PROGRAMMING IN PICTURES

by

Johan Georg Raeder

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Science)

November 1984

Table of Contents

Abstract vii
Acknowledgements ix
Figure credits x
1. Introduction 1
1.1. Background 1
1.2. Pictures and text 3
1.2.1. What are the main features of pictures vs. text? 4
1.2.2. Why is our society biased toward text? 10
1.2.3. Why should we use pictures when programming? 15
1.3. Goals of the thesis 27
1.4. Plan of the thesis 28
2. Related work 30
2.1. Primitive program displays 30
2.2. Program development environments 37
2.2.1. Syntax-directed editing 39
2.2.2. Style of interaction 41
2.2.3. Integrated environments 47
2.2.4. Program animation 49
2.2.5. Programming-in-the-large 52
2.2.6. Programming-by-example 53
2.3. CAD/CAM 55
2.4. Office information systems 59
2.5. Human-computer interaction 63
2.6. Other systems 66
2.7. Summary 76
3. The form of pictures in programming 79
3.1. General guidelines 79
3.1.1. What aspects of programs shall we show? 82
3.1.2. How shall we display it? 89
3.1.3. Summary of program display issues 100
3.2. Key decisions 101
3.3. The computational model 103
3.4. Graphical representation 108

3.4.1. Data layout 108
3.4.2. Function layout 110
4. A system for programming in pictures 112
4.1. System overview 112
4.2. Systemwide concepts 115
4.2.1. Mouse use 115
4.2.2. Command parameter input 118
4.2.3. The scratchpad 120
4.2.4. Global commands 120
4.3. Drawing pictures 121
4.4. Defining simple types 124
4.4.1. The type template 125
4.4.2. The library scroll window 125
4.4.3. Adding and manipulating elements 128
4.4.4. Saving and restoring 134
4.5. Defining functions at the object level 135
4.5.1. The function template 135
4.5.2. The library scroll window 136
4.5.3. Typing a function 136
4.5.4. Programming a function 141
4.5.5. Inspecting a function 147
4.5.6. Executing a function 148
4.5.7. Saving and restoring 151
4.6. Defining functions at the function level 151
4.6.1. The function template 152
4.6.2. Typing a function 153
4.6.3. Programming a function 153
4.6.4. Executing a function 159
4.7. Another example 162
4.8. The typing mechanism 169
5. Programming by example 177
5.1. Defining structured types 177
5.1.1. Sequence structuring 177
5.1.2. Recursive structuring 185
5.2. Programming with structured data 195
5.2.1. Executing with structured display 195
5.2.2. Programming by example 196
5.3. An example 203

6. Discussion and conclusions **212**
6.1. Casual programming 212
6.1.1. The need for a casual programming tool 212
6.1.2. Properties of a casual programming tool 215
6.1.3. PiP is for casual programming 217
6.2. Correctness issues 218
6.2.1. Syntactical errors 219
6.2.2. Semantic errors 220
6.2.3. Logic errors 221
6.3. Software engineering issues 222
6.3.1. Design 222
6.3.2. Prototyping 224
6.3.3. Testing 225
6.3.4. Maintenance 225
6.4. User reactions 226
6.5. Further work 229
6.6. Conclusions 232
Appendix A. Summary of the FP model **234**
Appendix B. Code generation **239**
Appendix C. Implementation status **243**
References **244**

List of Figures

Figure 2-1: Nassi-Shneiderman diagram. 32
Figure 2-2: State diagram. 33
Figure 2-3: Rothon diagrams. 33
Figure 2-4: Petri net. 34
Figure 2-5: Data flow diagram. 35
Figure 2-6: Structure of our PiP system. 36
Figure 2-7: Simulation process dependencies. 37
Figure 2-8: Conditional in FGL. 68
Figure 2-9: FADT function. 70
Figure 2-10: A list splitting routine in Cardelli's notation. 71
Figure 2-11: Program execution in the PV environment. 72
Figure 2-12: Visual specification hierarchy in PegaSys. 73
Figure 2-13: PECAN execution display. 74
Figure 2-14: OMEGA program display. 76
Figure 4-1: The toolmenu (legend added). 116
Figure 4-2: The picture editor. 122
Figure 4-3: The type editor template. 126
Figure 4-4: The scroll window. 127
Figure 4-5: Creating a type. 130
Figure 4-6: A type with background illustration. 133
Figure 4-7: The object level function editor template. 137
Figure 4-8: Input and output types inserted. 139
Figure 4-9: Zooming to show type structure. 140
Figure 4-10: Programming a simple function (1). 143
Figure 4-11: Programming a simple function (2). 144
Figure 4-12: Programming a simple function (3). 145
Figure 4-13: Inspecting a function. 149
Figure 4-14: Executing a function. 150
Figure 4-15: The function level template. 154
Figure 4-16: The loop frame. 156
Figure 4-17: The conditional frame. 157
Figure 4-18: Composing a function. 160
Figure 4-19: Types for Towers of Hanoi. 163

Figure 4-20: A program for Towers of Hanoi. 165
Figure 4-21: Moving the *n-1* discs. 166
Figure 4-22: Swapping two towers. 167
Figure 4-23: Subtracting a disc. 168
Figure 4-24: Moving a disc from A to B. 170
Figure 4-25: Prompting for a picture value. 171
Figure 4-26: Animating the disc moves. 172
Figure 5-1: A sequence drawn as a tree. 178
Figure 5-2: Sequence structuring. 181
Figure 5-3: Sequence structuring. 182
Figure 5-4: Display alternatives. 186
Figure 5-5: Recursive splitting. 189
Figure 5-6: Recursive structuring: a binary tree. 191
Figure 5-7: Zooming the subtree structure. 192
Figure 5-8: Recursive subtree display. 193
Figure 5-9: Recursive structuring. 194
Figure 5-10: Types for binary search. 204
Figure 5-11: Binary search function. 205
Figure 5-12: Extracting node data. 207
Figure 5-13: Moving down to a subtree. 208
Figure 5-14: Leaf node variant of left search. 209
Figure 5-15: General variant of left search. 211

Abstract

Programming in conventional programming languages is awkward because the resulting programs are quite far from how we like to think about them, both conceptionally and representationally. By representing programs as semantically suggestive graphical images we can shorten the gap between mind and medium and thereby make programming more pleasant, efficient and less error-prone.. A rich graphical interface can also aid naive programmers by making abstract concepts concrete.

In this thesis we examine the use of pictures in programming. We point out the salient characteristics of pictures vs. text. In particular, the concreteness, random access nature, high transfer rate, namelessness, multi-dimensionality, and possibilities for animation render pictures well qualified for representing programs. We discuss in more detail the best form of a pictorial program display, and arrive at a solution based on data structures as the primary displayed aspect. We also assign other program aspects, like control and hierarchy, to picture dimensions, obtaining a unified view that allows the representation of programs as a single object instead of a series of different views. We develop techniques for reading and writing programs through

sequences of pointing actions animated on top of the data structure display, mimicking the way people informally explain programs through handwaving on data structure illustrations.

We describe a concrete implementation of our ideas about programming in pictures. The system is based on a functional programming model. It allows the creation of functions where the data types of input and output objects are illustrated by the user. The user inputs pictures related to the application domain and inserts them in an algorithmic framework supplied by the system. We show how our style of programming leads us to a version of programming by example.

Finally, we examine what audience would benefit the most from a pictorial programming system, and what kind of applications these people would be interested in. We define the term "casual programming" as the creation of small programs by naive or casual users, and identify this as a useful application area of programming in pictures.

Acknowledgements

I wish to express my sincerest thanks to my thesis advisor, Lawrence Flon. His support and sound critique during the last few years have been invaluable for the progress of this thesis. Above all, I am thankful to him for believing in my ideas and letting me go ahead with the project. Thanks, Larry!

Many other people showed interest and offered encouragement. My committee members - Les Gasser, Ellis Horowitz, Alice Parker, and Walt Scacchi - represented an outstanding variety of expertise that I perhaps should have taken more advantage of. Thierry Paradan gave many valuable comments, carefully reviewed a paper based on this thesis, and is responsible for this immortal initial advice: "The hard part will be to design the aliens!" Peter Vanderbilt was instrumental in bringing to life our depressingly malfunctioning computer hardware.

During my years in Los Angeles I have had the privilege to count as my friends a collection of remarkable people from most corners of the world. I am deeply indebted to all of them for enriching my days in so many ways. Finally, my appreciation goes to a woman called Shahrzade for turning a time of pressure and hard work into quite a pleasant chapter in my life.

Figure credits

The following figures were borrowed from the references given:

Fig. 2-1	[Ng 79], p 241
Fig. 2-2	[Wulf 81], p 17
Fig. 2-3	[Brown 83], p 19-20
Fig. 2-4	[Baer 80], p 71
Fig. 2-5	[Davis 82], p 27
Fig. 2-7	[Berry 85]
Fig. 2-8	[Keller 81], p 158
Fig. 2-9	[Amble 83], p 18
Fig. 2-10	[Cardelli 83], p 146
Fig. 2-11	[Kramlich 83], p 144
Fig. 2-13	[Reiss 84], p 41
Fig. 2-14	[Powell 83], p 18

Chapter One

Introduction

1.1. Background

After decades of being concerned primarily with highly technical issues, like operating systems and compilers, mainstream computer science finally seems to direct substantial effort outward, and consider how computer systems are to be designed in order to accommodate the human user of the systems. A certain maturity in traditional fields has probably contributed to this development, but the main reason for the upsurge of interest in human-computer interaction is most certainly the widespread use of microcomputers: When millions of people use computer technology every day in their work and at home, the way computers appear to their users becomes important to the well-being and productivity of a large segment of our society.

One approach to building better user interfaces is based on artificial intelligence. The AI community has done extensive work on techniques that allow humans to communicate with computers via natural language. Currently, much work is also being done on expert systems, that simplify the dialogue by being capable of deductions within an application domain.

Another approach, which concerns us here, makes use of current technological advances that have made high-resolution bit-mapped graphics available on inexpensive microcomputers. Traditional computer systems show very little of their internal state to the user. This means that the user must mentally store and manipulate a large, complex, and abstract structure without getting more than hints from the computer whether his/her conjectures about the state are correct or not. Screen editors, spreadsheets, and electronic desktops are recent developments that attempt to remedy this situation by showing as much of the state as possible on the computer screen. High-resolution graphics allows more detailed and meaningful pictures to be drawn, thereby increasing the information content of the screen.

An additional aspect of the new technology is its interactive nature. Static graphics has traditions in fields like the arts and advertising, and dynamic graphics is being used in video and television. Dynamic *interaction*, however, has not been widely explored, with the exception of video games (which, since the input is not graphical, does not strictly qualify). Thus, there seems to exist a vast potential for sophisticated techniques of communication between humans and computers, and much current research is involved with finding systematic ways to exploit this.

In this thesis, we will examine how we can utilize interactive graphics to

improve computer programming. We will do this by developing a concrete system for programming in pictures. Current programming languages and environments are usually rather painful to use, mostly because they have come a very short way in adapting to the human user rather than to the technology they are based on. This lack of adaptation exists both at the representational (i.e. the graphical layout) and at the conceptual (i.e. the computational model) level. Research in, for example, functional and logic programming tries to remedy the latter. Concerning the former, we argue in this thesis that it seems worthwhile to attempt to make programming more pleasant by bringing more of the abstract properties of programs out on the computer screen.

1.2. Pictures and text

Before we attempt to design a picture-oriented programming system, we have to understand pictures and how they relate to programming. In this section we examine pictures to see what inherent features set them apart from textual forms of representation. Next, we show how these features match key aspects of programming, and therefore how pictures provide a better medium for representing programs.

1.2.1. What are the main features of pictures vs. text?

Pictures and text are the most prominent carriers of non-numeric information in our society, and this has been the case ever since the invention of writing 3500 years ago. Given these two media, we are constantly faced with the decision of which one to express ourselves in. We usually make this tradeoff without very much consideration, sticking to our habits and established conventions rather than making a conscious choice of the best strategy. Before we start investigating how computer programming relates to these media, it is useful to sum up the most important characteristics of pictures vs. text.

Random vs. sequential access. Exactly how we "access" a picture is hard to determine, but the parsing process is in any case so fast that it seems we have instant, random access to any part of the picture we are looking at. In fact, it is completely effortless for us to shift our attention between different parts of the picture and to alternate between detailed and overall views. Recognition of known features is usually also almost instantaneous. Text, on the other hand, is strictly sequential. Partitioning and good headlines can to some extent speed up the search for a particular piece of text, but at some level we always have to resort to sequential scanning. Also, it is usually hard to gain an overview of the matter described without first

reading the detailed representation. On the other hand, the sequential order of the alphabet makes it possible to sort and thereby easily search through large amounts of written information. It is very hard to sort pictures!

Another, related feature of pictures is that they provide several means for drawing attention to their individual parts. Colors or shades, prominent shapes, and geometrical composition can all be used for this purpose. When we use italics, bold face, capitals, or other forms of textual highlighting we are really borrowing from the pictorial domain to make text access less sequential.

Dimensions of expression. Text is sequential, and, moreover, it allows for only one kind of representation of information: a one-dimensional stream of words. Pictures are much richer in this respect. Not only do they provide two (or three, or even more) dimensions along which to lay out the information, but there is also a host of other properties borrowed from the physical world that we can utilize, like shape, size, color, texture, direction, and distance. Since the language for representation is richer, the encoding of each piece of information can be more compact in pictures, just as a number expressed in decimal notation generally contains fewer digits than when expressed in binary. The encoding and decoding processes needed for writing and reading should therefore potentially be much faster for pictures than for text.

Transfer rate. The above two aspects of pictures and text explain the fact that pictures generally can be read with a significantly higher information transfer rate than text: both the access and the decoding is more efficient. This is witnessed by the increased use of pictures in places where a lot of information has to be transferred rapidly, from advertisements to operating instructions (e.g. for telephones, [Karhan 83]), traffic signs to control system displays. Of course, it is possible to express something pictorially in such a way that no one can understand it, just as the advertising industry has shown that the proper words displayed in the correct way can have a tremendous effect (how much of this is graphics and how much text?), but the fact nevertheless remains that we are experts with pictures rather than text. Remember that our sensory system is set up for *real time* image processing, whereas text processing is a recent invention by humans.

Concrete vs. abstract. Pictures are direct descendants of the images we see around us every day, images representing concrete objects. Text mirrors our thoughts, detached from any physical reality. Pictures therefore appear concrete, regardless of how abstract the ideas they purport to illustrate may be. This makes them easy to think in terms of, since much of the apparatus we have to deal with our surroundings can be brought to bear on any picture. For example, our capabilities to judge sizes, shapes, distances, etc. can be

directly applied. Text, on the other hand, is an artificial invention, and so it is not tied to any existing framework. But this is also its strength: Textual abstraction allows us to describe concepts that are not easy to explain with bones and pebbles.

Compactness and preciseness. These issues are completely dependent on application. Even though we concluded that pictures yield more compact encoding, and a picture is supposed to be worth a thousand words (how many words do we need to describe a human face?), it is not hard to come up with examples where text is superior (how many pictures do we need to describe the word "culture"?). Textual abstraction is an immensely powerful tool that contributes at least as much to compactness and preciseness of textual information as richness of language and metaphorical content does for pictures.

Language independence. An interesting aspect of pictures is that they are not based on language, and hence are independent of it. Ideally, pictures should therefore be understood in any country without need for translation. This has been taken advantage of in many cases, e.g. European traffic signs, garment care labels and informative signs in airports. For products sold all over the world, textless icons in the user interface can save expensive multi-version manufacturing (for example of microcomputers, [Williams 84]). One

has to be careful not to include culturally dependent notions in the pictures, though.

Other issues. Pictures have more properties that are of interest to us here, and that we will say more about in connection with programming later:

- Pictures do not need to name their objects since these are present and can stand for themselves. Text must always use names and can therefore only refer to objects indirectly. Thus, pictures provide windows into the real world, whereas text can only point to the real world.

- Pictures can be set in motion to show dynamics directly. Text can only describe it.

- Text forces us to create our own mental images to grasp what we read. Pictures give us the images for free.

- Both text and pictures provide a medium for aesthetic expression. But pictures are a more easily accessible and more versatile medium, at least if we are to judge from our surroundings, the physical appearance of our society, where graphical designs flourish, but text is only trivially used.

The above characteristics of pictures vs. text should already tell us something quite clearly: When there is much interaction between representation and the mind, pictures appear to be the superior medium. In particular, the high information transfer rate, richness of language and random access nature of pictures contribute to a more effortless interchange. Text is better suited for sequential operations, like reading a novel or telling a story. Computer programming *is* a highly interactive affair, and it could therefore benefit greatly from a more graphical style of presentation.

But programming is not the only area that has been biased toward text. From news articles to shopping lists, textbooks to recipes, wouldn't the presentation be more attractive and the matter easier to absorb if the pictorial content were increased? If we look at present material, the answer is usually positive. A well-illustrated article is much more appealing than a dry piece of text. (And, we usually look at the pictures before we read the text, don't we?!) To the extent that pictures are used they are quite often not allowed to talk for themselves. They are demoted to illustrations, with an accompanying text stating essentially what can be deduced from the pictures as well.

1.2.2. Why is our society biased toward text?

With television and new printing techniques the amount of pictures that we see daily has increased markedly in the past. There are, however, some reasons why our society is still a text-biased one, reasons that may disappear in the near future due to the computer medium.

