAKE-ADDER N)
  (LAMBDA (X) (+ X N)))

(DEFINE (MAKE-COUNTER)
  (LET ((CURRENT-COUNT 0))
    (LAMBDA ()
      (SET! CURRENT-COUNT (+ CURRENT-COUNT 1))
      CURRENT-COUNT)))
```

6. Procedures both taking and returning procedures, e.g.

```
(DEFINE (TWICE FN)
  (LAMBDA (X) (FN (FN X))))

(DEFINE  (REPEATED FN N)
  (COND ((= N 0) FN)
        (ELSE (LAMBDA (X) ((REPEATED FN (- N 1)) (FN X))))))
```

7. Message-Passing procedures, e.g.

```
; An implementation of CONS, CAR, AND CDR in a message passing style.
(DEFINE (CONS1 X Y)
  (LAMBDA (M)
    (COND ((= M 0) X)
          (ELSE Y))))

(DEFINE (CAR1 Z)
  (Z 0))

(DEFINE (CDR1 Z)
  (Z 1))


; A more convoluted implementation of CONS, CAR, and CDR
; that causes great confusion among students.
(DEFINE (CONS2 X Y)
  (LAMBDA (Z) (Z X Y)))

(DEFINE (CAR2 Z)
  (Z (LAMBDA (P Q) P)))

(DEFINE (CDR2 Z)
  (Z (LAMBDA (P Q) Q)))
```

There are certain features of Scheme which I explicitly decided *not* to include in the initial Grasp model. As previously mentioned, Grasp does not have a rich set of data structures. Since the focus of Grasp is on procedures rather than on data structures, I decided to support only numbers, booleans, and procedures as data objects. This does not indicate that that other data structures or data-related concepts are unimportant. Certainly every programmer requires an understanding of compound data and data abstraction. However, my experiences in the Scheme course have made it clear that procedures are more common than data as a stumbling block for novices. Moreover, the first chapter of [Abelson & Sussman 85a] explores the properties of procedures using only numbers, booleans, and procedures as data objects. Note that the example programs listed above are restricted to these types of data as well. Given that procedures are a greater source of confusion and interesting examples can be done with simple data objects, I decided to deemphasize data structures in in this project.

In Grasp I also decided to avoid Scheme's metastructural operators, namely QUOTE and EVAL. These constructs are fundamentally expression-oriented in nature; it is not clear what they would mean in a device programming system. In a command language interface, expressions are evaluated by default, and QUOTE is necessary as a mark to prevent evaluation. Even if Grasp supported symbols and lists, QUOTE would not be necessary because Grasp clearly differentiates between data objects and the interpretation structures which manipulate them. EVAL does not make sense for Grasp because it does not provide explicit representations of expressions as data objects.

A third aspect of Scheme which is not incorporated into the Grasp model is tail recursion. Tail recursion is essentially an efficiency mechanism which allows certain programs to use less space during execution than their specification might indicate. It does not really extend the expressive power of the language. If memory weren't a constraint, computations performed using tail recursion could just as well be done without it. Given an infinite memory, Scheme programs would perform the same computation without tail recursion as with, though they might take more space. Because tail recursion doesn't increase the expressive power of a programming language, I see no pressing reason to include it in the Grasp model of computation.

In fact, the Grasp system generally ignores efficiency in deference to the clarity of its explicit model. Although efficiency is an important issue in computer science, it is much more important to present a simple and consistent framework for computation to novices. Once a novice understands the framework and the principles it embodies, efficiency matters can be introduced as modifications to the general framework.

5.3 THE PRINCIPLE OF REIFICATION

One trouble spot for novices is that the evolution of a process in many models of computation depends largely on the manipulation of implicit structures - structures which are not directly accounted for in the model. Many procedural languages, for example, implicitly use a stack of activation records for handling nested procedure calls and recursion. In the Scheme course, the problem with such implicit structures is that the explicit models to which the students are exposed in the early part of the course do not shed any light on these structures. New programmers must resort to induction to complete their models. As discussed in Chapter 3, however, induction is not a reliable method for forming robust mental models. Furthermore, novices cannot simply neglect the implicit structures; at least *some* model of them is required to understand the interpretation process. The upshot is that implicit structures are a central source of confusion for novices.

Consider the problem of understanding "how names work" in Scheme. Before the environment model is introduced, students can only understand the evaluation of names in term of some form of implicit contexts; the interpreter

uses these contexts to "do the right thing" when evaluating names. This fuzzy notion is hardly comforting to a programmer trying to predict the value of a variable in a situtation where many different contexts share the name of that variable. Without explicit models, people have little choice but to use their often misleading intuitions to reason about the details of interpretation.

The introduction of the environment model, on the other hand greatly increases the potential for robust reasoning. Environments are explicit structures which maintain the bindings between names and objects. The environment model uses them to show how the interpreter can "remember" the values of names and distinguish the use of names in different contexts. Explicit environments make it possible for the programmer to accurately predict and explain the treatment of names in almost every situation. Whether they actually *do* help programmers in practice depends on the extent to which people are aware of the environment model and the facility they have in applying it. Such principles as visiblity and manipulability may be useful for helping people become familiar with environments. The crucial point of this example, though, is that robust reasoning about implicit structures is only possible when they are made explicit in a model of computation.

It is not enough, though, to make implicit structures explicit - they must be made explicit in a way that supports structural reasoning. The explicit control evaluator for Scheme is an example of a model where explicitness of structure does not necessarily imply support for certain kinds of reasoning. This model makes the management of control and procedure activations explicit through the use of registers and a stack. These interpretation structures, however explicit, are too low-level. Registers and stacks may seem natural to those with hardware experience, but others find such elements confusing. Although the model provides detailed information, this information is encoded in a way that makes the model more of an intricate puzzle than an aid for understanding.

To support structural reasoning about implicit structures, Grasp subscribes to a principle of *reification*. The principle of reification says that implicit structures should be concretized into explicit structures that can be thought of as objects in the physical world. Reification is essentially a method for viewing computational elements from a device programming perspective. The major structures of Grasp are designed so that it is possible to reason about them in physical terms. Object-like structures also have the advantage of being clearly

amenable to visible and manipulable representations in an interface to the model of computation.

5.4 ELEMENTS OF THE GRASP MODEL

This section introduces the elements of the Grasp model and shows how they are used to construct programs. These elements arose out of the desire to reify traditionally implicit structures, such as those implementing procedure calls, environments, and control. The way in which the elements resulted from applying the principle of reification will be discussed where appropriate.

In describing the elements of the model, I will also show the graphical representations used to present these elements to the programmer through the interface. The focus of this chapter, however, is on the the abstract elements themselves rather than on the interface through which the user interacts with them. Thus, when I mention that elements can be connected together, I will not stress how the connections are displayed or how the programmer can modify the them. The interface and its properties are the subject matter of the following chapter.

5.4.1 Primitive Machines and the Controller

The basic building block of the Grasp model is the *machine*. Machines are the sites for computations in Grasp. Normally they compute outputs based on inputs; they may have side effects as well. Primitive machines are reponsible for the primitive operations of the system, such as adding numbers or testing for equality between elements. A primitive machine for multiplication is shown in Figure 5.2.

Machines have input variables to hold their arguments and output variables to hold their results. In Figure 5.2, the boxes labelled ARG1 and ARG2 represent the input variables and the one labelled ANS represents the output variable. Variables may contain data objects; Figure 5.3 shows a multiplication machine whose input variables contain the numbers 5 and -12.

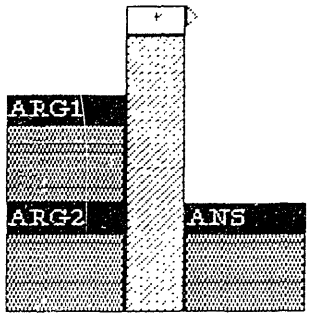


Figure 5.2: A primitive multiplication machine.

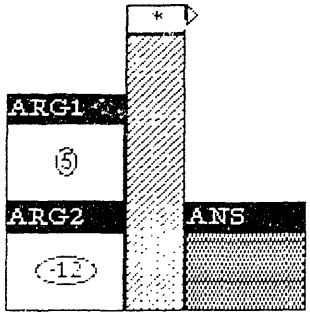


Figure 5.3: A multiplication machine with arguments 5 and -12.

Notice in Figure 5.3 that the presence of the input arguments does not mean that the result of multiplication is actually computed. Although this would be the case in a data flow model, Grasp uses explicit flow of control to determine when the machine *fires* - i.e. carries out its intended action. Control is associated with an explicit element known as the *controller*. The controller "walks" from machine to machine along explict *control paths*.

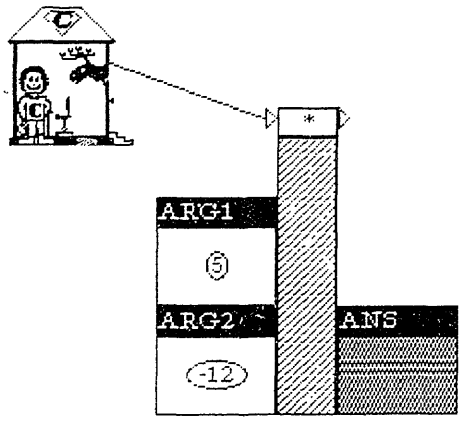


Figure 5.4: A configuration with a controller and control path.

Representations of the control elements appear in Figure 5.4. Here the controller is depicted as a human-like figure. The line with the triangle at the end is a control path to the multiplication machine. The notion that the controller moves from structure to structure along explicit control paths necessitates an element to serve as the original location for the controller. The *control house*, represented by the icon at the left end of the control path, fills this need. When the controller reaches a machine along the control path, it activates the machine. Thus, in Figure 5.5, the controller has activated the mulitplication machine.

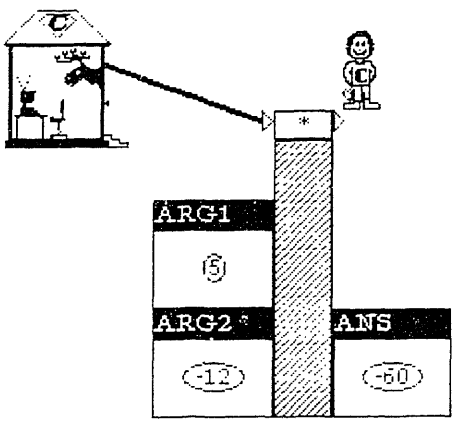


Figure 5.5: The result of activating the muliplication machine.

It is important to note that machines in Grasp are the analog not of procedures but of procedure activations. A procedure is a data object which describes a class of processes. An activation is an interpretation structure that maintains state for a particular instance of that class. A single procedure may have numerous activations; with recursive procedures, more than one activation may be "active" at the same time. The mulitiplication machine may be viewed as an instantiation of a primitive multiplication procedure. User-defined procedures exist in Grasp in the form of *blueprints*; these will be introduced later.

Grasp machines, however, have properties which distinguish them from conventional procedure activations. Machines in Grasp are one-shots - that is, they fire only once and cannot be reused. This fact has some interesting consequences. First of all, since it is not meaningful for the controller to activate a machine more than once during a computation, only a single control path may enter any machine. Second, each machine can be associated with a particular point in time - namely, the time at which it fires. Since flow of control and passage of time are related, we can think of the controller as progressing along a *computational time line* while the computation evolves. At any point during the

execution of a program, control paths to be encountered by the controller represent the future of the computation, while control paths already traversed by the controller represent the history of the computation. This point of view suggests the interesting possibility of running the controller *backward* in time along the computational time line. In fact, Grasp allows the controller to return to *any* previous state of the computation by walking in the reverse direction along the control paths and "undoing" each machine along the way. In the case of our multiplication example, undoing is straightforward - the result needs to be removed from the output variable. Sending the controller backward in Figure 5.5 would yield the configuration of Figure 5.4 again. Undoing is less straightforward with side effects; this point will be considered when mutating structures are introduced below.

The ability to return to a previous state of a computation implies that all of the intermediate state of a computation must be saved in Grasp. This contrasts with the common practice of discarding intermediate state whenever possible in most programming environments. Even in the simple multiplication example considered above, most programming systems would simply return the final answer and throw away the state of the activation used to compute it. Such an approach is necessary for running large programs within the memory limits of a computer.

Grasp, on the other hand, is intended for use by novices on simple programming examples. The amount of space required by these examples should not be exorbitant, and it is a small price to pay for the conceptual clarity it lends to the computational model. Saving the history of the process not only allows novices to run computations backward, but permits them to inspect *how* a computation reached a particular point. When the behavior of a program is incorrect, all the information necessary to debug it is available to the programmer. There is no need to invoke special debugging tools or restart the computation. Furthermore, the intermediate state is inspectable even when there is no error. Thus, in addition to seeing that their programs gave the right answer, novices can find out *why* it gave the right answer.

5.4.2 Reference Pipes

Suppose we want to extend our example to compute whether the result of the multiplication is greater than -60. We can modify the configuration represented in Figure 5.5 by introducing a greater-than machine and connecting a control path from the multiplication machine to the greater-than machine. The result is shown in Figure 5.6. Note that we are modifying a partially executed program. Grasp allows us to modify a program at any point in its execution.

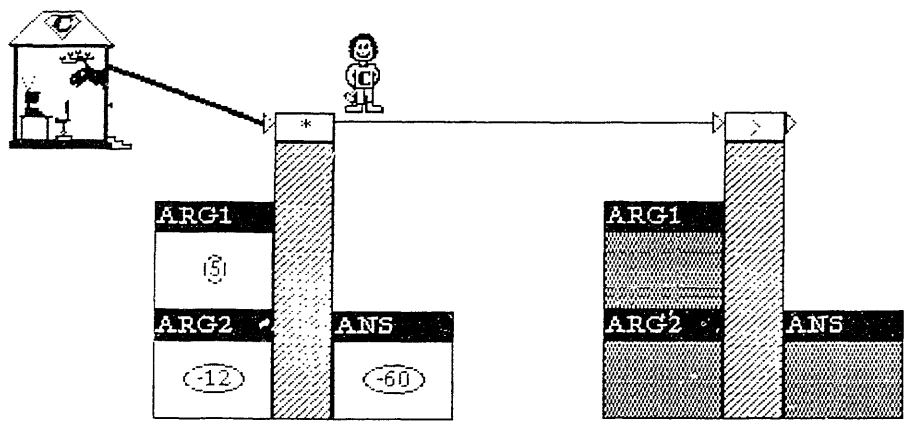


Figure 5.6: Extending the configuration to include a greater-than machine.

We see from Figure 5.6 that making the control connection is not enough. Obviously, we have to insert a -50 as the second argument (ARG2) to the greater-than machine. However, we also must specify that the result of the multiplication machine is to be used as the first argument (ARG1) to the greater-than machine. A *reference pipe* is an object used to specify that the value of a variable is to be used at another variable; it denotes a path over which data may flow in a program. A reference pipe is an objectification of a reference to a variable. Grasp does not use names and environments to specify a relationship between a reference and a variable. Instead, the programmer *points at* the variable providing the value (the source variable) to create a reference pipe and then attaches the reference pipe to the variable where the value will be used (the target variable). Because of the directness of the specification, there is never any question about what variable the reference pipe refers to; this information is hardwired into the reference pipe when it is created.

The result of specifying the arguments to the greater-than machine is illustrated in Figure 5.7. The icon  pointing into ARG1 of the greater-than

machine is the representation of a reference pipe from ANS of the multiplication machine to ARG1 of the greater-than machine. In this case, the representation is ambiguous since more than one of the variables pictured has the same name. The structure of the reference, however, is clear from the interactions that were used to create it. (Unambiguous visual representations do exist; why Grasp does not use them is a topic for Chapter 6.) Since names serve only as comments in Grasp, we can change the name of the output variable of the multiplication machine to expose the underlying structure. The result of changing the name from ANS to Prod is shown in Figure 5.8. In subsequent examples, similar name changes will be made to clarify the structure of the references.

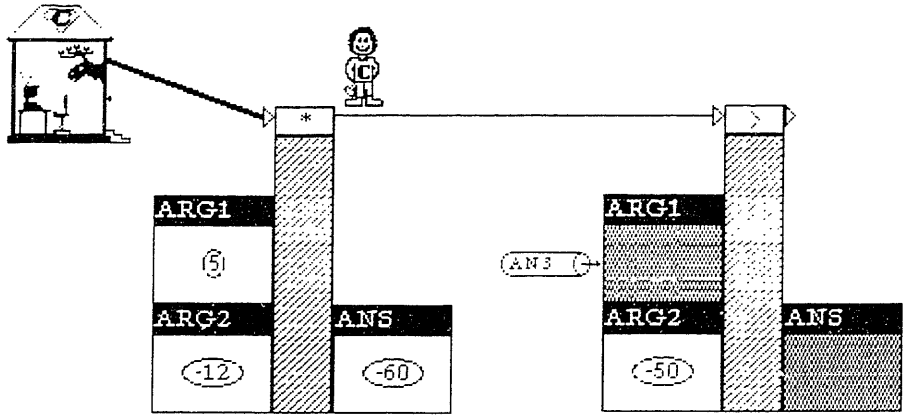


Figure 5.7: Configuration with a reference pipe.

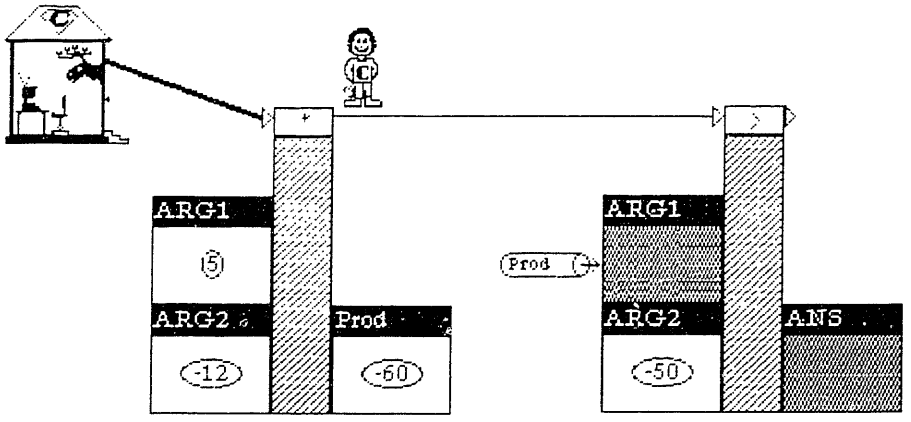
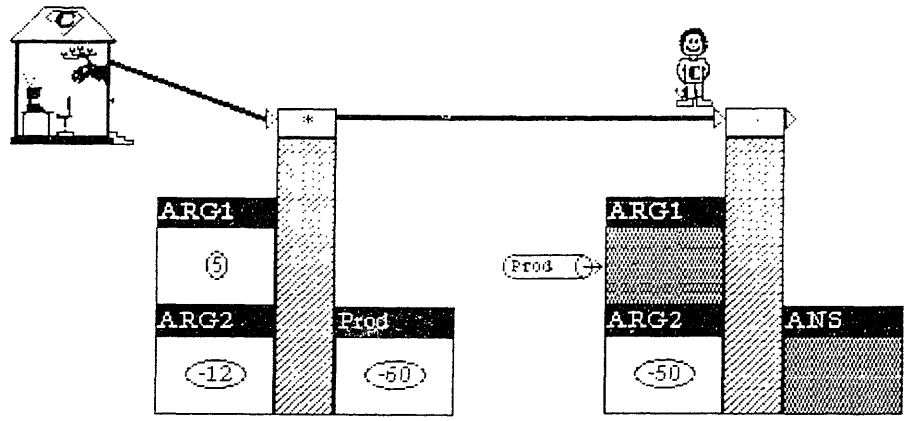


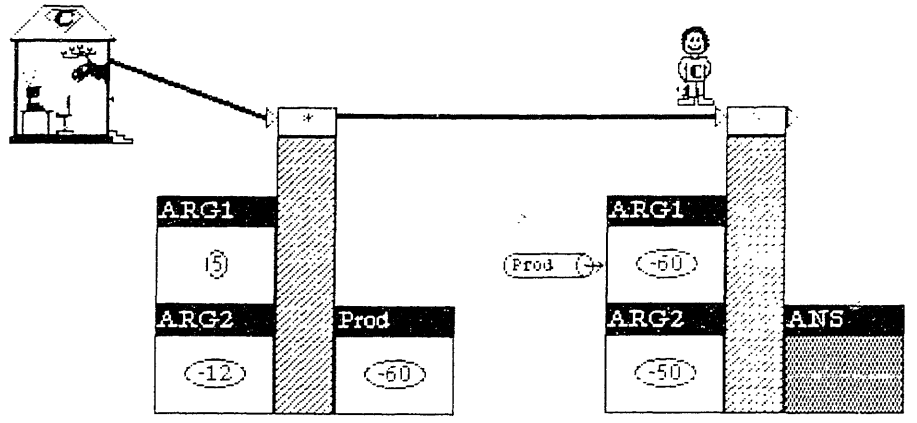
Figure 5.8: Renaming exposes the structure of the configuration.

Attaching a reference pipe to ARG1 specifies a connection between Prod and ARG1 but does not actually cause any flow of data to occur. The actual data flow is triggered when the controller "reaches" ARG1. Control reaches an input variable upon entering a machine and reaches an output variable upon exiting a machine.

The two views of Figure 5.9 show the controller at the entry point to the greater-than machine both before and after the value of Prod has been brought to ARG1.



(a) Before data flow.



(b) After data flow.

Figure 5.9: Two views of the controller entering the greater-than machine.

Upon entering the greater-than machine, the controller causes the machine to fire. As indicated in Figure 5.10, the result is a boolean false value, which is represented by a thumbs-down icon (truth is represented by thumbs-up). As in any configuration, the controller can be sent backwards to the beginning of the computation (see Figure 5.11). This allows us to try out our configuration for numbers other than 5 and -12.

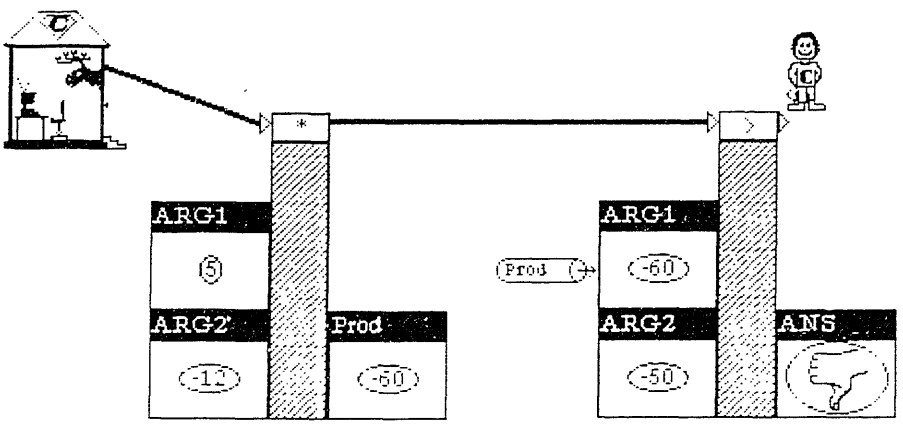


Figure 5.10: The final configuration.

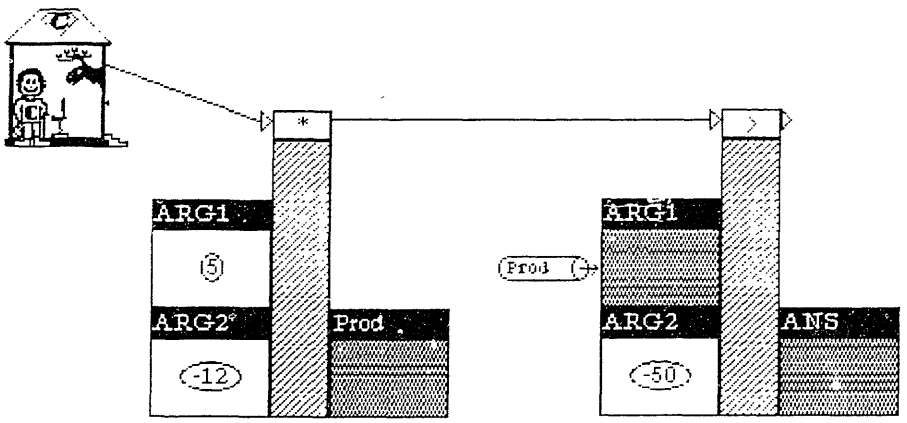


Figure 5.11: Returning to an initial configuration.

5.4.3 Variables

In the above examples, primitive machines used input and ouput variables for holding the arguments and results of the primitive operation. Variables are general Grasp interpretation structures whose purpose is to contain a value. Variables need not be attached to machines - they can be used anywhere to achieve a level of indirection in specifying a value. A Grasp variable which is not attached to a machine is called an *unattached variable.*

As an example, the configuration discussed above compared the result of a multiplication to -50. Suppose instead that we want to compare it to the value of an unattached variable named Num (although names are only comments in Grasp, I will continue to refer to them by name rather than structural features for ease of exposition). We can create a new variable by this name and attach a reference pipe from it to the second argument of the greater-than machine. The resulting

configuration is shown in Figure 5.12. If we now give the Num variable a value of -100, we can run the computation forward as shown in Figure 5.13.

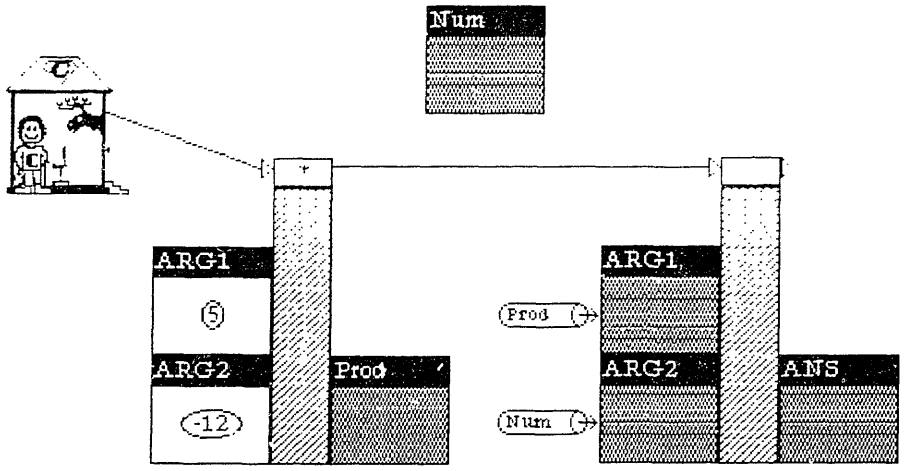
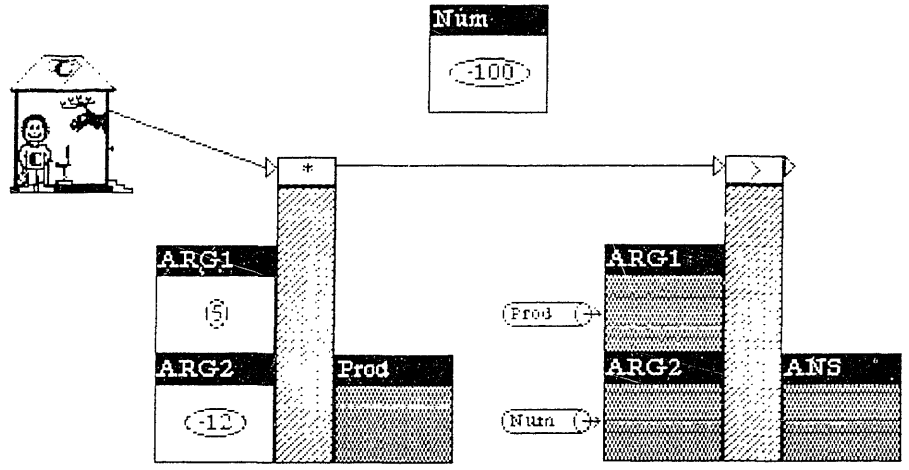
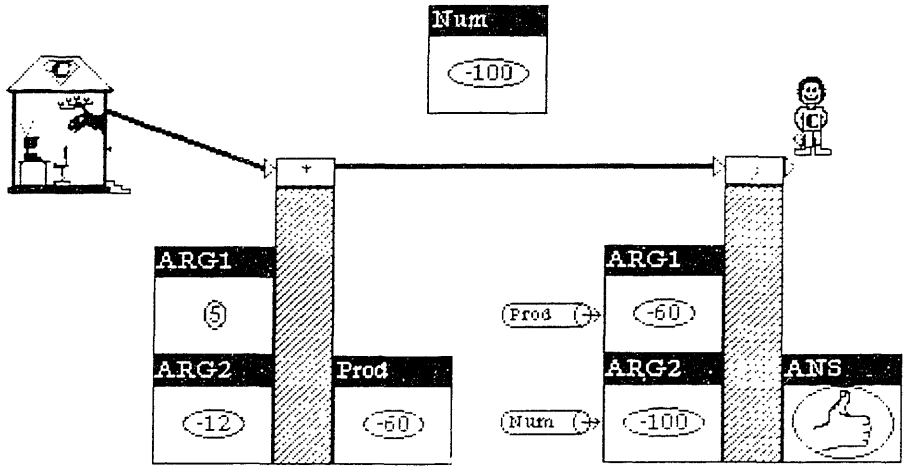


Figure 5.12: Adding an unattached variable to the configuration.

In this particular case, the introduction of the Num variable does not buy us much. We could have just as well changed the value of the second argument of the greater-than machine to -100. The variable does add an important level of indirection, however. If we want to use the value of Num in another part of our computation, we just need a reference pipe from it. In this way, unattached variables in Grasp provide the same level of abstraction for referring to objects that Scheme variables provide. In fact, a Grasp variable is no more than an objectification of a binding in the Scheme environment model. Rather than having environment frames which associate names and objects, Grasp provides an interpretation structure for every association between a "spatial" name and an object.



(a) An initial configuration where Num has a value of -100.



(b) The result of executing the configuration.

Figure 5.13: Before and after views of executing the new configuration.

5.4.4 Smashing Machines

The Grasp model is designed to allow the programmer to easily manipulate the state of the abstract machine. One option open to the programmer is to change the value of a variable. Changing the value of a variable is an example of a *side effect*, which refers to any mutation of the state of an abstract machine. Side effects have many undesirable properties, most important of which is that they obscure the natural parallelism inherent in a computation. An increased interest of the computer science community in parallelism has led to a recent emphasis on side-effectless models of computation, such as those embodied in functional languages. Although the functional programming paradigm has many

advantages, it did not fit into the goals of this project. Because a goal of Grasp is to handle procedures with state (as in Scheme), it must provide some method for programatically changing the value of a variable. Furthermore, since Grasp allows the programmer to interactively change the values of variables, it makes little sense to prevent a program from performing the same manipulation. For these reasons, a structure for mutating a variable is included in the Grasp model.

The *smashing machine* is a special machine in Grasp whose purpose is to change the value of a variable. It corresponds to the assignment operator in conventional langauges; with respect to Scheme, it is the analog of SET! . A representation of a smashing machine appears in Figure 5.14.

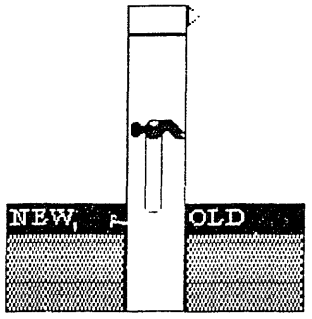


Figure 5.14: A smasher.

One specifies the variable to be mutated with a *smasher*. The smasher specifies a connection between a particular smashing machine and the variable to be mutated. A smasher is represented by the hammer icon,  .

As an example of mutation, consider the situation depicted in Figure 5.15.

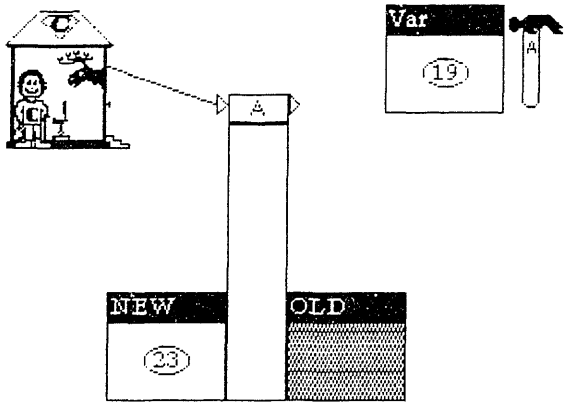


Figure 5.15: Configuration for mutating a variable.

In this configuration, the smasher of the smashing machine is connected to the variable Var. (As with reference pipes, the connection between smashers and smashing machines is represented by name in the interface.) Var contains the value 19, but when the controller runs through the smashing machine, the smasher changes it to 23. The result is pictured in Figure 5.16. The old value contained by the variable appears as the result of the smashing machine. This allows later machines to easily refer to the old value. Saving the old value in this manner means that the side effect can be undone in a simple fashion: when the controller moves backwards through a smashing machine, the result variable is emptied and its value is reinstalled in the mutated variable. Thus, moving the controller backward in Figure 5.16 returns the configuration to the state shown in Figure 5.15.

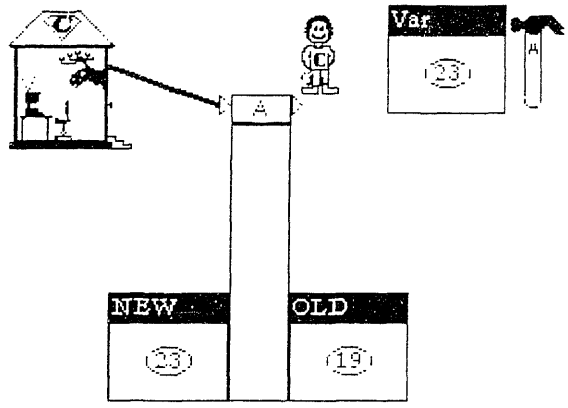


Figure 5.16: Configuration after the variable has been mutated.

Although the above example showed only a single smasher attached to a variable, any number of smashers can be connected to a variable. This contrasts with reference pipes, where only a single reference pipe may lead into a variable. An interesting property of smashers is that they allow the programmer to tell whether or not a variable can be mutated by a program. Furthermore, by finding the smashing machine whence the smasher came, the programmer can determine exactly what part of the program is responsible for changing a given variable.

5.4.5 Compound Machines

Abelson and Sussman note that a programming language is characterized by three mechanisms [Abelson & Sussman 85a]:

1. *Primitive expressions* for specifying basic operations.
2. *A means of combination* for forming more complex expresssions out of simpler ones.
3. *A means of abstraction* for naming expressions and treating them as identifiable entities.

The above sections introduced many of Grasp's primitive structures. Although machines can be wired together with data and control paths, these connections by themselves are a limited means of combination since the resulting configuration is not identifiable as a single unit. This section presents the *compound machine*, which allows simpler structures to be grouped into a single unit. In conjunction with data and control paths, the compound machine serves as Grasp's means of combination by allowing programmers to build complex machines out of simpler ones.

Like primitive machines, a compound machine has input and output variables. The number of these may be specified by the programmer. Unlike primitive machines, compound machines have an *internal structure* which determines what the machine does. The programmer fills in the internal structure of a compound machine in order to specify its behavior.

Consider building a compound machine for squaring a number. First we need to create a compound machine with one input variable and one output variable. The representation for such a machine is shown in Figure 5.17.

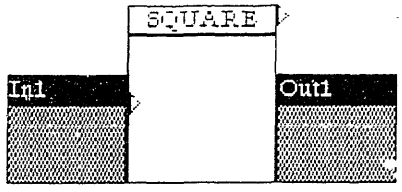


Figure 5.17: A compound machine with an empty internal structure.

The empty space between the two variables indicates that this compund machine does not yet have any interal structure. Grasp allows us to name the compound machine (in this case, SQUARE), but as with variables the name serves only as a comment and has no semantics associated with it.

To specify that this should be a squaring machine, we fill the internal structure with a configuration that multiplies the value of In1 by itself and puts the product in Out1. The completed internal structure is shown in Figure 5.18.

Note that control paths have to be specified in addition to the data paths. The control paths illustrated in Figure 5.18 indicate that the controller should move directly to the multiplication machine upon entering the squaring machine. After the multiplication is done, the controller should exit the multiplication machine and then exit the squaring machine. Note that values flow through the reference pipes labelled In1 when the controller enters the multiplication machine, and the result flows through the pipe labelled Prod when the controller exits the squaring machine.

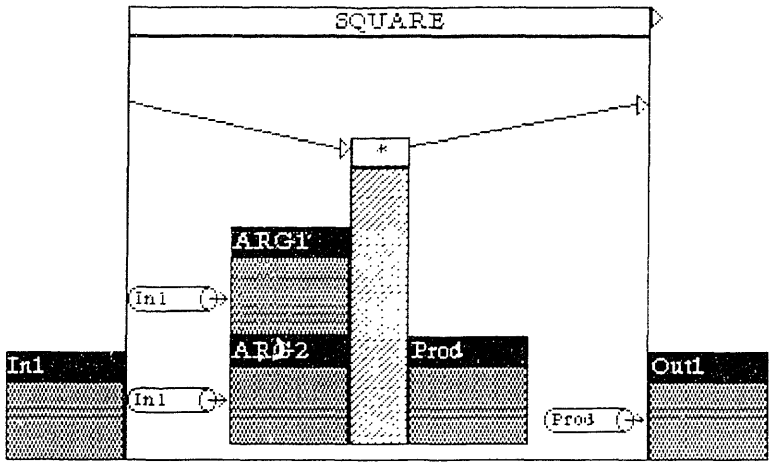


Figure 5.18: A squaring machine.

The important property of compound machines is that they can be treated as single machines. The interface to Grasp provides an abbreviated representation for compound machines which emphasizes their behavior as "black boxes" that compute outputs based on inputs. This representation is shown in Figure 5.19.

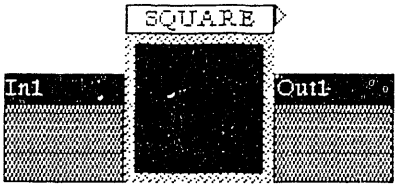
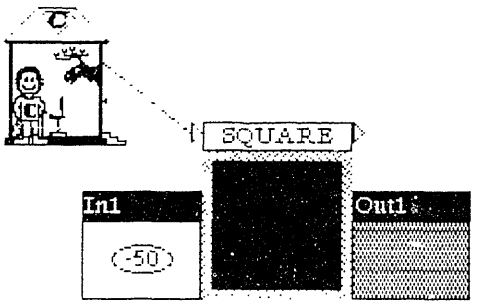
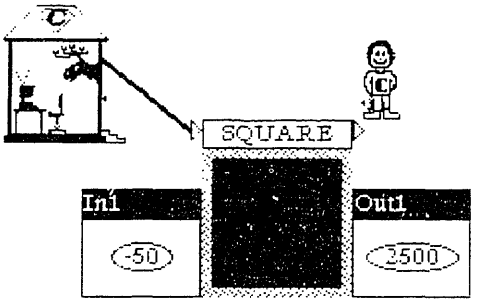


Figure 5.19: SQUARE as a black box machine.

Compound machines are manipulated like their primitive counterparts. Thus, we can form a configuration for testing out our squaring machine on an input of -50. Figure 5.20 shows an initial configuration for performing this computation and the final configuration showing the result.



(a) The initial configuration



(b) The final configuration

Figure 5.20: Initial and final configurations for computing the square of -50.

Since Grasp saves all of the intermediate state of a process, we can inspect this state by expanding the black box into its fuller form (Figure 5.21).

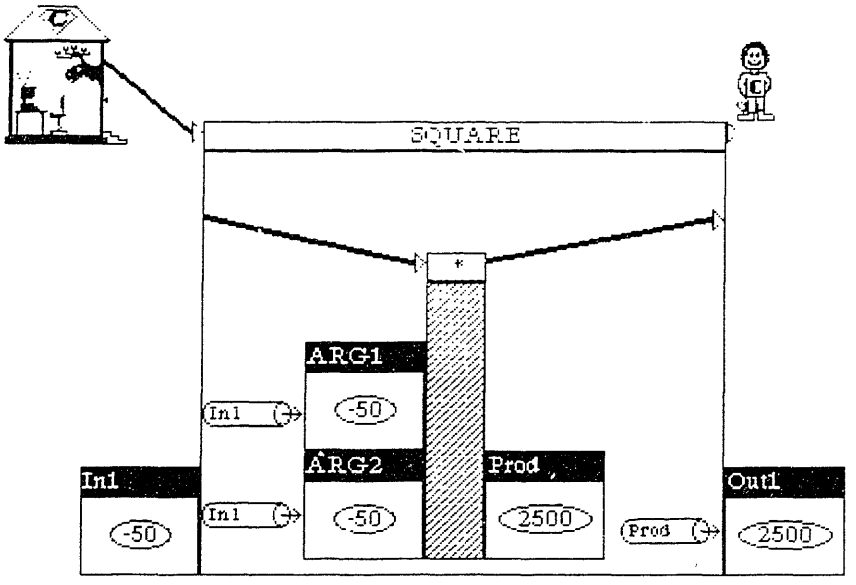


Figure 5.21: The saved state of the squaring computation.

To try the machine out on a different input, we can run the controller back to the control house, insert a new number into In1, and send the controller forward

again. Alternately, we can make a copy of the compound machine and execute it with a different input.

It is important to stress that, like primitive machines, compound machines are the analog of procedure activations and not procedures. Construction of a squaring machine and subsequently running it on -50 does *not* correspond to the following creation and application of a Scheme squaring procedure:

```
=> (DEFINE (SQUARE X) (* X X))
SQUARE

=> (SQUARE -50)
2500
```

Rather, the squaring configuration contructed above much more closely resembles the Scheme expression:

```
=> ((LAMBDA (X) (* X X)) -50)
2500
```

Although this expression creates a squaring procedure, it can only be applied once since it is not named in any way. Immediate application of an unnamed Scheme procedure bears some resemblance to the one-shot behavior of Grasp machines.

5.4.6 Blueprints and All-purpose Machines

The elements of Grasp which support a means of abstraction are the *blueprint* and the *all-purpose machine*. A blueprint is a data object which describes a configuration of Grasp elements. It serves as a template for automatically constructing such a configuration within a program. The all-purpose machine is a special machine in which the construction of the configuration specified by a blueprint takes place. Blueprints serve the purpose of Scheme's procedures, while all-purpose machines are Grasp's equivalent of Scheme's APPLY.

To illustrate these elements. first consider how an all-purpose machine makes use of the information held by a blueprint (we will discuss later how to construct a blueprint). Suppose we have a blueprint which describes the squaring configuration we explored above. We use an all-purpose machine as a site for automatically constructing a new squaring machine. As with compound machines, we need to specify the number of input and output variables that the

all-purpose machine requires. For squaring, both of these numbers are one. Figure 5.22 illustrates the representation of an all-purpose machine in Grasp with one input variable and one output variable.

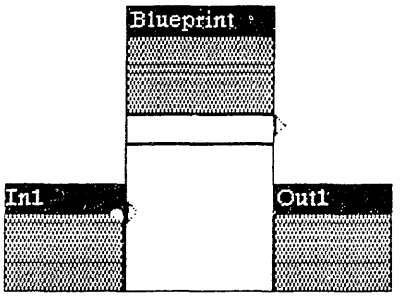


Figure 5.22: An all-purpose machine.

An all-purpose machine is similar in structure to a compound machine except that it has a distinguished variable for holding the blueprint that describes the computation to be built. Figure 5.22 shows that this special variable is located above the internal structure of the machine in the visual representation.

To configure this all-purpose machine for use, we give it a blueprint, fill in its argument (in this case, the number 12), and connect a controller to it. The resulting configuration is shown in Figure 5.23.

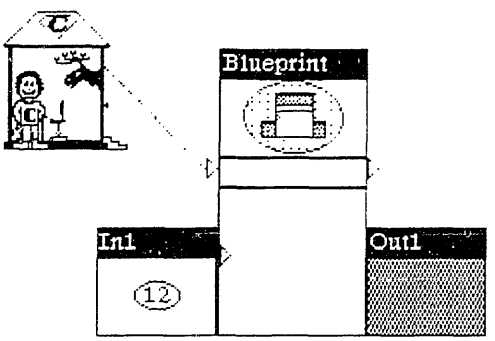
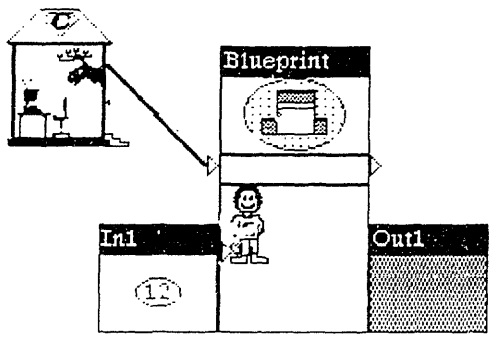


Figure 5.23: A configuration for applying a squaring blueprint to 5.

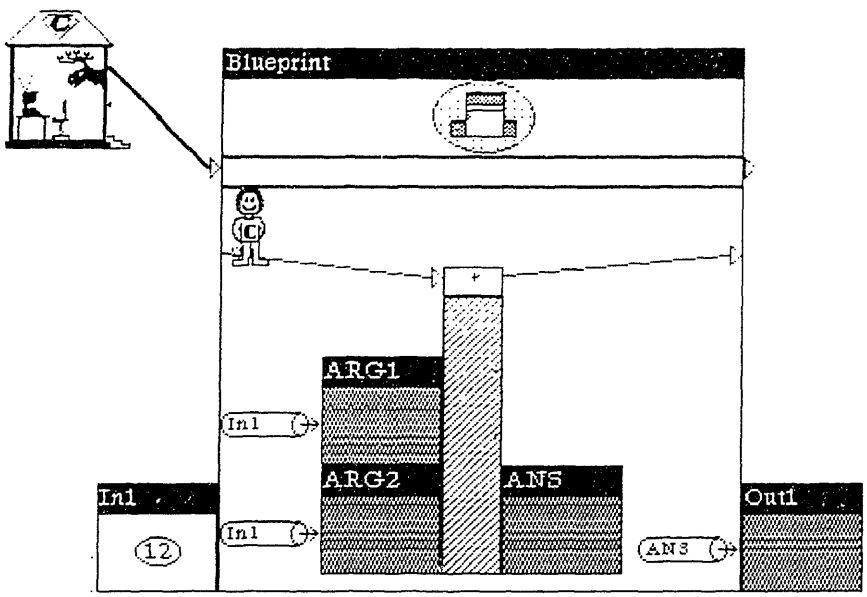
The squaring blueprint is represented as the icon  . This is a shrunken representation of its expanded form, which we will shall examine later.

When the controller enters an all-purpose machine, it first causes data to flow into all of the input variables of the machine. For this purpose, the blueprint variable is also considered an input variable of the machine. In our example, none of the input variables have reference pipes into them, so this detail does not come into play.

As its second action, the controller automatically constructs within the empty internal structure of the machine the configuration described by the blueprint. This process, called *construction*, is analagous to the application process in Scheme, in which a new environment frame is built for a procedure. In fact, we will adopt Scheme terminology and say that a blueprint appearing in the blueprint variable of an all-purpose machine is *applied* to the values in the input variables of that machine. Views of the configuration before and after construction are shown in Figure 5.24.



(a) Before construction



(b) After construction

Figure 5.24: The state of the configuration before and after construction.

As a third step, the controller moves forward as usual through the newly constructed elements. Figure 5.25 shows the final configuration reached for this program. As with compound machines, all-purpose machines can be depicted with

a shrunken representations that hide their internal details. Figure 5.26 shows an abreviated form of the above computation.

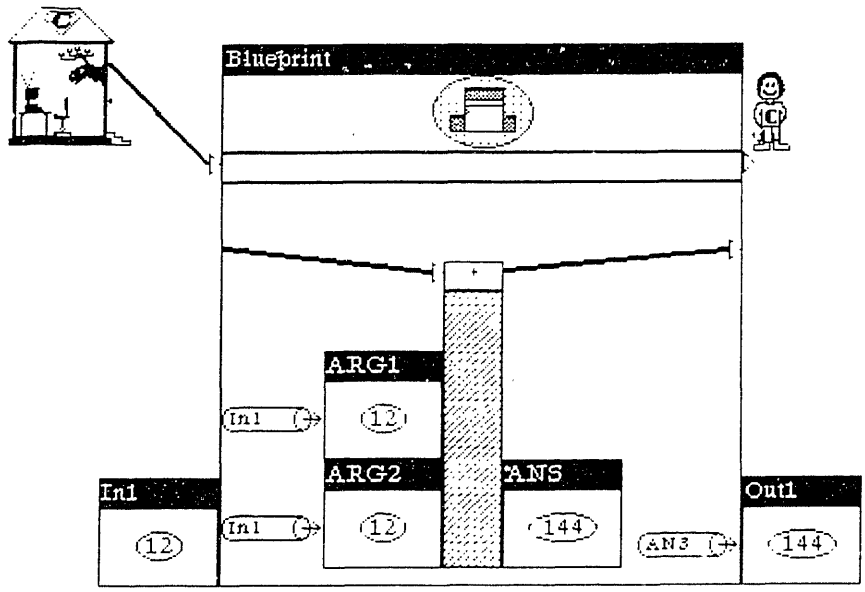
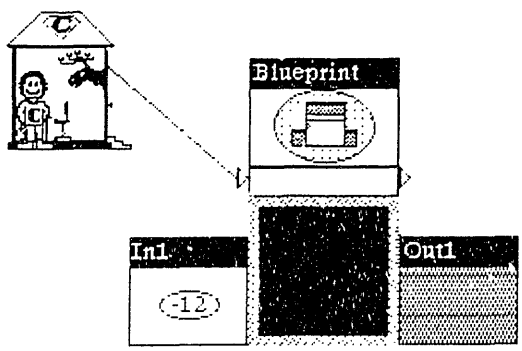
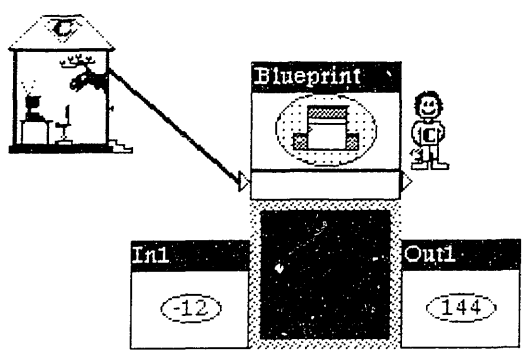


Figure 5.25: The final configuration.



(a) Before



(a) After

Figure 5.26: Black-box representations for the application of the squaring blueprint to 12.

Note that reference pipes may enter the blueprint variable like any other variable. This means that instead of hardwiring a squaring blueprint into an all-purpose machine, we could store it in a variable and refer to that variable via a reference pipe. Figure 5.27 illustrates such a configuration.

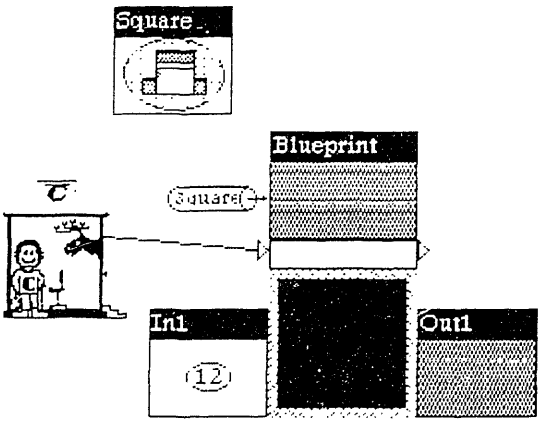


Figure 5.27: Storing the blueprint in an unattached variable.

As explained above, a data object flows into the blueprint variable when the controller enters the all-purpose machine. When the controller moves forward through the configuration shown in Figure 5.27, the result is the configuration displayed in Figure 5.28.

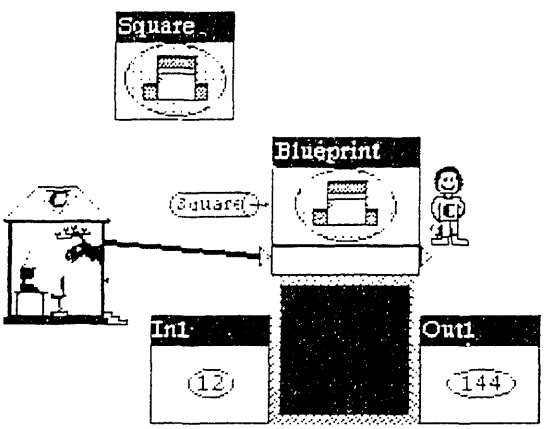


Figure 5.28: The squaring configuration after execution.

Thus far we have shown how blueprints may be used, but we have not shown how they can be constructed. A blueprint is a data object with a template for an all-purpose machine inside of it. We fill in the internal structure of that template in the same way we want the controller to fill in an actual all-purpose machine during the construction process. When it comes time to perform the

construction, the controller can copy the structural information supplied by the template into the internal structure of an actual all-purpose machine.

In order to build the squaring blueprint used above, we first need to create a blueprint for a machine which has one input variable and one output variable. The representation of the blueprint is given in Figure 5.29.

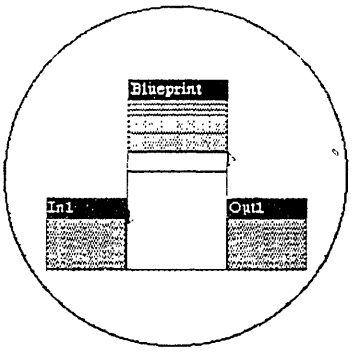


Figure 5.29: A blueprint for a machine with one input and one output.

Now we fill in the internal structure in the same way we want the internal structure to be filled in for each construction of the blueprint. The resulting blueprint is shown in Figure 5.30.

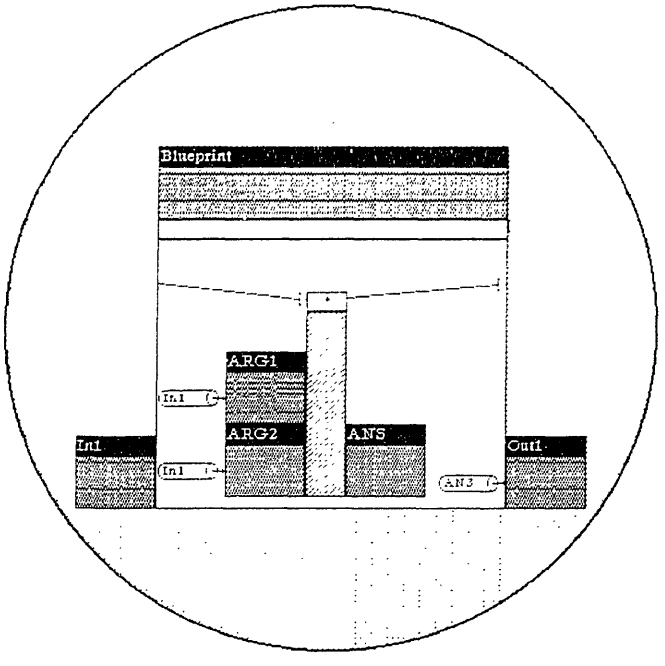


Figure 5.30: A blueprint for squaring a number.

During the construction process, the internal structure of the all-purpose machine template inside the blueprint is copied into the all-purpose machine where the controller is. The copying process must be carried out in a careful manner in order to preserve the appropriate structure. Complexities are introduced by the fact that blueprints may have reference pipes or smashing machines associated with variables not within the blueprint itself. These are Grasp's equivalents of free variable references within the body of a Scheme procedure. Scheme approaches the situation with lexical scoping; Grasp handles it with an appropriate copying algorithm. The copying algorithm is the equivalent of lexical scoping for a device programming system.

As an illustration of the copying scheme, consider a blueprint which describes adding a number to the value of variable A, where A is outside the "scope" of the blueprint. This situation is depicted in Figure 5.31.

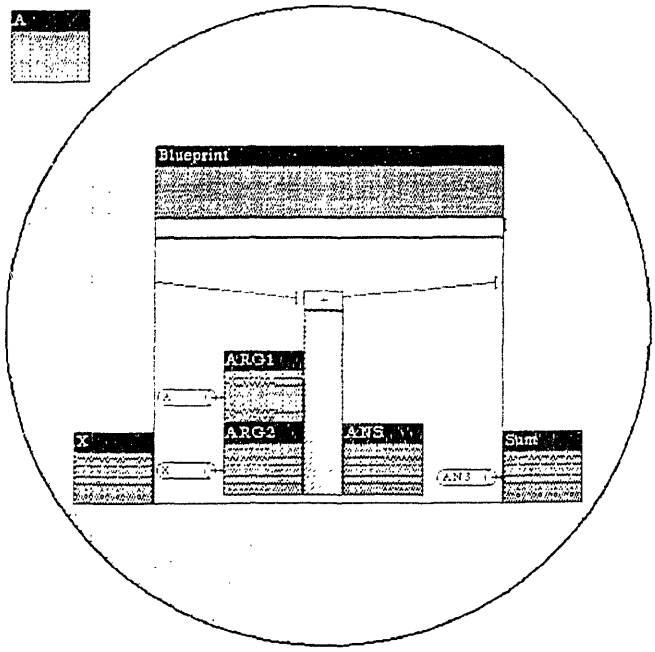


Figure 5.31: A blueprint for adding a number to the value of A.

Suppose we put the blueprint in a variable called ADD-A and use it in an all-purpose machine with the argument 3. This situation is indicated in Figure 5.32, which also shows that we have set the value of A to be 7.

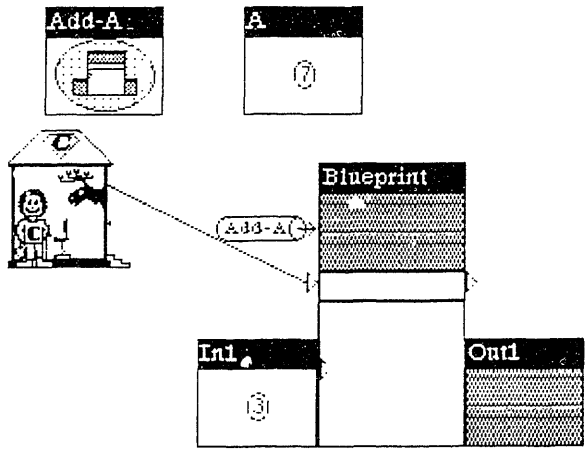


Figure 5.32: A configuration for applying ADD-A to 3.

When the controller builds the internal structure to the all-purpose machine, the variable In1 is matched with the variable X in the blueprint, and the variable Out1 is matched with the variable Sum in the blueprint. After a new primitive addition machine is placed within the internal structure of the all-purpose machine, reference pipes are used to associate the second addition argument with In1 and Out1 with the result of the addition. This copies the data flow connections between two variables that are both within the blueprint itself. Any reference pipes not copied by this scheme, such as the reference for A in our example, must refer to variables outside of the blueprint. For this case, a new reference pipe to that external variable is created and inserted into the internal structure at the appropriate point. The result of this process is shown for our example in Figure 5.33.

For side-effects, the copying of smashing machines introduces similar complexities. If both the smashing machine and smasher are within the blueprint, then copies of them are created and the structural connection between them is maintained in the newly constructed configuration. If a smashing machine is connected to a smasher that is attached to a variable external to the blueprint, a new smasher must be added for every application of the blueprint. As an example, observe the blueprint in Figure 5.34, which changes the value of the variable X to be one greater than its old value (and also returns the new value).

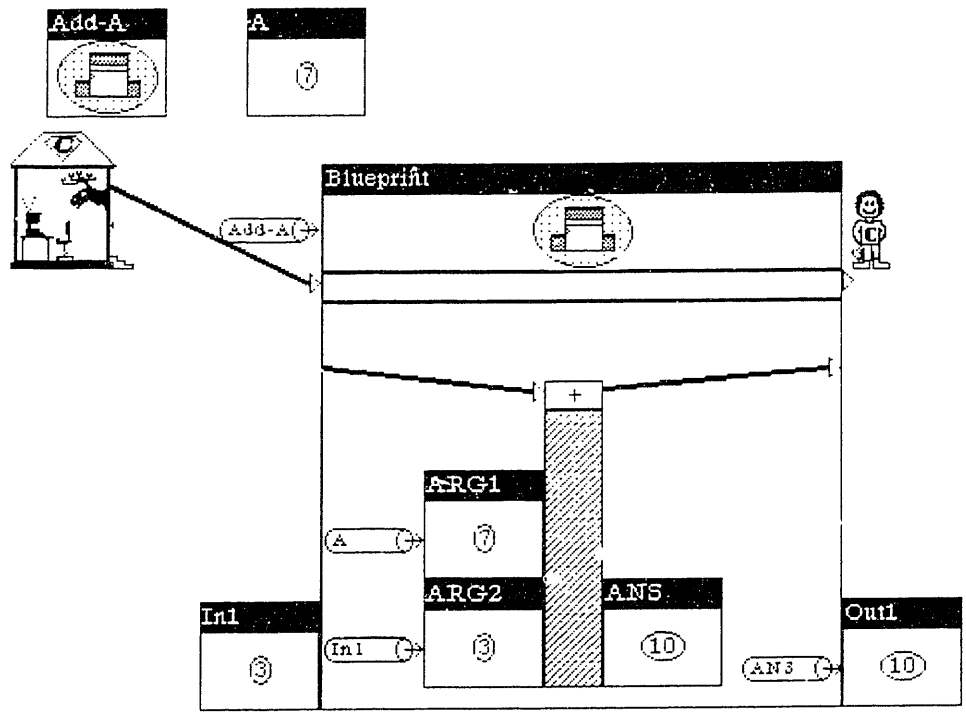


Figure 5.33: The result of the copying algorithm for the ADD-A example.

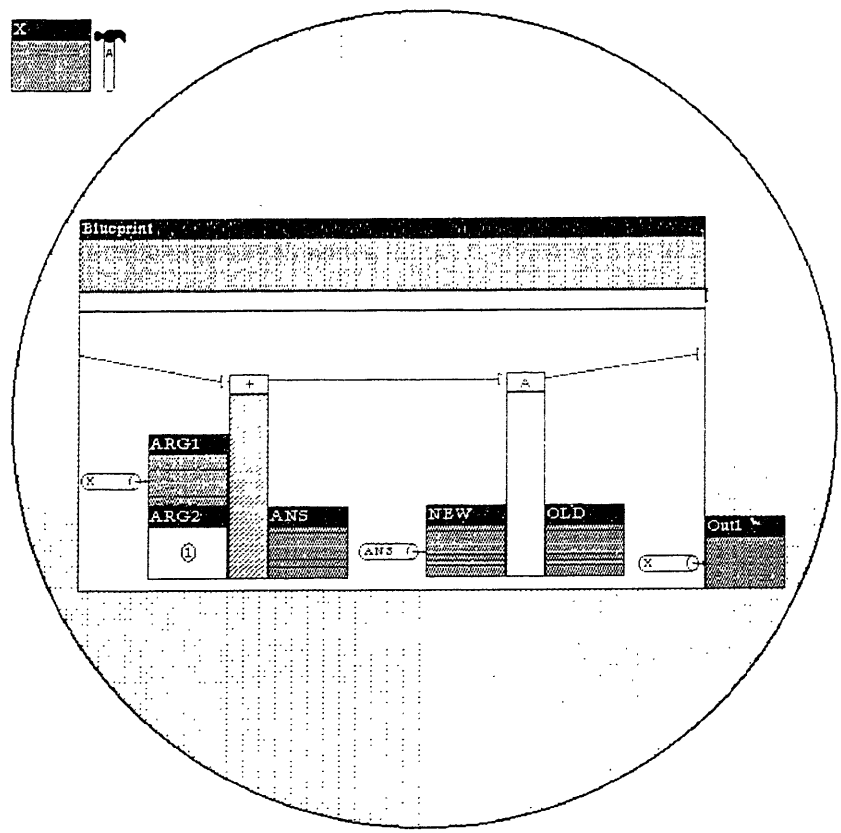
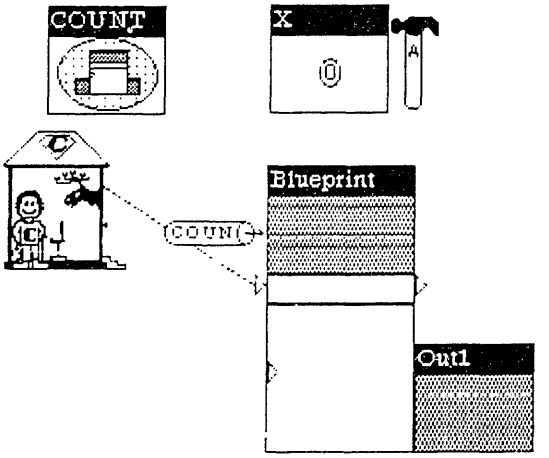


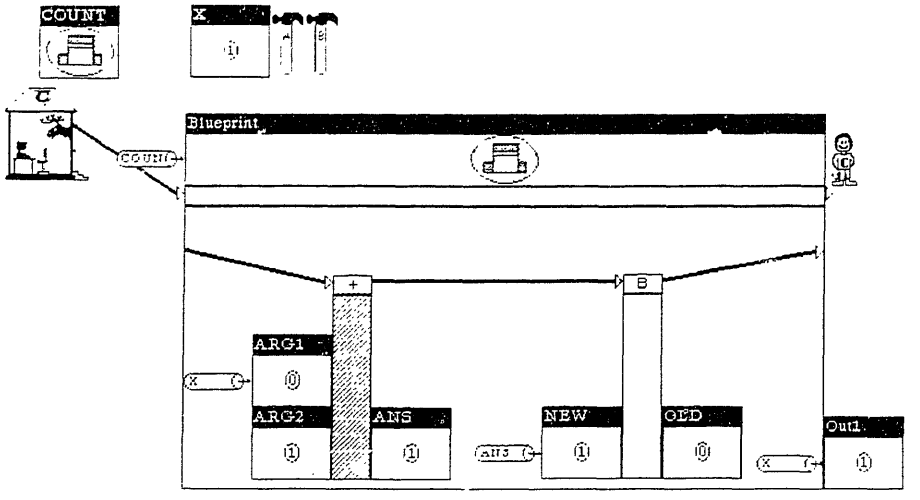
Figure 5.34: A blueprint which mutates an external variable.

If we apply this blueprint (which we shall store in the variable COUNT), then a second smasher is added to X during the construction process. This situation is illustrated by the before and after views of Figure 5.35. In this figure, the structural connection between smashing machine an smasher has been highlighted by changing the name of the new smasher to B.

This section has touched upon the basic properties of blueprints and all-purpose machines in Grasp. There are many interesting uses for these elements, such as the Grasp equivalent of higher order procedures in Scheme. For examples of blueprints being passed as arguments and returned as results, refer to the Appendix.



(a) Before



(b) After

Figure 5.35: Applying the COUNT blueprint adds a new smasher to X.

5.4.7 Conditional Machines

The last major element of the Grasp model to introduce is the *conditional machine*. Until this point, we have only considered examples where the controller could only move along a single control path through a configuration. Conditional machines allow for the structured branching of control paths. Modelled after Scheme's COND special form, conditional machines are built out of a number of clauses that consist of a predicate and a sequence of actions to be performed. Predicates are executed in order until one yields a boolean truth value; the sequence of actions in the clause of the true predicate are then performed. The representation for a conditional machine with two clauses is shown in Figure 5.36.

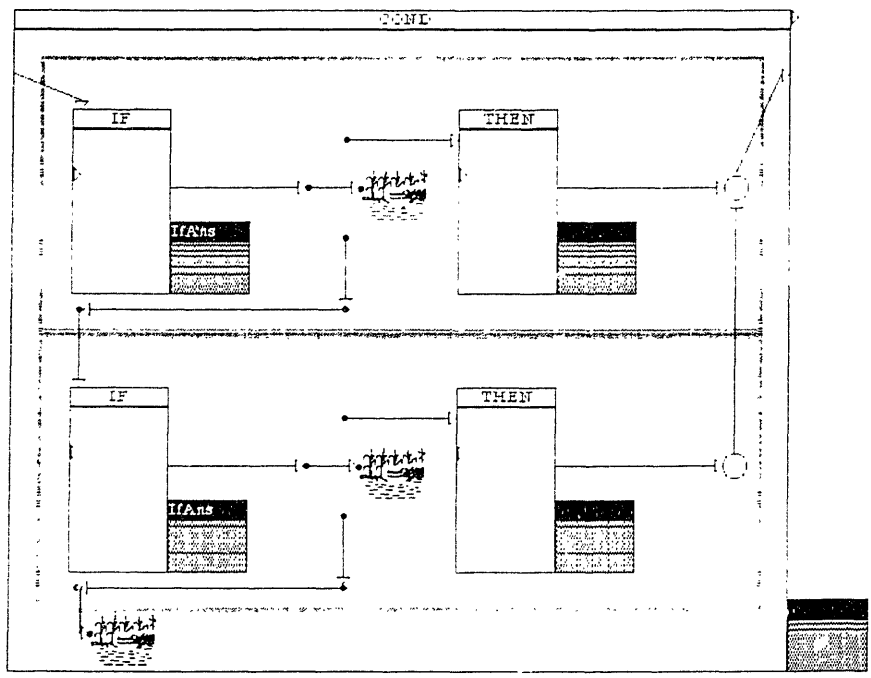
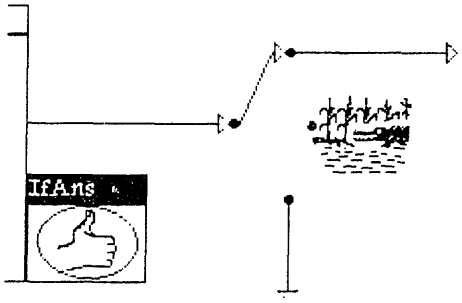


Figure 5.36: A conditional machine with two clauses.

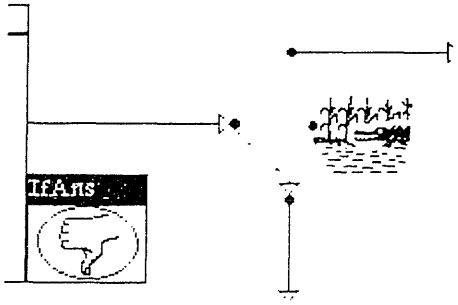
Each clause is enclosed in a gray rectangular border in the visual representation. The boxes labelled IF are *predicate machines*, whose purpose is to test for a truth value. Action sequences are performed by *consequent machines*, which are represented as the boxes labelled THEN in Figure 5.36.

A predicate machine is a special machine which controls the direction of a *control switch*. The relationship between the output variable of the predicate machine and the control switch is illustrated in Figure 5.36. If the output variable of the predicate machine contains the boolean true value (Figure 5.37a), the switch flips to a position which allows the controller to go forward to the

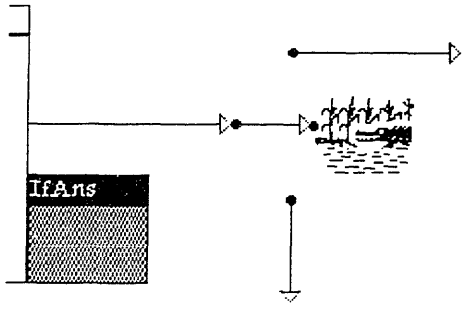
consequent machine in the current clause. If the boolean false value is present in the output variable of the predicate machine (Figure 5.37b), the switch flips to a position which send the controller to the next clause. If the output variable does not contain a truth value (Figure 5.37c), the switch flips to a position which sends the controller into a reified error state (indicated in the representation by the swamp icon, ).



(a) True value in output variable



(b) False value in output variable



(c) No value in output variable

Figure 5.37: The output variable value detemines the direction of the control switch.

A control switch by itself would be similar to a GOTO statement in conventional languages. In an attempt to avoid control going off in arbitrary directions, every control switch is associated with a *control join* in a conditional clause. The control join elements represents the confluence of two control paths. Control joins are represented by the icon ◯ in the representation of Figure 5.36. Their state is also dependent on the value in the output variable of the predicate machine. For a true value they appear as ◴; for a false value they look like ◵. Allowing control switches and joins to appear only in pairs within conditional clauses means that the possible ways the controller can mode forward is highly constrained. In particular, the controller must exit the conditional machine on its single control path regardless of the branches it took within the machine.

A structural aspect of the conditional machine which is not evident in the representation of Figure 5.36 is that the output variables of the consequent machines are all connected to the output variable of the conditional machine. Conceptually there is a data join at the output variable to the conditional. Since only one consequent machine can be activated within any conditional machine, there is no problem of multiple data values being specified for the output variable of the conditional. Right before the controller exits the conditional machine, the output variable of the conditional machine takes its value from the output variable of the consequent machine in the clause through which the controller moved.

The details of the structure of the conditional machine are best illustrated by an example. Consider the compound machine for computing absolute values that is shown in Figure 5.38. This machine uses a conditional machine to test the sign of the input number and negate the number if it is negative. The black-box representation labelled POSITIVE? denotes a machine that returns true if the value of NUM is greater than zero, and false otherwise. The black-box representation labelled NEGATE designates a compound machine that negates the value of NUM. Note that the output variable of the predicate machine in the second clause has already been assigned a the boolean true value; this situation is the analog of a conditional clause beginning with an ELSE in Scheme or a T in other Lisps.

Figures 5.39-42 show various points in the computation of the absolute value of -57. In Figure 5.39, the controller has just exited the predicate machine of the first clause. The "thumbs-down" indicates that the number in question is not

positive, and the control switch has flipped to allow the controller to go to the second clause. By Figure 5.40, the controller is on its way out of the conditional machine. Note that the final answer is sitting in the Clause2 variable but has not yet propogated. The controller exits the conditional machine in Figure 5.41, allowing the result of the second clause to appear in the output variable CondAns. Finally in Figure 5.42, the controller leaves the ABSOLUTE-VALUE machine and the final answer appears in the ouput variable Abs.

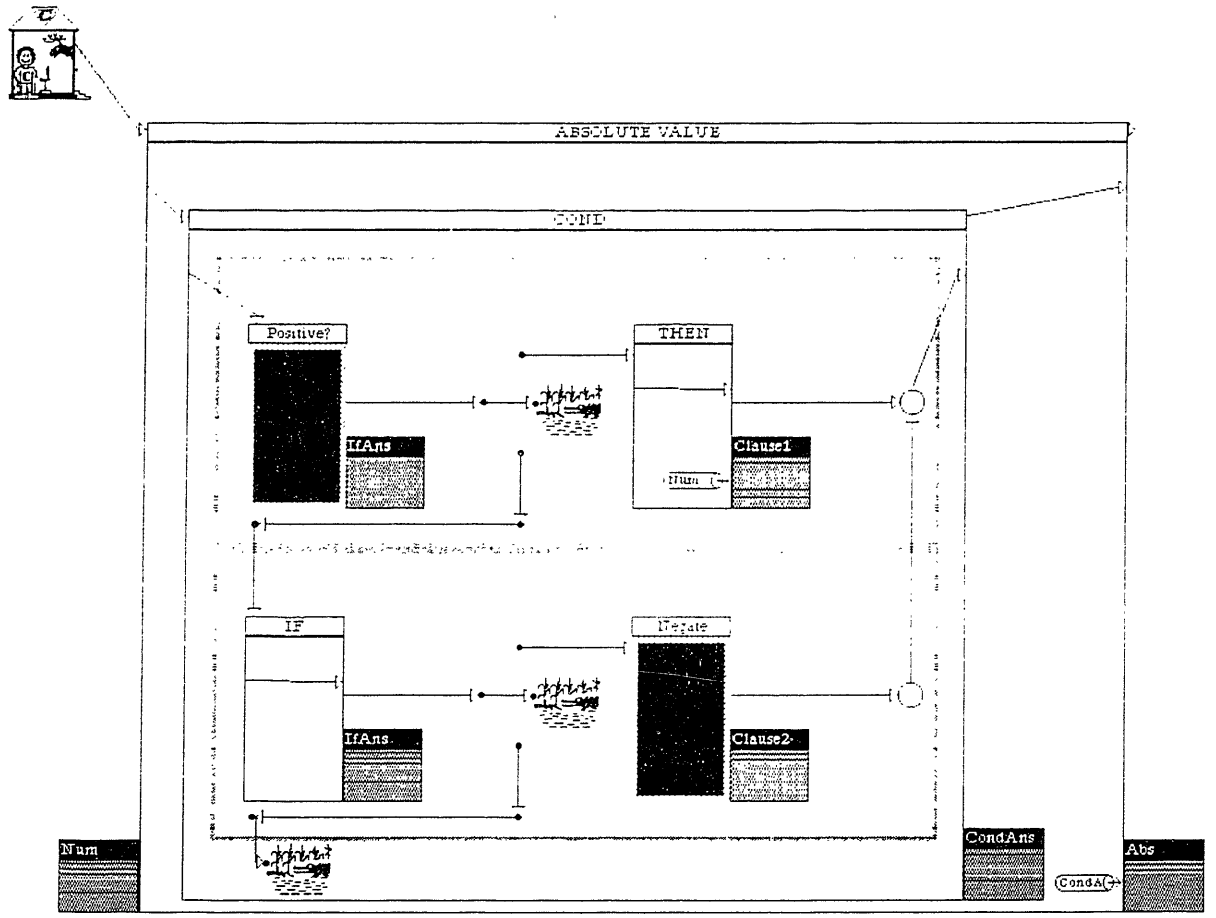


Figure 5.38: A compound machine for computing the absolute value of a number.

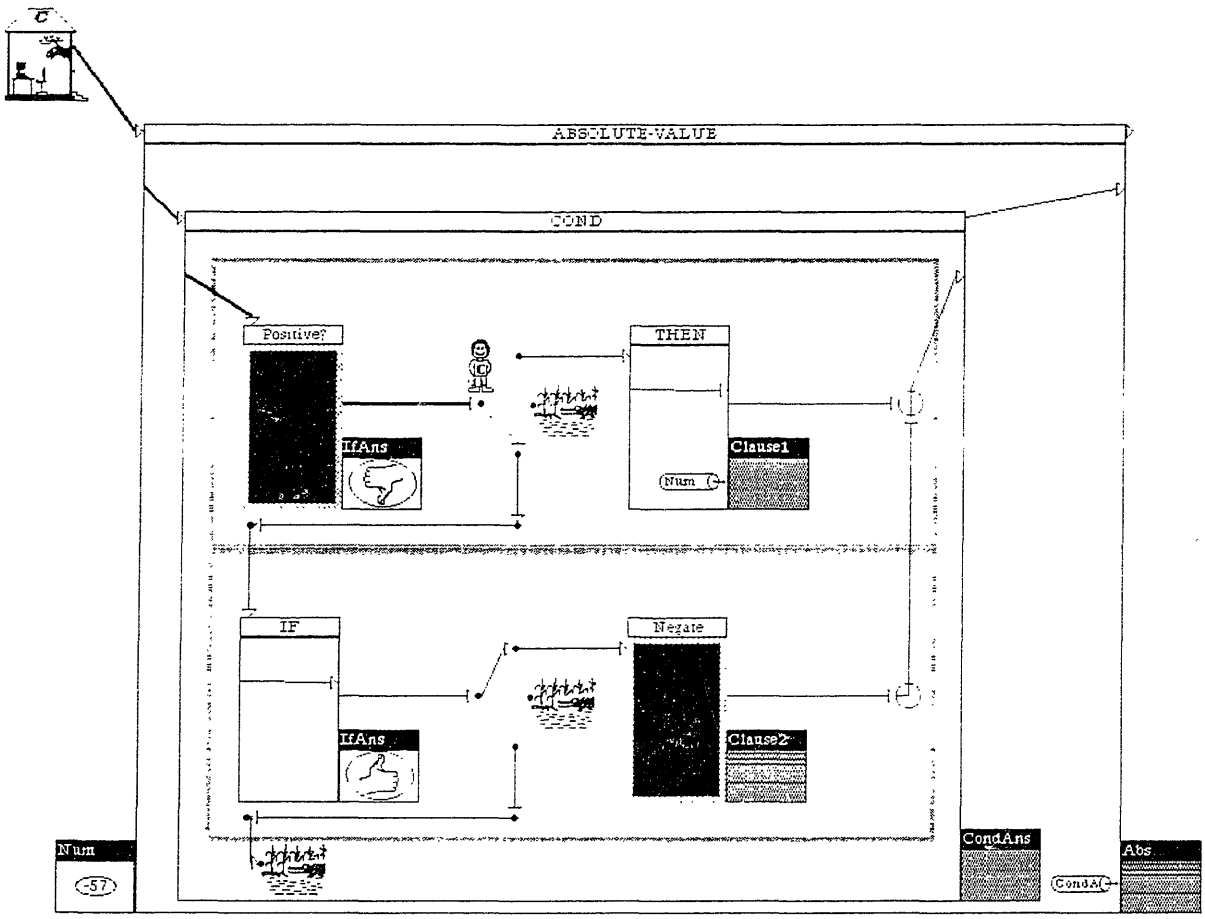


Figure 5.39: The controller exits the first predicate machine.

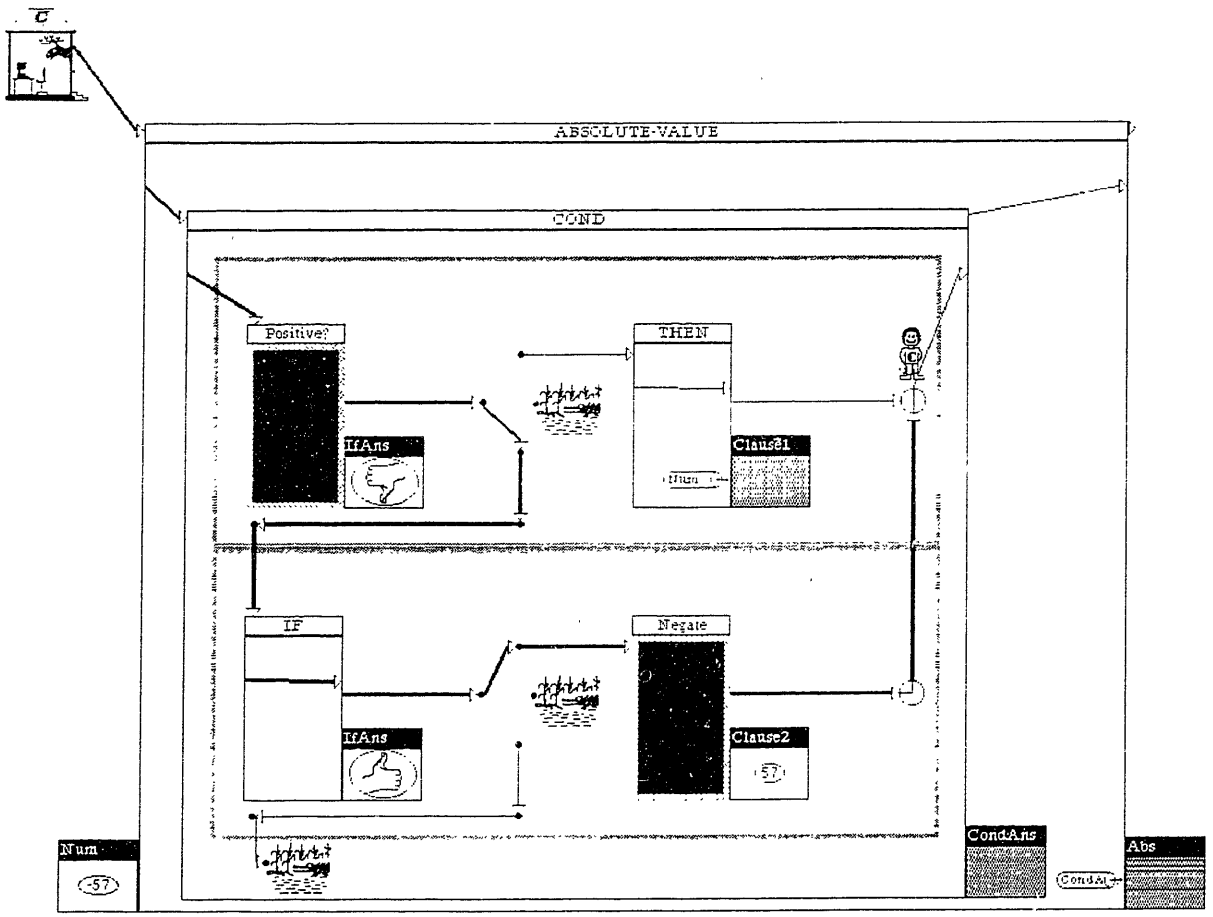


Figure 5.40: The controller before exiting the conditional machine.

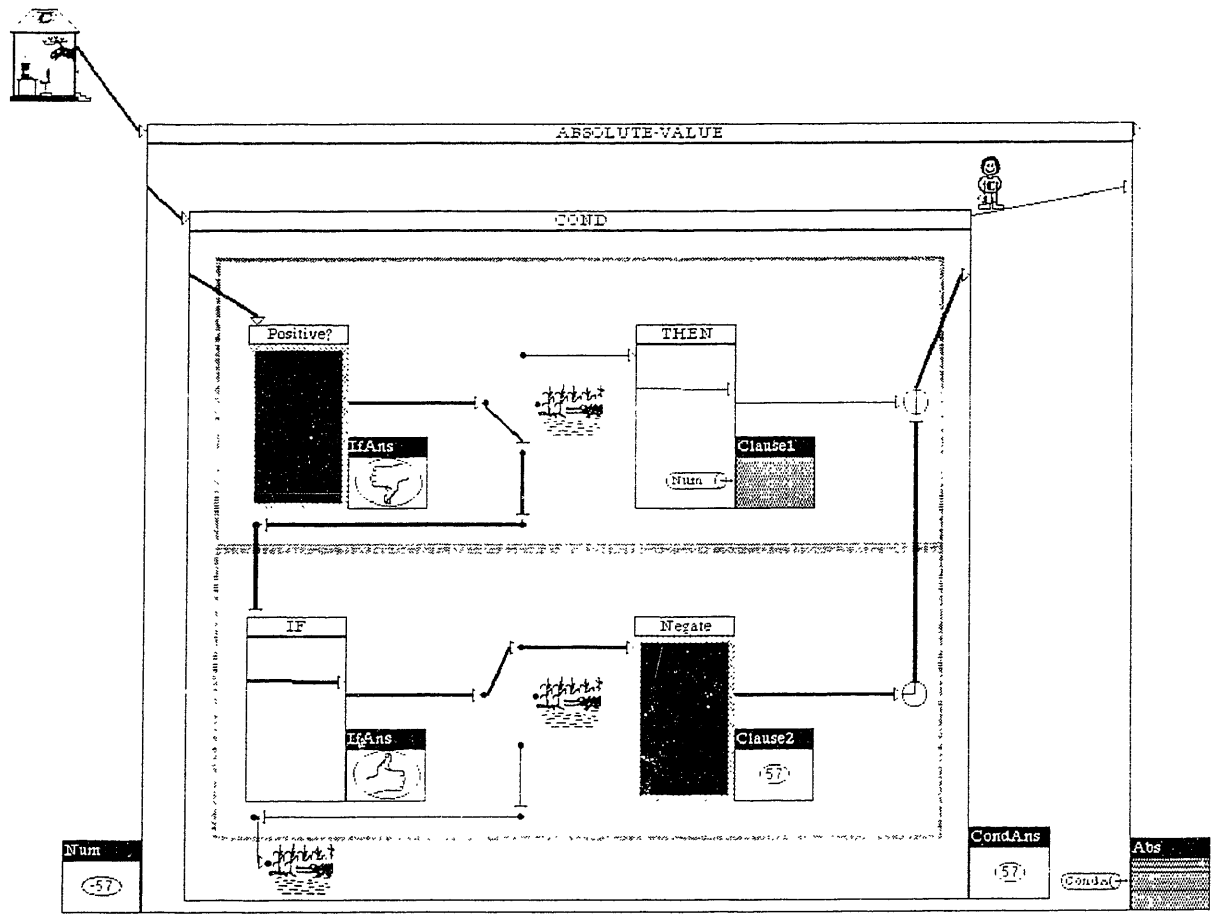


Figure 5.41: The controller after exiting the conditional machine.

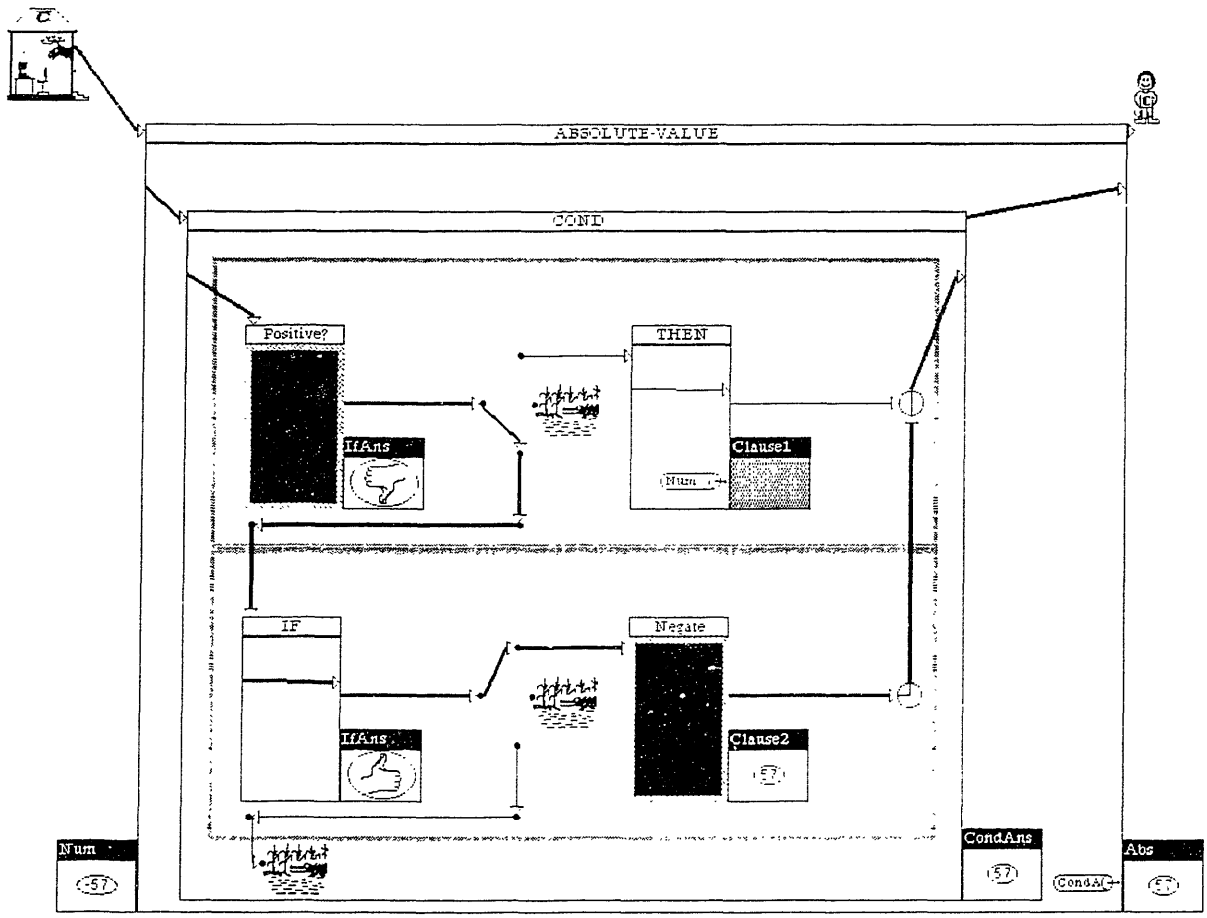


Figure 5.42: The controller after exiting the ABSOLUTE-VALUE machine.

5.5 Summary

Embodying the procedural paradigm in a device programming model requires viewing computational elements as physical devices. In the Grasp system, a principle of reification is used to turn many implicit structures of a procedural programming language into explicit ones. This chapter describes the reified elements of the Grasp model, including structures which implement procedures, procedure activations, variable bindings, variable references, and control flow. An interesting outcome of the reification process is that programs can be viewed as computational time lines where control can move forward and backward through a process. Although the device-based Grasp model differs in many respects from the expression-oriented Scheme model, Grasp is similar to Scheme in many respects, especially in its support of first-class procedure objects (blueprints).

CHAPTER 6

A VISIBLE AND MANIPULABLE INTERFACE TO THE GRASP MODEL

The previous chapter motivated the elements of the Grasp model. The purpose of this chapter is to describe the design of Grasp's user interface. As noted above, this order of presentation does not reflect the order in which the project actually progressed. Representational considerations were often a strong motivating factor of crafting the elements of the model; after all, it is important to have computational elements amenable to visible and manipulable representations if the goal is to display them through a graphical medium. The order of presentation followed here is intended to clarify the conceptual matter by factoring the representational issues from the issues more intrinsic to the formation of the model.

Visibility and manipulability are aimed at making a transparent interface that facilitates the programmer's interactions with the abstract machine. Allowing novices to easily inspect and manipulate the kinds of reified elements introduced in Chapter 5 should help them gain a firmer understanding of the structure of procedural programs. In Grasp, incorporating these two principles into the interface leads to a system where the user can believe that the visual information displayed on the screen *is* the state of the system. Since users will naively make this assumption anyway, it is better to purposely set out to maintain this illusion than to contradict their expectations and intuitions. DiSessa calls this powerful principle *naive realism*; this principle is the cornerstone of the Boxer system [diSessa 85a]. A similar idea is expressed by the designers of Star when they note how "the display becomes the reality" in their system [Smith *et al.* 82]. The following sections explore how visibility and manipulability make the structure of Grasp programs "real" to the user.

6.1 VISIBILITY

The principle of visibility is intended to help programmers build more robust models by making information about the state and behavior of the model available to them on the display screen. Two important questions to consider are:

1. *What* information should be displayed?

2. *How* should that information be displayed?

Since the goal of this project is to make the structure of programs explicit, the answer to the *what* question is fairly straightforward: display as much structural information as possible. Since all the relevant information may not be able to fit on a single screenful, some means must be provided for structuring the information so that only a part of it need be viewed at any one time. Yet, the programmer should have easy access to all the information which is not immediately visible on the screen.