Technological base. Starting at the technical end, we observe one reason why text has enjoyed such popularity: Text is easy to represent and distribute. The situation is parallel to analog vs. digital representation of electrical signals. Text is largely independent of variations in the medium (fonts, colors, print quality, layout of the pages, etc.) whereas a picture generally is sensitive to such changes. As computer professionals we also know how much harder it is to implement a good picture handling system than one which handles text, and how much easier it is to define standards for text representation than for pictures. The reason for this is of course that text is composed of a few atomic symbols, just as digital electronics use a few standard voltages. Text is also a sequential medium, making it more straightforward to handle than two-dimensional, let alone three-dimensional pictures, since much of our technology is based on sequential machinery and procedures. Today, cheap microelectronics seems finally to have made us able to handle, display, and distribute pictures more or less as easily as text.

Education. When asked why we would choose to express ourselves through text rather than pictures, most of us would probably answer that it is much easier for us, and that we have no idea of how to give a good graphical presentation. But this does not necessarily imply that pictures are harder *per se*. We all spent about a decade at school perfecting our writing skills, but how much was said about graphic design and visual communication? When we write, we can therefore rely on a lot of experience, as well as a sound base of rules on syntax, grammar, form and style, whereas when it comes to graphics, most of us are really illiterate. That is why even simple drawing work looks so much better when done by professionals. Much of what we think requires artistic talent could probably be taught, just as we can learn to improve our writing style. Modern computer-based graphics editors also provide support that can substitute for much of the virtuosity needed to handle a paintbrush. As the image becomes a more easily accessible medium, education in related areas needs to be improved in order to enable people to utilize it better.

Prestige. It is quite interesting to observe that, even though we take it for granted that people in the industrialized world can read and write, there is still a fair amount of prestige associated with textual presentation. A description in terms of pictures easily conveys a feeling of being too simple, as

if the pictures should suggest that the description is meant for people who cannot even read. It also evokes thoughts about pictorial product descriptions, "as easy as 1-2-3", and about comics and cartoons, some of which admittedly belong to the lower end of the intellectual scale. Text is considered more "formal", even though it is usually not hard to find simplistic and imprecisely written language, and there are no rules against formalization of figures. It is not that many years ago that the *London Times* for the first time printed a photograph on its front page, insulting the intellect of some of its readers. The prestige gap that traditionally has existed between journalists and photographers, and authors and illustrators, is also a case in point here. Today, as technology permits simpler and cheaper dissemination of pictures, and as the effectiveness of pictures as a communication medium is recognized, these attitudes are changing rapidly. In everything from photojournalism to computer interfaces, the value of textless communication is appreciated.

So, are we moving toward a textless society? Certainly, not. There are good reasons why text is the prevalent carrier of graphically encoded information between humans. Text is a direct representation of language, and language is the key to the advancement of human perception. Speech has been developed into such a powerful tool that it is usually more convenient for us simply to

say what we mean than to show it by means of drawing, acting, or actually doing what we have in mind. This fluency in language is transferred to the graphical domain and causes text to dominate over pictures. We are here only arguing that there are areas where an increased use of pictures seems to be appropriate.

We mentioned the prestige myth associated with the use of pictures. There are other problems with pictures that turn out not to be real. For example, one obstacle we encounter when we want to express something through a picture is the lack of predefined symbols. It is very hard to understand a description based on more than trivial depictions of well-known physical objects. But this is just a problem of consensus rather than a deficiency of pictures. Earlier civilizations have made perfect sense out of cryptic pictures, and, if we look around us, we can still see lots of pictures that are not only well defined, they also stand for quite complex structures (chess pieces, traffic signs, company logos, etc.). With graphical command symbols, called *icons*, attaching meaning to pictures has already become a part of application programming.

One of the ways language has achieved such great power and versatility is through the idea of abstraction. To let one word stand for a whole set of ideas and concepts that may need lengthy explanations to be defined (if they

are definable at all) is something that can seem relatively harder to achieve in pictures, particularly when the domain of discourse moves away from what we see around us. We have all used various diagrammatic techniques to represent abstract concepts like system structures, but we always run the risk of indicating more meaning than we intend to. Pictures are rooted in real life; we are constantly surrounded by them (some created by humans, some not). This makes pictures so replete with metaphorical content that it is always hard to use them without saying more than we had in mind. For example, in a diagram consisting of boxes and other shapes connected via lines, does it make a difference whether two boxes are beside each other or one above the other? Does shape have significance; for example, are objects of square shape easier to combine with each other than circular ones?

On the other hand, abstraction quite often uses metaphor to achieve its effect. Given the metaphorical richness of pictures it should therefore be possible to find good metaphors that allow us to construct the powerful abstractions we need.[1]

The concreteness and metaphorical richness of pictures can be seen as their great potential, and it is the topic of this thesis to show how this can be

---

[1]Apple Computer's trashcan for document and file disposal is already a famous example.

exploited to improve computer programming. It is still easier for most people to think in concrete terms, and if we can control the power of metaphorical associations, we can create systems that are very efficient and smooth to use.

1.2.3. Why should we use pictures when programming?

In the previous section we concluded that, in an interaction-rich activity like programming, the high transfer rate, richness and random access nature of pictures give the graphical medium strong potential advantages over text. In this section we will expand further on why pictures are suitable for programming.

Mental images. When we think about a computer program, or any other abstract structure, most of us form some kind of mental images of the structures we build or try to understand. Interviews with good programmers reveal that they support their creativity with rather vivid images of program statics and dynamics [Molzberger 83]. Many of us see other abstract concepts, like time or music, as pictures of some sort. It seems that our roots in a physical world demands physical interpretations of all our thoughts. This is because we learn by extending known structures through metaphors [Carroll 82], and the physical world contributes the largest base of good, consistent structures we can use. We see this in our languages, where abstractions often are expressed by metaphorically transforming the meaning of words of

concrete origin. The pictures in our mind's eye are in the same way graphical metaphors explaining abstract concepts in terms of concrete allegories.

The pictures in our mind are difficult to capture, though. They are usually quite imprecise, just flashing by in an instant, allowing us to use some aspect of them that helps to shed light on our thoughts. Bringing the pictures out in the open is neither possible, nor is it certain that it would be helpful to scrutinize them in too much detail. Still, isn't it a worthwhile challenge to try to illustrate our work with pictures that evoke some of the insight that we get from our own imagination?

Concrete pictures. We argued previously that pictures convey a degree of concreteness, whereas text is a good vehicle for abstraction. Computer programming is an intellectual exercise, building abstract structures based on an equally abstract computational model. Doesn't it seem paradoxical, then, to try to use pictures in this connection? Maybe so, but the paradox can still make obvious sense: We do need and we do use pictures to support our abstract thinking. Different people may use different images to explain the same concepts. It is not clear whether different pictures are optimal for different people, though. In fact, another person's view can sometimes contribute a better way to see things. Some images seem to be generally acceptable as useful illustrations. If we manage to capture the "right" kind

of images for a certain abstract domain this will help not only the specialist to formulate and communicate thoughts faster and better, but in particular it will aid the novice who is groping for concrete handles into unfamiliar matter.

The reason why professional programmers have been able to interact successfully with complex computer systems through primitive interfaces is largely that they, after several years of daily experience, have acquired a clear, fairly complete model of the system in their minds. The naive or casual programmer, however, does not have the opportunity or interest to accumulate this abstract knowledge. For such users we therefore need to present the model along with the interface, and this is probably best done via some graphical representation. Learning computer programming is best done by building a concrete model of the computer [Mayer 81]. In a way that text can never manage, a concrete picture can unveil abstract structure and in an instant bring understanding both individually and between people. Computer programming is one abstract discipline where we can use pictures, not in spite of their concreteness, but because it reduces the abstractness.

Random, fast access pictures. A program is a complex abstract object. There are many components, interrelated in many different ways and with a number of different attributes. When we reason about a program, we think about all these aspects in a fairly unstructured, random fashion, our thoughts

flowing freely over large parts of the program. We therefore need a representation that not only is capable of expressing this multitude of notions, but that also provides us with direct, effortless access to program information. Text can, with the aid of textual abstraction, represent anything, but it seems that text, in its sequential, encoded form does not provide random access, nor is it usually effortless to decode the symbolic code. Pictures, on the other hand, have a wonderful two-dimensional, random-access nature. It is also a medium so rich that there is room for direct representation of a large amount of concepts with a very shallow level of encoding. If we manage to pin down all our programming concepts in a graphically obvious way, we can therefore obtain a representation that supports our mind with constructive metaphors rather than presenting us with obstacles that have to be overcome.

Pictures of the real world. Programs may be abstract, but in most cases they actually deal with aspects of the real world. Whether it is a home accounting package or an airline booking system, a program is a model of a part of reality, or at least it builds a computational structure on top of it. Reality is usually rich in pictures of various kinds, and it is therefore natural to attempt to include some of these in our programs to help explain them. But this is possible and convenient only if we have a graphical framework for expressing programs. It should be up to the programmer to decide what

pictures from the real world and what pictures from the programming world are useful in the description of a particular program, and only a general tool providing complete freedom for creative graphical expression of programs can support this.

When we open up the world of programming to elements from the real world, we lose mathematical preciseness. We can no longer hope to find a unique, canonical, representation of a program, since two programs that are computationally equivalent can have different real world semantics. (A given program still has a unique semantics, of course!) But if the programs are different as seen from the programmer, why shouldn't the representation reflect this? In a sense, we are capturing part of the thought process that lead to the program in addition to the program itself, as suggested by others (Martin-Lof, p 15 in [Karlsson 82]; [Krueger 83], pp 186-187). A free form representation should also inspire some creativity and encourage the programmer to put his/her "personal touch" on the program. We will shortly expand further on the benefits of this in a paragraph on aesthetics.

Pictures without names. The idea of naming is a marvelous invention that allows *indirect* reference of objects, and is also a basis for abstraction. The indirection enables us to refer to objects that are not physically present, either because they are out of our immediate view, or because they do not

exist physically. Objects that are physically present can alternatively be referenced *directly* through pointing, touching, etc. Naming may be a powerful concept, but it nevertheless introduces a second level of reference that can be an obstacle from time to time. How often do we find ourselves uttering something like "you know, that ... er ... thing" and then waving our hands as if we were creating a picture of the object, or pointing in the direction of its location. When we know which object we have in mind, we can point at it without any effort if it is present. The name of an object, however, is an artificial attribute that requires mental activity to find.

In a program written in a conventional, textual programming language, our *only* way to refer to objects is by name. This means that *all* our objects are referenced indirectly and are, in fact, invisible to the programmer. (It is quite tempting to call the traditional textual approach to programming "Blind Programming", as opposed to Programming in Pictures.) This has the effect that all operations the programs perform are "covert actions" performed behind the scenes. We know they are performed, but we only represent them implicitly, and the programmer has to do the mapping mentally.

There are also other problems with not having any means for direct object reference. We are forced to attach names to *all* existing objects, whether interesting or not. In large programs this can really become a burden,

especially since most names have to be unique, and we often find ourselves inventing various naming conventions that will help us generate a large number of more or less meaningful names in a fairly mechanical fashion. A related problem exists in VLSI-CAD for off-sheet connections [Maling 81]. The typing of names also introduces many possibilities for error in referring to a specific object. When we point, we very rarely point to the wrong object!

Pictures lack the indirection that names have. Of course, one could claim that a picture is no less a label for an object than a name is, the only difference is that it is graphical. This is not entirely valid, though. Since we are surrounded by a real world full of pictures and most of us have an eye that parses these pictures quickly, a picture is rather perceived as a window into a piece of reality that otherwise would be outside our view at the moment. Hence, a picture of an object can be identified with the object itself in a way that a name cannot. When we program in pictures we therefore introduce a concreteness we have not enjoyed before. The pictures of our data, procedures, etc. become the constructs they depict themselves. The programmer is relieved from constantly having to bridge the gap to the invisible objects referred to by names. They are rather put in his/her hands, so to speak.

Pictures really remove most of the need for naming. If all persons and other

objects we cared about were occupying one room, we wouldn't need names, we could just point. When all our program objects are visible as "real" objects on a screen that we can point at, we likewise do not need names for them. The names become auxiliary, optional attributes, and several objects may have the same name.

Animated pictures. A program is a dynamic object. To study the details of a program we really have to see it in operation. Hence, one could argue that the depiction of a program should have a dynamic, or animation, aspect to it. But for this we need a dynamic medium. Text is by birth a static medium, being carved in clay or printed on paper. Thus, if we attempt to impose a dynamic component on it, this will necessarily have to be a rather artificial extension. For example, a cursor that jumps line by line, following the execution, can only capture the control structure of the program. It has to be augmented by a depiction of how the corresponding data are transformed. But this is drawing pictures of the data, escaping outside the text domain.

Pictures, on the other hand, are everywhere around us, full of dynamics. Taking a pictorial representation of programs as a starting point, it should therefore be easier to find good dynamic transformations of the pictures, that will exhibit the program dynamics in a more natural and coherent way than text.

Metaphorically rich pictures. Since pictures are interpreted as windows into the real world, they automatically provide a huge base of graphical metaphors. Along with this comes our ability to reason and make judgments in terms of these metaphors. We can utilize this to make it easier to think about programs, by attaching semantic significance to graphical relationships concerning shape, size, distance, etc. For example, it seems reasonable to require that programs that are similar in function also should *look* quite similar, and, conversely, that different programs should be easy to distinguish. This is definitely not the case with conventional program text, where the only distinguishing feature is the number of nested clauses. By the same token, we would like certain features, like parallelism, to stand out well. In general, "good" features (time/space efficiency, ease of understanding and maintaining) should make a program pretty, whereas "bad" features should make it ugly and unpleasant to look at. If there is an error, the program should ideally not "look right", as if it were out of balance somehow. These goals are very hard to achieve, but ultimately we will have to build this kind of mechanism into our programming systems if we want to make full use of an important part of people's reasoning power. This can only be achieved through pictures.

Aesthetic pictures. The aesthetic content of programs is an issue that

easily can seem rather controversial from within a field as technical as computer science, but that nevertheless has great importance if we try to relate to much of the real world around us. Purists might claim that the only real aesthetics a program possibly can contain is the mathematical beauty expressed in a compact, elegant algorithm. Many people cannot appreciate such abstract aesthetics, however, but this is the only form of aesthetics that a textual representation of programs can convey. If we look around us in society there are so many other ways people get aesthetically stimulated: music, poetry and literature, *haute cuisine*, and a plethora of graphical and plastic media, like architecture and city planning, ads and commercials, product design, and printed material. We realize particularly well how extensive this contribution is to the look of our society when we visit other countries with other aesthetic styles (which, if we look closely, can be surprisingly different, from huge buildings even down to the design of text fonts!). This texture of society influences how we feel about our surroundings, and it provides a collection of media for creative exploration.

The computer is a completely general tool that will have to fit into numerous slots in this fabric. It will therefore also have to subsume and contribute to the numerous aesthetic media. Thus, in our programs, we have to look for ways to express other kinds of aesthetics than the purely algorithmic one.

This work is primarily oriented toward graphics, so we will here only explore how to increase the pictorial content of programs. This is an important part, though. No other aesthetic domain influences our daily environment more than the graphical one.

How can the use of pictures in programs help more people enjoy their creation and use? Few things bring such satisfaction and can contribute to one's self-esteem in such a degree as when one is given the opportunity to unfold one's creative abilities. Of course, the amount of creative talent varies tremendously from person to person, and this has had the effect that people with only an average amount of talent have not pursued it at all. This is due to a fear of being compared to the masters, whose works are given ample exposure by the modern techniques of cultural dissemination, but also because it is usually associated with extra time and effort to contribute anything beyond the pragmatical minimum. It *is* a major undertaking to find drawing paper and special pencils and sit down to create something. Moreover, it becomes a rather meaningless activity if it is not attached to anything, if it doesn't have a particular purpose. We are so used to everything we do having some rational value that can be measured in dollars or seconds, that exploring our own imagination becomes an unnecessary luxury to many people. In order to put people into a position where they will naturally

explore their creative abilities, it has to be effortless, almost automatic, for them to do this. Creative self-expression has to be part of our daily routines in a much higher degree than is presently experienced by most people.

We believe that the success of the computerization of society depends in part on the degree to which we can utilize this new medium to weave creativity into daily life, at least if we define success as the increased well-being of people. The electronic revolution presents us with the unprecedented opportunity to design in detail a large portion of the environment and routines for work and private life. We should use this to build surroundings that contain the freedom needed for unconstrained creative exploration, yet provide the support necessary to do the job in question, thus helping people enjoy their own and others' personal contribution. Furthermore, we should give the worker the opportunity to apply his or her abilities to redesign the workplace according to personal taste. Opening up for spiritual and emotional expression along with the pragmatic and logical could enrich the individual's job experience and make the computer a tool for self-realization rather than a technological threat to the quality of our lives. After all, dehumanization of society is a result of decisions rather than technology itself ([Krueger 83], p 53).

Of course, one might fear that activities that are not strictly productive will

take the upper hand and reduce the workers' efficiency, but this need not be the case. Firstly, it may not really be a question of extra activities, but of a different way to do things, not necessarily less efficient, or of filling in spare moments. Secondly, a work environment that inspires people to have fun in a creative way may establish an enthusiastic mood that in itself may increase overall performance. This is of course rather speculative, but it does not seem unlikely that technology in this way can improve the creativity level, well-being and self-esteem of the workforce.

Part of the rationale for programming in pictures is to change the programming activity in this spirit, making it more pleasant and accessible for a wider audience. In doing so, we may in the process help more people discover the abstract beauty of programs as well. Whether we will be able to improve the real efficiency of programming or not, we will still have gained something if we can improve the *subjective* efficiency of programming, as seen from the individual programmer.

1.3. Goals of the thesis

The primary goal of this thesis is to make a strong argument for the case of using pictures to represent programs. We will do this by examining pictures to some detail, identifying the reasons why they provide a good medium for expressing programs, and what form pictures of programs should take. The

main contribution, however, will be the design and implementation of a concrete system that illustrates our ideas about graphical programming. Through our implementation we will try to further locate advantages and problems with programming in pictures. Specifically, we are interested in determining the class of users most likely to benefit from our approach, and the kind of programs our system is useful for. It is our view that in a field as applied and user-oriented as this one, the best way to make progress is to implement our ideas, draw experience from our experiments, consolidate what we learn, get new ideas, and start over.

1.4. Plan of the thesis

To relate our work to relevant parts of computer science, we briefly summarize in chapter 2 some interesting developments in other fields. A characteristic of our topic is that it touches a quite large number of areas within computer science (and some outside it, too).

We next continue where the informal thoughts about pictures in this introduction left off, and in chapter 3 discuss in detail what options we have for representing programs as pictures. We then make some key decisions with respect to our concrete implementation.

Chapter 4 introduces and describes our implemented system proper. The

presentation is rather informal, focusing on the interesting and novel aspects of the system, and interspersed with justifications of our design decisions.

Some advanced features of the system not covered in chapter 4 are dealt with in chapter 5. Here, we point out the connection between pictures and programming by example, and show how our system supports this.

Finally, chapter 6 discusses some remaining issues related to our design. In particular, we identify a class of programming for which a system like ours seems to provide an excellent tool. We call this class ***casual programming***, since it involves programming of small programs by people who do not spend a significant part of their time programming.

Chapter Two
Related work

There is really not much previous work in the area of interactive, graphical program representation. A few systems have been designed with more or less similar goals to our own, but few of these have been very innovative. On the other hand, designing a programming system raises a whole family of issues related to the computer programming activity, ranging from the purely technical, via human concerns, to the philosophical. We must therefore take a look at recent developments within a number of fields, all of which can demonstrate important results that have a bearing on our work. In this chapter we will highlight interesting research in the most important related fields and comment on how we can incorporate these results into our own work. We will also here give a brief account of other, similar projects.

2.1. Primitive program displays

Using pictures in connection with programming is certainly not new. Pictures of various kinds have been used, both formally and informally, as an *aid* for programmers, illuminating one or more aspects of the program.

The best known graphical programming aid is probably the *flowchart.* Flowcharts were developed as a tool for assembly language programmers. Assembly programming allows completely unstructured control transfers, and the data often consist of an uninteresting global pool of storage. Flowcharts match this kind of programming very well, in that they clarify the intricate control structure graphically, while leaving the trivial data structure undescribed. For present-day structured programming, however, disciplined control constructs, scoping, data types and non-trivial data structures render flowcharts emphatically obsolete as a general graphical tool. That they are still being used in education and industry testifies to the severe lack of good, systematic graphical aids supporting current programming methodology.

Structured charts, or Nassi-Shneiderman diagrams (fig. 2-1), impose structure on program control, and coupled with a syntax-directed editor correspond well to how we would like to see our programs built systematically. Still, data are all but ignored graphically, and the pictures used to display control structure are rather uninteresting and perhaps convey more information about program component breakdown than a feeling for how the control flows.

State diagrams and *Augmented Transition Networks* (fig. 2-2) can only be used for simple automaton-type program pieces, but for these they show very

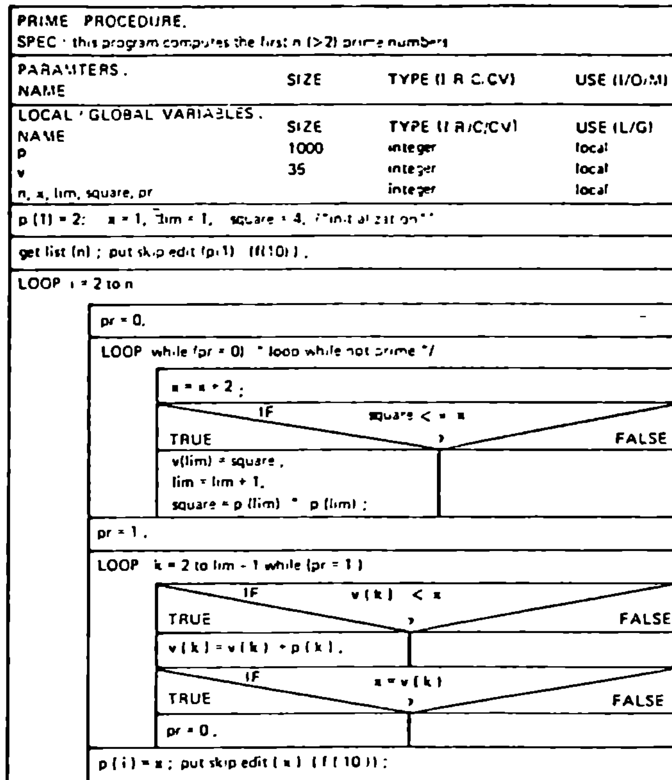


Figure 2-1: Nassi-Shneiderman diagram.

clearly how input is parsed and output generated. They are related to flowcharts and suffer from the same lack of support for data and control structure. *Rothon diagrams* (fig. 2-3) are another flowchart derivative that manages to enforce some structure on control [Brown 83]. *Petri nets*, often used for hardware descriptions (fig. 2-4), formally show how control flows through a network via a token mechanism.

The diagrams mentioned so far all describe program *control flow* in some way. One primary feature of control flow diagrams, especially node-arc based

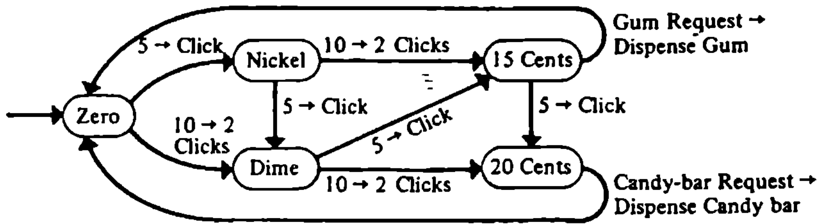


Figure 2-2: State diagram.

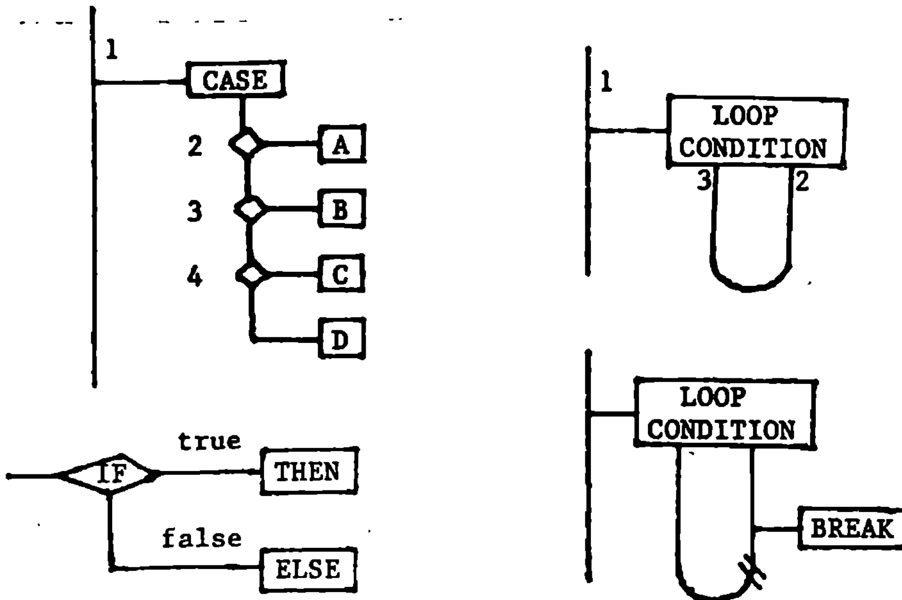


Figure 2-3: Rothon diagrams.

drawings like flowcharts, state diagrams, etc., is that they are excellent for animation. A system that "executes" a diagram by successively highlighting nodes as the computation proceeds provides great insight into the control

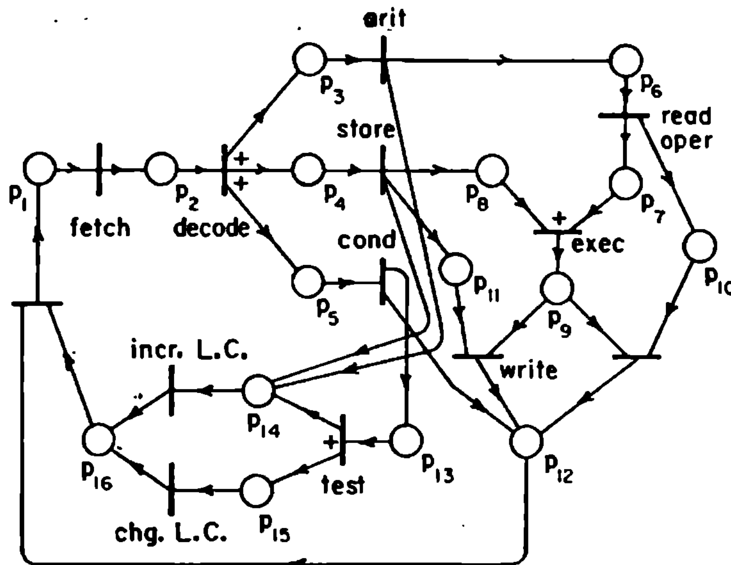


Figure 2-4: Petri net.

mechanism of an algorithm. In addition, flow graphs can show concurrency quite dramatically as multiple "foci of control" move around the network. For certain types of software, like control systems, control is the major aspect of the system, and a display focusing on the control flow would be the natural choice.

Pictures of *data flow* (fig. 2-5) have mostly been used in connection with dataflow languages and systems [Davis 82]. Here, as with functional programming and state diagrams, data flow is identical to control flow, so we might as well talk about the latter. Indeed, flowcharts and data flow graphs share the same problems: undisciplined structure leading to messy diagrams,

and no support for non-trivial data structures. As control flow graphs, data flow graphs are suitable for animation, with node highlighting showing the progress of data through the network and emphasizing the parallelism obtained.

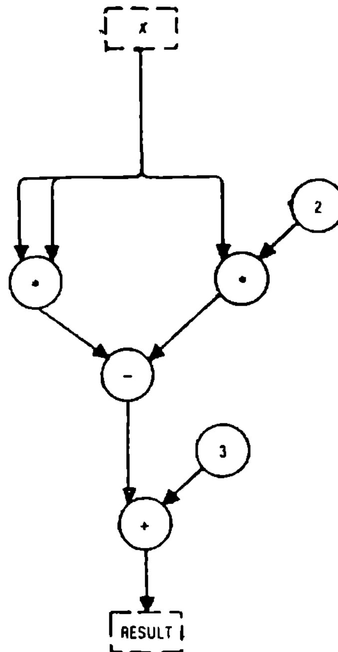


Figure 2-5: Data flow diagram.

Most programmers use some kind of "boxology" technique to describe the overall structure of their systems, like a module interconnection graph where the arrows between the boxes represent access to code or data. Figure 2-6 shows the module breakdown of our system for programming in pictures. As a basis for a system browser that allows the programmer to get an overview

of a large system, such a facility is very useful. It fails, however, to provide support for programming-in-the-small, where the simple, static component description must yield to dynamic control and data structures.

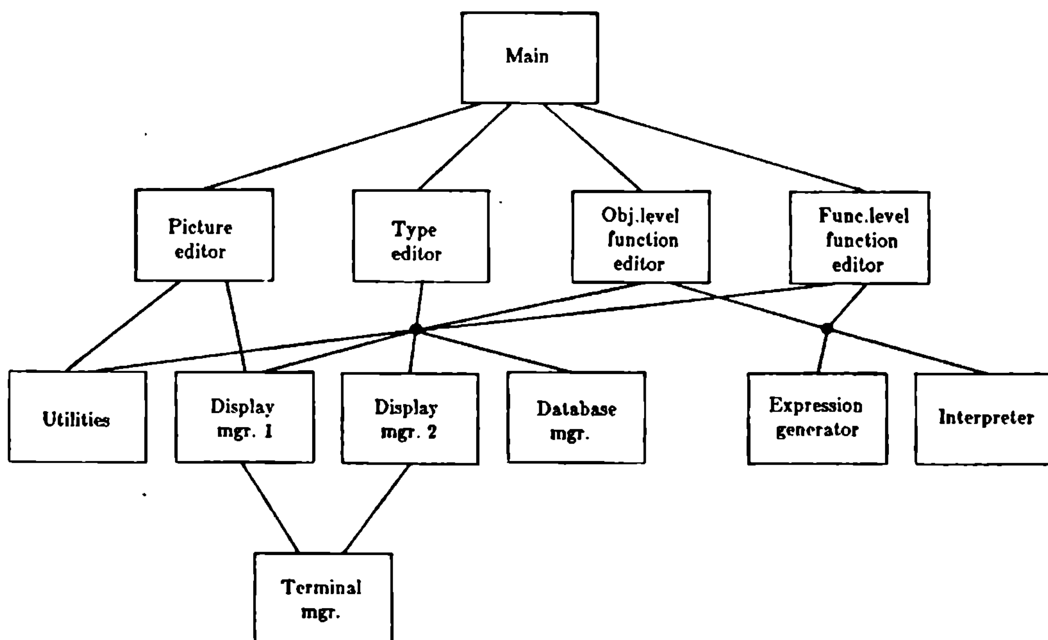


Figure 2-6: Structure of our PiP system.

There are certainly many other specialized diagrams that help shed light on some aspect of a program, especially if we consider high-level concepts. For example, figure 2-7 shows scheduling dependencies among simulation processes. General diagrams that cover more than one simple aspect of programs are hard to find, though.

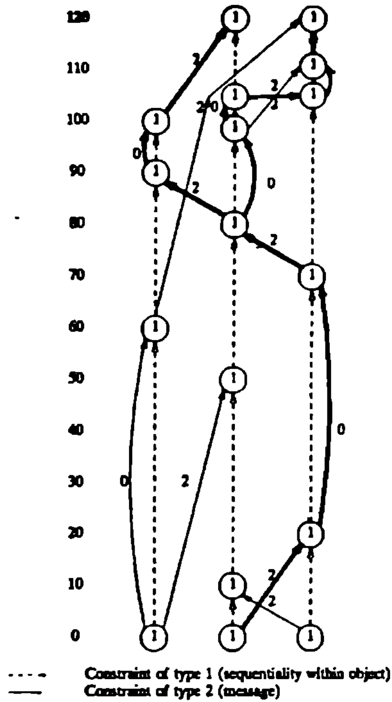


Figure 2-7: Simulation process dependencies.

2.2. Program development environments

Until now, the representation of programs, i.e. programming languages, has been designed with a purely static medium in mind, the printed paper. Defining a language as a sequence of symbols has certain advantages: The syntax can easily be described by a grammar, representing a simple and well-defined interface to a compiler. The static, one-dimensional view is also the basis for current program verification techniques. In daily life, however, these languages are used in a very dynamic fashion, as the representation of

structures that are created, inspected, modified, executed, and in general thought about in many more interesting ways than the program text can suggest. As a result of this, development environments have been created that in various ways "support" the process of program development. These environments are rarely very well integrated in the language definition, and conceptually they lag considerably behind the state-of-the-art of programming languages.

One of the founding ideas of our work is that, rather than treating a programming language as a sacred cow that we have to make practically usable by building development tools on top of, we should recognize the language as an interactive tool in itself and design it according to the needs of the programmer, not the compiler or the theoretician. Starting with the interactive computer system as our medium instead of printed text gives us new, powerful capabilities to express programs. This has been demonstrated by many applications and development environments, for which a dynamic sequence of graphical images has already become a common medium. Let us take a look at some recent work on program development environments and how it applies to the design of our programming system.

2.2.1. Syntax-directed editing

Since most programming languages are represented as text, it is common to manipulate programs via a standard text editor. Programs are not simply text, though, but compositions of computational structures, and it would seem appropriate to build editing tools that embody this view. Syntax-directed editors [Medina-Mora 82, Meyrowitz 82, Morris 81, Teitelbaum 81a, Teitelbaum 81b] use the syntactical definition of a programming language (or other context-free grammar) to operate on the syntactical categories of the language as editing units instead of each individual character in the text. Some systems actually represent the program as a parse tree and invoke a pretty-printer ("unparser") to display it on the terminal.

Syntax-directed editing has several advantages: First, it reduces the typing effort to enter programs and minimizes typing errors, since the language tokens are usually available through a few keystrokes. Second, it ensures that the program is always syntactically correct, eliminating many frustrating passes through the compiler, especially for programmers who are not very familiar with the language. This also speeds up the learning process for such users. Third, it allows for "intelligent" editing operations that seem natural to the programmer, like deletion of a compound statement or a search bounded by the enclosing procedure. Fourth, combined with modern

programming languages, it encourages a well-structured, top-down development style by enforcing the view of programs as hierarchical compositions of language constructs. Fifth, it provides an interface for debugging tools. Some systems generate code along with the parsed statements and allow the program to be executed (statement by statement, procedure by procedure, if desired) through the editor. This provides a unified interface that both shortens development turnaround time and conceptually simplifies the development process.

The main problem with syntax-oriented editors for text-based languages concerns the granularity of syntax checking. To modify a program, we often have to take it through incorrect intermediate states, and we are really not interested in the system getting upset about these. Some editors allow only changes in terms of syntactical entities, but this leads to serious inflexibility (for example, to change a conditional into a loop, the whole conditional has to be deleted, even if the loop may use the same boolean expression and body). Other systems flag incorrect statements in a fairly quiet manner (like inverse video) until the modification is completed.