On the other hand, the answer to the *how* question is not so clear. Because pieces of programming structure are abstract in nature they do not conjure up well-defined images. Other criteria must be used in order to choose the graphical representations for computational information. The following sections describe the criteria that were used in designing visible representations for the Grasp system.

6.1.1 Direct Mappings

Visual representations should demonstrate a clear relationship to the underlying structure they are presenting. A *mapping* describes the correspondence between the visual information and the structural information. The *directness* of the mapping is a measure of how well the properties of the underlying structure match the properties of the visible representation. In a one-to-one mapping, the most direct kind of mapping, the underlying structure can be completely deduced from the visual structure alone; the converse is also true. Less direct mappings may show important structural features, but not all of them. For example, the box-and-pointer notation discussed in Chapter 3 is a one-to-one mapping for list structures, while parenthesis notation is much less direct since it does not show sharing. Since the Grasp system is concerned with the presentation of structural information, direct mappings are a chief

consideration in the design of visible representations for the Grasp computational elements.

The key goal of direct mappings in Grasp is to visually represent the structure of the underlying computational configuration. Each piece of structure should have a corresponding visual representation so that the structure can be determined from the visual representation. Beyond this constraint, other guidelines may be used to determine exactly what the representation should look like.

Consider the representation of booleans in Grasp. There are two pieces of information that need to be represented for a boolean: its type and its value. Both of these are encoded in its visual representation. The boolean truth value is represented by a "thumbs up" icon; the boolean false value by a "thumbs down" icon (Figure 6.1).

Truth Falsity

Figure 6.1: Boolean icons.

Here the hand with protruding thumb represents objects of type boolean, and the orientation of the hand encodes the value. Of course, the number of different visual encodings for these two pieces of information are practically unlimited. The choice of a hand rather than some other representation is due to additional considerations which are discussed later.

Although one-to-one mappings are desirable for visual representations, in some cases they are not feasible. Consider the choice of representation for a reference pipe. A reference pipe has three pieces of information: its type, the variable that is the source of the pipe, and the variable that is the target of the pipe (the target of a reference pipe is undefined in the case that it is not connected to a variable). The type and target variable are represented in a direct manner in the Grasp interface. A unique pipe icon distinguishes the reference pipe from other types of objects. When a pipe is connected to a target variable, it appears in a special position relative to the variable. Thus, in Figure 6.2, the reference pipe labelled Ref has the variable A as its target, while the other reference pipes have undefined targets.

Figure 6.2: Connected and unconnected reference pipes.

The representation of the source variable for a reference pipe, however, is ambiguous. Grasp encodes this information by displaying the name of the source variable on the pipe. An ambiguity arises because more than one variable can have the same name in the Grasp environment. Consider the example in Figure 6.3 - it is impossible to determine from the provided representations which variable A is connect to variable B.

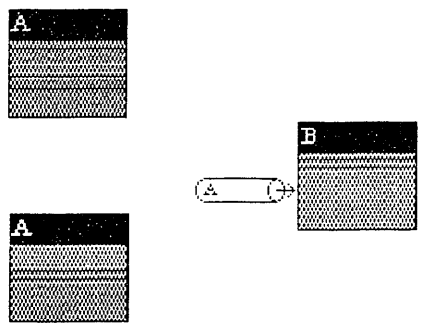


Figure 6.3: An ambiguous referencer.

This does not mean that the structural information is totally inaccessible to the programmer. The user is free to change the name of one of the variables labelled A; the effects of such a change are immediately displayed on the screen. Suppose the topmost variable A is really the source of the reference pipe. Then changing its name to Z results in the configuration illustrated in Figure 6.4.

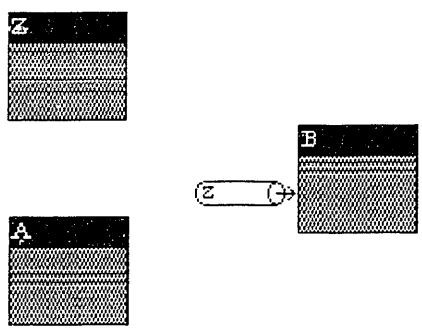


Figure 6.4: The result of changing a variable name.

By observing the effects of such a change, the user can indirectly determine the hidden structural connection.

Why not show an explicit structural connection between the reference pipe and its source variable? In the case of control paths, explicit connections are shown at both ends of the path. Why not do the same for reference pipes? The problem is that control is local in computations whereas reference is not. The notion of lexical scoping in a language means that the value of a variable can be used in a place arbitrarily far away from the location of the variable. To explicitly show both ends of a data flow connection would result in a veritable spaghetti of reference pipes crisscrossing the screen. There is a tradeoff here between the potential ambiguity which results from missing information and the potential confusion which results from presenting too much information. In this case, Grasp shows less information to keep the screen less cluttered.

Despite the above discussion, the method of determining the source variable of a referencer could clearly stand some improvement. Having to query a variable for its underlying structure smacks of the indirection of command language interfaces, in which the programmer indirectly pokes around to discover the structure of a program. A better approach would be to give the user the option of temporarily showing an explicit structural connection between the reference pipe and its source variable. It is easy to imagine querying a reference pipe to show an uninterrupted data path to its source variable, or asking a variable to display all outgoing paths. Such a path might be displayed as in Figure 6.5.

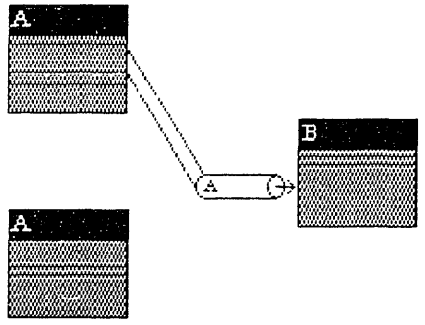


Figure 6.5: An uninterrupted data path.

In some cases, direct mappings conflict with the pedagogic goals of the system. Consider the case of blueprints. As discussed in Chapter 4, novices tend to attach too much meaning to the names of procedures; often they believe that a procedure knows the name by which it is referred to in the environment. The Grasp interface stresses the object nature of blueprints and shows that the only information they hold is a template for an apply machine. However, since the

visual representations for blueprints are huge, for the purposes of putting them in variables they are shown in the shrunken form shown in Figure 6.6.

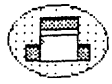


Figure 6.6: The shrunken icon for blueprints.

Note that the type of the object is indicated by the icon, but there is no way to distinguish one blueprint icon from another in the shrunken form. At such a fine level of detail, it is not possible to show structural features of the template. It is tempting to at least give each blueprint a name and to display that name within the shrunken icon. The name would not have any semantic import; much as in the case of variable names, it would serve as a comment and as a means of distinguishing objects. Yet, giving blueprints a name would also reinforce a model that a name is a piece of information intrinsic to a blueprint - a model Grasp strives to avoid.

The most straightforward way of handling the naming of blueprints in Grasp is to take the Scheme approach - make a binding between a variable and a blueprint. Grasp variables hold onto any object, including blueprints; furthermore, they have names associated with them as comments. For example, by storing a squaring blueprint in a variable named SQUARE, we are able to effectively able to name the blueprint. This siutation is illstrated in Figure 6.7. A blueprint that is not stored in any variable resembles the "anonymous" Scheme procedures created through LAMBDA but not named by DEFINE.

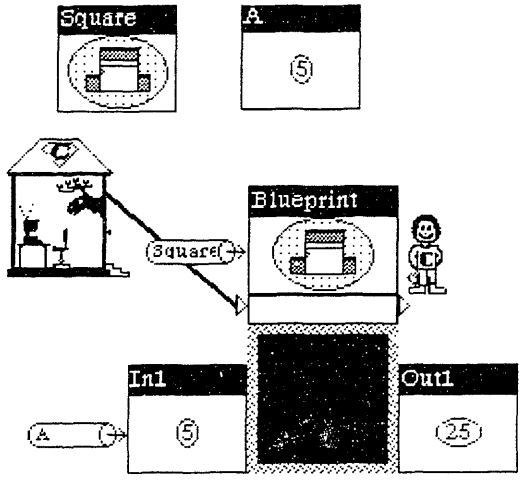


Figure 6.7: A squaring configuration.

Another structural connection which is not reflected in the Grasp interface is the sharing of blueprints. For immutable objects like numbers and booleans, a behavioral difference between shared and unshared objects is impossible to detect. Thus, in Figure 6.7, it does not matter whether the value in the variable In1 is a copy of the value in variable A or the exact same value stored in variable A.

Blueprints on the other hand are mutable objects - we can expand them and change the configuration which they specify. It therefore matters a great deal whether the value of the variable labelled Blueprint is the same object as the value in the Square variable or just a copy of it. In the underlying model, the two are in fact the same object. However, showing the explicit connections between blueprints would clutter the screen in much the same way as showing explicit data flow connections between variables. For this reason, Grasp does not show the explicit connection between blueprints.

As with reference pipes, however, the programmer should be provided with a way of unveiling hidden structure in the case of shared data objects. Perhaps the user could request that explicit sharing be shown for a particular blueprint. The representation for such sharing might appear as in Figure 6.8, where a graphical "tentacle" connects the blueprint in the Square variable to another variable that shares it.

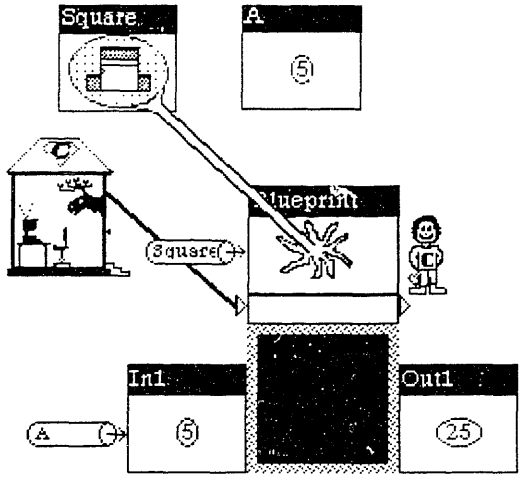


Figure 6.8: Possible representation for sharing of data objects.

The reason for choosing a tentacle over the traditional pointer notation for sharing is that it emphasizes in a physical way that that a data object may be "rooted" in several locations at the same time.

It should be clear from the above discussion that although the principle of direct mappings is desirable for making structure visible, it should not necessarily be incorporated into a system at all costs. In choosing visual representations, there are many tradeoffs. Though it is nice to explicitly show structural information through visual representations, there is the danger that too much structural information can overwhelm the user, leading to confusion and decreased usability of the system. In fact, the current Grasp system tends to show altogether too much structure - although this may be useful to novices using the system for the first time, it will probably soon become cumbersome. Representations which show less of the structure are required to improve the usability of the system for programmers more advanced than the first-time novice. This idea is explored further in the discussion of Chapter 7.

6.1.2 Continuous Representations

The direct mapping principle guides the static aspects of a visual representation - it applies to a representation at a particular point in time. Naive realism, on the other hand, requires that the visual representations on the screen should reflect the underlying state at *all* times. This requirement motivates a principle of *continuous representations*, which constrains the evolution of a visual representation over time by dictating that the state be represented continuously.

The principle of continuous representations is largely orthogonal to the issue of direct mappings discussed above. It is founded on two requirements:

1. All changes to the underlying structure must be reflected in visual representations.
2. Any changes made to the visual representations must be accompanied by appropriate changes to the underlying structure.

These ideas have been formalized by Ciccarelli in his notion of a *presentation system* [Ciccarelli 84]. A presentation system consists of two data bases: an application data base and a presentation data base responsible for visually presenting the application data base. In a presentation system, a *presenter* is responsible for maintaining a current view of the application data base, while a *recognizer* translates modifications to the presentation into actions performed on the application data base. If the Grasp system is view in this framework, the data

base consists of elements of the Grasp model, and the presentation data base is responsible for providing an interface to those elements.

As an illustration of the issues involved in continuous representations, consider possible interfaces for presenting Lisp list structures. The most common method of presentation is parenthesis notation within a text editor. Not only is the representation ambiguous and non-unique (see Chapter 3), but the interface does not meet the two requirements stated above. Changing parenthesis notation within a text editor does not effect a change in the corresponding list structure; moreover, mutations to the list structure will not be reflected within the text editor. In both the text editor and the READ-EVAL-PRINT loop, there is no guarantee that representations appearing on the screen faithfully mirror the state of the Lisp environment. Such interfaces impede the structural reasoning process and are partially responsible for the poor structural models of novice programmers.

The problems exhibited by the text editor and READ-EVAL-PRINT loop are not inherent in parenthesis notation. For example, in Interlisp-D's list structure editor, DEdit, changes to the notation *are* accompanied with changes in the underlying structure [Interlisp 83]. However, if the list structure being edited is mutated by some other means than through DEdit, the representation in the editor is not properly updated. We can also imagine a system which reflects mutations properly in its list notations, but does not allow the user to interactively edit the notation.

Of course, it is best to have an interface which meets both of the requirements listed above. Although its major data structure is a two-dimensional box rather than a one-dimensional list, Boxer is worth mentioning in this regard. Not only can boxes be modified interactively by making changes to their visual representations, but Boxer ensures that the current state of a box is correctly displayed regardless of how modifications are made to its underlying structure. A similar approach can be taken for list structures; the benefits are even greater if the approach is applied to box-and-pointer diagrams rather than parenthesis notation. The KAESTLE system includes an interactive box-and-pointer editor, though it is unclear whether the editor properly reflects modifications made from sources outside of the editor [Bocker *et al.* 86].

In order to maintain the illusion that the representations which are visible on the screen are the state of the system, Grasp meets the two requirements of continuous representations. This means that all state changes in the underlying model of computation, whether resulting from the user's direct interactions with the interface or from the running of a program, are continuously reflected in the interface.

Consider assigning a value to a variable. The programmer may do this directly through the interface by picking up a data object and placing it over a variable. As discussed below in the section on manipulabiity, this action is interpreted as assigning the data object to the variable. Figure 6.9 demonstrate how the representation of the variable changes to show that an assignment has taken place.

Before assignment After assignment

Figure 6.9: Changes in representation indicate an assignment.

The grayshade background indicative of an empty variable changes to white when the variable contains a value. Furthermore, the iconic representation of the data value appears centered in the variable box after an assignment. If the variable or a structure to which it is attached is moved, the variable's value always remains visibly contained by the variable in this manner. When the data object is "picked up" from a variable, the box assumes the grayshade background associated with an empty variable.

Variables can also dynamically take on values during the running of a program. For example, when control reaches a machine, all input variables to the machine which are the targets of reference pipes receive the value stored in the source of their reference pipe. When control exits a machine, values appear in its output variable. In both cases, the change in the state of the variable is shown in the interface in the same way as if the user had explicitly assigned a value to the variable. The key aspect of these interactions is that the state of the variable is continually refelected in its representation irrespective of the manner in which it arrived at that state.

Grasp supports continuous representations to a similar degree throughout the interface. Control path connections, reference pipe connections, and composition of internal structure - whether directly manipulated by the programmer or carried out by the system (during the construction process, for example) - are consistently mirrored in the interface at all times. This continuity of representation is a chief way in which Grasp supports naive realism; they allow the programmer to readily believe that the representations on the screen *are* the Grasp elements of computation.

6.1.3 Additional Representational Considerations

The design of visual representations involves considerations beyond the goal of representing information. Certainly it would be possible to represent booleans in a much different fashion. One could choose the symbols ***true*** and ***false***, the numbers ***0*** and ***1***, the shades white and black, or even an obscure and arbitrary association like **%** for truth and ***&*** for falsity. Yet none of these choices for booleans seems as appropriate for Grasp as the hand icons. What are some of the other forces at work in the design of a visual representation?

6.1.3.1 Familiar Representations

One goal in designing representations is to make them as "familiar" as possible. In the world of document preparation systems, this task is simplified by the existence of straightforward physical analogs for most of the elements of the system. Star's iconic representations are designed to take advantage of an office worker's familiarity with sheets of paper, envelopes, in and out boxes, and other objects common in the office environment. For the Grasp system, the notion of a "familiar" representation is much hazier, since computational elements are not physical objects we encounter in everyday life. Yet, commonalities in the representations generated among teachers and students suggest that certain representations are intuitively appealing for computational elements.

Grasp's representation of procedure activations as input/output machines is an example of such a "natural representation." Algebra and calculus texts often introduce functions as machines which compute outputs based on inputs (Swann & Johnson's delightful introduction to calculus uses this approach extensively

[Swann & Johnson 77]). When asked to describe procedures in a recent questionnaire [Turbak 86], several students in the Scheme course gave responses which stressed the input/output nature of procedures. Students were also asked to sketch the way they visualized the following interaction with Scheme:

```
=> (DEFINE A 5)
A

=> (DEFINE (SQUARE X) (* X X))
SQUARE

=> (SQUARE A)
25
```

Of those who provided a sketch, most showed a box with an input coming in on the left and an ouput leaving the box toward the right, as in Figure 6.10.

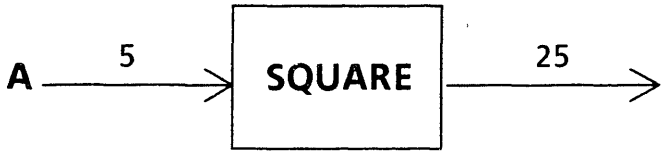


Figure 6.10: Typical student sketch of (SQUARE A)

These sketches bear a great resemblance to standard graphical representations used for describing data flow programs (see, for example, [Dennis 75]).

The iconic representations chosen for Grasp machines are harmonious with the representations which programmers sponaneously generate. Grasp machines are essentially box-like, with inputs arriving on the left hand side, and outputs leaving on the right hand side. (The common tendency to show data flow from left to right is probably related to people's familiarity with reading from left to right in English.) The Grasp representations differ from programmers' sketches mainly in their explicit representation of control. Grasp machines have input and output control paths in addition to the specifications for inputs and outputs. People's sketches of programs rarely represent control in such an explicit fashion.

There are many other cases of "natural" representations chosen for Grasp. Variables are often introduced in programming courses as boxes or containers for values; Grasp represents them visually as boxes. The term "data flow" is often bandied about in computational lingo. Grasp makes use of this notion in its choice of reference pipes to allow values to "flow" from one variable to another. The visual representation of a a reference pipe as a section of pipe (⟹) supports the view of data structures as physically flowing objects. The analogy is made even

stronger by the fact that the icons for data objects actually move in a continuous path across the screen from one spatially-located variable to another.

Representing the controller as a little person is intended to take advantage of people's familiarity with animate agents. Grasp is designed so that action in a program is localized around the position of the controller in the visual representation. The novice is thus led to believe that the homunculus on the screen is actually responsible for all of the action in a computation. The fact that the controller actually "walks" along the control paths from machine to machine reinforces the view of the controller as an active agent and also provides a visual representation for "control flow."

Figure 6.11: Icons used to animate the "walking" of the controller.

Figure 6.11 shows some of the icons used to animate the walking sequence. The object in the controller's hand is a flashlight; this corresponds to a model (popular at Xerox) that views control as a pencil-beam flashlight which illuminates small areas of code as it traverses the expressions of a program in the order they are executed.

The choice of "thumbs-up" and "thumbs-down" icons for booleans is an especially interesting one. In programming semantics, the only interesting property of booleans is that they specify which branch of a conditional to take. However, the terms "truth" and "falsity" have important connotations in English. Not only are they deemed to be opposites, but they are generally asociated with goodness and badness, respectively. The hand icons also share these connotations. They are not the *only* representation to share these properties; icons of an angel and a devil would have similar characteristics. However, "thumbs-up" and "thumbs-down" also have a clear directionality associated with them. This directionality is useful for showing how the result of a predicate machine in a conditional clause determines which control branch is taken. In the Grasp representation of a conditional clause, the control switch flips in the direction pointed to by the thumb in the output variable of the predicate machine (Figure 6.12).

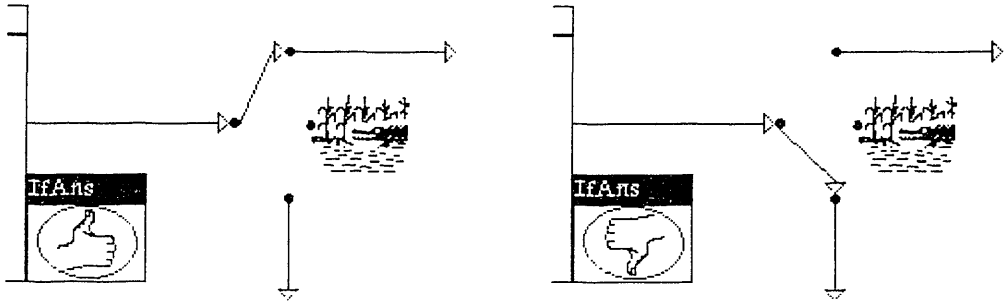


Figure 6.12: Relationship between thumb direction and switch direction.

Regardless of the design criteria used to develop visual representations, the results will always possess some *accidental properties*. These are visual properties to which users will assign a meaning not intended by the designer. In the case of the hand icons for booleans, for example, some users might associate the "thumbs-up" with success and the "thumbs-down" with failure. In such an interpretation, the "thumbs-down" might be disturbing, and the user might want to avoid it. This may sound silly, but users can find certain representations particularly annoying or frustrating. Experience from the Scheme course shows that part of the trouble students have with procedures is that the the procedure-creating construct is named LAMBDA. Not only does the name of a Greek character give no clue as to the purpose of the construct, but I even think that the obscurity of and mathematical ring to the name fuels the view that procedures are hard to understand. A different name, such as MAKE-PROCEDURE, would be much more suitable for avoiding this type of problem. One representation with which Grasp tries to carefully avoid undesirable accidental properties is that for the controller. The icons for the controller are designed to be unisexual in order not to offend programmers of a particular gender.

6.1.3.2 Visual Hints

Certain properties of Grasp representations are useful for reasoning about programs even though they are not strictly necessary for conveying structural information. The properties are called *visual hints*. For example, the lines representing control paths thicken after the controller has traversed them. The thickening does not represent a piece of information stored in the control path element of the abstract machine. Conceptually, a control path keeps track of just two pieces of information: the machine whence it comes and the machine to which it leads. The thickening of the control path is a purely visual mechanism that

provides a "trace" of the program executions and is especially useful for seeing the path taken by the controller through a conditional machine. Another example of a visual hint is the relationship (illustrated above in Figure 6.12) between the thumb direction and the control switch direction in a conditional clause,

Some visual hints are intended to emphasize the similarities and differences between Grasp objects. Representations adhere to the principle that structurally similar objects should look similar and structurally different elements should look different. Thus, all machines share a central rectangular shape flanked by variables; data objects are always enclosed by elliptical borders; connectors (reference pipes and control paths) are long and thin with arrows indicating the direction of flow. These shape differences underscore the different uses for the objects. For instance, even brief interactions with the system make it clear that only the elliptically-shaped data objects can be the value of a variable; it makes no sense to consider interpretation structures such as machines or reference pipes as a variable value. The consistent representation chosen for data objects also stresses that blueprints are closely related to numbers and booleans in terms of their properties as data objects. Unlike Scheme, Grasp succeeds at making "procedures" *look* like data.

6.1.3.3 The World is not a Box

Although rectangular shapes are particularly easy to display in a graphical environment, they are not the only shapes available. Bitmapped display screens allow for more complex shapes than simple rectangles. Grasp tries to make use of some of these shapes. A greater repertoire of shapes is certainly not necessary for achieving visibility, but it helps. If elliptical shapes were not provided, some other representational feature would have to be used to distinguish data objects from other objects. Furthermore, as discussed in a later section on manipulability, interactions with non-rectangular shapes makes manipulability more apparent. Finally, representations that are not mere boxes provide for a more varied and visually pleasing graphical environment. For these reasons, Grasp supports non-rectangular representations for data objects and various other icons.

6.1.4 Information Suppression

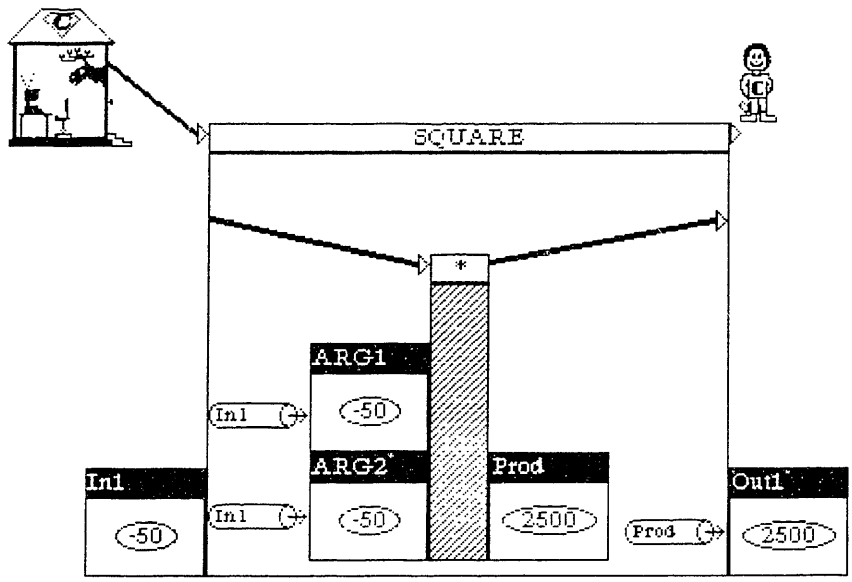
The limited size of the display screen makes it impossible to show *all* of the state and structural information associated with a computational environment. Even simple programs contain a surprisingly large amount of information. The user must be provided with some means of navigating through a potentially large space of information. Grasp provides two simple mechanisms for suppressing unwanted information: scrolling and shrinking.

Grasp interactions take place through windows supported by the Interlisp-D window management system. Each Grasp window is scrollable in both the horizontal and vertical directions. This allows the window to contain more visual information than can fit on a single screenful. However, it is not an extremely useful way to maintain large amounts of information. In the case of a computation, scrolling is not a particularly convenient way to access a particular point in a program or to get a high-level view of the "shape" of the program. The situation is similar to viewing the world through a long tube; many details are apparent, but the "big picture" is hard to see.

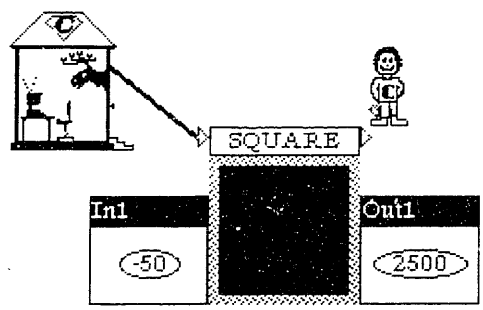
Shrinking is a more effective way of structuring visual information. This mechanism essentially involves having expanded and shrunken forms of representations. The expanded form is used for the examination of the detailed structure of a representation. When the detailed view is not necessary, the representation can be shrunken to a form which takes up much less space. This method of hiding detail is common in user interface designs. The Star designers refer to this type of mechanism as "progressive disclosure" - hiding detailed information until the user indicates that he or she wants to see it. For example, the shrunken form of a Star document is a small rectangle which can be positioned on the "desktop" represented by the screen. The expanded version much more closely resembles the 8.5 X 11 inch paper which writers are familar with. [Smith *et al.* 82] Boxer provides a mechanism for shrinking and expanding any of the boxes in the system. Not only does this help users suppress unwanted information, but it allows them to treat a hierachically arranged collection of boxes as a simple file system through which they can easily navigate via shrinking and expanding [diSessa 85a].

The main shrinkable unit in Grasp is the machine. Primitive machines have no shrunken representation, but both compound and all-purpose machines

have "black-box" shrunken forms which hide the details of their internal structure. Figure 6.13 shows how the internal details of a compound machine may be hidden by shrinking. The internal details of conditional machines, predicate machines, and consequent machines can be hidden in a similar fashion.



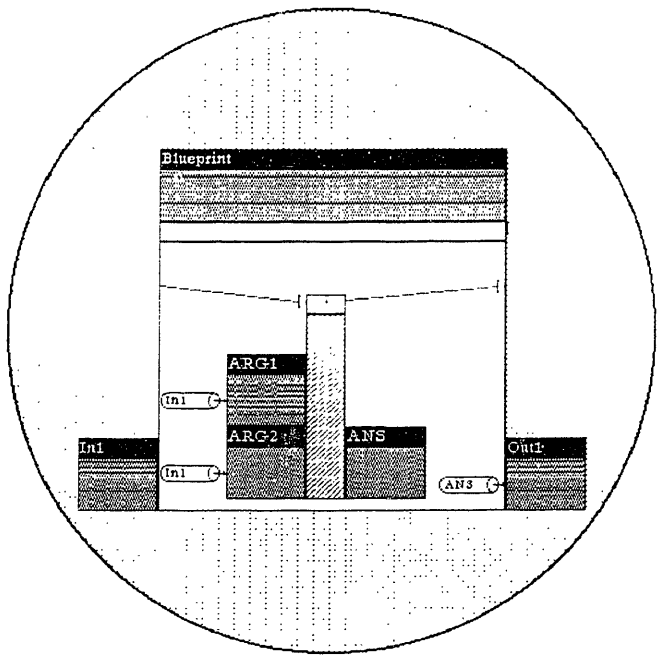
Expanded form



Shrunken form

Figure 6.13: Expanded and shrunken forms of a compound machine.

Another shrinkable unit in the Grasp interface is the blueprint. Since blueprints have a visual representation which consumes a large area of space, Grasp allows the user to shrink them to a small iconic representation. The shrunken representation is used when assigning a blueprint to a variable. Both expanded and shrunken forms are shown in Figure 6.14 for a simple blueprint. As noted above, the current implementation of Grasp does not provide a way to distinguish between the shrunken representations of different blueprints.



Expanded form

Shrunken form

Figure 6.14: Expanded and shrunken forms of a blueprint.

An important property of the expansion scheme Grasp utilizes is that it maintains the context of surrounding objects. Consider the ABSOLUTE-VALUE machine shown in Figure 6.15.

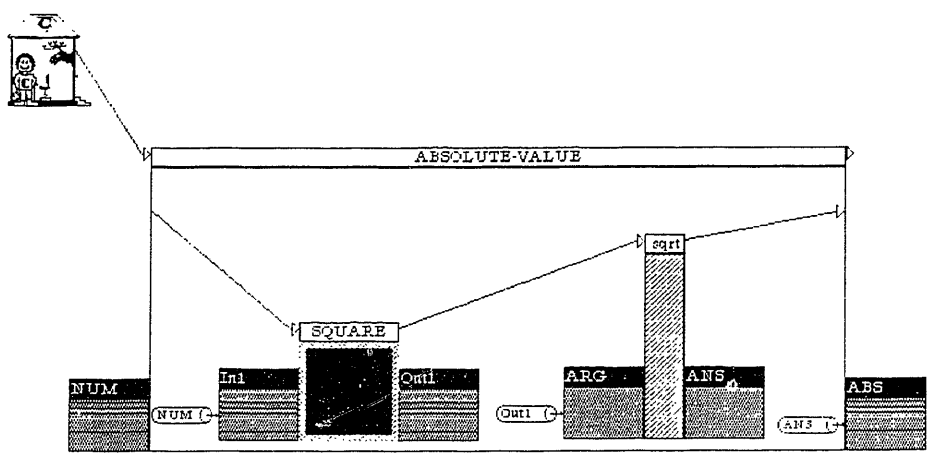


Figure 6.15: An absolute value machine.

This machine computes the absolute value of a number by squaring it and then taking the square root. If we expand the black box SQUARE machine, we get the configuration illustrated in Figure 6.16.

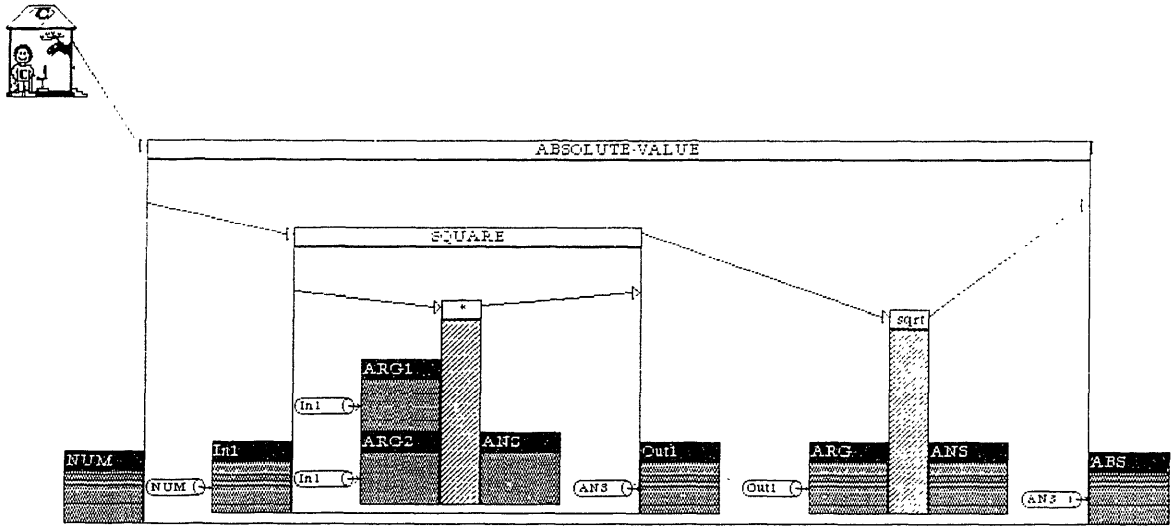


Figure 6.16: After expansion of the SQUARE machine.

Note that the elements surrounding the shrunken SQUARE representation remain in view when it is expanded. This type of context is important for determining where the expanded information belongs in the larger information space.

Many other systems do not show the context of surrounding objects when a shruken object is expanded. The Star system is a good example - expanding a shrunken document icon makes it appear in an expanded form in a totally different part of the screen. [Smith *et al.* 82] The Pict/D graphical programming system [Glinert & Tanimoto 84] and Robot Odyssey [Dewdney 85] use what might be called a "room-based" approach to expansion. In these systems, an expanded object takes up the entire screen; the context of surrounding objects is not shown. This approach is similar to entering a room through a doorway - one loses sight of the areas outside of the room.

In contrast to these systems, Boxer [diSessa 85a] and BOCHSER [Eisenberg 85] have a more versatile expansion mechanism. In these systems, a shrunken box can be expanded in two ways: one in which the context of the surrounding regions is shown, and one in which the expanded box fills the entire screen. This type of expansion scheme gives the user a finer control over the amount of context to display on the screen.

6.2 MANIPULABILITY

The principle of manipulability dictates that interactions with the visual representations displayed in the interface should parallel interactions with objects in the physical world. In a device programming system, the principle is intended to give the programmer a more intuitive understanding of the devices and the way they may be interconnected to form useful configurations. From the standpoint of usability, manipulability allows the programmer to construct configurations in a more direct way than entering expressions in a command language interface.

Possible interactions with a two-dimensional display medium are rather limited by the widely available technology - typically a keyboard and a pointing device known as a "mouse." There is little about these means of interaction which mimics a physical system. Feeling the plastic of the keys or mouse buttons against our fingertips is hardly the kind of tactile feedback we would experience if we were interacting with physical manifestations of the objects represented on the screen. Realistic tactile feedback, though, is not a requirement for a manipulable system. The important feature of such a system is that it encourages the user to manipulate the structure of the system in a straightforward manner. The structure of Grasp programs is mainly determined by the way reference pipes and control paths connect machines. For these simple kinds of structures, *visual feedback* is sufficient for achieving manipulability. Manipulability in this context means that interactions with visual representations *look like* interactions with physical objects.

If visual feedback is so crucial to manipulabilty, then what is the difference between visibility and manipulability? Visibility is the principle which governs the visual representation of the information maintained by static structures. Manipulability is concerned with representing the dynamic interactions which bring about changes to those structures. Certain characteristics of the visual representations in Grasp are designed to bear information about the underlying structure; these are the features of visibility described in the previous section. The representations are also endowed with other characteristics to improve their resemblance to physical objects. The purpose of this section is to describe those characteristics which support manipulability in the Grasp interface.

Not all of the characteristics described below are directly related to the goal of manipulating structure. A manipulable interface for Grasp would certainly be possible without such features as continuous motion, fine-grained positioning, and animated control and data flow. Although many of these might be criticized as unnecessary frills, I strongly disagree with such an assessment. These kinds of features are important for supporting the illusion of naive realism. It is much easier for the user to believe that the "display screen is the reality" if it *acts* like reality. The fact that Grasp objects can be "picked up", moved smoothly across the screen, and positioned at arbitrary locations may not have a direct relevance to the underlying computational model, but it certainly gives Grasp the "feel" of a real physical system. Since understanding computation in terms of what appear to be physical devices is the focus of Grasp, it is desirable to incorporate into the interface any features that help to support this illusion. In addition, the extra manipulability features make the system fun to interact with.

6.2.1 The Physical Object Metaphor

Iconic representations in Grasp were chosen not only to convey information about the underlying structure but also to emphasize their resemblance to physical objects. In this way Grasp embraces what might be called a *physical object metaphor*. Of course, Grasp is rather limited in the ways that it can represent objects. A two-dimensional display screen cannot capture the essence of three-dimensional objects, but two dimensions suffice for displaying the kinds of structure which Grasp tries to show.

All icons in Grasp have distinct borders to show the spatial limitations of the representations. The borders help to clarify exactly where the object is on the screen. Bordered representations, like other Grasp features supporting manipulability, emphasize the objectness of objects. Expression-based systems such as Scheme do a poor job at displaying what is and what is not an object. One of the reasons why novices erroneously include quoted symbols in box-and-pointer diagrams (see Chapter 3) is that the interface does not make the extent of objects clear - is it A or 'A that is the object? Since Grasp objects are not plain rectangles in general, bordered representations are useful for helping users determine the extent of the objects they are manipulating.

6.2.2 Fine-grained Positioning

Objects in Grasp may be positioned at arbitrary locations within a Grasp window. This is in contrast with systems in which the placement of objects is much more constrained. In Star, for example, icons can only appear in one of the 154 invisible grid boxes tesselating the screen [Smith *et al.* 82]. In Boxer, boxes are constrained to appear as large characters in a line of text [diSessa 85a]. The positioning provided by Grasp more resembles the physical world, in which objects can be placed arbitrarily in the continuum of space.

6.2.3 Occlusion

In the Grasp interface, an object is allowed to be "on top of" another object. Here certain qualities of three-dimensional space are ignored. In particular, objects placed on top of other objects in the physical world are often unstable and fall off. In the Grasp system, the representations do not "fall off" - perhaps it is best to think of them as being suspended above the other objects rather than resting on them.

An object suspended above other objects occludes them in an appropriate fashion. That is, the area within the border of an object will hide objects below, while any area outside the border of an object will show any objects below. An example of this type of occlusion occurs in Figure 6.17, where a blueprint icon partially hides the truth icon.

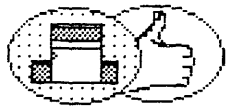


Figure 6.17: Blueprint partially obscuring truth.

Many windowing systems deal with occlusion in this manner (the window system provided by Interlisp-D is an example [Interlisp 83]). However, windows are generally monolithic rectangular shapes for which this behavior is easy to implement. Providing the proper occlusion for the more complicated shapes in Grasp requires much more care in the implementation.

6.2.4 Mouse Sensitivity

A pointing device known as a "mouse" is used to interact with the visual representations in the Grasp window. When the user moves the mouse, a cursor (represented as ↖) follows the motion on the screen. In order to manipulate a Grasp object, the user points the tip of the cursor at the desired object and presses one of the three[2] mouse buttons to specify an action:

Left Button - used for "picking up" the object to move it. See the section on movement of objects below.

Middle Button - used for actions specific to an object. Depressing the middle button often brings up a menu of options for an object. Menus hardly fit into the physical object metaphor, but they are useful for extending the functionality in a prototype system. For example, the middle-button menu for a variable appears in Figure 6.18.

Figure 6.18: Middle-button options menu for a variable.

The top option creates a reference pipe for the variable, while the ChangeName option prompts the user for a new name for the variable.

Right Button - used to bring up a window menu for the Grasp window. By convention, almost all windows in the Interlisp-D world support this behavior for the right button.

An object in Grasp is *selected* by positioning the top-left point of the cursor within the borders of an object and depressing the left or middle mouse button. Thus, in Figure 6.19a no object can be selected, in Figure 6.19b the blueprint can be selected, and in Figure 6.19c the boolean can be selected.

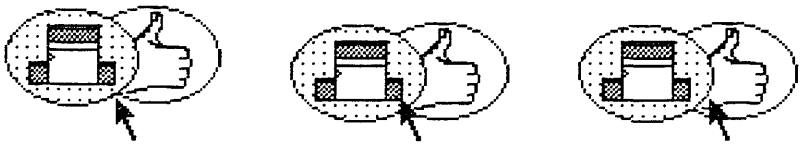


Figure 6.19: Selecting different objects.

Note in particular that the sensitive region of an object is actually within its visible borders and not simply within a rectangular region surrounding the object. Many other systems take the latter approach, which is considerably easier

to implement. For example, in the Interlisp-D window system, clicking a mouse button in the configuration shown in Figure 6.20 will select the LOOPS icon and not the History icon to which the cursor seems to be pointing.

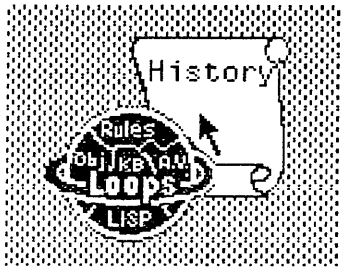


Figure 6.20: A selection in the Interlisp-D window system.

Such behavior results from a simple implementation strategy. Great care is taken in Grasp to ensure that objects can only be selected when the cursor is directly pointing to them. This is an important way in which Grasp embraces manipulability.

6.2.5 Movement of Objects

Perhaps the most obvious way in which Grasp supports manipulability is the way it allows objects to be moved. Most interactions with Grasp are carried out by moving objects. Creating a computational configuration requires "wiring" together machines with control paths and reference pipes and putting data objects in appropriate variables. All of these manipulations are carried out by moving the visual representations on the screen. The Grasp system guarantees that the appropriate changes are made to the underlying structure.