There are two reasons for the granularity problem: First, these systems are really of a hybrid nature. We represent programs as sequences of characters, but we still want to manipulate them as program structures. Thus, it is

possible to introduce errors that are even below the token level, i.e. we can create structures that are not even legal programs. Second, the structure of languages is not designed with interactive modification in mind. Program maintenance has been studied at the macroscopic level (program structuring, information hiding, integration of programming tools), but how language structure influences changes at the microscopic level has not been a primary issue.

In the system presented in this work, we have attempted to avoid these problems. Since we design the system directly with the graphical, interactive medium in mind, we can pick representations that correspond atomically to the language concepts. Further, we have chosen a computational model, functional programming, that has a simplicity and orthogonality that relieves us from some of the editing problems mentioned above.

2.2.2. Style of interaction

One of the few technological advances during the past years that seems to be able to contribute more than a simple capacity increase is the introduction of cheap high-resolution bit-mapped displays. After two decades of character-based display terminals, which really didn't take us very far from their "teletype" ancestors, we suddenly have virtually unlimited possibilities to express things graphically. Now a rapid sequence of two-dimensional images

is our medium. There are of course many directions in which we could develop this potential. There already exists one paradigm that has become fairly popular and is utilized by several manufacturers. This is the pictorial design pioneered at Xerox PARC through the Smalltalk system [Goldberg 83]. Although the Smalltalk language is itself fairly conventional when syntax is concerned, the development environment builds upon a few ideas that have also inspired other systems [Teitelman 77, Williams 83]:

Menus. The inclusion of the mouse as input device makes moving a cursor around on the screen extremely fast and picking entries from menus very convenient. Combined with rapid screen updates to reflect the menu choices, this provides interaction that is simple enough for users not familiar with the system, yet fast enough for expert users. The primary benefit of menus versus stated commands is that a menu does not require the user to memorize specific syntax. Another interesting aspect is that a menu-based system makes system *exploration* quite easy. This is very hard in a command-based system, although it is to some extent possible to anticipate what commands users will attempt [Wixon 83]. The use of menus has been subject to research in itself. [Norman 83] investigates the tradeoff between information conveyed by menus and the time it takes to display them. Fast screen updates makes large menus with much associated information feasible. The size of menus is

still limited by the human capacity to scan images and recognize entities. [Warman 81] gives 5-7 as the maximum number of items we easily can distinguish among. This means that the depth of the menu hierarchy in large systems can become a problem (especially if the user has to move level by level out of the system again), if we do not employ special techniques, like displaying a large menu as several submenus of related options.

Windows. The high resolution of modern bitmapped displays makes it feasible to display a large amount of information on the screen simultaneously. This has spawned the idea of showing several more or less independent activities in separate windows on the screen at the same time. The reason why this is so appealing to most users is that most of the tasks we perform, whether we use a computer or not, are composed of several sub-tasks, and most humans tend to shift their attention from one sub-task to another in a fairly random fashion. The window technique thus makes the screen look like a familiar desktop, with windows corresponding to sub-tasks spread around the screen. If moving the mouse from one window to the next is all it takes to shift from one sub-task to another, the shift of attention is made virtually effortless, supporting people's work habits.

Modelessness. A related issue that has been brought up in this context [Tesler 81] is the issue of modelessness. Many systems have rigid "modes"

that take care of different parts of the system's function. If it is (1) awkward to go from one mode to another (cf. moving from editing mode to compiling mode in most program development systems), or (2) the same commands mean different things in the different modes (or, equivalently, different commands must be given to obtain similar results), we say that the system has a *rigid mode structure*. Since users like to switch their attention between different aspects of a task, a rigid mode structure both hampers this shifting of attention and also causes confusion with respect to command usage. (A case in point for the latter occurs when looking at incoming mail through one of our text editors at USC. The regular commands for moving about in buffers cannot be used; the mail system requires its own.) The window technique combined with mouse input can be used to break down the mode barriers, since mode switching can occur simply by moving the mouse.

System transparency. Computers are powerful tools for information processing, but, just as most Americans find it more convenient to grab a hamburger with their hands instead of using knife and fork, there are significant differences in computer system tool convenience, too. When the computer is used for manipulating some object, we are really interested in dealing with the object as directly as possible. The computer should not get in the way, it should be transparent.

Two components of system transparency can be identified [MacDonald 82, Tesler 83]. When the user initiates a command, *immediate feedback* causes changes on the display reflecting the result of the command as soon as the command is executed. This principle is violated by most operating system interfaces. Who hasn't often issued a separate command to examine the print queue just to make sure that a print command succeeded? The popularity of screen-oriented editors also illustrates the importance of command confirmation. Without immediate feedback, the user must remember much of the system state and must make conjectures as to how it changes, without getting this confirmed until later.

The second component of system transparency is the *What-You-See-is-What-You-Get* paradigm (and What-You-Get-is-What-You-See; there should be a one-to-one correspondence). For example, with a graphics screen, a text formatter can directly display the final layout of the document instead of being based on cryptic commands for font changes, subscripts, special characters, etc. If the user does not see the end result directly, he/she must perform a mental translation. This translation cannot always be correct, and this can lead to much irritation and waste of time.

Mixed text and graphics. With bitmapped displays (and dot-matrix or laser printers), technology seems finally to have caught up with the artificial

boundary that has existed for centuries (since Gutenberg, actually) between the ease of handling textual and pictorial information. It is an indication of the utility of pictorial representation that after all this time our books and documents still contain a fair amount of figures. We can of course only speculate as to what effect several decades of convenient creation and distribution of pictures will have on information material around us, but it is clear that this is a significant component of the new interactive medium about to be explored.

The components just described together make up a coherent interaction language that is becoming adopted by an increasing number of system developers. This acceptance may indicate that the style has certain inherent positive features, most of which were outlined above, and it may also be a first small step towards the user interface standardization advocated by [Newman 80]. We will follow this style whenever it will serve us, but we will feel free to invent our own designs when we need to. This will help both users of our system and ourselves to focus attention and effort on the genuinely new aspects of the system.

2.2.3. Integrated environments

Most computing environments, be they intended for office automation, program development, or other purposes, consist of several software tools each designed to aid in one more or less well-defined sub-task within the overall goal of the system. To accomplish a task, the user has to make use of several such tools, passing some object (a form, a set of data, a program) from tool to tool. Thus, even for simple programming environments it is clear that several benefits can accrue from some kind of integration of the tools that the environment offers, and this becomes increasingly clear for larger environments that can provide dozens of tools, e.g. for the requirement specification, design, development, verification and maintenance of programs [Howden 82, Sandewall 78]. The integration can occur at two levels:

Internal integration refers to integration in the communication between the various components of the system. This usually means that the components communicate via some standard data format. In the primitive case this format is simply a file (as in UNIX), but by deciding on a format with more structure, more interesting information can be exchanged between tools. In essence, each tool knows more about what the others are doing, so that it can respond more intelligently to the results of the use of other tools. For example, a program editor can use the diagnostics from a compiler to position the cursor at the offending token of a parsed program.

The tools may not only share data format, they may share data themselves in the form of a database containing descriptions of the current state of the system. In this way all information about any component is available to all tools, thereby maximizing the utility of the information, and at the same time centralizing it, making sure all information is consistent. This concept is used in design databases [Williamson 84] that keep track of system components and their interrelationships during all phases of a design project.

External integration refers to how the system components appear to the user, i.e. the user interface. Different command syntax for different components can only confuse the user and make the system hard to learn. A well-defined, unified user interface helps making rapid context (tool) switching painless (cf. our discussion of modelessness above), and it can encourage users to use parts of the system they otherwise would not attempt to learn. Integration of user interfaces also forces serious thinking about standardization. In the end this may lead to fairly standard user interfaces across machines, so that moving from one system to another can be fairly easy, i.e. we obtain integration even between different systems [Newman 80].

The system developed here will consist of a few distinct tools, and since we design them simultaneously, we can assure that they will benefit from proper integration, both internal and external.

2.2.4. Program animation

The aspects of program development considered so far concern only the static representation of programs. The dynamic behavior of a program during its execution is usually hidden from our view, but it is nevertheless this aspect that is of primary concern to the programmer. Several attempts have been made at visualizing this internal behavior as an aid during program development.

There are basically two approaches to program animation. The simplest technique is to use a graphics library and insert calls to image drawing routines whenever the state of the program changes in an interesting way. This in essence makes the programmer write two programs in parallel, one for the simulation and one for the graphics, and there is of course no guarantee that the two will agree with each other. Sometimes, however, we may only be interested in the graphics part, as for the production of animated films. [Brown 84], [Langlois 84], and [Magnenat-Thalmann 84] describe some recent contributions to this method.

In the other approach, hooks are inserted into the low-level run-time software. Procedures that display various parts of the program state as a graphical side-effect are then attached, so that the progress of a program can be monitored without any modification to the program itself. In simulation, for example,

the routines for process scheduling can be modified in this manner, allowing the simulationist to follow the life of processes moving between queues (see e.g. [Birtwistle 84, Dewar 84]). This provides an insight into the behavior of the simulation model that trace dumps of scheduling actions can never accomplish. Such a model visualization aid is another instance of the What-You-See-is-What-You-Get principle decribed earlier. A similar technique can be used when developing distributed software. [Unger 84] describes such a system that intercepts all message passing between processes and displays the action graphically.

These systems display a few rather high-level structures of programs, and benefit from it in that non-trivial semantics can be contained in the pictures (e.g., a queue can be drawn to look like a queue). It seems to be harder to draw interesting pictures automatically for user-defined structures made up of general components. [Dionne 78] describes a LISP system that automatically draws S-expressions as they are evaluated. Except for simple list operations, these pictures tend to get unwieldy and perhaps even obscure the high-level semantics of the function being executed.

Common to these latter systems is that they attempt to automatically display the progress of programs without gathering more than a very narrow slice of their behavior hierarchy. If we want to examine the programs at precisely

this level, they can be very helpful, but if we want to get a more detailed or overall view, they do not provide much assistance. Also, we see that there is no graphical communication between the programmer and the display functions. The programs are represented in the usual textual way, and it is up to the display software to invent good graphical metaphors. In our system, we will support the display of structures at all levels of abstraction, and it is one of our basic principles to let the programmer decide what pictures should represent these structures.

Related to program animation is program *monitoring*. Here, techniques are developed for selectively displaying parts of the program state, and also for collecting statistics about the program behavior [Plattner 81]. Debugging, performance evaluation, and optimization are the application areas. Unlike animation, monitoring usually employs a separate, independent monitoring process that can extract the desired information from the program in operation. The final display is therefore not only the result of executing the program itself through a layer of animation utilities, but the output can be processed further by the monitor program. Such facilites undoubtedly have enormous practical value, but we will not attempt to develop this issue fully in connection with our own design.

2.2.5. Programming-in-the-large

DeRemer and Kron, in their landmark paper [DeRemer 76], argued convincingly for the recognition of programming-in-the-large as a distinct activity, different from the conventional programming-in-the-small, and requiring its own language tools. Such a utility has four major functions: It serves as a *management tool* for organizing and breaking down the various parts of the system by its manager. It is a *design tool* for actually describing the structure of the system in terms of its components and their interconnections. It serves as a precise *communication tool* enabling individuals working on the project to interchange information in a precise manner. Lastly, it is a useful *documentation tool* for automatically generating formal descriptions of the system.

While such a two-tiered approach seems natural within a traditional language environment, the situation can look different for systems based on other languages. For a purely functional language, for example, the programming modules correspond to functions, the building blocks of the language itself. Further, these modules/functions only communicate via data passed from one function to another. Hence, there is apparently no need for a special module interconnection language [Keller 81]. In practice, there is still a need for grouping related functions together into modules and controlling resource usage by some kind of scoping mechanism.

As we have hinted at earlier, we have based our system on functional programming, so we avoid these issues to some degree. Since programming-in-the-large is not central to our thesis, and to limit the scope of our work, we will not address the problems in detail, but only indicate possible solutions that may fit our framework.

2.2.6. Programming-by-example

Programming-by-example refers to techniques whereby a program, formulated in terms of its behavior on one or more sets of example data, can then be synthesized by a support system into a general program. Ideally, the programmer should only need to specify pairs of corresponding input and output data, and the system should find the "natural" extension to the general case. This is very hard to do, since the generalization can always be done in many directions. This approach therefore requires that the search space be limited in some way, for example by restricting the application area or the computational power of programs generated.

Query-by-Example [Zloof 77] provides programming-by-example in the restricted context of database operations. Here, example data satisfying a query are set up by the user, and the system responds by extracting all data satisfying the same query. That is, in contrast to regular programming, where a program is first created, then applied to a set of data at the user's

request, the program created in QBE is automatically applied to all available data immediately. QBE has been extended to include common office tasks (word and data processing, report writing, graphics and electronic mail) [Zloof 82].

The problem can also be simplified by limiting the data structure. [Summers 77] describes techniques for inducing transformations of LISP list structures from input and output lists.

Another approach to making programming-by-example feasible is to supply traces of intermediate states along with the desired input and output [Biermann 76]. Alternatively, the desired function can be illustrated by example computation. In its simplest form, the programmer performs a complete computation on example data, much like programming of pocket calculators. This gives only a description of a straight computation, and it is difficult to include information about conditionals and loops. [Halbert 81] describes such a system, enhanced with a loop construct, but without conditionals. More advanced systems can discover the structure of conditionals and loops from several example computation traces (e.g. [Bauer 79]), combined with common knowledge about programming concepts. Ultimately, artificial intelligence techniques will have to be employed to mimic the kind of generalizations humans make.

Programming-by-example has mostly been motivated by the need to let people from outside the computer profession create their own programs. Programming today is too difficult for non-specialists for several reasons. One major obstacle is the extent of intricate detail that needs to be mastered. Programming is also very abstract, since computational structures must be formulated and generalized in the programmer's mind before they can be encoded in a program. Furthermore, this encoding only indirectly describes the program functionality, without giving the programmer any clear indication about how it will perform on given data. Programming-by-example attempts to alleviate this by letting the system take care of extraneous detail and allowing the programmer to concentrate on specific, concrete examples. These are goals that coincide with our own, so we will explore example-oriented programming as part of our work.

2.3. CAD/CAM

Computer-Aided Design and Manufacture is one of the oldest application areas for interactive computer graphics, and is perhaps the area that currently exhibits the closest parallels to interactive program development. Indeed, we might well view program development as computer-aided design of programs. The similarities are particularly clear in CAD of VLSI circuits, a discipline which more and more closely resembles its software counterpart

[Maling 81, Shrobe 83]. Both are creative activities, concerning the design and implementation of non-trivial, large systems of hierarchical structure, and they encounter many of the same problems when development tools are to be built. Hence, much of the research on CAD systems applies to program development systems as well:

Interaction language: The language of VLSI itself (circuit components) may be different from the language of programming, but the language of interaction can be the same, and faces the same challenges. The system should display as much information as possible to maximize the user's control of what the state of the system is. But this information has to be displayed in such a form that it is easily extracted by the user, and so that only necessary information is actually extracted. The information transfer rate must be increased in the other direction too: By designing intuitive representations of concepts, and formulating powerful, yet precise tools, a naturalness of expression can be achieved that greatly simplifies the user's design task. Certain general principles, such as minimizing memorization, providing immediate feedback on all user actions, and reducing the risk for misinterpretation and confusion, further enhances the quality of the interaction. We will say more about these issues in the section on human-computer interaction.

System management: Another aspect of CAD systems is the facilities for support of large systems. Viewing and simulating hierarchical designs at different levels; integrating modules developed by different people; providing libraries of components that can be reused; keeping track of interdependencies among modules; version control; all these are valuable mechanisms we recognize from discussions on software development environments.

Testing: In a VLSI circuit development system, a powerful simulator is just as important as the tools for putting the design together in the first place, since this allows the chip to be tested before a costly implementation is attempted. For programs, the computer itself usually serves as a simulator. It is nevertheless desirable to monitor and control programs more carefully than what can be achieved through a simple execution of the programs. Hence, there is a need for versatile debugging tools for programming, too. Interactive, graphically supported testing of VLSI designs is fast, since immediate feedback is provided and modifications can be made as soon as errors are found. With good probing and display tools it makes it easier to locate problems by actually showing what is going on internally. It supports a structured approach to modifications, since smaller entities usually can be isolated and tested, and subsequently reinserted in the main structure. It can reduce introduction of new errors by keeping track of dependencies between

parts and automatically checking for design constraints. These are all well-known merits of interactive programming environments with symbolic debuggers.

There are, of course, differences too. Pictorial diagrams of electronic designs have been in use since long before the first VLSI CAD system was built, and they continue to be a useful representation of the circuits. At the lowest level of design, the physical layout of the chip provides another natural pictorial representation. Thus, in CAD (for VLSI and also for most other uses of CAD) the pictures are given and we are only faced with the task of utilizing them in a useful way. In programming, as we shall see, it is mostly up to the programmer to decide what kind of pictures to show. The underlying reason for this is that programs can model anything in the real world, whereas CAD systems concern a specific domain.

Another characteristic of CAD systems is the large number of constraints on the designs. This is related to the fact that a VLSI circuit must be viewed from several angles: Timing and control, dataflow, schematic (logic diagram), and physical layout are different aspects of the same design imposing different constraints. Physical alignment, wirability, timing dependencies, noise, power consumption and various parameter optimizations are some of the factors that must be considered, and CAD systems should contain components that do

corresponding analyses [Renfors 83, Revett 83]. While it is necessary to control most of these factors to obtain a working chip, we usually have at most overall performance demands in programming, related to the single dominating (functional) view of programs. Our task is in this respect simpler, although we should add that for very large systems more sophisticated tools are needed.

Our task is ultimately also made simpler by being able to test in the real environment. For example, an electronic circuit contains much real concurrency, and however good simulators are built, they cannot provide more than approximations to the real behavior. This can cause problems in unfortunate cases, since small inaccuracies in concurrency can cause substantial differences in behavior. In general, VLSI testing really means *simulation*, which means that we have to stop somewhere down the hierarchy of components and base the tests on simplified models.

2.4. Office information systems

Information systems based on networked microcomputer workstations are rapidly being introduced in the office. Their chief function is to store, retrieve, manipulate and control documents within a distributed environment to aid in document preparation, information management, and decision making [Ellis 80]. Thus, unlike traditional business data processing, these

systems are highly interactive in nature. There are a number of issues related to the human-computer interface that are of interest to us here:

Programming languages: The dynamic nature of modern offices, with changing conditions and requirements causing changes in the office worker's routines, makes it necessary to allow the end user of the OIS to program his/her own applications to some degree. The traditional, centralized data processing department approach is too slow, since computer professionals are scarce and communication between users and programmers is difficult and unreliable; it is too expensive, since the scarcity of specialists causes salaries to skyrocket; and it is not flexible enough to take care of individual users' needs [McNurlin 81a, McNurlin 81b, McNurlin 82]. There is also a growing realization of the importance of how the individual feels about his/her work. The ability for each office worker to redesign job procedures according to taste is therefore becoming increasingly important. For end-user programming to be feasible, we need very high level languages that are simple enough for fairly unskilled (in the computer domain) people to use, yet powerful enough to allow non-trivial applications to be generated. These languages can be domain specific, since it is known within what area they will be used. For the same reason, computational power can to some degree be sacrificed, making the systems easier to use and allowing for better support in

terms of consistency and other checking. In our system, generality precludes these simplifications, making a well-engineered human-computer interface all the more important.

Programming-by-example: One way simplicity of use has been obtained in OIS while at the same time keeping a fair amount of computational power is by exploiting examples [Zloof 82] (cf. sect. 2.2.6). By specifying the desired operations on the resulting document itself, the gap between an abstract description of the computation and the resulting operations is obliterated, significantly reducing the complexity of the programming process. This is exactly what we are doing in our system here, when we draw data instead of abstract control structures and let the user directly show how the data are to be transformed by the program.

Graphics: Graphics has of course been recognized within the business environment as a powerful means to convey information. But there have also been experiments on using graphics in the information management process itself. [Herot 80a, Herot 80b] explore a system that displays a database on a plane surface and allows the user to browse through the data displayed, zooming in on entities of interest. This extremely fast and convenient way to search the database makes it feasible to look for things without knowing much about them. Even if the user knows exactly what he/she is looking for,

it may be easier just to zoom in on it graphically than specify a lengthy predicate. Thus, the pictorial representation lowers the memorization requirements and provides a more convenient channel for communicating with the computer. Moreover, spatial positions and shapes of icons can convey information that is hard to store symbolically. These factors are central to our work as well.

Concrete models: For OIS systems to work well, it is important that the conceptual model embodied in the user interface closely resembles the "real life" model the user has of the organization in which he/she works and how his/her job fits into it. This makes the medium (the computer) transparent as a tool to support already well-known concepts. In the same way, the implementation of the tools can be made transparent by always displaying as much as possible of the current state of the application. This is the "Visual Programming" paradigm [MacDonald 82] (or, "What-You-See-is-What-You-Get"), as embodied in screen editors and many form-based data entry systems. Query-by-Example [Zloof 77] also follows this spirit by showing directly the resulting data. In our programming system, this principle applies both to the representation of programs and to their display during execution.

2.5. Human-computer interaction

The field of Human-Computer Interaction (or Human Factors, or Ergonomics) has gained substantial popularity in the past few years, and there is now a steadily growing body of significant research in the area [Bo 82, Carey 82, Moran 81, Sondheimer 82]. The main reason for this interest is of course the proliferation of inexpensive microcomputers. When a large part of the population spends its workday interacting with a computer, the design of human-computer interfaces becomes extremely important for the well-being of many people. The increased capabilities of these computers (fast processors, large memories, graphics) heighten the need for sound principles to guide in system development, and at the same time represent an opportunity to implement the findings being made in this research area.

The problem of human-computer interface design is attacked from several angles. From the theoretical end, concepts are borrowed from other fields, e.g. cognitive psychology and linguistics, to form a model of the user. These models can be at a very low level (like the keystroke model [Card 80]) or at a high level (like activity organization [Bannon 83]) in the task hierarchy. Analytical models are also attempted to capture user behavior in a mathematical framework (e.g. [Norman 83], which studies tradeoffs in user satisfaction). Good models of the user can make it possible to predict user reactions to new interfaces and deduce practical design rules.

Another class of efforts attempt to formalize the human-computer dialogue via some specification technique (e.g. [Jacob 83, Kieras 83, Lawson 78, Roach 83]). This is useful if we want to apply automatic checking of the dialogue, for example for security against misuse of systems, or for dialogue evaluation by expert systems. It also is the basis for rapid prototyping techniques. Since interface design is still very much an art the ability to quickly build and explore prototypes is extremely valuable.

From the empirical end, a number of interesting experiments have been conducted to determine the effect of various factors on user satisfaction and productivity. Issues like screen layout [Teitelbaum 83], menu size [Warman 81], mouse design [Price 83], color use [Frome 83], and reponse time [Butler 83] have been investigated to determine the effects on the user at a fairly microscopic level (e.g. response latency and error rates). More global performance aspects of interface design, like learning time [Gomez 83], the time needed to perform a certain task, or the effect on how the user organizes his/her work [Murrell 83], have also been studied. New forms of interaction are investigated [Krueger 83, Lippman 81, Negroponte 81], and existing systems are evaluated [Bewley 83, Yavelberg 82].

Several aspects of computer programming have also been studied from a human-computer interaction point of view [Sheil 81]. For example, [Sime

77] empirically establishes that an *if-then-else* nested program really is easier to understand than a similar *goto* based one. [Miara 83] investigates program indentation and finds that moderately indented programs (2-4 spaces) are optimal. [Shneiderman 77] finds no indication that flowcharts really help in the program development process. How computer programming is learned is studied by [Mayer 81]. Mayer conjectures that novices support their reasoning with a concrete model of the computer, and largely confirms this with experiments. This suggests that programming environments should provide such a model directly. A common result quoted in several reports is that expert users are relatively immune to interface and language design, whereas novices and casual users are easily affected [Moran 81]. Thus it is important to determine the audience when developing programming systems.

The most directly applicable result of the research on human-computer interaction is a growing catalog of design philosophies [Nakatani 83], principles [Gould 83, Jones 78, Mooers 83, Morse 79], methods [Kelley 83, Topmiller 78, Wixon 83], and rules of thumb. We are still far from being able to guarantee the quality of a new interface, but these guidelines already supply the designer with concrete aids in the design process.

In this work we are concerned with exploring new ways to represent computer programs, i.e. we are interested in the human-computer interface aspect of

programming concepts. The research outlined above is therefore of great interest to us here and we will refer back to specific results as we go along.

2.6. Other systems

Diagrams have long been used to display the *global* structure of programs, since this can usually be shown as a simple static picture of objects and their relationships. Most software projects employ at least some informal way to express the overall design pictorially, usually as a graph of modules connected by control and data references. [deBalbine 78] describes a system that automatically generates Modular Tree Representations of programs, supplying the system developers with correctly updated overviews at all times. The TELL system described by Hebalkar and Zilles [Hebalkar 79] is an interactive editing facility that uses a graphics display to provide development support of systems as hierarchical block diagrams. The system supplies icons and arrows for standard code and data modules and control and data flow among them, but the user can also define his/her own. At the bottom diagram level, conventional program code is entered. The system maintains a database that is amenable to various kinds of system analysis. None of these systems address the topic of our main interest, though, viz. the issue of representing graphically the detailed program itself and its dynamic behavior.

Frei, Weller and Williams at IBM, San Jose [Frei 78], acknowledge the fact

that programming does not have a "natural" graphical representation, unlike most other areas. In their search for a useful representation, they end up with structured charts (Nassi-Shneiderman diagrams; cf. fig. 2-1) and claim that a graphical programming system based on these will support structured, top-down program development by visually exhibiting program structure and enforcing page-sized modules. A pictorial representation will also more clearly exhibit the meaning of programs and result in better coding, improved productivity and better documentation. [Ng 79] describes an implementation of the system.

Structured charts still go a very short way towards pictorially displaying programs. They really are little more than "structured" flowcharts, giving graphical representations to control structures and leaving expressions, assignments, procedure calls, and data, parameter and type descriptions still to be written in some regular programming language (in this case PL/1).

Keller and Yen of the University of Utah have built a graphical programming system based on functional programming (FGL) [Keller 81]. Since there is a direct correspondence between the functional model and the dataflow model, they can display their programs as graphs consisting of nodes representing function applications and arcs corresponding to the data flow between functions (fig. 2-8). A functional graph model has several advantages: A

program is a composition of functions, so the control and data flows are the same, thereby obviating the multiple views necessary in more conventional models. This simplifies graphical representation significantly. A flow-graph brings out clearly any concurrency that may be obtained and errors are likely to stand out well. Function graphs have a well-defined semantics, so they can be executed by an underlying system. The simple functional structure encourages modular design and can be used at all levels of system description, making a separate module interconnection language unnecessary (cf. sect. 2.2.5).

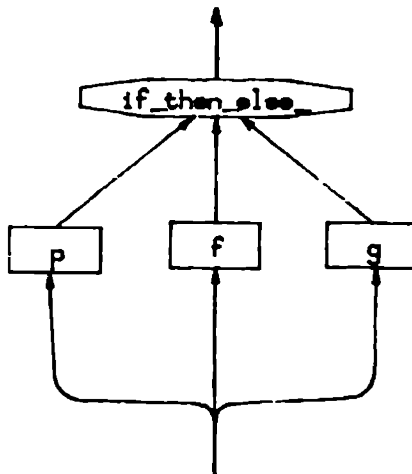


Figure 2-8: Conditional in FGL.

We think, though, that a graph-based design will be awkward to use in practice, since graphs take a lot of space and can get messy if they are non-trivial. More importantly, a graph does not convey any semantic information

beyond node and arc names and connectivity, so we still have not exploited the graphical medium very far. The use of a functional base is interesting, though, and we have found it a useful foundation for our work as well.

FADT is both a design technique and programming tool being developed at the University of Trondheim [Amble 83]. It is based on an extended, formalized version of flowcharts that map into PROLOG-like code (fig. 2-9). The criticism given for graph-based languages above applies to this system as well: The pictures can get unwieldy and they cannot represent in a graphical way more than the simple data dependence aspect of programs.

Cardelli [Cardelli 83] makes a rather elegant effort to design a language for manipulation of two-dimensional data structures, like boxes with letters and pictures (fig. 2-10). He acknowledges that a conventional sequential language can never express spatial objects and relations in a natural way, and defines a two-dimensional, functional language with pictorial operators and function lay-outs. In addition to the usual integers, booleans, etc., his primitive data types include a simple *box* for creation of two-dimensional structures.

Cardelli's system seems to work quite well for the graphical data structures it supports, like lists. But since the pictures displayed are generated by the system, there is no way for the programmer to attach semantically interesting

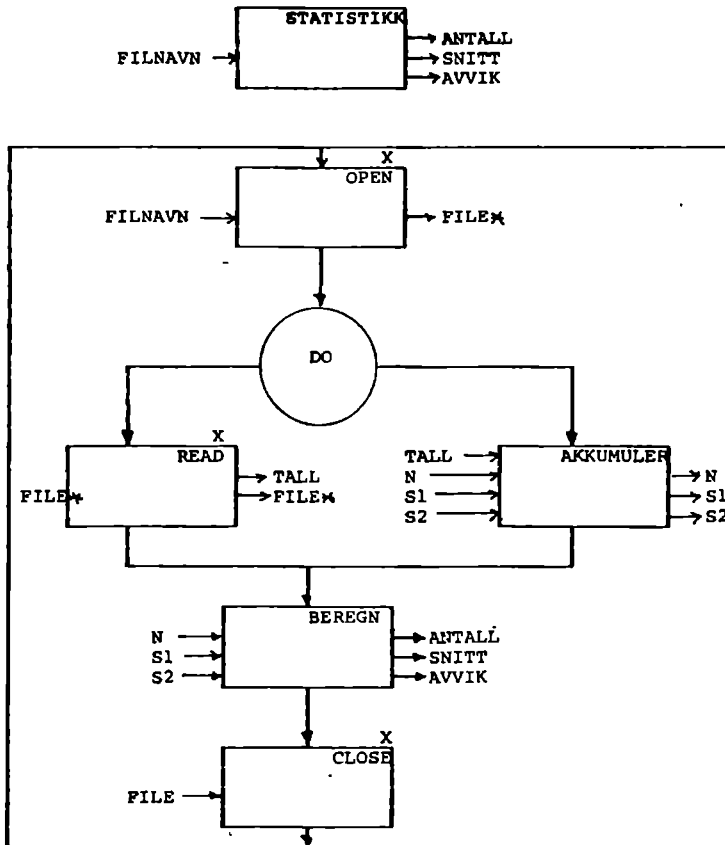


Figure 2-9: FADT function.

figures to the programs, i.e. one can still not get an immediate feeling for what the program does just by taking a brief look at it.

The Program Visualization (PV) environment being built at the Computer Corporation of America [Kramlich 83] is primarily devoted to the *dynamic* visualization of programs. The system presents an integrated view of code and data structures during program execution, allowing the programmer to

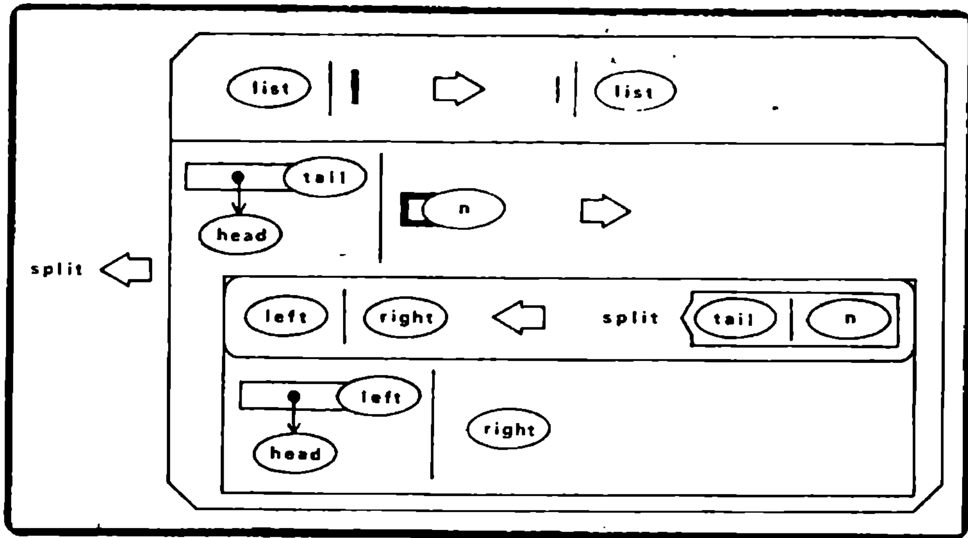


Figure 2-10: A list splitting routine in Cardelli's notation.

monitor directly both the control sequence and the data modifications (fig. 2-11). The user can further create graphical pictures and link them to code and data, establishing high-level icons that can be used during the execution monitoring. The level of abstraction shown can be chosen by the programmer, ranging from top-level drawings of the whole system being developed to the bottom-level language code.

Since the PV system addresses program simulation, the authors have recognized the importance of displaying data structures. The data-oriented view is one of the main principles of our work. The PV system does not, however, attempt to use the displayed data for more than program

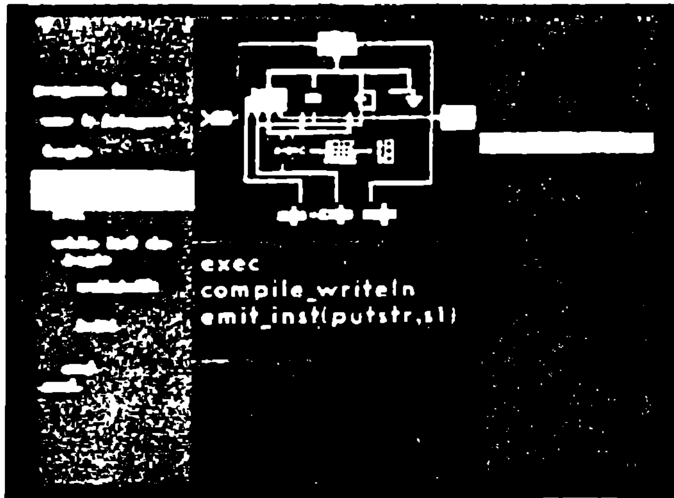


Figure 2-11: Program execution in the PV environment.

illustration. The program itself is displayed along with it as conventional text, and the progress of its execution shown by highlighting program statements.

The PegaSys system developed at SRI [Moriconi 84] also does not represent the programs themselves in terms of pictures, but it is an attempt at using pictures formally for program design and documentation (fig. 2-12). Pictures are mapped into calculus formulae, and the system can check that the composition of picture elements is consistent. The system also supports a refinement methodology for the visual specifications.