Objects are moved in the following way. The user points the mouse at the desired object and depresses the left mouse button. This action "picks up" the object. If it was partially occluded by other objects, picking it up brings it to the forefront of the display; this behavior is illustrated in Figure 6.21.

Before picking up After picking up

Figure 6.21: Picking up an object brings it to the forefront of the display.

Keeping the left mouse button depressed, the user can move the object to its new location. The object moves smoothly across the screen along with the cursor. The bitmap representations are not constrained to appear on a coarse grid but can fill any area of pixels on the screen. The same kind of occlusion which occurs in static representations applies to moving objects as well. When a moving object passes over other representations on the screen, it occludes them in an appropriate manner. The smooth motion and dynamic occlusion contribute greatly to the physical "feel" of the system.

In the case of control paths, the user picks up the arrow at the end of the path. As the arrow is moved, the line connecting it to the source object follows the arrow in a "rubber-band" fashion. This kind of behavior is illustrated in Figure 6.22.

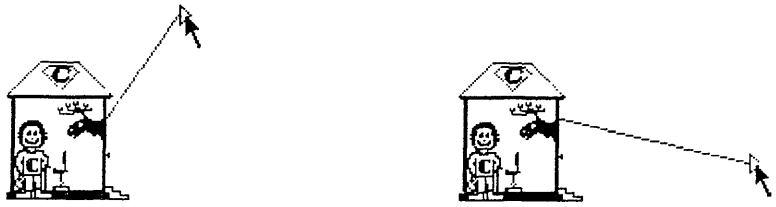


Figure 6.22: Snapshots of a moving control path.

Dynamically maintaining the control path connection is another way in which the system attempts to resemble the physical world.

To place a moving object, the user releases the left mouse button. Depressing the button corresponds to grasping the object; releasing the button means to "drop" the object at the current postion. As noted above, objects may be placed in arbitrary postions on the screen.

6.2.6 Making Connections

Fitting objects together by positioning them in an appropriate way is a familiar experience from everyday life. We are used to plugging in power cords, buttoning up shirts, and inserting keys into locks. Grasp takes advantage of these physical experiences by allowing the programmer to specify computations by properly positioning visual representations on the screen. Constructing programs by moving and placing objects is the essence of the device programming style.

In some cases, placing an object in Grasp has no great structural significance. "Dropping" a number in the Grasp window or a variable within the

internal structure of a compound machine simply indicates that the window or machine now contains a new object. However, in other instances, the placement of an object may carry an important semantic interpretation. For example, placing a data object on top of an empty variable means that the variable now has that object as its value. In such an instance, the system may reposition the object and make other representational changes as well (see Figure 6.9 for an example of placing a data object in a variable).

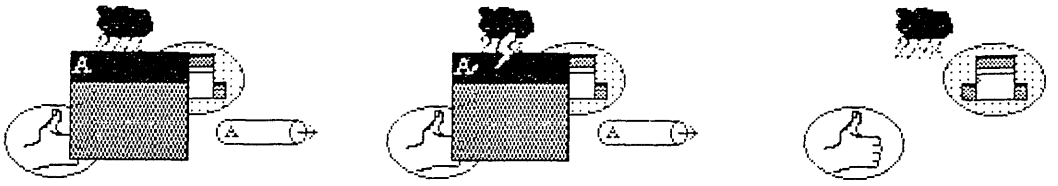
Wiring together machines with data and control paths procedes in a similar manner. When a reference pipe is placed over a variable, it becomes "connected" to that variable. Positioning a control arrow above a machine indicates that the control path will terminate at that machine. These connections may be broken by picking up the reference pipe or control arrow. However, data and control connections are maintained when the user moves the elements to which they are connected.

Making structural connections by placement clearly resembles the physical world. The physical world, though, is not as forgiving as the Grasp environment. We cannot expect to plug in a power cord simply by placing it near an electrical socket - we must carefully orient the prongs and insert the plug into the socket. In Grasp, we can connect a reference pipe to a variable by positioning its right-hand side anywhere above a variable. Grasp could more accurately model the physical world by having distinguished reception sites for connectors like the reference pipe and control arrow, but such strict adherence to the physical object metaphor impedes the user's interactions with the system. There is a clear tradeoff between faithfulness to the physical object metaphor and usability; on this issue I favored usability.

6.2.7 Permanence

Grasp objects exhibit permanence. Once the programmer creates a Grasp object, it remains in the Grasp environment until the programmer explicitly destroys it.[3] This resembles the situation in the physical world, where objects tend to remain in place except when explicitly removed.

Objects are destroyed in Grasp via the *thundercloud*. When the thundercloud is placed over an object, it "zaps" the object, and the object disappears. Figure 6.23 shows three snapshots of a thundercloud destroying the variable A.



Impending doom . . . Zap! Bye-bye.

Figure 6.23: Destroying the variable A.

Notice that the reference pipe to A disappears when its source variable does. The Grasp system must guarantee that no "dangling references" or other meaningless structures are left behind when objects are destroyed.

6.2.8 Animated Control and Data Flow

Both control and data flow in the Grasp system are animated. When control goes forward through a computation, the controller homunculus walks along or slides down the control paths (see Figure 6.11 for some appropriate icons). Data flow occurs into input and output variables when control enters and exits a machine. In this situation, a copy of the value from the source variable for the reference pipe moves across the screen and enters the target variable. Such animation is not necessary to illustrate the semantics of the underlying computation - the system could simply show the controller and data objects "jumping" through space to their new positions. However, since physical objects can only get from one position to another by continuous motion through space, the animation reinforces the view of the controller and data values as physical objects.

6.3 Summary

This chapter discusses the representational principles used in designing the interface to the Grasp system. The principle of visibility dictates that visual representations appearing on the display screen reflect the structure of the underlying computational model. Visibility is achieved in Grasp by direct mappings, continuous representations, familiar representations, and visual hints that aid reasoning about program structure. The principle of manipulability states that interactions with the representations on the screen should resemble interactions with physical objects. The most important features of manipulability in Grasp are those which allow the user to make and break structural connections

by "picking up" and moving object. Other features, such as continuous motion, fine-grained positioning, and occlusion, though not necessary for structural manipulation, make the system appear more "real." In conjunction with visible representations, these features support naive realism - the illusion that the representations on the screen are themselves the elements of the Grasp model of computation.

CHAPTER 7

DISCUSSION

The previous two chapters concentrated on the important principles behind the design of the Grasp model and interface. For the Grasp model, the key principle is reification. In order to build programs out of compuational pieces in a device programming style, it is necessary to make traditionally implicit structures explicit elements in the Grasp model. The application of reification results in such elements as machines, variables, reference pipes, the controller, and control paths. In the interface, the important principles are visibility and manipulability. Visibility is achieved by continuously representing direct mappings of structure in visual representations. Manipulability results from extending the ideas of direct manipulation interfaces to elements which more closely resemble physical objects. Together, visibility and manipulability support naive realism, the illusion that the visual representations on the display are the actual pieces of the underlying model.

The purpose of this chapter is to evaluate the Grasp system by discussing its advantages and drawbacks. I will argue that embodying the device programming style in a visible and manipulable interface is superior to textual notations for presenting the major ideas of the procedural programming paradigm. However, the approach taken by Grasp introduces new problems and limitations which are not associated with the text-based view of programming. In discussing these issues, I will also compare Grasp to several other novice-oriented programming systems.

Of course, I will be in a much better position to evaluate the Grasp system when I actually test it out with novices. The major thrust of this project up to this point has been a principled design of a device programming system for procedural programming. Although a prototype implementation of Grasp exists, the system is still very much under development. Several crucial features, such as blueprints and all-purpose machines, have not yet been completely implemented. Thus, the

system has not yet reached a state where it can be adequately tested. Most of the comments and observations I make in this chapter are of necessity based only on my own interactions with the system.

7.1 ADVANTAGES OF GRASP

In this section , I explore the key features of the Grasp model and system and argue why they should help novices build more robust models of the procedural paradigm. In particular, I explain why the Grasp model should help eliminate many of the confusions about the procedural paradigm discussed in Chapter 4.

7.1.1 Device Programming Style

Many advantages of the Grasp system stem from the device programming methodology which it supports. In contrast with the linguistic approach to programming, this methodology focuses on the run-time structures used in the evolution of a process rather than on the textual expressions which describe a process. Grasp provides a microworld for exploring how processes can be constructed out of computational devices in an erector set style. Such an environment encourages people to use their familiarity with connected objects in the physical world to reason about the structure of computations.

When embedded in a visible and manipulable interface, the device programming style allows programmers to get "hands-on" experience with the elements of the abstract machine. Rather than typing expressions at a command language interface to indirectly manipulate implicit interpretation structures, the programmer can make direct modifications to explicit visual representations of those structures. The visibility of computational elements and the directness with which programmers can interact with them should help make the structure of programs much more apparent to the novice.

As an example, reconsider the two models for variable binding which I induced from examples when I took the Scheme course (see Chapter 3). The reason that two very different models were possible was that the interface did not provide enough visual information about the state of implicit binding mechanisms; futhermore, the bindings could only be indirectly manipulated by

DEFINE and SET! expressions, whose semantics were not obvious from their form. In Grasp, the implicit binding becomes reified as a variable object which has an associated value. The variable object and its contained value are graphically represented on the display screen. DEFINE expressions are not necessary in Grasp; the programmer directly creates variables and manipulates their values on the display screen. The information provided through the static graphical representations and the dynamic interactions make it obvious that variables in Grasp are associated with data objects. The alternate model that variables could be associated with unevaluated expressions would simply never arise in such a system. By making the structural features of a computational model more evident, a transparent interface to a device programming system helps the programmer form more robust structural models.

In addition to making structural information more accessible, the Grasp system has the important property that visual patterns make good structural sense. A problem with text-based languages such as Scheme is that syntactic patterns do not necessarily indicate a semantic relationship. This is the basis for the "procedure as pattern" bug in which the programmer attaches undue importance to the syntactic similarities between the forms for defining a procedure and for invoking it. Since the visual representations in the Grasp system are intentionally chosen to convey the structure of the computational elements, reasoning based on visual similarities is valid in Grasp.

7.1.2 Primacy of Procedure Activations

Unlike Scheme and other procedural programming languages, Grasp focuses on the procedure activation (machine) rather than the procedure (blueprint) as its basic unit of computation. Traditional programming languages embody a linguistic approach to programming in which textual specifications are easy to construct. In these languages, procedure activations are created only as a by-product of using a procedure. In the device programming style that Grasp embraces, however, procedure activations are more natural than procedures as the fundamental "device."

This shift in focus has several advantages for novice programmers. First of all, a model which emphasizes the primacy of procedure activations is likely to prevent the formation of the "procedure as doer" bug. Novices have a desire to

associate observed action with a computational element, and the interface to the Scheme language makes procedures seem the logical choice. However, the common intuitive notion of active computational units that produce outputs based on inputs better fits a procedure activation than a procedure; procedures are abstract specifications for a collection of "active" units. In the Grasp abstract machine, the emphasis on procedure activations should lead novices to attribute activity to the activation rather than the procedure.

Second, Grasp permits novices to experiment with individual concrete activations without having to first construct abstract specifications. In Scheme, this style of interaction is not possible. The Scheme equivalents of activations (i.e., environment frames) can only be created after constructing a procedure, and even then they are only accessible to the programmer through arcane debugging tools.

Third, the machine-based view provides a new way for understanding procedures. An important advantage of the Grasp approach is that once the novice gets used to primitive and compound machines, blueprints and all-purpose machines can be motivated as a means for automating the process of building machines by hand. Blueprints can be understood as templates for the internal structure of an all-purpose machine; the programmer builds structure in the blueprint in the same way the controller should build structure in the all-purpose machine. With this representation, the actual application process can be understood in terms of a semantically straightforward copying scheme.

7.1.3 First-Class Procedures

From the standpoint of computational power, the device progamming approach has a serious drawback: the lack of linguistic abstraction mechanisms. If a device programming system too closely resembles the physical world, it will suffer the limitations of the physical world as well. An important characteristic of the linguistic view of programming is that it allows abstraction to be handled in a way unattainable with physical devices. As an illustration, suppose we combine several physical components to construct a device that exhibits a desirable behavior. Then we can use this device as a higher order component in other systems based on its behavioral characteristics without worrying about the details of its implementation. As with Scheme procedures, there seems to be an

abstraction barrier here which separates the use of a physical element from its implementation. However, we cannot take full advantage of two important Scheme abstraction mechanisms with the physical device:

1. *Procedural Abstraction:* Procedural abstraction allows us to build procedures, each of which is an abstract description of a class of processes. To construct an actual instance of a process, we simply call the appropriate procedure with arguments. In the physical world, we can similarly create an abstract description for a device. However, every time we want to use an instance of the device, we must construct it by hand. In computation, the interpreter automatically builds an activation of it for us based on the specification information in the procedure; in the physical world, humans are the only interpreters of specifications for devices.
2. *Naming:* In the procedural paradigm, we can name a procedure and indicate a use of the procedure by referring to its name. Naming provides a level of indirection which ensures that future modifications to the procedure will be reflected in all the places it is used. Indicating the use of a device in the physical world implies having a distinct physical copy of it. Should we ever decide to update the design of the device, it is not enough to change its specification; we must also individually modify every instance of the device already in existence.

If a device programming system also incorporates enough of the linguistic view of programming to support the two abstraction mechanisms described above, then it is possible to reap the benefits of both views. This is the approach taken in the Grasp model. On the one hand, reified interpretation structures allow programmer to user their physical intuitions to build executable configurations in a device programming style. On the other hand, the Grasp model supports both procedural abstraction and naming. A blueprints acts as template for a family of Grasp machines in the same way that a procedure is a template for a class of procedure activations. All-purpose machines are used to automatically construct machines based on blueprints. The level of indirection provided by names in textual languages is realized in Grasp with variables and reference pipes. By incorporating the ideas of two very different views of programming, Grasp is able to support a device programming style without sacrificing computational power.

In addition, the Grasp approach to the procedural paradigm exploits the full power of procedures as first-class objects. Blueprints share the same data structure properties as numbers and booleans - they can be stored in a variable, given as arguments to a machine, and returned as results from a machine. In fact, since inputs and ouputs of machines in Grasp are just variables, the second two properties are subsumed by the first. Treating blueprints as first-class objects gives Grasp all the advantages that Scheme derives from first-class procedures.

Grasp blueprints have some interesting properties to help the programmer understand their status as first-class objects and their purpose as abstraction mechanisms. First of all, Grasp procedures *look like* the other data objects in the Grasp system - all are encased in elliptical borders. This notation underscores the properties of blueprints as first class objects. Second, the "body" of the blueprint is constructed with the same manipulations used to build any of the programming configurations. Thus, building a blueprint may be thought of as forming a template or prototype for a class of machines. Third, Grasp has no element equivalent to Scheme's LAMBDA. Rather than writing an expression which dictates that a procedure is to be created at a later time, the user interactively creates blueprints and specifies their templates before the program is even exectued. The effect of procedures creating and returning other procedures is achieved by putting blueprints inside of other blueprints; the copying mechanism guarantees that a new blueprint will be returned for every invocation of the surrounding blueprint.

Other novice-oriented systems are characterized by restrictions and limitations on the use of procedures. Like most conventional languages, many of them support procedure-like specifications but do not treat these specifications as data objects. Sutherland's system provides a macro facility for representing data flow configurations by a single symbol; data lines in his system, however, can only carry only numbers, booleans, and symbols [Sutherland 66]. In systems based on the PASCAL view of the world, such as Pict/D [Glinert and Tanimoto 84] and PECAN [Reiss 84], first-class procedures have no place because PASCAL does not support them.[1] Boxer marginally allows the use of higher order procedures in the sense that it allows "data boxes" to encapsulate specifications that can be applied to other objects. However, the "doit boxes" normally used in Boxer for specifying procedures cannot be treated as data objects unless explicity enclosed within a data box [diSessa 85a]. Eisenberg's BOCHSER, which provides a Boxer-style

interface to the Scheme environment model, is the only other novice-oriented system of which I am aware that supports first-class procedures in the Scheme sense [Eisenberg 85].

7.1.4 Structural Reference

Another important aspect of Grasp is that it supports a binding mechanism and a means of reference that are close in spirit to the environment structures and lexical scoping provided by Scheme. The reason I use the phrase "close in spirit" is because lexical scoping is inherently a text-based notion and Grasp is not a text-based system. However, the *intent* of lexical scoping is that references be wired down at the point of procedure creations rather than at the point of procedure call. Textual languages must indirectly achieve this affect by using names, environment structures, and closures in the mannner dictated by Scheme's lexical scoping rules. Grasp implements the same intent in a way which underscores the structural connection between the variable and references to it - when programmers want to refer to a variable, they directly *point* to it, request a reference pipe, and show where the referred value is to be used.

Note that because reference is determined by structure rather than name, it is possible in Grasp to have more than one variable with the same name in the same "environment." The utility of such a situation might seem questionable, but note that it means that the programmer is not required to be aware of all the other variables and their names when he or she creates a new one. In Grasp, creating a variable always indicates the introduction of a new binding structure which is different from all those that were created before it. Compare this with the use of DEFINE in Scheme, where the programmers can never be sure whether they are creating a new variable or modifying one that is already present in the current environment frame.

Since Grasp allows variables to be placed anywhere, it supports a mechanism resembling Scheme's block structure. Because reference is determined by structure rather than name, however, there is no notion of a "hole in the scope" of a variable. Consider the following Scheme code:

```
(DEFINE A 5)
(DEFINE (ADD-100 A)
  (+ A 100)
```

The A within the body of the ADD-100 procedure refers to the formal parameter of the procedure, not the value of the global variable of A. The body is said to be in a *hole in the scope* of the global variable A because the choice of parameter name make it impossible to refer to that variable within the procedure. If we wanted instead to write and ADD-A procedure which took an argument and added the value of the global A to it, we could no longer use A as a formal parameter to such a procedure. Grasp has no such problem because the reference is purely structural.

Although Grasp emphasizes structure, it does not totally ignore text. Well chosen textual names provide important information about the function of a program. Although text does not have any meaning in Grasp, variables (and machines) may be given textual names as comments. In fact, textual names are important in the interface for showing the relationship between a reference pipe and its source, or for describing the function of a shrunken machine. In its support of textual comments, Grasp differs from some other graphical systems which try to avoid names entirely. Sutherland's data flow system, for example, has no names; programs are specified totally by how functional units were wired together with data paths [Sutherland 66]. Names however, give important clues about how a program works, and without them some of Sutherland's example programs are nearly impossible to decipher (as an example, see Figure 7.1 later in this chapter). The Pict/D system scorns textual names. All names, even those of variables, are denoted by icons. The designers of this system admit that a total avoidance of names is too restrictive and that some form of textual names should be added to future versions of the system [Glinert & Tanimoto 84].

Grasp's representations of variables, their values, and their reference pipes should help clear up some of the misconceptions novices have about names and environments. Because Grasp does not rely on names and because its representations emphasize the distinction between variables and values, it is unlikely that novices would develop the "procedure as name" bug or the "formal/actual" confusion in Grasp. In addition, the stress Grasp puts on the structure of reference clarifies exactly what reference means in a program. Models of alternate parameter-passing mechanisms and scoping rules are not well-supported by the visual evidence available on the display screen.

As with procedures as first-class objects, other novice-oriented systems often deal with variables in a nonstandard or restricted way. The Pict/D system, for example, displays four variables on the screen at all times. When defining a

procedure, the programmer specifies for each of the variables whether it is to be local or non-local to the procedure. Since the four displayed areas for variables represent more than four conceptual variables, this visual representation is a potentially confusing one. Furthermore, the value of Pict/D variables is limited to six-digit integers. Compare these properties with those of Grasp variables, where the number of variables is arbitrary (limited only by the memory space of the machine), each variable has a distinct visual representation, and variables can contain not only any number supported by Interlisp, but booleans and blueprints as well.

Unlike Pict/D, Boxer handles variables and references to them in a much more standard way. Variables are named boxes on the screen, and references to them are specified by use of the names within textual code. It employs a *copy and execute* model for procedure application to determine what the references mean. In this model, the code for a procedure is conceptually copied to the point of application and is then executed in the calling environment. Since names refer to the closest box with the same name in the hierarchy of surrounding boxes, this gives the effect of dynamic scoping. However, a mechanism known as a *port* is available to achieve the effect of lexical scoping when the programmer wants it. A port is a method of hardwiring a reference to a box into a particular spot in a program. In this respect, ports have a similar functionality to Grasp's reference pipes.

It is interesting to note that Grasp, like Boxer, uses a copy and execute model for procedure application. When the controller comes to an all-purpose machine, it first copies the structure specified by the blueprint into the internal structure of the all-purpose machine, and then forges ahead. The important difference is that since Grasp copies *structure* rather than text, there is no possibility of getting into any situation akin to dynamic scoping. Dynamic scoping is purely the result of a text-based model and has no analog in a structure-oriented model. Copying models in a text-based languages can lead to other semantic subtleties as well. In Algol 60, for example, a copying model for procedure execution introduced the need for a call-by-name parameter-passing mechanism. Since blueprint application in Grasp is based on the copying of structural connections (i.e., reference pipes and control paths), a copying model is semantically more well-defined for Grasp than for textual languages.

7.1.5 Agent-Centered Control

Grasp's representation of the controller as a little person takes advantage of people's tendency to associate action with animate beings. The Grasp model is designed so that activity in the system is localized around the controller, both at the conceptual level and in the visual representations displayed on the screen. In this manner, the controller may be considered the true "agent" in the model; machines may be locations of actions, but it is the controller who sets them into motion. The controller appears responsible for activating machines, constructing the internal structure of all-purpose machines, and causing data to flow through reference pipes.

Because of its close relationship to activity in Grasp configurations, the controller can fruitfully be viewed as a reification of the interpreter itself. There are two advantages of this point of view. First, understanding the distinction between procedures, procedure activations, and the interpreter is a stumbling block to many novices since all three seem to be lumped together in the "action" of a program. Grasp clearly distinguishes these elements: procedures are represented as blueprints, activations as machines, and the interpreter as the controller. The explicit differences between these elements should help novices appreciate the subtle distinctions between them.

Second, the choice of a human-like representation of control makes it likely that programmers will project themselves onto the controller when reasoning about programs. Predicting and explaining the behavior of code requires an ability to "play interpreter" - to imagine oneself as the interpreter and to simulate the actions specified by the code. It can be difficult for novices to understand that the behavior of the computer is totally determined by a small set of interpretation rules until they become good at "playing interpreter." The representation of the controller as an animate being will hopefully lead novices to reason naturally along the lines of "If I were the controller, I would be doing the following in this situation . . ."

Several other computational models make use of animate entities to explain the dynamic behavior of the computer. The actor model taught in the Scheme course represents procedure activations as actors standing in an unemployment line; a procedure application is handled by taking an actor off the unemployment line and handing him a "script" containing the body of the

procedure. Other computational models that rely on human-like elements include the "little-man" model of LOGO [diSessa 86a] and Malone's organizational model of Lisp [Malone 85].

7.1.6 Computational Time Line

The device programming style is motivated by the desire to take advantage of people's intuitions about physical objects and devices. Such intuitions can be valuable for reasoning about the static structure of a configuration - what the elements are and how they are connected. However, these intuitions are not necessarily appropriate for helping people reason about the *dynamic behavior* of a given structural configuration. Determining the behavior of a system from the behavior of its parts is not always the easiest of tasks. Sometimes it is possible to envision the behavior of a system in a step by step fashion, but many physical systems are notoriously hard to simulate in this manner.

Consider an electrical system of interconnected resistors, capacitors, inductors, and op-amps. Understanding the topology of the system - how the elements are connected in a network - is straighforward. However, determining the behavior of the system knowing the behavior of the parts is decidedly nontrivial. Trying to envision the time behavior of the circuit by imagining the time variations of voltages and currents throughout the circuit is often infeasible due to overwhelming complexity. Moreover, time simulation is simply the wrong approach for most circuits; considering the network from the viewpoint of constraint satisfaction is often a far more fruitful approach for analyzing electrical systems.

An important property of the procedural paradigm is that step-by-step simulation *is* a valid approach for reasoning about the dynamic behavior of programs. The procedural paradigm is intrinsically related to procedural epistemology - the study of knowledge based on the imperative notion of "how to" rather than the declarative notion of "what is" [Abelson & Sussman 85a]. A major difficulty novices have in learning programming is that they are not comfortable with expressing their knowledge in procedural terms [Sheil 81]. A major thrust of the LOGO movement is that procedural specifications can, in fact, often be a more effective means of description than declarative ones [Papert 80]. The way "how to" knowledge is normally expressed is as a series of steps to perform. In the

linguistic view of procedural programming, programs consist of sequences of expressions to be evaluated. The dynamic behavior of the program derives wholly from the sequential evaluation of its component expressions.

This imperative approach generally imposes too much sequentiality on programs. In many cases the exact order of evaluation does not affect the behavior of the program. In side-effect free Scheme programs, for example, the order of evaluation of arguments in a procedure call can have no bearing on the computed result. Recent research in parallelism has aimed to remove much of the arbitrary sequentiality from procedural programs. The data flow approach, for example, is based on the idea that order of expression evaluation in applicative programs is constrained only by data dependencies between expressions, not by the order in which a programmer happened to include expressions in a program (see, for example, [Arvind & Ianucci 85]).

Although this shift of attention away from sequentiality is a fruitful avenue of exploration for computer scientists, it is not clear how good it is from the viewpoint of novices trying to understand computation. It is good in the sense that it separates the novice from details which are not intrinsic to the problem domain. In financial spreadsheet programs, for example, specifying a computation is simplified for the user because the order of expression evaluation does not have to be given - it is determined from data dependencies alone.

On the other hand, removing sequentiality adds nondeterminism into the process of envisioning the evolution of a computation. Although this nondeterminism cannot effect the behavior of a program, it means that the envisioner has to make the choice of which order to use. A problem here is that it is easily possible to get lost in simulations when too many things are going on in parallel. Consider a program from Sutherland's data flow system for computing square roots (see Figure 7.1). The trapezoidal forms are functional units; the lines connecting them are data paths. Most of the functional units compute familiar arithmetic functions, except for the unit marked with a ←, which acts as a filter for passing its upper input if only if the lower input is a boolean truth value. This particular program computes square roots by Newton's method (compare with the Scheme code for this computation presented in Chapter 4).

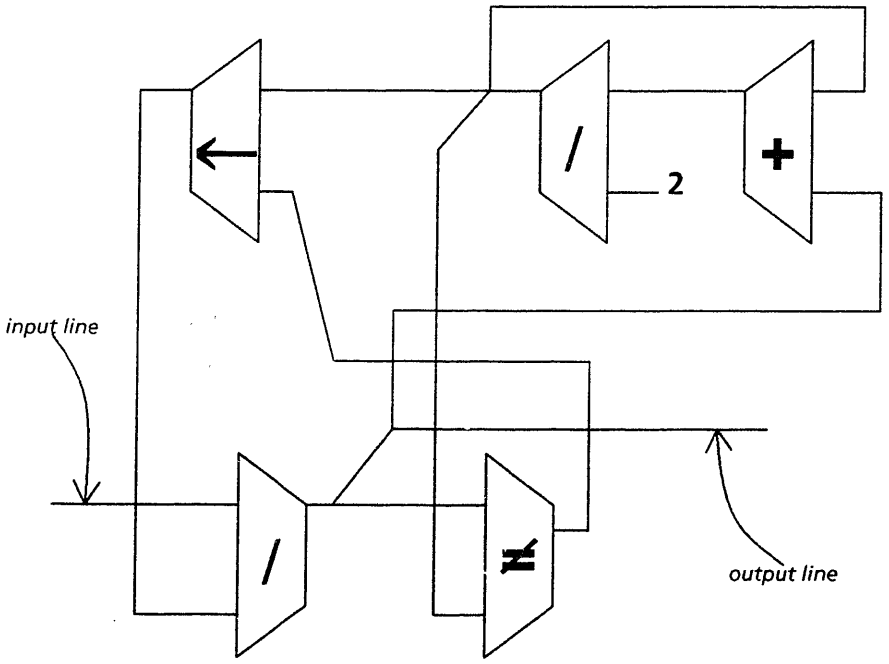


Figure 7.1: Square root program in Sutherland's system.

Trying to mentally run this example on a particular input quickly leads to confusion. Part of the problem is that there are no textual names to help the programmer understand the purpose of different parts of the configuration. A more intrinsic problem is that there are simply too many things happening in parallel to keep track of. Of course, we can impose our own sequentiality on the program, but we must be careful to remember all of our choices along the way.

To avoid the possible confusions of parallelism, Grasp embraces the the traditional sequential approach to procedural programming. This approach has the advantage that it is ruthlessly deterministic; the programmer never has to make any choices when simulating the program. Sequential programs are simply more conducive to mental simulation. In this respect, Grasp departs from the device programming style envisioned by diSessa, who sees parallelism as one of the most important features of physical devices [diSessa 86b].

Just because the procedural paradigm is more amenable to sequential simulation does not mean that all models for this paradigm aid in the envisioning process. Many models have two properties which make envisioning difficult:

1. *Creation of structures at run-time* - In many models, new interpretation structures are created at many steps during the evolution of a process. In the procedural paradigm, this is an

inherent property; new procedure activations must be created every time a procedure is invoked. In the environment model, for example, a new environment must be created whenever a procedure is applied because an environment is the repository of much of the state information of the activation.

The problem with creation of structure at run time is that there is a tendency to lose track of the envisioning process by getting bogged down in the details of structure creation. Students often apply the rules of the environment model blindly without trying to relate the intermediate results to a high-level view of what is happening in the process. Many never become facile enough with the model to realize that the *shape* of the environment frames for an application of a particular procedure is always the same and is in fact easily derivable from the text of the procedure (see [Sussman 85] for an explanation of this point). Thus, experts can look at the definition of a procedure and quickly mentally contruct the configuration of environment frames which corresponds to a call of that procedure. Novices tend to apply the rules that dictate that environment frames are not created until they are needed. Rather than envisioning the environment configuration in a single step, they take many steps. With each extra step, there is a chance to make a mistake or get lost in the simulation.

2. *Reuse of structures over time* - Interpretation structures are often reused during the course of a computation, usually for efficiency reasons. In the actor model, an actor returns to the unemployment line once he is done with his script. The contents of the registers and stack in the explicit control evaluator change at almost every step of the computation. Sometimes this type of reuse is intrinsic to the model of computation. Reuse of variables is intrinsic in the procedural paradigm, since the value of variables may be changed over time via assignment. Often, though, reuse is for efficiency reasons alone. If infinite memory were available, there would be no compelling reason to throw away the structures associated with a procedure activation when it returned its result.

> Reuse of interpretation structures makes reasoning about the past and future of a computation difficult. It requires the programmer to keep track of the uses of the structures at different points in time. For example, when using the explicit control evaluator, it is helpful to unfold the contents of the registers and stack over time by considering sequences of contents. By examining such sequences, it is possible to see the common patterns of register movement and stack use which take place over time. Such patterns are difficult to detect if the elements are reused since information about the past is lost along the way.

Run-time creation and reuse of interpretation structures underscore the intimate realtionship these structures have with time. What a model of a process describes at any particular point in time is a *static snapshot* of the process at that time. The dynamic evolution of a process must then be viewed as a sequence of these static snapshots, much in the same way that movies are created out of sequences of discrete frames. The unfolding in time of the register and stack contents suggested above is an example of this idea.[2]

In order to better support the envisioning process, Grasp attempts to reduce the dependence between interpretation structures and time. Consider the problem of creation of structure at run-time. As noted above, structure creation is inherent in procedure applications. In Grasp, however, not all work is done by procedure applications. It is possible to use primitive and compound machines anywhere in a program - these are in effect hardwired procedure activations. They give an effect similar to what one might see when procedure calls are open-coded by a compiler. In Scheme, it is not possible to manipulate activations in this manner - only procedures can be manipulated. When machines are used in the Grasp fashion, the creation is done at define-time rather than run-time. When reasoning about the evolution of the process, the programmer does not have to worry about a structure being created on the fly - it was already explicitly created before the execution of the program even began.

This line of reasoning does not hold true for all-purpose machines. The internal structure of an all-purpose machine is created on the fly when control actually reaches the machine. This corresponds to the dynamic creation of a procedure activation when a procedure is called. Yet, all-purpose machines themselves are explicitly inserted into the program at define-time. They mark a

site where creation of structure is going to occur. Even though Grasp cannot avoid the creation of structure, at least it localizes it to areas which are determined at define-time. In reasoning about a program, a programmer can use the fact that any new structure dynamically created during the execution of the programs must be associated with an all-purpose machine.

Grasp also avoids reusing interpretation structures when possible. To do this, it employs the same trick used by applicative languages to get rid of side effects on variables - mapping aspects of the time dimension into the space dimension. Applicative languages conceptually deal with unmodifiable sequences of values to represent the history of a changing variable over time. In Grasp, processes are represented as machines connected together by control paths and reference pipes. The key property of the machines is that they are *one-shots* - when control reaches them, they fire, but once they have fired, they cannot be used again. In this way a machine has a direct correspondence to a procedure activation at a particular point in time. To represent a use of the same procedure at a different point in time would require a separate machine.

The choice to make machines one-shots has some consequences for control flow. Since it is not meaningful to reuse a machine, there is never a case where control would need to come back to the same machine. As a consequence, machines have a single control path entering them and a single control path exiting them. This also means that as a process evolves, the controller moves ever forward from machine to machine and never retraces its path. Each machine represents a point in time, and the traversal of the controller over the control path can be interpreted as a passage of time.

I refer to the above idea as a *computational time line.* As a process evolves, the controller is essentially navigating along a time line through a space of interpretation structures. The time line is not simply linear - Grasp includes structured branches of control paths to handle conditionals. The notion of a time line is intended to aid reasoning about the evolution of a process. A particular point of the computation is marked by the current postion of the controller. The past of the computation is exactly the set of paths traversed by the controller up to its current position. Its future could be any of the possible paths which lead forward from its current position along the time line.

Grasp machines differ from traditional procedure activations in that they do not go away once they have finished computing. Rather, like all object in the Grasp model, they remain a part of the computation until the programmer explicitly destroys them. This means that after a process has evolved, a complete history of its evolution is maintained in the interpretation structures used for the process.

Saving the entire history of a computation may seem wasteful, but it is an important aid for reasoning about the structure of programs. The most straightforward use is for debugging programs. If a program does not exhibit the expected behavior, the programmer merely need examine the history of the process. It is not necessary to enter a separate debugger. Rather, the interpretation structures of the history can be inspected and manipulated like any other interpretation strucuture. It is a low level property of the Grasp model that the programming environment and the debugger are the same. This feature is designed for novices, who typically have a hard time learning a separate debugger.

Interpreting control flow as a passage of time has important consequences for debugging in Grasp. Because distinct points in time map to different structures in space it is perfectly meaningful to run a process backwards in time by having control traverse control paths in the opposite direction. At each point in backward flow, the controller undoes action. If the sites for smashing machines store the old values of variables, even side effects can be undone in a reasonable manner. This feature is valuable for debugging. If a programmer finds a fault in the program, he or she can send the controller back to the point of the error, fix the error, and send the controller forward again.

The saved state information is not only useful for debugging - it can aid the explanation process as well. In traditional programs, the returned value is the only inspectable result of a process. In Grasp, however, the entire history of the process is avaiable for inspection. It is thus possible to inspect all the intermediate steps used in reaching the result. Programmers can see not only that they got the right answer but *how* they got the right answer. The notion of "process" is often a hazy one for novices; the ability to examine the state associated with a correctly working process might give novices a firmer understanding of this idea. In addition, the process history is useful for reasoning about about the time and space requirements of a process. The programmer can

easily find out how many times a particular procedure was called in a computation. Explicit process histories also facilitate the comparison of processes.

Other novice-oriented systems share some of the aspects of Grasp's computational time line. In the Pict/D system, control is reified as a white box which travels forward along explicit control paths through an iconinc program [Glinert & Tanimoto 84]. Reiss's PECAN system allows the user to go both forward and backwads in their programs [Reiss 84]. The design of Boxer includes a single stepper which shows intermediate steps in an evaluation using the copy and execute model. As with the stepper in Scheme, however, the Boxer stepper must be explicitly invoked before the beginning of a computation.

7.1.7 An Integrated Environment

The Grasp environment is *integrated* in the sense that it allows programs to be defined, executed, and debugged within a single environment. The level of integration in Grasp makes inspection and manipulation of the structure of programs more direct than in programming environments that support separate editors, interpreters, and debuggers.

Consider the task of inspecting the state of a procedure activation. In Scheme, one needs to TRACE a procedure to see the arguments it receives and the result it computes. If one desires to change the state associated with a particular activation, one must BREAK the procedure, and use an arcane environment inspector to modify the state of the activation. This approach suffers from the problems associated with any command-language interface. At all times the user must mentally keep track of where he or she "is" in the system. Furthermore, the programmer must have familiarity with the special debugging procedures and tools. The task is usually so imposing to novices that many never learn how to use the debugging facilities to their full extent. Furthermore, even those who know how to use them will only resort to them in cases of dire problems. It is *possible* to manipulate an activation in Scheme, but it is not very likely that novices will do so.

In contrast, the Grasp interface is designed to facilitate such interactions. Since the environment is always in "debugging mode," relevant state information is always available through the interface. To manipulate a particular activation, the programmer searches for it spatially and, upon finding it, manipulates it in

the same direct he or she is accustomed to. The Grasp system facilitates the kinds of manipulations which novices can find difficult or impossible in conventional environments.

Several other novice-oriented systems are worth mentioning in this respect. Boxer, for example, is integrated in the sense that the text editor, file system, and interpreter are all rolled up into one environment. As in Grasp, the programmer creates and runs a program within a single environment. However, Boxer stresses the static rather than dynamic aspects of a program. The effects of a program on data structures is visible, but how those effects come about is generally *not* visible. The programmer must use a special stepper in order to see the details of program execution. Since Boxer is a text based environment, however, integration means that the user can easily put text anywhere. Compare this to Grasp, in which text can only be associated with variables and machines in the form of short names. A key advantage of the Boxer approach is that it supports a general computing environment rather than just a programming language. It is just as useful to people who want to do word processing as those who want to do programming [diSessa 85a]. Grasp is aimed only at programmers.

Several other interfaces provide more of the Grasp style of integration. In Sutherland's data flow system, the programmer constructs data flow configurations by moving visual representations on the screen. The resulting configurations can then be run in the same environment. The values on the data paths can also be viewed, but only in a special "debugging mode" [Sutherland 66]. Pict/D allows users to construct and run their programs within a single environment. Like Grasp, Pict/D has an explicit representation of control flowing through a program, and thus displays the dynamics as well as the static changes [Glinert & Tanimoto 84]. The Programming by Rehearsal system [Gould & Finzer 84] provides an integrated environment in which the graphical representations of computational performers can be auditioned, rehearsed, and performed. The dynamic actions of the program can be observed, although there is no explicit representation of control. Curry's prototype Programming by Abstract Demonstration system also uses a single integrated environment for program definition, review, and execution [Curry 78].

7.2 DRAWBACKS OF GRASP

Conventional wisdom proclaims that "nothing is for free." This notion is quite true in Grasp, where the above advantages are not gained without cost. The purpose of this section is to describe some of the drawbacks of the Grasp system.

7.2.1 Too Much Information

Without a doubt, the major problem with Grasp is that it tries to display *too much* information. A major goal of Grasp is to explicitly represent the traditionally hidden state and behavior of programs. Unfortunately, even small programs have a nontrivial amount of state information associated with them. In trying to display *all* the structural information associated with a program, Grasp makes it unwieldy to deal with anything other than small programming examples.

The main problem here is that the principle of direct mappings is double-edged. In specifying that the structure of the underlying model should be fully reflected in the visual representation, this principle is a statement about the *quality* of the representation. However, since visual information takes space on the screen, the *quantity* of information which can be displayed at any one time is limited. Presenting the detailed structural information implied by direct mappings means that the bandwidth of the display is being used to convey a fine-grained view of a small piece of the structural space. Relaxing the directness of the mappings allows for a coarser-grained view of a larger region of the structural space. Displaying a wider range of information at the cost of reduced detail is important for gaining a high level view of the structural space and determining the context of a particular piece of information.

To see the tradeoffs involved with direct mappings, reconsider the visual representations of list structures discussed in Chapter 3. Based on the analysis in that chapter, it appears that embedding the box and pointer notation into a direct manipulation editor would clearly aid the programmer in reasoning about list structures. Why is it that such editors are so rare? Part of the reason is tradition - textual notation and the READ-EVAL-PRINT loop have been carried down from the era of the teletype to the present day. However, I believe the problem is more fundamentally rooted in tradeoffs implied by direct mappings. It is important for new programmers to see how pairs are chained together to form lists, but once

they become familiar with this idea, the extra information conveyed by box-and-pointer diagrams becomes cumbersome. For anything larger than simple examples, box-and-pointer diagrams simply become unmanageable. Textual representations, though hiding some information, are much more concise and scale more gracefully. These characteristics make them the notation of choice in the Lisp community.

Notice that the problem here is with direct mappings and not naive realism. Naive realism is still a useful principle for interfaces aimed at the programming community at large. This is the driving force behind the Boxer project - that naive realism and the spatial properties of boxes can enhance a text-based programming language for programmers of all levels. Direct mappings, on the other hand, tend to limit the complexity of the examples which can be represented in a system. They provide a useful pedagogical environment for the novice, but tend to be not well-suited to the experienced user. A box-and-pointer editor is a great idea for students learning about list structures, but is of limited utility to the experienced Lisp programmer.

In Grasp, the problems inherent with direct mappings are most clearly exhibited by the conditional machine. The visual representation designed for the conditional machine emphasizes how control flows through the machine. The splitting and joining of control paths are represented explicitly, as is the relationship between the result of a predicate and the direction of its associated control switch. I believe that this kind of detailed representation clarifies the evaluation rules for Scheme's conditional statement. However, after those rules have become clear to a novice, the utility of such a detailed representation is suspect. Although first helping the novice to reason about the structure of the program, such space-consuming representation may later impede the novice's interactions by making it more difficult to access other information. As with box-and-pointer diagrams, the representation of a Grasp conditional may be useful for simple examples in the programmer's early experiences with the language, but soon after it becomes very cumbersome to deal with.

The consequence of direct mappings in Grasp is that they limit the system to handle only simple programming examples. (The limitation is purely a practical one; in theory, Grasp can handle any programs manipulating numbers, booleans, and blueprints.) This does not mean that the system will not not helpful to novices. Even simple examples are nontrivial to understand at a deep level in

Scheme. Grasp provides a method of viewing the important structural concepts underlying the simple examples. This information is valuable when linguistic intuitions begin to fail the programmer. What it does mean is that, at this stage in its development, Grasp can hardly be considered a true general programming environment. As it stands, Grasp is a tool for illustrating the ideas of the Grasp model, but it cannot be used for serious programming.

Some other novice-oriented systems have fallen prey to the same problems as Grasp in this regard. In Sutherland's system, complexity quickly takes over - his example of the square root program (Figure 7.1) is all but undiscernible. Pict/D also suffers limitations in this regard - it is suitable only for simple numerical problems. Other systems, however, have dealt with the complexity problem much more gracefully. By espousing concise textual representations, Boxer makes it possible to construct fairly complicated programs. Eisenberg's BOCHSER is able to handle nontrivial Scheme programs - his report shows how the system could be used with object-oriente and production-based examples [Eisenberg 85] . Gould and Finzer's Programming By Rehearsal system has proved quite successful in one test study. Two curriculum developers with weak programming backgrounds were able to design and implement an exciting game for teaching fractions within the span of a few days [Gould & Finzer 84].

The reason the above systems are successful at allowing complex programs is that their representations stress function rather than structure. By not focusing on the structural details of a program, the systems admit to much more concise representations of computational elements. The representations stress what the elements are for rather than how they do what they do.

Simple detail suppression schemes such as scrolling and shrinking only partially mitigate the problem of too much structure. Even the shrunken representations take up much more space than the information they convey merits. Further, expansion generally totally overwhelms the surrounding information. The expanded form of a shrunken conditional machine takes up the better part of the display screen.

What can be done to improve Grasp in this respect? The problem is rooted in the goals of the system. The goal of representing structure is in basic conflict with the goal of allowing nontrivial examples. The first goal must be relaxed if the second is to gain any ground. Functional representations which concentrate less on structural details are necessary to support more complex programs.

Perhaps Grasp could provide a more layered approach to structure in which structural details are normally hidden but programmers could explicitly request more detailed views. Incorporating into Grasp the notion of fisheye views - interfaces which allow the user to examine the details only of the object in focus [Furnas 86] - would also provide the user with a better way of navigating through the large space of structural information.

7.2.2 Usability Problems

Some of the usability problems with the Grasp system were described above. Other problems result from the device programming style embraced by Grasp. Even simple programs require a fair amount of work to construct. Consider making a squaring machine. We must first create a compound machine, and then put a multiplication machine inside of it. But this is not enough - we must also explicitly connect three reference pipes and two control paths. It would be desirable to be able to specify such a simple operation with less work.

One approach would be to have the system make some connections by default. For example, when we put a machine in the internal structure of the compound machine, we most likely want a control path to pass through it. The system could automatically make such connections for us by default. If the connections were not the ones we wanted, we could always change them. Another such usability change could be made for determining the value of output variables. In most cases, the output variable of a compound machine is wired to the output variable of the last internal machine. Again, the system could make such connections for us by default.

The kinds of suggestions above certainly help the situation, but do not greatly improve the usability. A more significant gain in usability could be gained by allowing structures to be specified through some kind of textual language. We can imagine typing in the following Scheme-like expression to some interpreter

```
(MAKE-COMPOUND-MACHINE SUM-OF-SQUARES (A B) (C)
  (C ← (+ (SQUARE A) (SQUARE B)))
```

and having it create for us the machine shown in Figure 7.2.

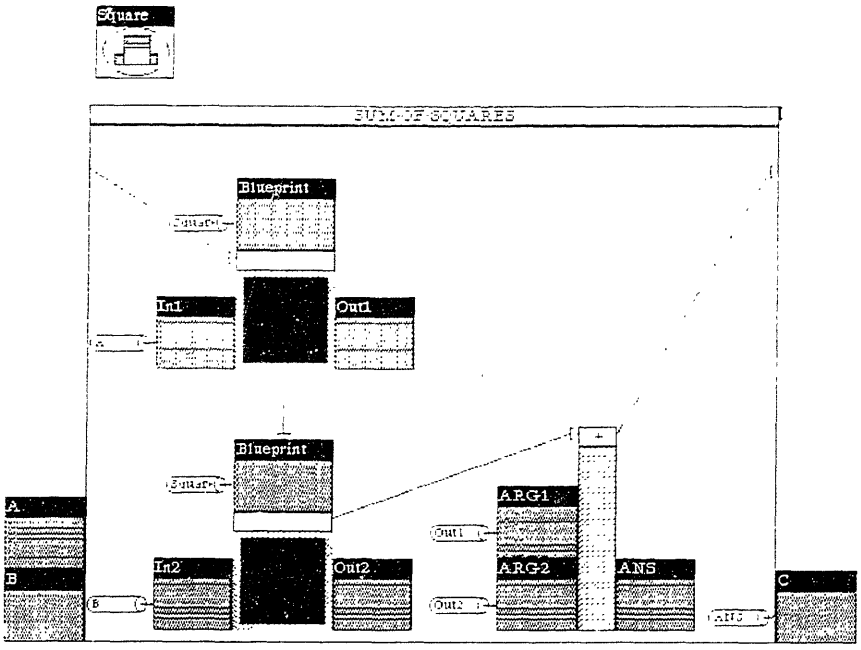


Figure 7.2: Machine resulting from the above expression

Such an interaction would greatly increase the usability of the Grasp system. It would also help novices to make the tie between the world of expressions and the structures they denote.

I should be careful to point out that there would be many subtleties in designing such a langauge. First of all, naming would pose a problem. Since reference in Grasp is based on structure rather than name, using simple names in expressions could not in general unambiguously describe a machine. If there are two variables with the name A in the Grasp window, how could Grasp decide what the following piece of code means?