PegaSys combines graphics with formal logic quite interestingly, and represents programs as a hierarchy of precise, yet fairly meaningful pictures. The plans for extending the system with animation also seem exciting. The

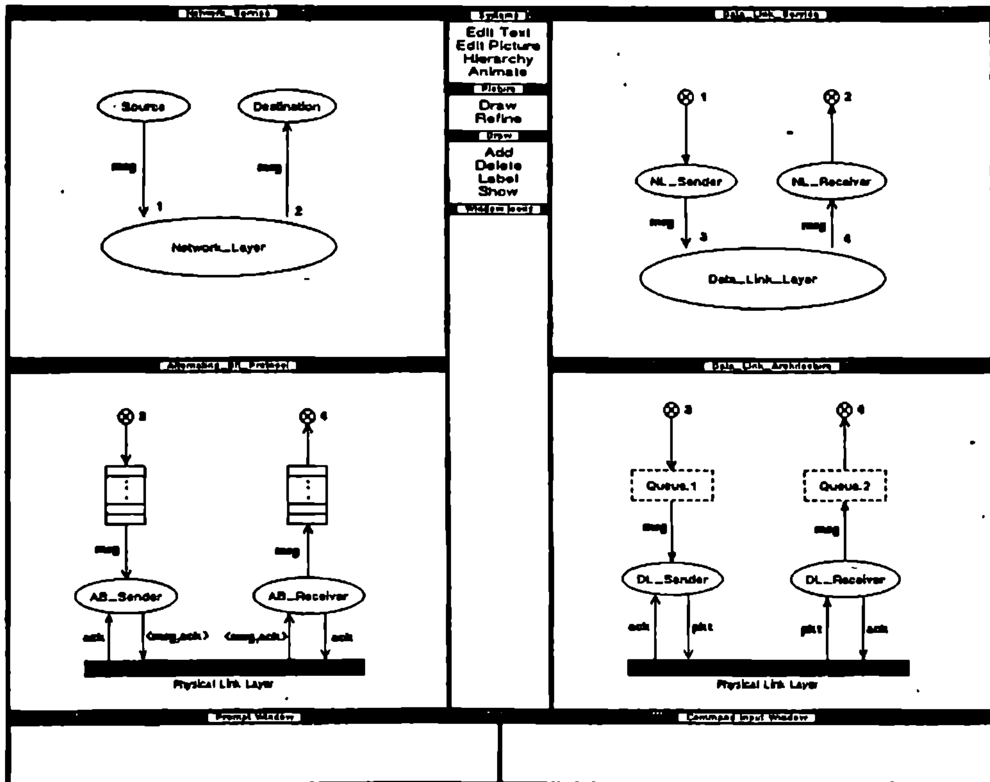


Figure 2-12: Visual specification hierarchy in PegaSys.

amount of graphical semantics the user can contribute is still limited, though, and the final refinement step, down to the actual program code, is missing.

The PECAN system developed by Reiss at Brown University [Reiss 84] utilizes high-resolution graphics to show a large amount of information during program development (fig. 2-13). The system can present a collection of all the traditional views of a program at the same time on the screen: program listing, data type schema, parse tree, symbol table, flow graph,

execution stack, and input-output dialogue. Having instant access to all this information can give the programmer a good view of what is going on in well-known programming terms, and the system is indeed a powerful debugger. This work does not, however, attempt to contribute any novel paradigm that utilizes new technology to improve program representation.

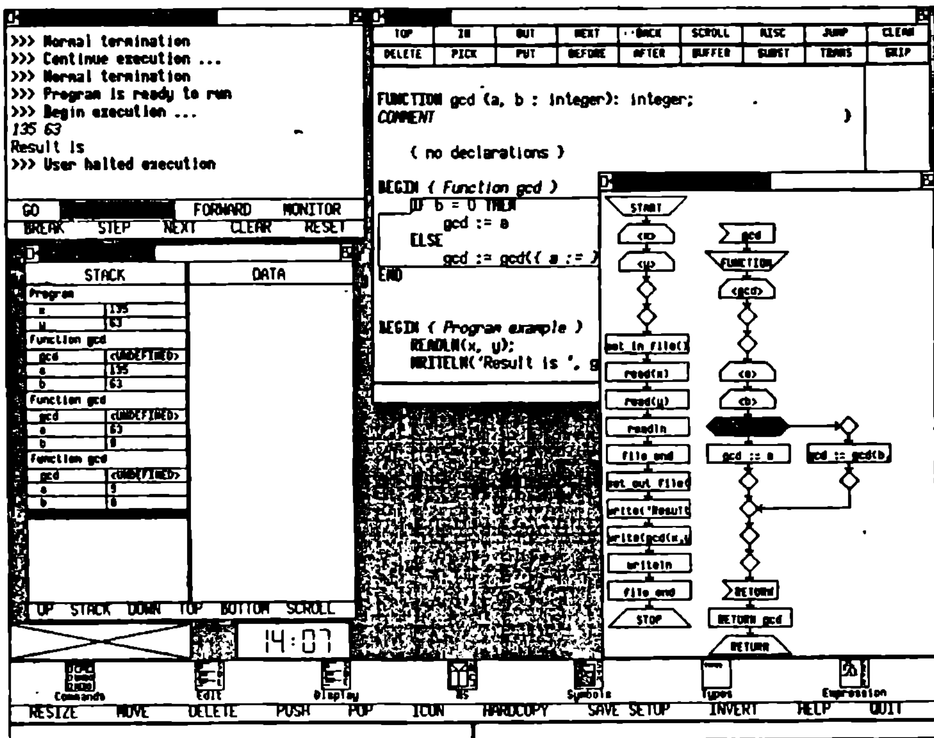


Figure 2-13: PECAN execution display.

The OMEGA system being built by Powell and Linton at UC Berkeley [Powell 83] is the system that in goals and design comes closest to our own. With OMEGA, the programmer will be able to interactively build programs

by pointing to graphical structures and move them around on the screen (fig 2-14). Multiple windows can be used to show different parts of the program and view the database of already defined entities. Central to this work has been the idea of separation of the objects stored by the system from their representation. Thus, the user can choose between several ways to display objects, and the input of objects can be done in a syntax-free manner by pointing to graphical icons. These icons ("pictographs") are visual abstractions of the code or data they represent. The actual details have to be specified in a textual form close to regular programming languages. The system also features a multi-threaded database that will enable the programmer to associate program components along several lines, e.g. same type, used in same module, executed after each other, etc.

While the designers of OMEGA recognize the virtues of graphical abstraction, they are still tied to a conventional view of programming. This prevents them from fully exploiting pictures; the old textual form is still there. It is also not clear whether the separation of objects from their representation actually helps the user. One of our slogans is "make the abstract concrete", meaning that the more concrete the objects seen on the screen are, the simpler will it be for the user to think about and manipulate them. We have therefore chosen to tie the objects and their representation together as closely as possible.

procedure readdata

```
initialize [|||||]
while not full [|||||] do
   tmp := new [|||||]
   { set contents of tmp here }
end while
```

B and not eof ○

read tmp from ○

glossary for readdata

[|||||] : aggregate

○ : file of text

tmp : element of [|||||]

B : boolean

files

read from file
write to file

data structures

fill data structure
sort [|||||]
sort ○→○

read from file

eof ○ : boolean
eoln ○ : boolean
read [ ] from ○
readln ○
reset ○

boolean operations

B and B : B
B or B : B
not B : B

Figure 2-14: OMEGA program display.

2.7. Summary

We have surveyed recent highlights in several related fields. Many different ideas are incorporated into new, powerful program development environments to make computer programming more manageable and available to a wider audience. Many of the same techniques are found in similar design tools, like CAD systems, and in major application areas like office information systems. The principles that explain why some systems are better to use than others are investigated within the field of human-computer interaction.

From this survey we can discern the present situation:

- Even though pictures are used in several systems, their nature has not been explored in its own right to determine their potentials and limitations.

- Still, there is a very clear trend toward an increased use of pictures, but without much guidance for the system designers. This trend is more based on available technology than clear human interface considerations.

- Nevertheless, some important notions, like immediate feedback and see-what-you-get, have surfaced. Once identified, they have a substantial influence on designers hungry for guidelines.

It is therefore essential that the use of pictures in several contexts be investigated as a topic of its own. This work is an attempt at relating pictures to programming more clearly than has been done in the past. We also observe that

- Both empirical research on programming interfaces and the wide distribution of microcomputers dictate that the main target for new developments in interface design should be directed at naive and casual users. These are the users that need and are most influenced by better interfaces.

Hence, we will focus our work on the kind of programming done by this user group. Finally, both our own and other researchers' experience indicate that

- The present (von Neumann) computer model is not particularly suitable, neither for teaching simple programming, nor as a basis for graphical displays. In contrast, simpler models, like the functional programming model, promise great advantages in both respects.

With these observations in mind, we can proceed to our own design.

Chapter Three

The form of pictures in programming

3.1. General guidelines

In the introduction we argued that there seem to exist many good reasons for expressing computer programs in a more pictorial fashion, and we will now try to identify more precisely what form these pictures should have, and what they should depict.

Simpler structures than computer programs have long had established graphical forms. For numbers, for example, the unit is identified with a certain amount of ink. By combining ink amounts corresponding to the numbers to form rods, pies, etc., we obtain a representation that makes it easy to compare the numbers. For mathematical functions of one argument the line is the metaphor that allows us to talk in terms of intersection, angles, curvature, etc., instead of more abstract concepts.

What real-world metaphors can fit programming? To get a feeling for how immense our initial search space is, let us amuse ourselves with a few possibilities. Is a program like a road with intersections and forks where we

have to make decisions depending on our current errand, and with roadside inns where we can spend computation time or, if we are careless, completely overflow our stack? Or shall we abolish the "meta-view" and, instead of seeing the program from the sky, put the programmer inside the program by presenting a view of a tunnel he/she can drive through? At each intersection one might have to throw dice or make guesses to introduce some of the challenge, fantasy and curiosity found in computer games [Malone 81]. Perhaps we should rather present programs as physical, three-dimensional objects that we can turn around and view from different angles and in different light so as to reveal the various facets of their structure. Can we present a program as a play with actors and objects on a stage? Before our imagination runs astray, let us start discussing the topic more systematically.

It is useful to start the discussion by examining how we use pictures in programming today. The only forms of graphical effect we can see in program listings are indentation to reveal block structure and delimiting lines inserted as comments. But even though the finished product is an encoding in some textual form, the process of creating it is usually sprinkled with illustrations of various kinds. When we think about an algorithm, or explain it to someone, what kind of images do we support our explanation with? We draw pictures of the data structures, and show how the algorithm works by

pointing to the data, drawing arrows and filling in example numbers. For instance, when explaining the Quicksort algorithm, we draw an array and show how it is split by choosing some example numbers and indicating by arrows the swapping of elements. Only when the algorithm is understood do we take the pains to formulate it in a programming language, a form that very precisely defines the algorithm, but which is not very suitable for thinking about it.

There are a few points worth noticing here. First, the pictures that we draw when we have complete freedom to illustrate exactly the aspects of programs that we are interested in, and in exactly the way we like, are probably closer to the internal, mental images we talked about in the introduction. If we tried to base our pictorial programming system on this kind of pictures, we would considerably narrow the concept-representation gap that is presently making programming such an arduous task. Second, we notice that by far the most commonly depicted structure is *data*. If the pictures we like to put up on the blackboard are of our data, doesn't it make sense to attempt to focus a pictorial programming system around graphical data descriptions, too? Third, as soon as we have to deal with non-trivial data structures, we have a strong tendency to support our explanation with *examples* rather than general pictures. Fourth, the pictures are, if not fully dynamic, at least

semi-dynamic, in that we constantly modify them during the explanation, and show different aspects of the pictures at different times. That they are not fully dynamic is probably an implication of limitations in the medium (paper or blackboard) rather than a result of preference. Fifth, the pictures by themselves usually do not contain enough information to document the whole algorithm. The hand-waving and spoken words that go with them complement them.

With these indications of what we are looking for in a good way to display programs, let us explore some options.

3.1.1. What aspects of programs shall we show?

Primary aspects

There are several aspects to a computer program that we may want to depict graphically in some way. The most important ones are:

- Control flow

- Data flow

- Data structure

- Topology

It is not immediately obvious which one of these, or which combination of them, best lends itself to graphical display, and which is best for the programmer to work with in a pictorial way. Pictures allow us to show all interesting aspects of programs simultaneously. But although it is true that the more we see the easier it will become to discover relationships and spot errors, it would be useful to find out whether there is one aspect that the presentation and the programmer can focus on, letting the other views build upon that aspect. Let us examine each of the components in turn.

Control flow. In chapter 2 we saw several examples of control flow display, including flow charts, structured charts, state diagrams, and Petri nets. Indeed, graph or network type diagrams have been the predominant way to describe programs pictorially. We also gave in chapter 2 the primary benefits and drawbacks of such illustrations: If kept simple, they can provide a very clear view of how control proceeds, including a lucid presentation of parallelism. They are also a good base for animation. On the negative side, they easily tend to get unstructured and confusing. Moreover, they represent only the control aspect of the programs. It seems to be hard to include other important aspects, like data structure, in the diagrams.

Data flow. The only use of data flow diagrams has been for the dataflow model of computation. Since in this model data and control flow are identical, what we said about control flow diagrams applies equally well here.

Data structure. Curiously, there is not much work on formal or standardized ways to display data structures. Even though it is the data that often contain the graphically most interesting structures in a program, their picture usually remains hidden and must be illustrated manually by the programmer. There are some possible reasons for this. First, the data can be rather complex and its content and shape change dynamically. Sophisticated display technology is required to maintain updated pictures, and it is not always obvious which parts of a large structure to display. A hierarchical schema of code or data can always be displayed neatly one level/branch at a time, but the *instances* of the schema can easily get unwieldy. Second, we tend to think about our data at many levels of abstraction, and the display should reflect this. For example, a graph may be depicted as a network of nodes and arcs in one place, but as a connection matrix in another. This is of course what data type abstraction is all about. However, the translation between the different views is not trivial to do since it involves graphically interpreting the mapping between specification and representation of the abstraction, a mapping which is usually only implicitly defined via a collection of operators.

It is still the case that a major part of the program illustrations we make are of data structures. In fact, programming often starts by defining the data structures and then proceeds by building the procedures around them ([Sandewall 78], p 44). With the development of abstract data types and object-oriented programming the emphasis on data as semantically interesting pictures around which the algorithm revolves has only increased. It therefore seems that work on data-oriented program display is an important, needed contribution.

Topology. Diagrams of program structure can only capture the *static* interrelationships within a program. The more detailed a level of programming we are considering, the more interested we get in the *dynamic* behavior. Topological diagrams can therefore be useful as a background framework, but the diagrams central to programming-in-the-small must focus on other aspects.

If we now look back at what we said earlier about the nature of pictures and how this relates to programming, it is not hard to discover several points leading in the direction of data structure display. The *mental images* we use and the drawings we make to support our programming are mostly concerned

with data. This is because we like to work with *concrete* concepts. Data are concrete, they have a layout in the computer, and they are usually associated with the *real world* in a much more direct way than control structures. Control does not have a natural picture, it is an abstract concept that we at most can help make somewhat more concrete by attaching abstract pictures to it. Finally, even though programs are *dynamic* objects, it is hard to manipulate something which is constantly in motion, so we would like to find a representation that is at least semi-static. Data are the most *static* aspect of programming-in-the-small, so by basing our system on data display, we can provide a user interface consisting primarily of static pictures without artificially binding the dynamic aspects of programs to static images. Control and data flow are then best animated on these pictures instead of being the subject of static pictures by themselves.

Despite the difficulties associated with data structure display described above, we will therefore here develop a system for programming in pictures based on the display of data structures.

Secondary aspects

Orthogonally to the primary aspects of programs discussed above, there are several more aspects that we usually do not see displayed anywhere, but that

a pictorial programming system can let us examine in detail, thereby providing important information about a program.

Correctness. The main obstacle when searching for errors in a program is simply that we cannot see what the program does in detail. We have to rely on what we *think* it does, but this is not always correct. A program *debugger* can assist us simply by allowing us to observe more of the program in action. A pictorial system showing how the data is modified and how control proceeds can similarly make program operation more lucid and thereby simplify correction.

Performance. An animated display where the time spent in each program component always is in the same proportion to the total program time can give valuable information on where most of the execution time is spent. A good display of how data expands and contracts likewise shows space requirements. These measures have traditionally been very hard to observe, but a good pictorial system can make them a self-evident part of a programmer's knowledge of a program, simplifying code optimization.

Parallelism. A performance measure that is very important in certain kinds of programming is parallelism. The best way to give the programmer an idea of the parallelism obtained by his/her design is probably to show the program in operation, highlighting the multiple concurrent actions graphically.

Typing. The *type* of a program object is usually a feature which cannot be deduced simply by looking at it (e.g. looking at a variable name[1]), although it is an important attribute that can cause irritating errors if misjudged. In real life, we are usually able to tell the difference between apples and oranges, and if they are packed in cartons, these are usually clearly marked. Similarly, our objects should be marked in programming, and with a pictorial system we are able to do this, for example through the use of shapes or colors.

Abstraction. We would like our programming system to advocate a view of the software as a hierarchical structure of neat abstractions. In conventional languages, we can syntactically "support" this view by enforcing certain access rights within the program text. But the programmer, who is not supposed to think too much about the implementation of an abstraction when he/she uses it can probably see it just by flipping the page! If, on the other hand, we integrate the language with the programming system, we can unify the access path and the abstraction path, and let the system embody the desired view rather than merely support it. A pictorial system is the best vehicle for this, since it imposes no constraints on the graphical interpretation of program navigation. For example, we can obtain the underlying implementation of a piece of abstract data by expanding its picture with a kind of zoom-effect.

---

[1] In retrospect, FORTRAN's variable naming convention was perhaps not such a bad idea.

3.1.2. How shall we display it?

Having decided on data structures as the basis for program display, we now make several more comments about how we can exploit the features of pictures in this connection.

Multi-dimensionality. There are several aspects to programs, and the optimum display would probably show all of these at once and also how they are interrelated. This is extremely hard to do in one picture without making the result more confusing than illuminating, but the multi-dimensionality of pictures can help us to some extent here. For the best way to display several things at once without having them interfere with each other is to show each aspect along its own dimension, orthogonally to the others. With pictures, we have two dimensions to begin with, but with the proper display techniques and technology we can achieve several more. The matching of program aspects to dimensions is an important design issue in that the decisions we make can influence how the user thinks about programming. Lacking stronger guidelines than that we should try to minimize surprises and support people's preconceptions [Jones 78], we have based our design on the following, which we find intuitively appealing:

- We are used to laying out data graphically in the plane, so we need these two dimensions to show good illustrations of data. Since these are

the most obvious dimensions of plane pictures, this is also in keeping with our intent to focus the display on the data structures.

- Control and data flow are dynamic aspects, so the natural dimension to project these along is time. This means that we will promote animation to an integral part of program display, instead of just leaving it as a debugging aid. To make it easy to compose and modify programs we probably have to give some static representation of the control and data flow, but the way to study these aspects should still be by animation.

- This leaves the third spatial dimension, which we can conveniently utilize for hierarchy (topology), both for data and program components. One way to achieve this is to show essentially two-dimensional pictures of programs and data, and use a zoom-effect to "open up" more detailed structure underneath or to pan back for an overview.

- With a color display, we actually have a fifth dimension, which we can suitably use for data types or other category information. It is interesting to compare this to how colors are used in other fields to reveal attributes that are otherwise not visible, without changing any other attributes (e.g. red/blue marks on hot/cold water taps, color codes on gas bottles, painted curbs as parking restrictions).

Real world semantics. We talked earlier about how pictures are rich in metaphorical content, in that they tend to make connections to the real world, and how this represents both an opportunity and an obstacle when we want to use pictures to express programs. On one hand, the pictures we use should give associations about the high-level application domain to aid in program understanding by reducing the level of abstraction. On the other hand, it is difficult to attach too much new semantics to shapes, since most pictures already have a large amount of meaning associated with them and it is hard to restrict the interpretation. The key seems to be to utilize the semantics that is already there and be careful about illustrating new concepts with more than trivial, abstract figures.

Data structures model objects in the real world, so this is the part that naturally lends itself to illustration by real world pictures. Control now becomes actions that we perform on these real world illustrations, but actions that often do not have real world counterparts and therefore are best represented as abstract pictures or animation sequences. So we introduce new things that can be done with familiar pictures rather than new interesting pictures. By letting the programmer draw arbitrary pictures of data and using these in a framework determined by the system, we allow the human to attach real world semantics precisely where the interface to the real world is

the strongest. We then rely on the automated system to present pictures of the abstract computational structures that are the only aspects the machine really can know about.

This provides a reasonable division of labor, but it also ensures that the programmer's efforts are minimized. Only the aspects of the program that belong outside the machine world have to be illustrated by the programmer. The system can further aid the user by reusing pictures as much as possible, e.g. by using the picture of a data type to describe all objects of that type. Tying the user-defined data pictures together in a standard framework also has the advantage of rendering the program easily readable by those other than the program author.

Other graphical programming systems have invariably failed to include more than an abstract, uninteresting set of pictures to describe part of a program. They have therefore been unable to profit from much of the strength of the graphical medium.

Use of real world semantics gets more appropriate the higher the level of abstraction we consider. At the lowest level this benefit from pictures is therefore reduced,[2] but the freedom to view data as any graphical shape still remains.

[2] Just as Smalltalk's object-message view gets artificial at the primitive level.

No names. Pictures remove most needs for naming, and we must exploit this to simplify programming. This means that all objects the programmer may wish to refer to must be identifiable by pointing. A name is a very compact, convenient way to refer to an object that is not present. Thus, names may still exist as an option to be used to retrieve objects temporarily out of view, but no subtask must rely on the existence of unique names. In fact, names should be a mere attribute, not necessarily unique. Objects with the same name can still be distinguished by their structure, and selected by pointing.

Animation. We have already established the connection between control and data flow, and animation. While data provide the static base for our program displays, control and data flow are the dynamic aspects that should be shown essentially by animation on the data display. We are now in a position to expand in more detail on exactly how picture sequences can be used to illustrate the dynamics of programs.

Program execution. In a data-based display environment, the simplest way to visualize program execution is to repeatedly update the data pictures as changes occur. This gives a very good understanding of the effect of the program, but it does not say much about how control and data flow between different parts of the program, i.e. about the anatomy of the program. In a

flowchart or dataflow graph we can follow the flows through a picture of the program structure, but this is at the cost of completely losing the picture of the corresponding data. We can, however, invert the situation: By *surrounding the data display with the context in which it is currently being processed*, we get complete animation of how and why control and data flow through the program, and at the same time a full and undisturbed display of how the data change. By *context* we here mean an indication as to which procedure is currently in control of the data.[3] This can be just a simple name or icon hinting at the procedure in question. It can also be a more involved display showing by means of arrows and other figure elements how the procedure actually works, something akin to our style of explaining programs on the blackboard.

This solution gives us a fairly local view of the flows that may not be adequate to obtain a global overview of the program. The third dimension comes to our rescue, however, and in a fairly elegant way. We mentioned above that we would use depth to show hierarchical structure of data and programs. If we pan back from a view showing the data and its immediate context, what do we get? The data will in a sense get smaller and smaller,

---

[3] In a concurrent programming environment we automatically get a graphical constraint that only one process can access shared data at a time, since the data can only be surrounded by one context.

until only their top level structure is visible. Then, the context will get larger and larger, but this means that we show more and more of the program structure and less and less of the data. On the other hand, if we zoom in, we will see more details of the data, but only a small piece of context. We have therefore obtained a unification of data and program structure display, planar pictures tied together via movement along the depth axis. Since control and data flows are easily animated along program structure, we can use this solution to show all aspects of programs in a unified way, combining the views by carefully exploiting the various dimensions of pictures.

Note that a similar dual display would not be feasible if based on a flowchart or dataflow graph type display. First, we would violate our observation that the data are the centerpiece around which the rest of the program revolves. Also, unless we disconnect the display of data from the program structure and show it in a separate window, we are forced to animate pieces of data as control and data flow proceed, thereby animating the most static aspect of the program and leaving the dynamic aspects as a static picture. Second, while control follows a structured tour through program pieces in a manner that is usually easy to follow without a global display of the complete program, data accesses are usually much more random in nature, jumping back and forth between different sub-structures in a fashion that would make a solution showing only a partial data display very hard to follow.

The above solution, unifying the most interesting aspects of a program in one multi-dimensional picture centered around the display of data, has an obvious alternative. Why can't we show the various aspects of a program as separate illustrations in different windows on the same screen? This would disentangle the pictures and give us full freedom to look at the aspect that is of interest at the moment. One problem with this solution is precisely its strength: It separates the aspects, leaving it up to the viewer to find out how they are interconnected, even though it is these interrelationships that make up the soul of the program.

The most important reason that we will not pursue this solution here is more philosophical, however. We are trying to make programming more accessible by making it more concrete. If we show a program as several different pictures, we fail to bring the object we are working on completely out in the open. We merely provide a few peepholes into a reality we cannot grasp in its entirety. The "program proper", whatever it is, remains hidden and we only see its traces. If, on the other hand, we manage to unify the views into one picture, we have created an artificial object that becomes real. The program is *one object* that we can "touch and feel" and manipulate and think about as an object appearing before our eyes.

Animated writing. While program execution is the most obvious target for

animation, we must not forget the other ways in which we interact with programs. Program writing and modification are especially demanding with respect to the insight required, so these are areas where proper system support can be very helpful.

Since a program executes dynamically, we can argue that we should also create it by showing the computer in the same dynamic way how each piece of data is computed. In the light of what we have developed above, this means that we would like to specify how the program works by manipulating example data much in the same way we do when we explain a program on the blackboard. This has some implications:

- The data we use for program specification are *examples*. This is at least true for non-trivial data structures where we cannot simply represent a piece of data as a simple box or icon, but have to work on an instance of a more complex structure.

- To specify the program, we would like to interact with the system by pointing to the structures of interest and stating in a simple, direct way (e.g. by menu selection) the operations to be performed on them. This would replace the hand-waving that accompanies pictures of data on the blackboard.

- The program itself will thus not be "written" in the usual sense. Rather, the specification actions will have to be represented by graphical constructs imposed on the data pictures, e.g. like arcs connecting the pieces of data that are involved in a computation.

We refer to this style of programming as *animated writing* because, rather than giving a static specification, the programmer *does* the program as a sequence of actions. If example data are displayed, we might even see the effect of the program piece immediately on the example data. The style is a direct consequence of our previous decisions about animated data displays, and is an important part of a coherent approach to programming in pictures.

Reading by execution. Since we cannot appreciate a program fully without observing its dynamic behavior, executing the program should be part of any reading effort. To get an overall impression of what the program does, we could watch its execution in its entirety. However, when we look at the program in detail, we are usually not interested in observing the effect of the whole program. Each part of the program has its own dynamic attributes, so we would like to be able to execute selectively the piece we are inspecting. This implies a few points:

- As for writing, example data should be available when reading. These could be created by the reader or supplied by the system.

- The effort associated with triggering the execution of a piece of program must be minimized. Remember that all attributes of the program should ideally be equally easily observable. This means that a trivial action, like pointing, is all that should be necessary to invoke the execution.

- Since the central part of the display is the data, these are the objects that it is most natural to point at in this connection. This means that the execution will be data-oriented, in that it will show how a certain piece of data is computed or how it is used to compute other data.

By *reading by execution* we mean a style of program appreciation in which the reader can look at the various pieces of a program and by "touching" them can actually see them in operation. Only through such an interface can all aspects of the program be easily observed. What distinguishes this from simple program animation is the locality. Reading by execution means that any program piece can be picked up and looked at and executed in a purely local context. We also note that, since reading is the inverse of writing, we can expect to see much the same pictures as those we created when specifying the program. That is, the graphical structures we used to show how the data got connected can be used here, too, to indicate how data are obtained.

3.1.3. Summary of program display issues

In the introduction we investigated pictures in general as they compare to text, and we have here shown how their characteristics can benefit computer programming. Our main observations are:

- The random access, rich language, and high transfer rate of pictures make them well suited for an interaction-intensive task like programming.

- The concreteness of pictures allows us to reduce the abstractness of programming and make it simpler and more accessible to non-experts.

- Pictures are windows into the real world and this can be utilized to put more interesting semantics into the program representation.

- Pictures should be dynamic to capture the execution aspect of programs.

- Pictures obviate the need for indirect references through names.

- The many dimensions of pictures can be utilized to unify many aspects of a program into one view.

- Pictures present a rich medium for aesthetic exploration.

By looking at these characteristics, and at how people like to illustrate their explanation of programs, we have arrived at a design based on:

- Data-oriented pictures.

- Multi-dimensional, unified view manipulated by pointing.

- Free form pictures input by the programmer in an abstract framework supplied by the system.

- Animation-based program reading and writing.

3.2. Key decisions

For the idea of programming in pictures to have any practical value, we have to construct programming systems that implement the concept. When we build a concrete system, we are forced to make choices among a large number of possible incarnations of the general ideas we have developed. In doing so, we make it difficult to determine whether the merits of the finished system stem from the principle of programming in pictures, or whether we just have made clever implementation choices. We would really have to construct a whole family of systems, implemented along different lines, to be able to

factor out the effects of implementation. There are two levels of implementation choices that have to be made:

1. We have to choose a *computational model*. The model is the basis for the underlying semantics of the programming system. It also determines all the higher-level concepts that the user will form about the system. Thus, the choice of computational model is an important one. If the underlying model is not simple and well-defined, and does not promote concepts that are compatible with the ideas of programming in pictures, the benefits of these ideas will never materialize.

2. Having found a suitable computational model, we have to decide on its *representation*. This is the focus of our work here. The preceding section formulated guidelines for how we can benefit by using pictures in programming, and we now have to engineer a specific design that violates as few as these principles as possible. The search space is vast, but with the basis developed in the previous section, it will not be as vast as we may think.

This research includes the implementation of a simple system for programming in pictures, hereafter referred to as the PiP system. As indicated above, we will not be able to draw firm conclusions from this, but

we nevertheless see it as a very valuable first step in experimenting with the practicality of programming in pictures. This section justifies the key decisions we had to make regarding the computational model and the graphical representation. The next chapters describe in detail the particular implementation we have undertaken.

3.3. The computational model

The imperative von Neumann computer model is overwhelmingly the most used computer model today, whether we consider hardware implementations or research work. It was therefore quite natural for us to first investigate this model as a basis for programming in pictures. Our initial attempts were not successful, however, and it did not take long before we realized that this model is not very well suited for the purpose.

There are several important reasons why this model fails: First, it is a rather messy world, with many intricate concepts and mechanisms that would clutter the otherwise simple idea of programming in pictures. Second, the visibility rules in most von Neumann programming languages do not easily lend themselves to pictorial representation. The amount of data visible from one place in the code is usually large.[4] Third, the strict sequentiality of the

[4]This is of course a direct consequence of the von Neumann model's centerpiece, the large, complex memory state.

imperative model is not compatible with pictures, where there is usually no sequential order imposed on various picture elements visible in a two-dimensional plane.

Indeed, it seems quite fitting that we should encounter problems in attempting to force a novel user interface on an old-fashioned model. Choosing a new model also has the advantage of forcing even experienced programmers to take a whole new look at what programming is about, rather than just mapping the pictures to their predefined view of this activity.

We therefore set out to locate another existing model, more suited to the task. Backus' Functional Programming model was finally chosen as the best candidate. This model is fully described in [Backus 78], and we have included a brief overview in appendix A.

The salient features of FP for our purpose can be summed up as follows:

Simplicity. "Things should be as simple as possible, but not simpler", goes Einstein's famous quotation. In this connection, we would like our model to be simple for several reasons. First, we do not want issues irrelevant to programming in pictures to disturb our discussion. Second, we have said that we are targeting relatively naive or casual users, since these are the people

most likely to benefit from this work. Third, one of the principles we build upon is that the display should explicitly show all the aspects of the system that the user may want to think about. This is to enable thinking in terms of concrete metaphors and to identify the model with the display. The complexity of the model therefore becomes critical since a complicated model will easily clutter the display with too much detail. The FP model is simple, but at the same time it is an established, powerful computational model.

Unified data and control flows. One way FP achieves its simplicity is by using data flow as the basis for control. This is of particular importance here, since it relieves us of the need to display two separate aspects of programs that are usually rather difficult to combine in one view.

No global data. It can be difficult to keep a display of all the data that are accessible from a given point in an ALGOL-like language, both because of the amount and because of its dynamic variation. In FP, each function can only access one input and one output data structure, again simplifying the display. It is also good for the reading-by-execution technique, since only local data need be filled in.

No implied sequentiality. As we said above, pictures do not, unlike text, imply sequence. FP builds functions as algebraic expressions. The FP

concept closest to sequence is functional composition, but, in contrast to the sequence of von Neumann languages, it is explicitly stated. It is also used in a more coarsely grained manner. This makes it much easier to handle graphically. FP functions are also often evaluated in parallel, making the program structure less linear and more graphically interesting than von Neumann programs.

Few names. FP does not name its function arguments. In fact, data objects are in general nameless, making FP very suitable for an environment based on pointing at known objects rather than naming them. This lack of names comes from the locality of data. Functions are still referred to by name since they are selected from a global library.

No parameter substitution. Parameter substitution is a powerful, but intricate mechanism, hard to describe both mathematically and graphically. FP is one of the very few models that does not use parameter substitution, and it therefore again simplifies our display.

Small program components. FP encourages splitting up a program into many, very small components, since the overhead in setting up a function definition is minimal. This means that the components we display are usually very simple, both benefiting program understanding and display clarity.

In addition, the simple, almost trivial, FP data structures, and the single function argument, contribute to a simple interface that allows us to focus on issues regarding pictures in programming, like building high-level data semantics into the pictures. FP has almost no syntax as defined by Backus, so we are not constrained in any way to design our pictorial interface. Lastly, the precise algebraic definition of FP ensures a coherent underlying model that is straightforward to implement and simple to learn. If it is easy to handle mathematically, it is probably easy to think about, too.

Other programming models, such as pure LISP and logic programming, do not contribute anything that FP does not. Indeed, the parameter passing mechanism of lambda calculus and the conversational style of logic programming, with a large database of rules and facts, make them less suited for our task than FP.

One may of course ask why we should pick an existing model at all. If the pictures we present to the user are so central, couldn't we start by designing a graphical, algorithmic world and then come up with a computational model for it? Besides causing us extra work, there are reasons why this would really be to put the cart before the horse. We do not think that we can anthropomorphize computers completely. Computers are inherently new. Rather, we are investigating a way to make the burden of metaphorically

extending people's knowledge to cover computers less formidable. Thus, the abstract computational model is always there, and we might as well pick a good one. If we want a good total solution, finding the model before the pictures instead of the other way around limits the search space considerably.

3.4. Graphical representation

The first section of this chapter laid down the guidelines for how we shall utilize pictures in programming. Having chosen our computational model, we can now decide how the concepts of the model map into concrete pictures.

3.4.1. Data layout

As suggested in the previous section, we will lay out the data objects in the plane. But rather than supplying predefined formats for data, we will leave it to the programmer to design the data representation using a free format picture editor. This way the user can include as much semantic information about the application domain as desired. Since data are hierarchical structures, we will use the depth axis to show levels of detail through a zoom mechanism. So the recursive sequences handled by FP are translated into hierarchical free form diagrams with figure elements representing each element of the sequence.

These pictures of data are to be used as descriptions of objects operated on by

functions. In drawing these pictures, we really define data *types*, since a function works for a whole class of objects rather than just one specific object. Drawing data types is also much more economical than drawing data objects, since the drawings can be reused for each object. Moreover, much of the semantic information that we associate with an object really belongs to its type description rather than to its value. If we want to express object-specific information graphically, we should include a separate data type *picture* whose objects take graphical values.