```
(MAKE-COMPOUND-MACHINE (X) (ANS)
  (ANS ← (+ X A)))
```

In such cases of ambiguity, some method would have to be provided for the programmer to indicate the intended meaning. In the above example, the system might request that the user point to the variable being referred to.

Another point is that the explicitness of structure in Grasp means that elements can be specified to a fine degree. For example, unlike Scheme, Grasp allows users to deal with activation-like elements without defining an associated procedure. This distinction must somehow be embedded in the textual language. In the SUM-OF-SQUARES example above, for instance, is + a special keyword denoting the primitive addition machine, or is it referring to a blueprint stored in variable

named + ? In the former case, how would a programmer specify that the value of the variable + was intended? In the latter case, how would the programmer specify the use of the primitive procedure? When issues such as the explicitness of control and allowance for multiple returned values are also considered, it is clear that there are indeed many thorny issues to be resolved in the development of such a language.

7.2.3 Semantic Questions

Because programmers are able to manipulate the abstract machine elements directly in the Grasp environment, many common errors which occur in textual programming environments are not possible in Grasp. For example, in Grasp it is impossible to refer to a non-existent variable. Since names have no bearing on the meaning of a program, typing errors or misremembered names cannot possibly lead to any semantic error. Reference pipes in Grasp are made by pointing to a variable, so a prerequisite of obtaining a reference pipe is that the variable actually exits. Further, when a variable is destroyed all reference pipes created from the variable must also be automatically destroyed to avoid the problem of "dangling" reference pipes.

Although Grasp avoids many of the errors common in conventional text-based programming languages, it unfortunately introduces new types of errors and ill-defined configurations. For example, in constructing a compound machine for squaring we could accidentally leave out an internal control path (Figure 7.3) or forget to connect a reference pipe from the output variable of the multiplication to the output of the compound machine (Figure 7.4). Since control flow and return of variables are implicitly handled in most programming languages, it would not be possible to make these kinds of errors in those languages. For example, in Scheme it is impossible for a procedure *not* to return a value.

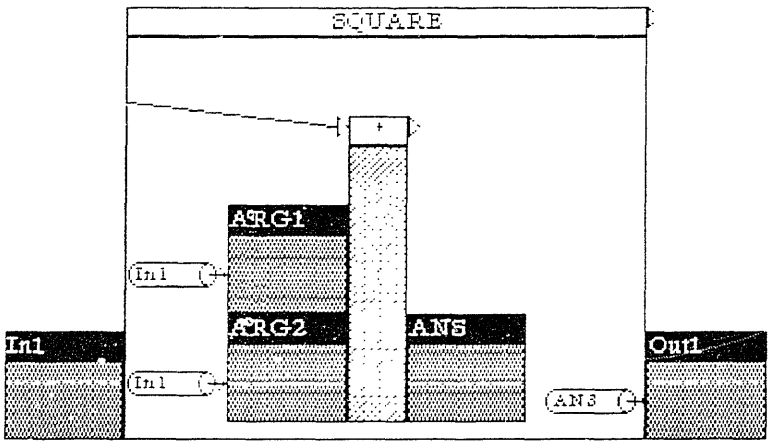


Figure 7.3: Compound machine with missing control path.

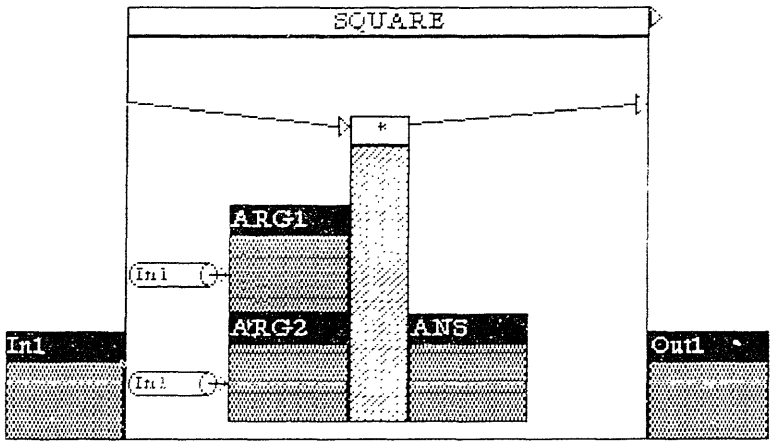
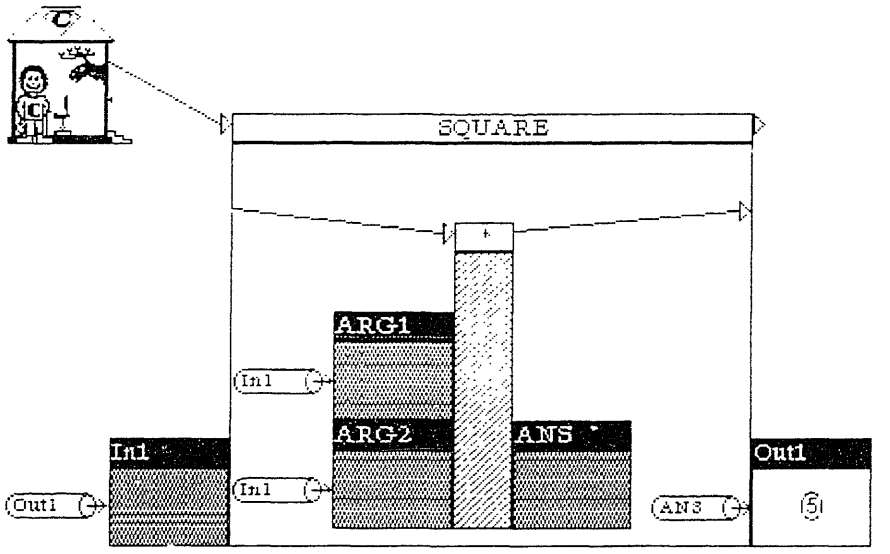
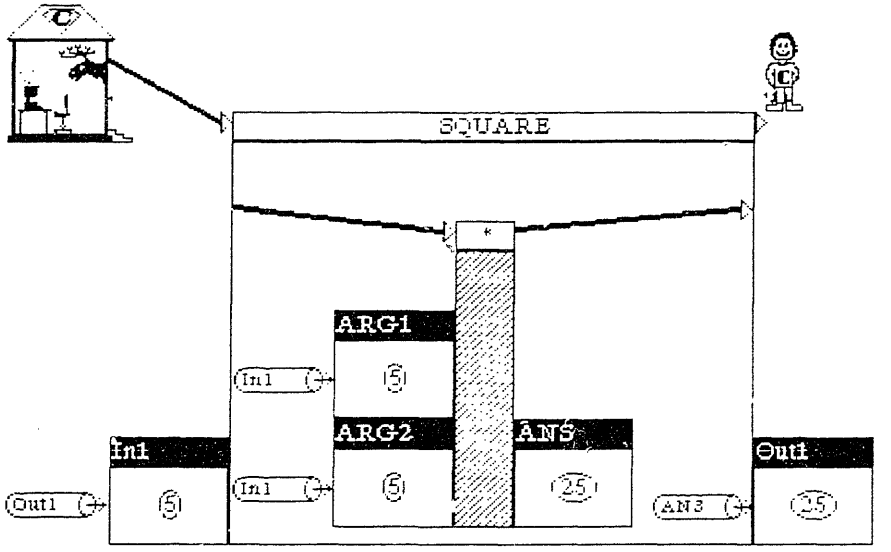


Figure 7.4: Compound machine with missing reference pipe.

Certain Grasp configurations that do not contain errors still seem rather ill-defined when compared to programs in Scheme. Consider the configuration pictured in Figure 7.5. Here the input of the squaring configuration is taken from the output variable of the same configuration. The output variable value changes when the controller exits the compound machine.



Before



After

Figure 7.5: A strange way to get inputs.

The use of reference pipes in this way is somewhat disconcerting - we would normally like to think that we can't refer to the ouput of an activation before its value has been computed. However, due to the Grasp's notion of a computational time line, it is possible to manipulate some of the run-time structures used by a computation before that computation ever begins. An even stranger use of reference pipes is exhibited in Figure 7.6, where the squaring machine gets its input from the output variable of the multiplication.

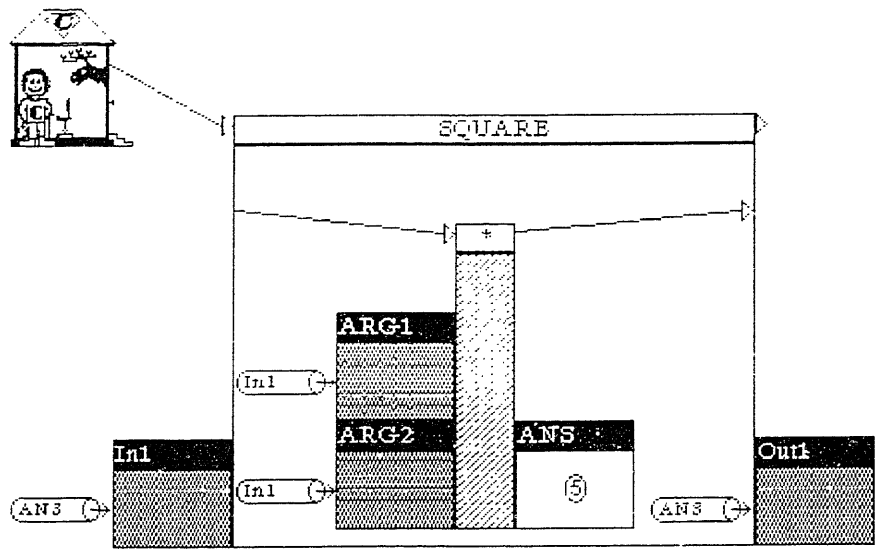


Figure 7.6: A stranger way to get inputs.

Not only does using a reference to the output variable of the multiplication for the input of the squaring machine appear to contradict causal time constraints in a program, but it also seems to violate an intuitive spatial scoping rule which says that reference pipes shouldn't be allowed outside of the internal structure in which their source variable lies.

It would be possible to incorporate into the Grasp system features to detect or prevent the kinds of configurations defined above. For example, to prevent discontinuous control paths, the system could include a control terminus and maintain an invariant that a continuous path of control must lead from the control house to the control terminus. This basic approach is followed in Pict/D. Similarly, the system could maintain a constraint on output variables, dictating that they must either contain a value or be the target of a reference pipe. To prevent the use of reference pipes in undesirable places, the system could prevent their attachment to variables that occur earlier on the computational time line than or lie outside the spatial scope of the source variable.

It is not clear that the features suggested above, though, are worth the complications that they would introduce into the system. Some of the "undesirable" configurations might actually be useful in some situations. For example, we could include a control "break" in any machine by simply disconnecting a control path; when the controller reaches the break in the path, the interface could focus in on that part of the computation. The intuitive rules of

spatial reference are actually broken by the system when using the copying algorithm on blueprints maintaining local state. This style of reference is inherent in systems supporting lexical scoping and procedures with state. Since the system builds such structure during the construction process, there is no good reason to prevent the user from building similar kinds of structure by hand. Furthermore, it is important to realize that it is possible to write obscure code in any language, but that every language is accompanied by an acceptable style of programming. Programs written in good Grasp style would not make use of the kinds of configurations illustrated in Figures 7.5 and 7.6.

Despite the above discussion, there are several situations in the Grasp model where the system must be careful to preserve an appropriate semantics. The two most important situations involve blueprints and the copying algorithm:

1. A reference pipe whose source variable is inside a procedure cannot be attached to a target variable outside the procedure. The copying algorithm implies that each application would add an extra reference pipe to the external target variable, which does not make any sense.
2. A smasher may not be inserted into a blueprint from a smashing machine outside of the blueprint. In such a situation, the copying algorithm would indicate that a single smashing machine would mutate several variables, which does not fit into the Grasp semantics.

7.2.4 Space

The Grasp model of computation requires saving all of the intermediate state of a program. Saving the history of a process in this manner allows the novice to inspect the computational time line of a completed computation and to run the computation backwards to an intermediate state, if so desired. Of course, Grasp can only be so inefficient in its use of space because it is intended for small programs. More complex programs could easily produce a bewildering amount of information which might even exceed the storage capacity of a machine. It is for this reason that in most models of computation, the state associated with an activation is discarded when the activation is done. As memory becomes cheaper and larger storage capacities are realized, however, it may not be too farfetched to

use the Grasp style of debugging even for more complex programs. Also, as will be mentioned in Chapter 8, the illusion of saving state can be maintained to some degree without actually saving state.

7.2.5 Grasp is not Scheme

The discussion of section 5.1 should make it clear that *Grasp is not Scheme.* This fact raises some important questions. How is Grasp supposed to help alleviate the very problems from the Scheme course which motivated its development? If taught to Scheme students, won't the differences of the Grasp model confuse them rather than help them?

The goal of Grasp is to help students understand fundamental structural ideas inherent in the procedural paradigm. Although the project is certainly motivated by problems which occur in Scheme, I believe the problems are basic to procedural programming languages in general. Notions such as first-class procedures, references to variables, and control flow are difficult to grasp in almost *every* procedural language. Futhermore, the expression-oriented approach taken by most procedural models, Scheme in particular, are simply not suitable for the approach of conveying the structure of programs through a visible and manipulable interface. It is for these reasons that Grasp is not Scheme.

However, I believe that the ideas embodied in Grasp will be useful to all programming novices, including those taking the Scheme course. The unique way in which Grasp represents first-class procedure objects, for example, should help novices gain a new perspective on what they are. Showing explicit structural connections between variables and references can give the programmer a better insight into the effect lexical scoping tries to achieve. Certainly there is a possibility of confusion between Grasp and Scheme. For this reason, I think that the two should be clearly distinguished in any teaching situation. The way I envision using Grasp is to introduce people to the basic ideas of procedural programs *before* they learn a language such as Scheme. Then the models for the more advanced procedural language can be viewed in terms of the way they deviate from the Grasp model.

7.3 Summary

Incorporating a device programming style of procedural programming into a transparent interface gives Grasp several advantages. Most importatant among these is that the programmer gets "hands-on" experience with the elements of the Grasp model. Inspecting and manipulating such elements as machines, blueprints, variables, reference pipes, control paths, and controllers, the novice should be able to get a good "feel" for the elements of procedural programming. In particular, Grasp should help clear up the kinds of the difficulties which novices encounter in learning the Scheme programming language. The Grasp approach does suffer several drawbacks, however, most important of which are space-consuming representations and the lack of usability features. These disadvantages constrain Grasp to be a pedagogical tool for illustrating simple programs. In order for Grasp to become a more versatile, general-purpose programming environment, it must concentrate less on structure and provide more functional view of programs.

CHAPTER 8

CURRENT STATUS AND FUTURE DIRECTIONS

8.1 CURRENT STATUS

A prototype implementation of the Grasp system has been partially implemented. The system is written in LOOPS, an object-oriented version of Lisp which runs on top of Interlisp-D. The Grasp system runs on the Xerox set of "D" machines, including the Dorado (1132), DandeTiger (1109) and DandeLion (1108). These machines support a 32 megabyte virtual address space, a high-resolution bitmapped display screen, and a "mouse" as a pointing device.

Most of the Grasp model as described in this document has actually been incorporated into the working system. Many of the figures in this document were taken directly from the current Grasp system. At the present writing, side effects and certain aspects of blueprint application are the only major features which remain to be implemented. Of course, there are a host of minor features which would increase the usability of the system if they were included as well.

The Grasp system is integrated into the normal LOOPS environment. All interaction with Grasp takes place through a Grasp window, whose shape and location are determined by the user. All other functionalities of the LOOPS system are accessible to the user through normal interactions with Interlisp-D's window-based environment.

The performance of the system tends to be slow, which is not surprising considering that it is implemented in a straightforward and inefficient style. On the Dorado, the fastest of the "D" machines, the response for the manipulability features is actually quite fast. In particular, objects can be picked up without much delay, and the continuous representation of motion is very smooth. The other machines (DandeTiger and Dandelion) do not provide as fast a response time. On these machines, picking up an object can take longer than expected, and

the representation of motion is not very smooth. Slowness of response makes the manipulations tedious to carry out and reduces the resemblance of the representations to physical objects.

8.2 FUTURE DIRECTIONS

The Grasp project can progress in several different directions. This section describes some avenues for exploration.

8.2.1 Evaluation with Novices

The most important future direction for this project is to test out the Grasp system with novices. Now that the system is finally reaching the stage where nontrivial programs may be written, it will soon be possible to evaluate Grasp system with students. It is important to assess to what degree the Grasp system actually meets its goal of helping novices build better structural models of the procedural paradigm.

I plan to evaluate the system in sessions with MIT students. I am interested in testing students in the Scheme course as well as those who have not had any experience with Scheme. My main interest is in the extent to which Grasp helps them understand procedures, especially their properties as first-class objects. Towards this end, would like to explore how well novices can read, modify, and debug the Grasp equivalents of the kinds of examples suggested in Section 5.2.

Novices will also be able to provide much-needed feedback about the interface to the system. There are many important questions which can only be answered by evaluating the system with subjects. How easy is Grasp to use? Do the chosen representations seem natural or are they confusing? What are some other choices of representation? What additional features would make Grasp more usable? Does the system make the notion of first-class procedures more understandable? Answers to these questions will be the basis for future improvements to the Grasp system.

8.2.2 Extensions to the System

An obvious direction for this project is to improve and augment the implementation. The first priority is to complete the implementation of blueprint application and to incorporate side effects into the system. After these are done, many other extensions are possible. One direction is to improve the performance of the system. In terms of time, special attention needs to be paid to the code for manipulation in order to improve the response time of the system to users' interactions. Blueprint application is another area where a faster implementation is desirable.

Space has not yet proved to be a limitation for Grasp, but this is due to the fact that it has only been used for simple examples. If Grasp is to allow more complicated programs, it will not be possible to keep a history of *all* the intermediate state of every computation. However, I still believe it will be possible to maintain the *illusion* that this state is in fact being kept. The key idea is that the state only has to exist if the user wants to inspect it. Since users will not want to view the saved state of a process in most situations, there is a clear advantage to discarding the major portion of the intermediate state. In the instances where they do want to inspect it, reconstruction of the intermediate state should be possible for many common cases. In a purely functional program, for example, all intermediate states could easily be regenerated from scratch simply by running the computation from the beginning. When side effects (both from the program and the user) are allowed, the situation becomes much trickier. I expect that maintaining the illusion to a great degree would still require a fair amount of space and would introduce other complications to boot. However, I believe that Grasp could handle many of the common situations using only a small amount of extra space beyond that used by conventional systems. Exploring tradeoffs between the quality of the illusion and the space requirements is a fruitful area for future research.

Another direction is to add more features to Grasp. For example, it would be nice if default data and control path connections were made by the system when adding machines to the internal structure of other machines. Allowing blueprints to be constructed automatically based on the pattern of an existing compound machine is a feature that would greatly facilitate the creation of blueprints. These are only two of the many extensions to Grasp which would improve its usability.

8.2.3 More Functional Emphasis

As noted in the previous chapter, Grasp's emphasis on structure limits the size of the examples which can be effectively constructed in the Grasp system. An interesting avenue for exploration is to see if a greater functional emphasis can be incorporated into Grasp. Since this shift in emphasis would allow more complex programs to be constructed in Grasp, it is an important possibility to consider.

I can see three approaches for achieving more of a balance between function and structure in Grasp. The first method is to choose visual representations which show less structure than the current ones. The Grasp conditional structure, for example, is unwieldy because it tries to explcitly represent the process by which conditional branching is determined. Although this representaion might help a novice initially, it is likely that it will become cumbersome very shortly thereafter. It would be desirable to provide an alternate representation for conditionals which showed less of their structure. We can even imagine a sequence of alternate representations, each of which shows less structure than the previous one. The programmer could then choose the level of detail that he or she wanted to see. This idea is applicable not only to conditionals, but to all Grasp representations.

A second way to allow for more complex programs is to have a better scheme for information suppression. Scrolling and explicit shrinking are not sufficient for managing a large amount of visual information. Although the context provided by the current expansion mechanism is nice, it also results in the screen being cluttered with many unimportant objects. Perhaps a mechanism along the lines of a *fisheye view* would be a more effective means of detail suppression. Fisheye views show objects at the center of attention in detail, but peripheral objects at a level of detail which decreases as a function of their distance (by some metric) from the center of attention. (See [Furnas 86] for a discussion of fisheye views).

A third method of striking a balance between structure and function is to allow textual representations of programs in the system. For example, blueprints need not be specified graphically in a device programming manner - we can easily imagine that they be described in some form of textual language instead. However, their meaning in an all-purpose machine would still be the same - i.e.

the controller would still build structure inside the all-purpose machine based on the code in the blueprint. There are many subtleties involved in introducing textual program representations into Grasp, however. As discussed in the previous chapter, there need to be ways to refer to objects by name and to distinguish particular machines from blueprints.

8.2.4 Variants on the Procedural Paradgim

Grasp was designed to handle the procedural paradigm in a way similar to Scheme. However, the Grasp elements can support many variations on the procedural paradigm with relatively few changes to the system. It would be interesting to allow the user to choose among these variations and experiment with them.

Some of the different models the Grasp system could represent without a great amount of effort are described below:

Data Flow: The Grasp system was designed with explicit control flow in mind, but there is no reason why the pieces could not be used in a variation without explicit control flow. Consider making the foloowing three changes to the system:

1. Machines are not connected by explicit control lines. Instead, they fire when all of their inputs become available.
2. Reference pipes are no longer controlled by the controller. Instead, they allow data objects to flow at all times. When the value of a variable changes, all of the variables to which it is connected by reference pipes are immediately updated.
3. Removal of side effects.

If these three changes were made, Grasp would become a data flow system.

Note that the Grasp version of data flow would differ from Sutherland's data flow system in a crucial respect. Sutherland's system has the property that the functional elements and data lines are reused over time. On the other hand, as in the case with explicit control flow, Grasp would still have one-shot machines. That is, Grasp would not allow machines to be reused in the data flow scenario;

activations differing in time would still be represented by spatially distinct machines.

Clairvoyance: Clairvoyance is a term I use to describe a variant of the procedural paradigm in which structural properties of the "future" of a computation can be inspected. In computations from standard programming languages, much of the structure of a program determinable before the computation ever starts. Consider the following piece of Scheme code:

```
(DEFINE (SQUARE X) (* X X))
```

In evaluating an expression such as `(SQUARE A)` where `A` is bound to 3, it is apparent that an activation of the * procedure will be created for an input of 3. Yet, in most computational systems, the structure associated with this activation is not actually created until control reaches the point of the call to the multiplication procedure. In Grasp, the same is true; activations are not created and data does not flow until the controller reaches an appropriate point of the computation. For example, in Figure 8.1, we can clearly see that a construction of the SQUARE blueprint will occur with the argument 3, but because the controller has not reached the all-purpose machine, none of this structure is actually visible.

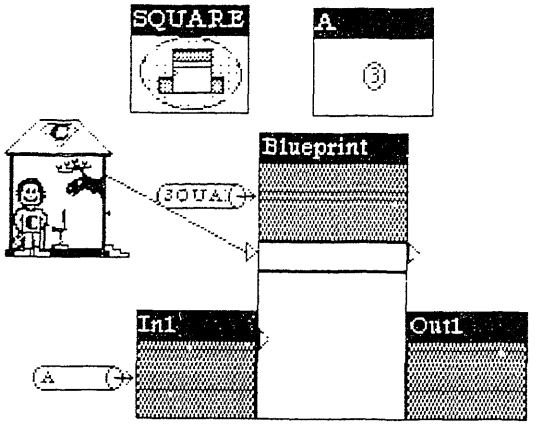


Figure 8.1: No structure is yet built.

In keeping with Grasp's notion of a computational time line, it would be nice to allow the programmer to manipulate the future structure of a program even before it would be created in more traditional models. In the above example, we could "predict" that an

activation of the SQUARE blueprint would exist in the future and that its argument would be 3. Figure 8.2 shows how this might appear in a clairvoyant version of Grasp.

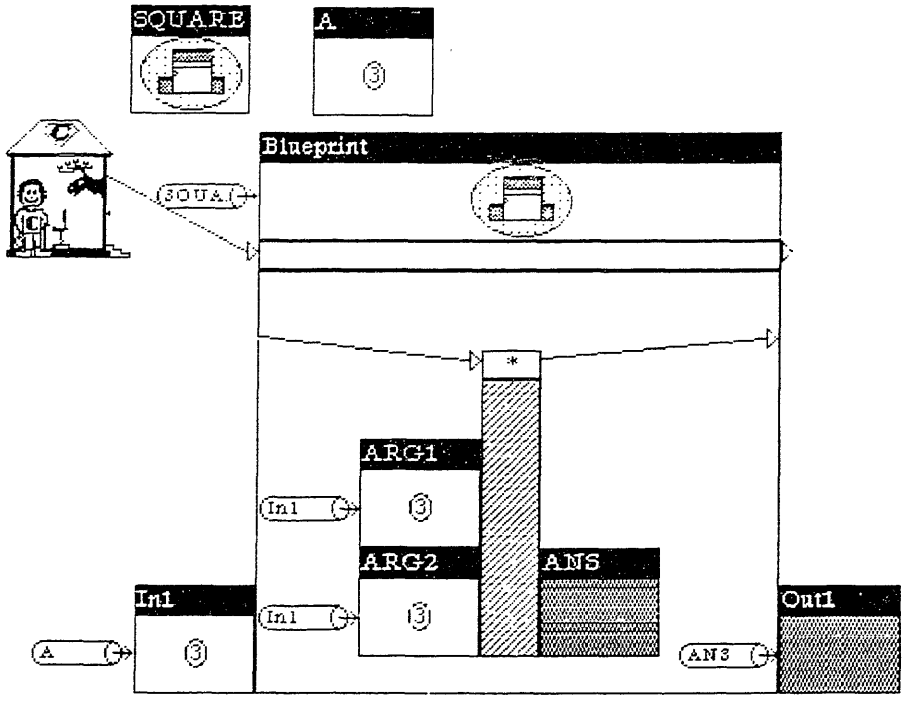


Figure 8.2: Clairvoyant version of squaring.

Note that certain points about the future cannot be easily predicted. In particular, the results of primitive machines and conditionals are not in general known in advance.

Clairvoyance has the advantage that it helps the programmer envision the future of a computation. As pointed out in Chapter 5, prediction is hampered by having to imagine the creation of structure. By showing the programmer what the "future" looks like, clairvoyance aids the programmer in prediction tasks.

The disadvantage of clairvoyance is that it conflicts with the principle of naive realism. The structures presented clairvoyantly aren't "really" there; they will not be created until control actually reaches the point in question. Because of the presence of side effects, the predictions are not always guaranteed to be correct. It will be interesting to implement a clairvoyant version of Grasp and to test

whether it helps novices reason about the programs or actually confuses them.

Multiple Controllers: The pieces of the Grasp system could be used to allow novices to experiment with explicit parallelism via multiple controllers. It is easy to imagine allowing more than one controller to run at a time. Multiple controller systems could provide an interesting representation of race conditions in a program. We can also imagine new controllers being spawned at control forks (as in a conditional, say), as a means of illustrating parallelism.

Scheme: The pieces of the Grasp model could also be modified to present a model much closer to that embraced by Scheme. The following changes would result in system much closer to the Scheme model of computation:

* Removal of output variables - Scheme has no equivalent of an output variable. Instead, the controller "carries" the single value returned by an activation to the next activation.
* Combination structures - the main type of Scheme expression is a *combination*, in which a procedure is applied to arguments. A structure corresponding to a combination could be added to Grasp. This special structure is needed to collect the values which result from the subexpressions in the argument positions of a combination.
* Machines that are automatically destroyed after they are fired. Although this makes for a model closer to Scheme, it also gets rid of one of Grasp's key features - the maintainance of state along a computational time line.
* Tail recursion - Upon entering the last machine in an internal structure, the structure surrounding that machine should be destroyed.

Note that incorporating these features means that significant advantages of the Grasp system will be lost. In particular, the whole notion of a computational time line will no longer hold. Although it would still be possible to return to some previous state of the computation, one could not return to an arbitrary previous state.

Furthermore, only the result of the computation and not the entire process would be inspectable after the computation is done. Despite these problems, having a Scheme-oriented version of Grasp would be useful for helping novices understand the differences between Grasp and Scheme. Such a system could serve as a stepping stone between Grasp and Scheme.

8.2.5 Exploring Other Programming Paradgims

A final area for exploration would be to consider applying the principles of visibility and manipulability to very different paradigms of programming. Because it supports an object metaphor, object-oriented programming is ripe for visible and manipulable interfaces. What about logic programming? Could visibility and manipulability be used to help programmers understand the structure of logic programming? Perhaps the physical object metaphor might be useful for explaining certain aspects of the unification, a pattern matching process at the heart of most logic programming systems.

CHAPTER 9

SUMMARY

Programming languages bear little resemblance to natural language. Interpreters for a programming language treat programs with a precision and determinism unparalleled in a human context. Unlike communication with people, communication with computers is marked by well-defined structure. This structure, however, may not be immediately apparent to the programmer because evidence of its existence is hidden by opaque interfaces that fail to convey information about the state and behavior of programs. Such a situation poses a barrier to students trying to understand the model of computation embodied in a programming langauge; based on their intuitions and the paucity of evidence with which they are presented, they will often construct poor mental models of the abstract machine that interprets their programs. Experience with students in MIT's Scheme course has shown that these factors prevent many students from acquiring a robust enough understanding of the programming model to read, write, and debug programs effectively.

This report describes the principled design of Grasp, an explicit model and programming system to help novices form more robust models of procedural programming languages. The key principles that Grasp adheres to are visibility and manipulability. Visibility decrees that the high bandwidth of a bitmapped display screen should be used to convey structural information about the interconnected elements of the abstract machine. Providing novices with more information about these elements should give them a more fundamental appreciation of the structure implied by a program. Manipulability exploits people's familiarity with interconnected physical devices to help novices interact more directly with the elements of the model of computation. The intention of manipulability is to allow novices get a better "feel" for the interpretation structures of a program through "hands on" experience with them.

These principles alone do not guarantee the transparency of a model of computation. If important structures of the abstract machine remain implicit in the model presented by the interface, programmers will have a hard time reasoning about situations where these implicit structures come into play. Subscribing to a principle of reification, Grasp represents many of the implicit structures of the procedural paradigm as concrete objects or devices. Thus, the Grasp programmer can directly manipulate structures resembling procedure activations (machines), environment bindings (variables), data paths (reference pipes), and control (the controller). These elements have the interesting property of mapping aspects of time into space; incorporating them into a device programming system allows a process to be viewed as a computational time line along which control can move forward and backward. The programmer can explore the shape of a computation before it begins, and can inspect the history of the computation once it completes.

Adhering too closely to physical object metaphor restricts the abstraction mechanisms which are available to the programmer. Grasp embodies enough of the linguistic view of programming to support procedural abstraction and indirection by naming. Procedures in Grasp appear in the form of blueprints which can be "applied" to arguments at the site of an all-purpose machine. Naming in Grasp is accomplished in a spatial manner by variables; references are specified by pointing rather than textual names. With these elements, Grasp is able to reap the full-power of first-class procedures without resorting to the use of text.

Emphasis on structure limits the Grasp system to small programs. Although Grasp's visual representations provide a wealth of structural information useful to novices, these representations become unwieldy for anything larger than simple examples. Representations that are more functional in nature are required to extend Grasp from a pedagogical tool to a full-fledged programming environment. With its current design, however, Grasp is still a valuable tool for helping novices gain a deeper understanding of the structural aspects of procedural programs, especially the use of procedures as first-class objects.

APPENDIX

Chapter 5 introduced the elements of the Grasp model and presented examples of some simple programs. This appendix presents more examples of Grasp programs (along the lines of those suggested in Section 5.2), including some which exploit the power of procedures as first-class objects.

A.1 FACTORIAL

No description of a procedural language would be complete without a definition of a factorial procedure. In Scheme, this procedure is defined as below:

```
(DEFINE (FACTORIAL N)
  (COND ((= N 0) 1)
        (ELSE (* N (FACT (- N 1))))))
```

In Grasp, we first create a blueprint which contains a conditional machine, as shown in Figure A.1.

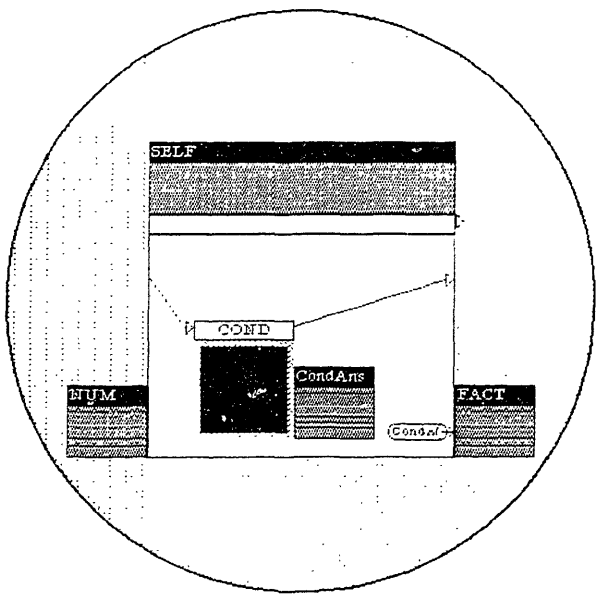


Figure A.1: The top-level blueprint for factorial.

Since it is difficult to show all of the structure of a Grasp program in a single picture, we will explore them in a top-down manner, exposing details as we progress. Although Grasp currently has no facility for printing out programs, it might be done in a similar manner.

The expanded conditional appears in Figure A.2. It tests whether the input number (NUM) is equal to zero. If it is, it returns the number one; otherwise it performs a recursive call to the factorial blueprint.

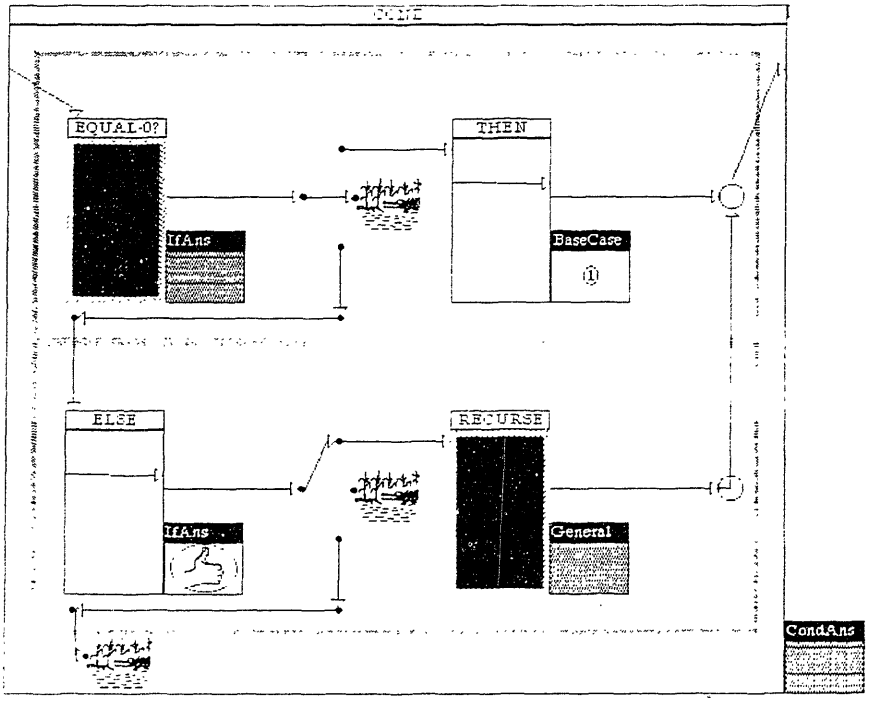


Figure A.2: The conditional for the factorial blueprint.

The EQUAL-0? predicate machine, shown in Figure A.3 is straighforward. The RECURSE consequent machine, appearing in Figure A.4, is more interesting. Here we recursively call the blueprint from which the activation is constructed by applying it to the number one less than NUM. In Chapter 5, the motivation for using an all-purpose machine rather than a simple compound machine for the template within a blueprint was not made clear. The only extra feature the all-purpose machine provides over a compound machine is the blueprint variable. We see in this example that having access to that variable simplifies the creation of recursive blueprints. Rather than creating a blueprint, storing it in a variable, and then using a reference pipe to that variable within the blueprint, we simply refer to the blueprint variable of the all-purpose template. Since that variable

must contain the factorial blueprint when factorial is being constructed, it is a logical place to look for the recursive blueprint. In this example, we have a reference pipe to SELF, which is the blueprint variable in Figure A.1.

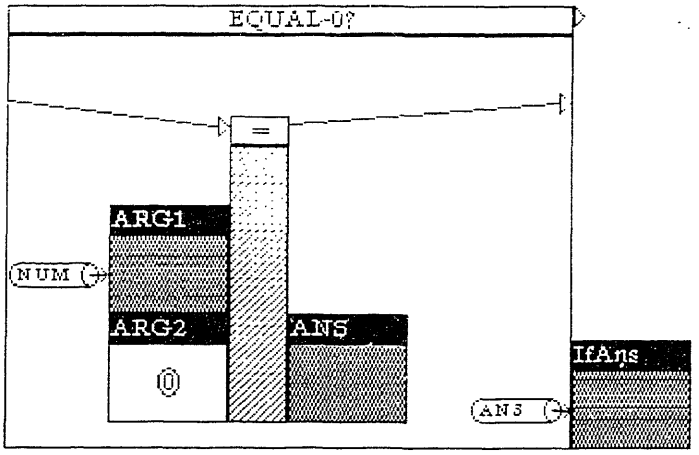


Figure A.3: Details of the EQUAL-0? predicate machine.

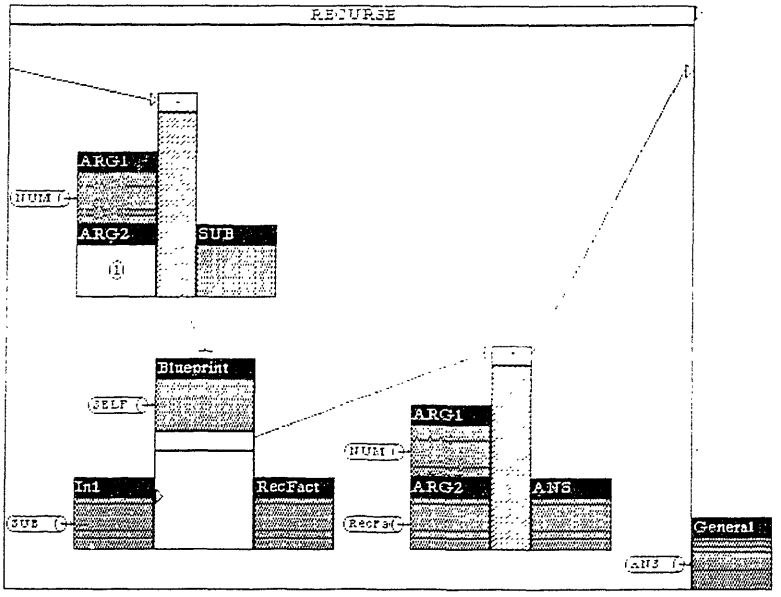


Figure A.4: Details of the RECURSE consequent machine.

A.2 APPLY-TO-FIVE

No examples of higher-order procedures were given in Chapter 5. APPLY-TO-FIVE is an example of a procedure which takes another procedure as an argument. In Scheme it is defined as:

```
(DEFINE (APPLY-TO-FIVE PROC)
  (PROC 5)
```

The Grasp version of this procedure appears in Figure A.5.

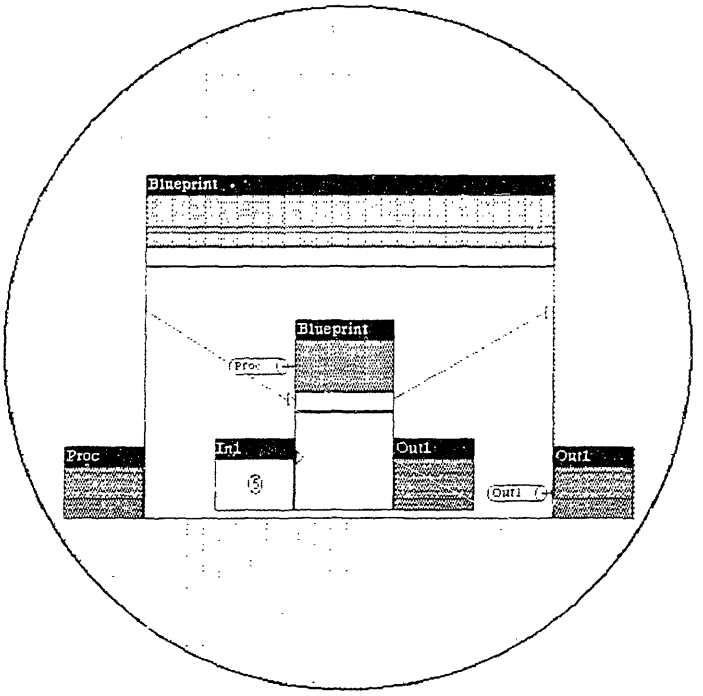


Figure A.5: A blueprint for applying a blueprint to 5.

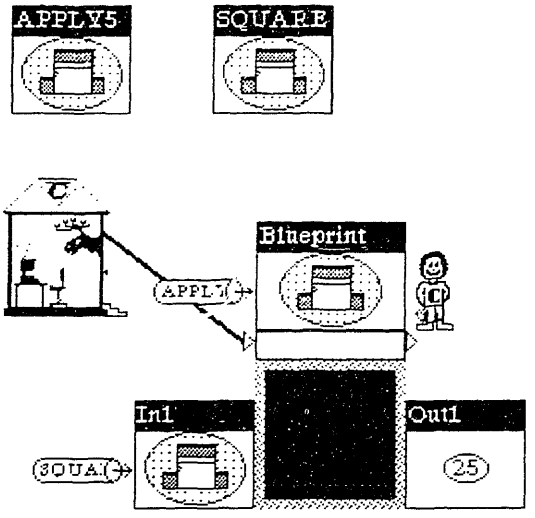


Figure A.6: Using APPLY5 on a squaring blueprint.

This blueprint shows how an all-purpose machine takes the input blueprint (from the variable Proc) and applies it to five. Note that the visual representation makes the details of the application explicit and also emphasizes how blueprints are data objects. A sample use of this blueprint is shown in Figure A.6, where the

blueprint has been stored in the variable APPLY5. Here the blueprint is being applied to a squaring blueprint.

A.3 MAKE-ADDER

MAKE-ADDER is an example of a procedure which returns a procedure as a result. In Scheme, we would express this procedure as:

```
(DEFINE (MAKE-ADDER NUM)
  (LAMBDA (X) (+ NUM X)).
```

In Grasp, the corresponding blueprint is shown in Figure A.7. To specify that it is supposed to return a blueprint, we insert the form of the blueprint we want returned into the output variable. Grasp's copying mechanism dicatates that blueprints occuring inside of blueprints are copied to the construction site in the all-purpose machine. The copied blueprint has a different template than the original, so that a totally new blueprint is created for every construction. Rather than putting the blueprint in the output variable, we could store it in an internal variable and connect a reference pipe to the output, as shown in Figure A.8.

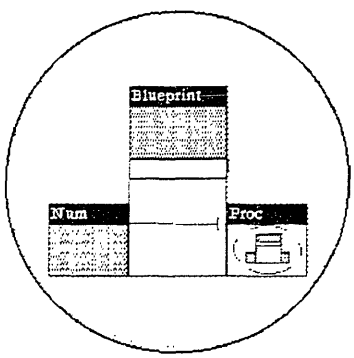


Figure A.7: A blueprint for the Grasp version of MAKE-ADDER.

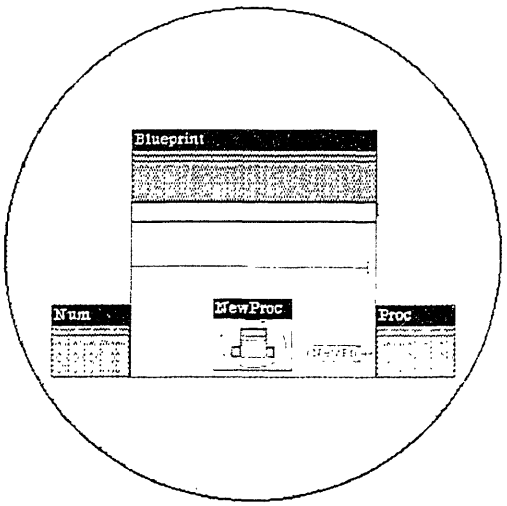


Figure A.8: Alternate version of the Grasp equivalent of MAKE-ADDER.

The structure of the procedure being returned is shown in Figure A.9. We see that it takes its first argument from the input of the MAKE-ADDER template. In structural terms, this means that when copied into a construction site, the internal procedure will refer to the input of the all-purpose machine into into whose internal structure it is put. Thus, the Num reference pipe refers to a different variable for every application of the MAKE-ADDER blueprint. Figure A.10 shows a sample use of the MAKE-ADDER blueprint, which is stored in the variable +Maker in the figure. The blueprint returned by the application of +Maker to 7 is an "add-7" blueprint - a blueprint of one argument which adds 7 to its input. By using the resulting blueprint as the blueprint for another all-purpose machine, we can see that it indeed has this effect.

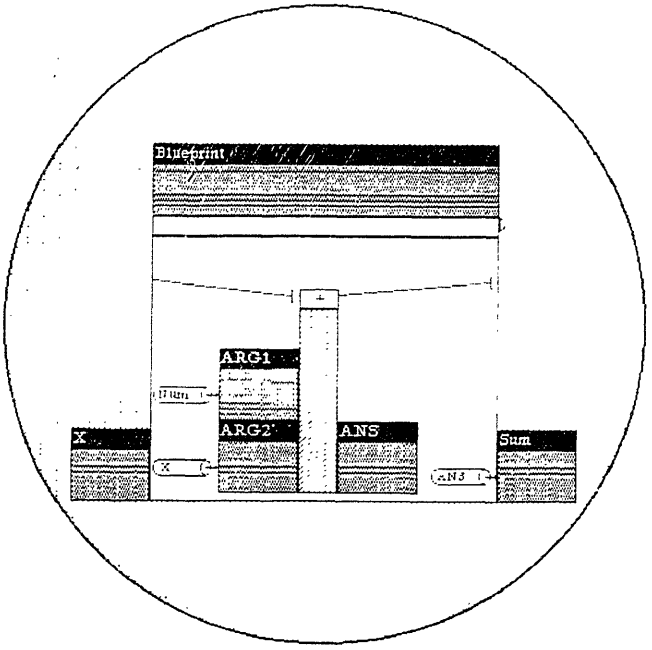


Figure A.9: The internal procedure in the MAKE-ADDER blueprint.

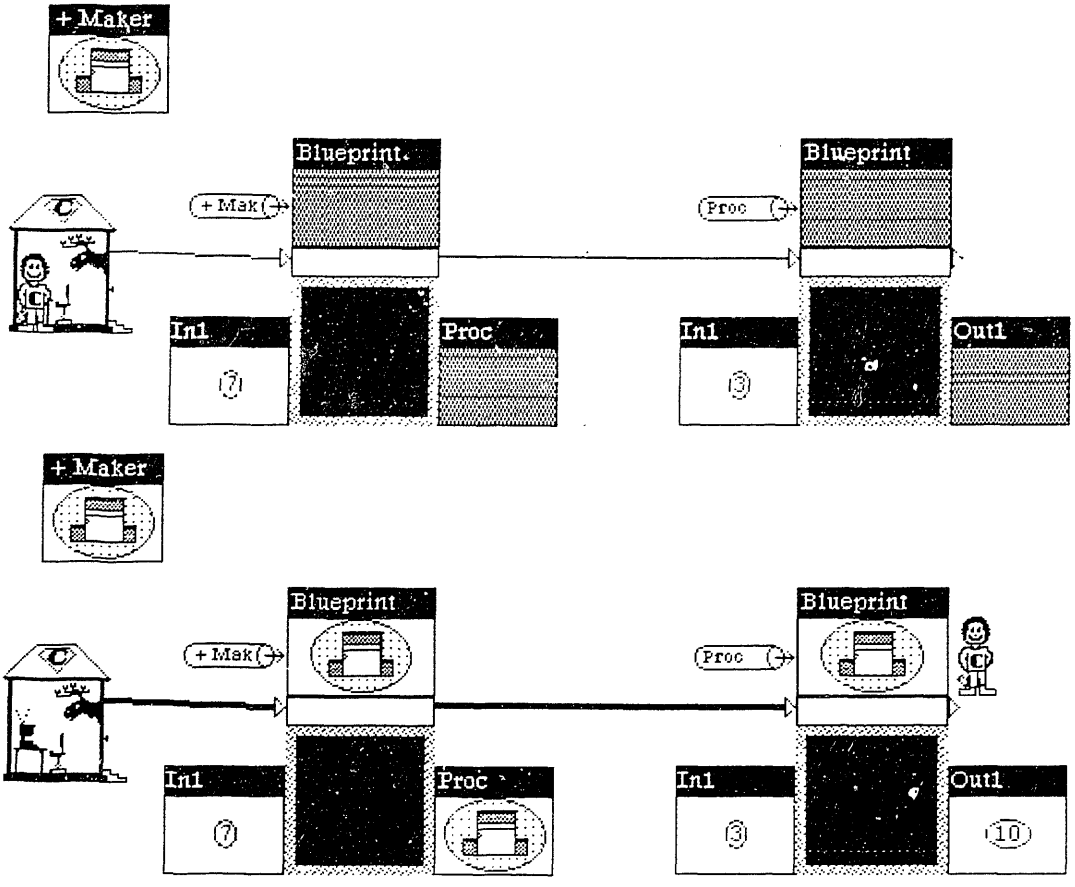


Figure A.10: A sample use of the MAKE-ADDER blueprint.

A.4 MAKE-COUNTER

As a final example, consider the Grasp analog of the Scheme MAKE-COUNTER procedure given below.

```
(DEFINE (MAKE-COUNTER)
  (LET ((COUNT 0))
    (LAMBDA ()
      (SET! COUNT (+ COUNT 1))
      COUNT)))
```

This is a procedure which creates procedures with state - each procedure it creates refers to its own local COUNT variable. In Grasp, we essentially want a blueprint which creates blueprints similar to the COUNT blueprint in section 5.4.6. The top level of such a blueprint appears in Figure A.11.

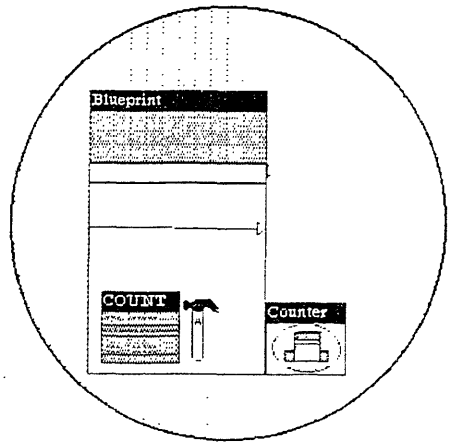


Figure A.11: The MAKE-COUNTER blueprint.

The appearance of the COUNT variable inside the template means that each construction of this blueprint will give rise to a new COUNT variable at the site of the all-purpose machine. Each construction will also create a new blueprint of the form shown in Figure A.12. Since each new blueprint mutates the variable at the site where it was created, we have the effect of blueprints with local state.

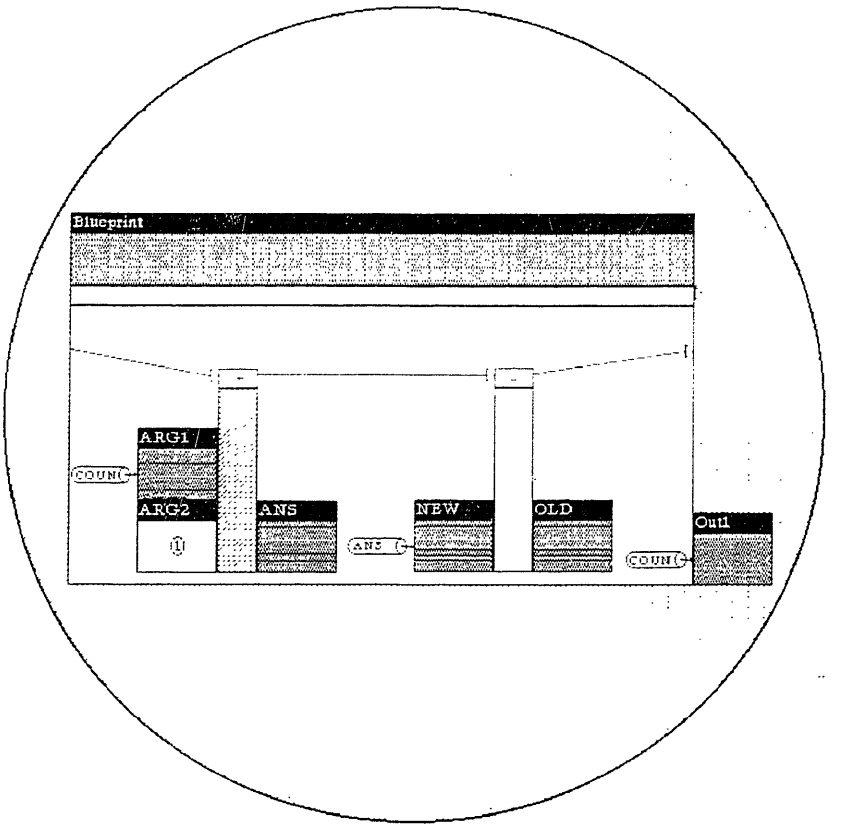


Figure A.12: The style of blueprint returned by MAKE-COUNTER

NOTES

Chapter 1

[1] Like almost all MIT courses, this one is known more widely by its number - 6.001 (six double oh one) - than its name.

[2] A first-class object is one which shares all the normal properties usually associated with a data object. For a further explanation, see the discussion of Scheme in Chapter 2.

[3] Several papers from [Gentner and Stevens 83] are relevant here, especially [diSessa 83], [McCloskey 83], [Clement 83] (motion); [Williams et al. 83] (heat flow); [Gentner 83] (electricity). [diSessa 82] is also relavent on the topic of motion.

[4] The design of Boxer, however, includes a stepper for showing the execution of a program one step at a time.

Chapter 2

[1] An argument for microworlds and an introduction to turtles are given in [Papert 80]. A more in-depth study of turtle geometry is provided by [Abelson and diSessa 82].

[2] All versions of Scheme treat procedures as first-class objects, but not all give the same rights to environments and continuations. In MIT Scheme, however, both of these are first-class.

[3] `EVAL` is a procedure which takes an expression and an environment, and evaluates the expression in the environment.

[4] A metacircular interpreter is an interpreter for a programming language which is written in the same programming language. For an example of a metacircular interpreter for Scheme, see Chpater 4 of [Abelson and Sussman 85a].

Chapter 3

[1] Actually, when I took Abelson and Sussman's course, Scheme was not yet available. We programmed in ULisp, another dialect of Lisp. Although the example associated with this note uses `DEFINE` for clarity, the actual ULisp construct was `SETQ`.

[2] A counterexample for which the model does not work involves the procedure `TEST` defined below:

```
(DEFINE (TEST) '(1 2 3))
```

Evaluating `(TEST)` gives the list `(1 2 3)` as we expect. However, if we evaluate `(SET-CAR! (TEST) 5)`, then subsequent calls to `(TEST)` will in fact return `(5 2 3)`. In my model, subsequent calls to `(TEST)` would still return `(1 2 3)`. I am endebted to Jeff Shrager for enlightening me with this example.

[3] MAPCAR is a procedure which takes as arguments a procedure and a list and returns a list of the results of applying the procedure to each element of the list.

Chapter 4

[1] Many other Lisps use a the form (DEFUN <NAME> <ARGLIST> <BODY>) for procedure definition, a syntax which does not exhibit the form of a procedure call.

[2] This is clearly true for user-defined procedures, but is far less clear for the primitive operations supported in hardware. These in some sense are hardwired into the interpreter and are much closer to activity.

[3] There is a version of LAMBDA called NAMED-LAMBDA which associates a name with a procedure and is useful for handling recursion in a proper fashion. Procedures made with NAMED-LAMBDA print out with the name; those created with plain LAMBDA print out with a number.

[4] See [Sussman 85] for a description of this method.

Chapter 6

[1] The name LAMBDA is taken from Alonzo Church's lambda calculus, upon which Lisp is loosely based.

[2] The Xerox Dorado (1132) has a three-button mouse. The Xerox Dandelion (1108) and DandeTiger (1109) are provided with two-button mice. The middle button is simulated on the two button mouse by "chording" - the process of depressing both buttons as the same time.

[3] The system, however, can effectively destroy objects by uncreating them when the controller walks backward along the computational time line.

Chapter 7

[1] PASCAL allows procedures to be passed as arguments, but not assigned to variables or returned as results.

[2] Viewing program execution as "movies" made out of a sequence of time-dependent frames is the basis of David Canfield Smith's PYGMALION system [Smith 75].

BIBLIOGRAPHY

[Abelson 86] Abelson, Harold. Personal communication, 1986.

[Abelson & Sussman 85a] Abelson, Harold, and Sussman, Gerald J., with Sussman, Julie. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw Hill, 1985.

[Abelson & Sussman 85b] Abelson, Harold, and Sussman, Gerald J. *Computation: An Introduction to Engineering Design.* Unpublished MIT paper, 1985.

[Abelson & diSessa 82] Abelson, Harold, and diSessa, Andrea. *Turtle Geometry: The Computer as a Medium For Exploring Mathematics.* MIT Press, 1982.

[Anderson *et al.* 84] Anderson, John, Farrell, Robert, and Sauers, Ron. "Learning to Program in LISP." *Cognitive Science.* 8(1) 1984. Pp. 87-129.

[Arvind & Iannucci 85] Arvind and Iannucci, Robert A. *Two Fundamental Issues in Multiprocessing: The Dataflow Solution.* MIT Computation Structures Group, Memo 226-3. August 1985.

[Bocker *et al.* 86] Bocker, Heinz-Dieter, Fischer, Gerhard, and Nieper, Helga. "The Enhancement of Understanding Through Visual Representation." *Proceedings of the CHI '86 Conference on Human Factors in Computing Systems.* Boston, Massachusetts, April 13-17, 1986. Pp. 44-50.

[Bolt 84] Bolt, Richard A. *The Human Interface: Where People and Computers Meet.* Belmont, California: Lifetime Learning Publications, 1984.

[Brown & VanLehn 80] Brown, John Seely, and VanLehn, Kurt. *Repair Theory: A Generative Theory of Bugs in Procedural Skills.* Xerox Palo Alto Research Center, Technical Report CIS-4. August 1980.

[Brown & Reiss 82] Brown, Marc H. and Reiss, Steven P. *Toward a Computer Science Environment for Poerful Personal Machines.* Brown University Department of Computer Science, Technical Report CS-83-04. December 1982.

[Burton 81] Burton, Richard R. *Diagnosing Bugs in a Simple Procedural Skill.* Xerox Palo Alto Research Center, Technical Report CIS-8. March 1981.

[Burton & Brown 79] Burton, Richard R. and Brown, John Seely. "An Investigation of Computer Coaching for Informal Learning Activities." *International Journal of Man-Machine Studies*. 11(1) January 1979. Pp. 5-24.

[Ciccarelli 84] Ciccarelli, Eugene Charles, IV. *Presentation Based User Interfaces*. MIT Artificial Intelligence Laboratory, Technical Report AI-TR-794. August 1984.

[Clinger 85] Clinger, William, editor. *The Revised Revised Report on Scheme, or, an UnCommon Lisp*. MIT Artificial Intelligence Laboratory, AI Memo 848. August 1985.

[Clement 83] Clement, John. "A Conceptual Model Discussed by Galileo and Used Intuitively by Physics Students." In Gentner, Dedre and Stevens, Albert, editors, *Mental Models*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983. Pp. 325-340.

[Curry 78] Curry, Gael Alan. *Programming by Abstract Demonstration*. University of Washington PhD. thesis, 1978.

[deKleer & Brown 83] deKleer, Johan, and Brown, John Seely. "Assumptions and Ambiguities in Mechanistic Mental Models." In Gentner, Dedre and Stevens, Albert, editors, *Mental Models*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983. Pp. 155-190.

[Dennis 75] Dennis, Jack B. *First Version of a Data Flow Procedure Language*. MIT Laboratory for Computer Science, Technical Memo TM-61. May 1975.

[Dewdney 85] Dewdney, A.K. "Computer Recreations." *Scientific American*. **253**(1) July 1985. Pp 14-19.

[diSessa 82] diSessa, Andrea. "Unlearning Aristotelian Physics: A Study of Knowledge-Based Learning." *Cognitive Science*. 6(1) 1982. Pp 37-75.

[diSessa 83] diSessa, Andrea. "Phenomenology and the Evolution of Intuition." In Gentner, Dedre and Stevens, Albert, editors, *Mental Models*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983. Pp. 15-33.

[diSessa 85a] diSessa, Andrea. "A Principled Design for an Integrated Computational Environment." *Human-Computer Interaction*. 1(1) 1986. Pp 1-87.

[diSessa 85b] diSessa, Andrea. *Knowledge in Pieces*. Unpublished paper, 1985.

[diSessa 86a] diSessa, Andrea. "Models of Computation." In Norman, D. A. and Draper, S. W., editors, *User-Centered System Design: New Perspectives on Human Computer Interaction*. Hillsdale, N.J.: Lawrence Earlbaum Associates, 1986.

[diSessa 86b] diSessa, Andrea. "Notes on the Future of Programming;Breaking the Utility Barrier." In Norman, D. A. and Draper, S. W., editors, *User-Centered System Design: New Perspectives on Human Computer Interaction.* Hillsdale, N.J.: Lawrence Earlbaum Associates, 1986.

[du Boulay *et al.* 81] du Boulay, Benedict, O'Shea, Tim, and Monk, John. "The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices." *International Journal of Man-Machine Studies.* 14(3) August 1981. Pp. 237-249.

[du Boulay & O'Shea 81] du Boulay, Benedict, and O'Shea, Tim. "Teaching Novices Programming." In Coombs, M. J. and Alty, J. L., editors, *Computing Skills and the User Interface.* Academic Press, 1981. Pp. 147-200.

[Eisenberg 85] Eisenberg, Michael Allen. *BOCHSER: An Integrated Scheme Programming System.* MIT Laboratory for Computer Science, Technical Report TR-349. October, 1985.

[Furnas 86] Furnas, George W. "Generalized Fisheye Views." *Proceedings of the CHI '86 Conference on Human Factors in Computing Systems.* Boston, Massachusetts, April 13-17, 1986. Pp. 16-23.

[Gentner & Stevens 83] Gentner, Dedre and Stevens, Albert, editors. *Mental Models.* Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983.

[Gould & Finzer 84] Gould, Laura, and Finzer, William. *Programming by Rehearsal.* Xerox Palo Alto Research Center, Technical Report SCL-84-1. May 1984.

[Glinert & Tanimoto 84] Glinert, Ephraim P., and Tanimoto, Steven L. 'PICT: An Interactive Graphical Programming Environment." *IEEE Computer.* 17(11) November 1984. Pp 7-25.

[Grafton & Ichikawa 85] Grafton, Robert B., and Ichikawa, Tadao, editors. *IEEE Computer*, Special Issue on Visual Programming. 18(8) August 1985.

[Halasz 84] Halasz, Frank. *Mental Models and Problem Solving in Using a Calculator.* Stanford PhD. thesis, 1984.

[Halbert 84] Halbert, Daniel C. *Programming by Example.* Xerox Office Systems Division, Technical Report OSD-T8402. December, 1984.

[Henderson 84] Henderson, D. Austin, Jr. Personal Communication, 1984.

[Interlisp 83]. *The Interlisp Reference Manual.* Xerox Palo Alto Research Center. October, 1983.

[Kahney 82] Kahney, Hank. *An In-Depth Study of the Cognitive Behaviour of Novice Programmers*. The Open University Human Cognition Research Laboratory, Technical Report No. 5. 1982.

[Lieberman 82] Lieberman, Henry. *Seeing What Your Programs Are Doing*. MIT Artificial Intelligence Laboratory, Memo No. 656. Februrary 1982.

[Liskov *et al.* 79] Liskov, Barbara, *et al.* CLU Reference Manual. MIT Laboratory for Computer Science, Technical Report TR-225. October 1979.

[Malone 85] Malone, Thomas W. Unpublished notes for teaching Lisp to management students in MIT Sloan School of Management course 15.560, 1985

[Malone *et al.* 86] Malone, Thomas W., Grant, Kenneth R., and Turbak, Franklyn A. "The Information Lens: An Intelligent System for Information Sharing in Organizations." *Proceedings of the CHI '86 Conference on Human Factors in Computing Systems*. Boston, Massachusetts, April 13-17, 1986. Pp. 1-8.

[Mayer 81] Mayer, Richard E. "The Psychology of How Novices Learn Computer Programming." *Computing Surveys*. 13(1) March 1981. Pp. 121-141.

[McCloskey 83] McCloskey, Michael. "Naive Theories of Motion." In Gentner, Dedre and Stevens, Albert, editors, *Mental Models*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983. Pp. 299-324.

[Myers 86] Myers, Brad A. "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy." *Proceedings of the CHI '86 Conference on Human Factors in Computing Systems*. Boston, Massachusetts, April 13-17, 1986. Pp. 59-66.

[Norman 83] Norman, Donald. "Some Observations on Mental Models." In Gentner, Dedre and Stevens, Albert, editors, *Mental Models*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983. Pp. 7-14.

[Papert 80] Papert, Seymour. *Mindstorms: Childrens, Computers, and Powerful Ideas*. New York: Basic Books, Inc., 1980

[Reiss 84] Reiss, Steven P. *Graphical Program Development with PECAN Program Development Systems*. Brown University Department of Computer Science, Technical Report CS-84-04. January 1984.

[Rovner & Henderson 69] Rovner, P. D. and Henderson, D. A., Jr. "On the Implementation of AMBIT/G: A Graphical Programming Language," *Proceedings ot the International Joint Conference on Artificial Intelligence*. Washington, D.C. May 7-9, 1969. Pp. 9-20.

[Sheil 81] Sheil, B. A. *Coping with Complexity*. Palo Alto, CA: Xerox Palo Alto Research Center, Technical Report CIS-15. April 1981.

[Shneiderman 83] Shneiderman, Ben. "Direct Manipulation: A Step Beyond Programming Languages." *IEEE Computer.* **16(8)** August 1983. Pp. 57-69.

[Smith 84] Smith, Brian Cantwell, and des Rivieres, Jim. *Interim 3-LISP Reference Manual.* Xerox Palo Alto Research Center Technical Report ISL-1. June, 1984

[Smith 75] Smith, David C. *PYGMALION: A Creative Programming Environment.* Stanford PhD. thesis (also Stanford Artificial Intelligence Laboratory, Memo AIM-260). June 1975.

[Smith *et al.* 82] Smith, David C., Irby, Charles, Kimball, Ralph, Verplank, Bill, Harslem, Eric. "Designing the Star User Interface." In Degano, Pierpalo and Sandewall, Erik, editors, *Integrated Interactive Computing Systems: Proceedings of the European Conference on Integrated Interactive Computing Systems (ECICS 82).* North Holland, 1983. Pp. 297-313.

[Soloway *et al.* 83] Soloway, Elliot, Bonar, Jeffrey, and Ehrlich, Kate. "Cognitive Strategies and Looping Constructs: An Empirical Study." *Communications of the ACM.* **26(11)** November 1983. Pp 853-860.

[Strassman 84] Strassman, Steven Henry. *Learning Lisp: The Barriers to Novice Programmers at MIT.* MIT Bachelor's Thesis. May 1984.

[Sussman 85] Sussman, Julie, with Abelson, Harold, and Sussman, Gerald Jay. *Instructor's Manual to Accompany "Structure and Interpretation of Computer Programs."* MIT Press and McGraw Hill, 1985.

[Sutherland 66] Sutherland, William R. *The On-Line Graphical Specification of Computer Procedures.* MIT PhD. thesis, 1966.

[Swann & Johnson 77] Swann, Howard, and Johnson, John. *Prof. E. McSqaured's Fantastic, Original & Highly Edifying Calculus Primer.* Los Altoc, CA: William Kaufmann, Inc. 1977.

[Turbak 86] Turbak, Franklyn. Results from a questionnaire presented to two recitation sections in the MIT Scheme course. February, 1986.

[Williams *et al.* 83] Williams, Michael, Hollan, James, and Stevens, Albert. "Human Reasoning About a Simple Physical System." In Gentner, Dedre and Stevens, Albert, editors, *Mental Models.* Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983. Pp. 131-153.

[Young 81] Young, Richard M. "The Machine inside the Machine: User's Model's of Pocket Calculators." *International Journal of Man-Machine Studies.* **15(1)** July 1981. Pp. 51-85.

[Young 83] Young, Richard M. "Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive Devices." In Gentner, Dedre and Stevens, Albert, editors, *Mental Models*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983. Pp. 35-52.