We have therefore included a typing mechanism in our system. It should be clear that the main reason for this is to allow the programmer to capture semantic knowledge about the application domain and express it directly in the program. In addition, we can enjoy all the traditional benefits that a typing mechanism gives us (increased reliability and more efficient correction through automated compatibility checking; clearer structure by forcing the programmer to classify data and operations). We will comment further on our typing scheme in a later chapter.

The equipment at our disposal for this project does not include a color display. We can therefore not utilize color to show object types, so this has to be indicated by other means. In our system, a simple zoom operation will reveal the structure of a piece of data and thereby also its type.

3.4.2. Function layout

A function is next defined by animated writing on a picture of its data. In FP, a function maps an input data object to an output object. It is therefore convenient to display a picture of the function's input data and a picture of its output data and then show by means of pointing actions how the output is obtained from the input. Similarly, reading by execution will show how the input is combined to yield the output. The pictures shown are those of the data *types* in question, and a context of arrows and predefined operators showing how the data are used, i.e. the data flow.

As we saw earlier in this chapter, this gives a *local* view of the function, suitable for fairly low-level programming where each part of the data structure is involved in small computations. We also saw how we could use the depth dimension to unify views of data and program structure. Our system implements this idea by providing a second level of program composition. If we pan back from the data-oriented view just described, the next higher level of abstraction will be an iconic picture of the function defined, rather than a picture of data (we had reached the top level of data abstraction anyway). Other function icons can then be brought into the picture and combined in various ways. This is where the FP *functional forms* enter the game. These are precisely the operators we have for

combining functions. The function icons are free form drawings allowing the programmer to attach real world semantics to the functions as well as the data.

Thus, we have a two-level system. At the low level we specify simple functions by animated writing on data. We refer to this level as *object-level* function editing since we work in terms of the data objects. At the high level we combine predefined functions into new ones via functional forms. We refer to this level as *function-level* editing since we work in terms of functions. The two levels visually unify data and function hierarchy along the depth axis, since the data structure can be inspected by zooming the data at the object level, and the function structure can be examined by panning back into the function level.

These are the basic ideas on which we have built our implementation. The next chapters will describe in detail how it is actually done.

Chapter Four

A system for programming in pictures

4.1. System overview

The preceding chapter established the computational and graphical bases for our system, the PiP system. The implementation incorporates these ideas into a dialogue style that has much in common with contemporary menu, window, and mouse based systems. Our system is an interactive, graphically oriented computer programming system. That is, by drawing pictures on a terminal screen, pointing at the pictures and moving them around, the user can compose structures which the computer will translate and execute as programs. The system assumes a high-resolution bit-mapped graphics screen and a pointing device (like a mouse) for normal interaction, as well as an ordinary keyboard for entering names and numbers.

Nothing but a live demonstration can fully describe all aspects of such a dynamic design, but there is nevertheless much that can be said via text and static pictures. The description that follows is the user's view, since this is what is of primary concern to us here. We will describe implementation issues only when they help to clarify matters.

The system consists of four main tools, each designed to handle one kind of object that the user will have to manipulate. In FP, the center of attention is the function, so it is not surprising that one of these tools is dedicated to the creation, modification and execution of functions. We want to provide both object level and function level editing, so we have found it convenient to actually supply two tools for function manipulation, one for each editing level. We refer to these tools as the *function editors*. We decided to include a typing mechanism to reap the well-known benefits of type checking, but also as a facility for increasing the use of pictures. Building types is therefore an important activity requiring its own tool, the *type editor*. Both functions and types use pictures in their definitions, and we have to provide some means of creating these. This is a task that is completely orthogonal to any language construct, so we have included a separate *picture editor* for this purpose.

For a multi-tool system like this to work smoothly in practice, we have to consider carefully how control and data are transferred among the components. In the PiP system, the tools can be "picked up" and "laid down" in any random sequence, i.e. any tool can be invoked from another. Furthermore, when reentering a tool after using another one, the situation is exactly as when the tool was last exited. There is only one incarnation of each tool available. This configuration gives a very convenient and

conceptually simple total system. The system consists of four "boxes" that can be entered and exited, and there are no surprises (hidden actions) connected to the switching between tools. Each tool is available at the touch of a menu icon.

The four tools produce functions, types and pictures, and can pass these among each other via a *scratchpad* that is accessible (and visible on the screen, of course) from all four tools.

Notice that the system only provides *editors*, and that function execution is performed through the function editors. This is a result of our view that program behavior is just another program attribute that should be observable during editing just as easily as other attributes.

Each tool provides a template, a simple frame structure, that the programmer can fill in. For the picture editor, the frame is a blank canvas that the user can draw any picture on. For the type editor, it is a blank field that the user can fill in with pictures (created via the picture editor) representing data elements. At the same time, type information is attached to each picture. For the object-level function editor, a double frame is provided, with one field to depict the input data and one for the output (using type pictures created via the type editor). For the function-level editor, built-in functional forms can be used to combine pictures representing functions within a frame.

When the system is first invoked, the tool menu is shown (figure 4-1). The icons represent the following commands (bottom to top): exit system, picture editor, type editor, object level function editor, function level function editor, cancellation, and a help facility. When a tool is entered, the tool menu is still visible on the screen, so that any tool can be invoked at any stage of using another tool. It is not possible simply to exit a tool without entering another one at the same time. Selecting the *exit* icon on the tool menu will exit the whole system.

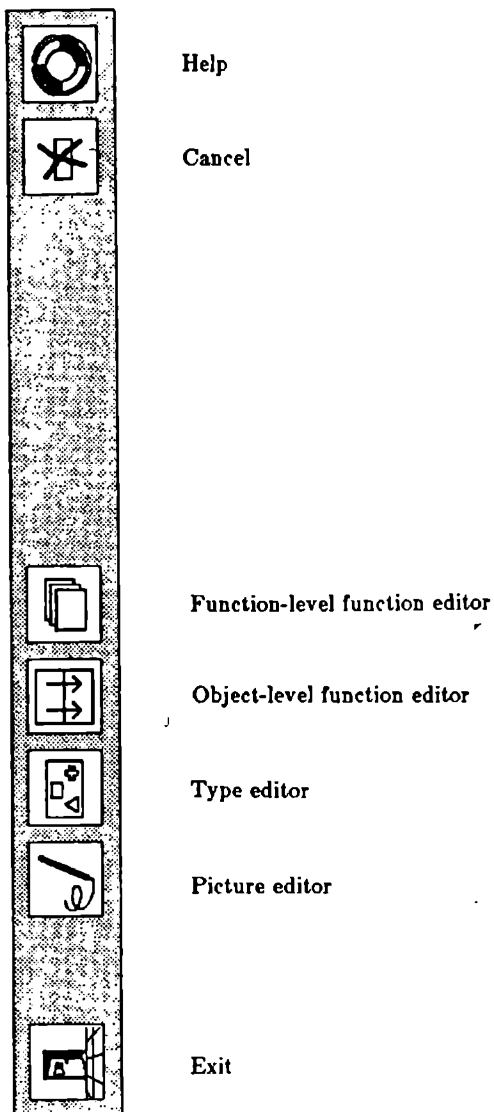
4.2. Systemwide concepts

4.2.1. Mouse use

The system is based on a three-button mouse. The buttons have the following assignments:

- The left button is used for all picking and drawing. This button is henceforth referred to as the **select** button. Using one button for all normal functions reduces confusion and minimizes errors. Indeed, the system does not really need more than this one button, but since our computer is equipped with a three-button mouse, we have assigned some simple functions to the others as well. Only areas of the screen that are sensible to select are sensitive to the mouse.

Figure 4-1: The toolmenu (legend added).



- The sole purpose of the right button is window overlap control. Since the system can display several windows at the same time (for example, a function definition and the scratchpad), some windows can overlap others. All available windows will have at least some part visible, though, and by positioning the cursor on part of a window and clicking the right button, the **top** button, the window selected will be moved to the top of the stack of windows. Pushing the *top* button does not affect the state of the system; it is purely a display function and can be used at any time.

- The middle button is another display facility. If the cursor is positioned over a data structure that has an underlying, refined structure, pushing the middle button, the **zoom** button, will display the next refinement level of the data selected. The *zoom* button can be pushed again to reveal even deeper levels of hierarchy. By double-clicking the *zoom* button (two clicks within a short time), the reverse operation will be performed, panning back to the next higher level of hierarchy. As with the *top* button, the state of the system is not affected by this facility.

4.2.2. Command parameter input

We mentioned earlier how FP avoids parameter substitution, and how this helps simplify the graphical representation of functions. For the system *command interaction*, however, there is still a need for specifying parameters. For example, when issuing the command that deletes a type element, we have to indicate which element to delete.

Parameter specification is a tricky aspect of human-computer interaction. The number of possible values that are acceptable is usually so large that a menu solution is unthinkable. Yet, there are usually so many constraints the parameter value has to meet that it is futile for the user simply to guess a format or sequence. In conventional systems, explanatory text is therefore often displayed to aid the user, triggered either by the user him/herself or by incorrect usage.

How can we utilize the graphical medium to make parameter specification less painful? This question fits into what we have said earlier about the reduced significance of names, and the opportunity to show a larger portion of the system state on the screen than is commonly done. By the former, we mean that when an object is shown on the display, we can refer to it by pointing instead of through a name. This removes the obstacles regarding special name formats. By the latter, we want to indicate that by displaying the

relevant parts of the system state, we reduce the number of choices significantly, making it easier to pick the right parameter values. In essence, we present the whole screen as a menu for parameter input. It is possible to keep a working set displayed in this manner, even if a full menu approach to parameter input usually is infeasible.

Still, there are cases where more support is needed. For example, when more than one parameter is to be supplied, it may be obvious which parameters should be given, but not in which order (or if it matters at all). Also, some commands might be applicable only to a subset of the state displayed, and it may not be entirely clear to the user what the subset is. For these reasons, the parameter input style of our system is as follows: Whenever the system expects an input, a region of the screen is highlighted, corresponding to the region of possible inputs. Outside this region the pointing device will have no effect. When the parameter is given, the highlighting is turned off as a feedback. This is a graphical form of prompting. It gives more support than textual prompting, though, because it also presents a menu of possible values within the indicated region.

4.2.3. The scratchpad

We mentioned the scratchpad as the vehicle for communication between the editors. Since the display space after all is limited, and we would like to have ample space both on the scratchpad and in the editors, the scratchpad is displayed overlapping part of the editors. The system will attempt to find out when the scratchpad is needed, and will display it only when it is being used (e.g. when a new object is put on the pad). The user can easily hide the pad and bring it back again by using the *top* button, as explained above. The scratchpad has a *clear* "button" in the upper right corner, i.e. a field that can be selected to clear its contents.

4.2.4. Global commands

In addition to the commands leading to the selection of an editor, there are a few commands visible on the main menu.

- **Help facility.** The system is easy to use and guides the user to a large extent, but if at any time the user does not know what to do, pushing the *help* icon will display a window with some explanation. Pushing the icon again will cause the help window to disappear, or, if needed, will page through the explanatory material. This facility does not affect the current state of the system.

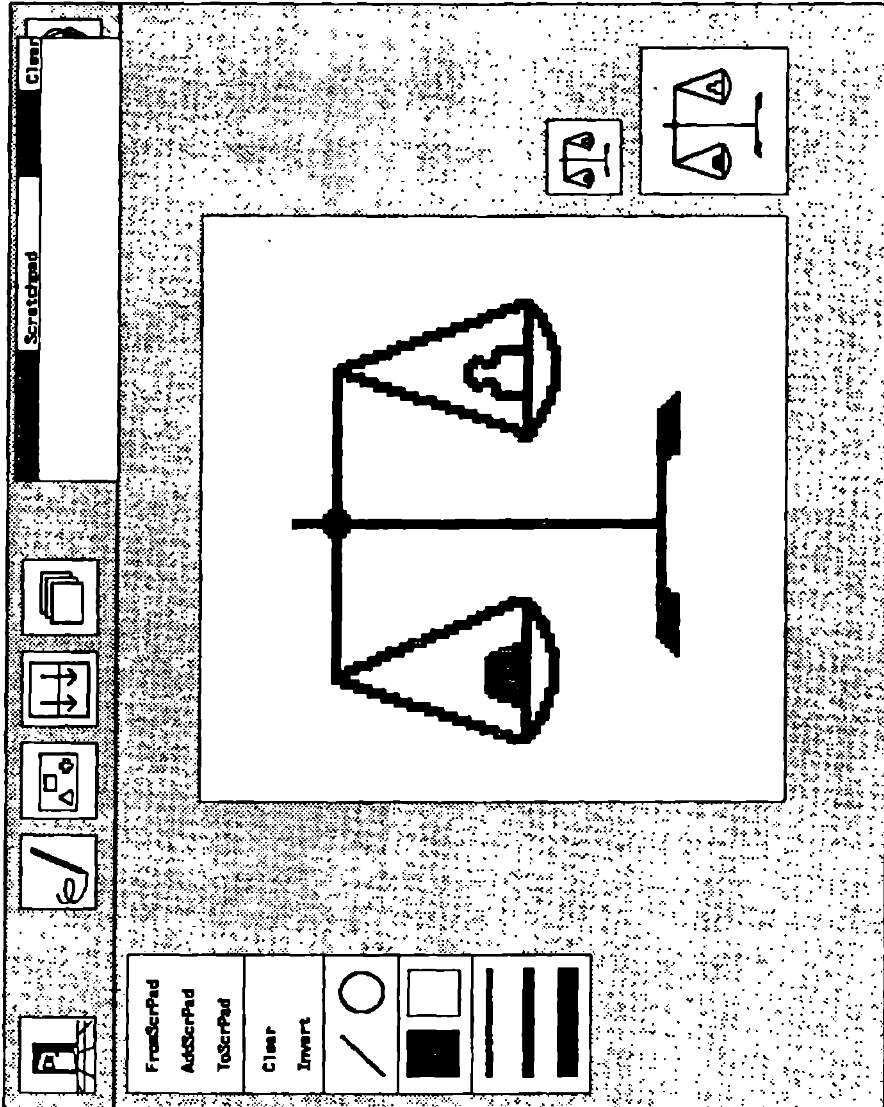
- **Command cancelling.** By pushing the *cancel* icon, the current command is aborted, and the command state of the current editor is brought back to the same as when the editor is entered (main menu for the active tool). The state of the object being manipulated by the editor is not changed.

4.3. Drawing pictures

Figure 4-2 shows the screen layout of the picture editor. Briefly, the mouse is used to draw pictures on the big, blank canvas, and these pictures can be transferred to the scratchpad for use in other parts of the system. The version of the editor described here is fairly rudimentary, and it should be clear that any decent graphics editor can be substituted in its stead (see e.g. "MacPaint" described in [Williams 84]).

When the cursor is moved over the canvas with the *select* button depressed, bits in its path will be painted. A picture is a fixed-size bitmap, much smaller than the big canvas shown, so each bit in the resulting map is shown as a black or white square several pixels wide. To help judge how the picture will look when it is used scaled-down in programs later, two small copies are shown in the lower right part of the screen. The small pictures are updated concurrently with the canvas, but one cannot draw on them directly.

Figure 4-2: The picture editor.



The menu on the left side of the screen supplies the following functionality:

At the bottom of the menu, lines of different widths are shown. These are used for selecting drawing with "brushes" of different sizes.

The small black and white squares above the lines can be used to choose the *paint color*. Normally, the drawing is painted black on white, with white color used for erasure. The paint color is picked by "dipping" the mouse in one of the colors (moving the cursor on top and clicking the *select* button).

There is a menu showing a line and a circle. This allows drawing of straight lines and nice circles in an easy way. For the lines, the user specifies a starting point and then stretches the line to the end point. For the circle, a similar approach is used. The stretch effect lets the user see what is going on: After the first line point is specified, there is no doubt that the next thing to do is to click the mouse again over the end point.

Clear colors all canvas bits white, and **invert** changes all white canvas bits into black, and vice versa.

Finally, there is an operations menu that lets the user transfer pictures between the canvas and the scratchpad. Any picture currently shown on the scratchpad can be transferred to the canvas by selecting the **from**

scratchpad function. The top of the scratchpad is always visible above the editor area, and the whole scratchpad can be brought out in front for inspection with the *top* button. The canvas is cleared before the new picture is brought in. Alternatively, the picture on the scratchpad can be added ("ORed") bitwise to the current content of the canvas via the **add scratchpad** facility. **To scratchpad** transfers the current picture on the canvas to the scratchpad, for use in type and function definitions.

This is all we really need to create pictures. There are, however, many other useful functions that we could think of to make drawing a lot easier. Examples include the ability to move picture elements around on the canvas, duplicating picture elements, built-in functions for more shapes, like boxes and ellipses, as well as various text fonts, and filling of regions with textures.

4.4. Defining simple types

The type editor provides the means for defining the layout of data to be executed by functions. The editor provides a template that the programmer can fill with various data components. The pictures of the components together make up a picture of the structure of the type. A picture icon can also be attached to the type to describe it as an entity when used as part of other entities.

When entering the type editor we see, on the left side of the editor space, a comprehensive menu of functions used to build and modify types.

4.4.1. The type template

Figure 4-3 shows the type editor after entering it and selecting the **create** option from the main type editor menu. The display is dominated by a blank type template. The template contains a large area that can be filled with a pictorial description of the data. It also contains two smaller frames, one for the type name and one for the date the type was created. The template is automatically dated and has the default name "NewType" and no data elements.

4.4.2. The library scroll window

If we choose the **search type** command, a scroll-menu is displayed to the right of the editor menu (figure 4-4). The scroll-menu is a window into the library of already defined types, and by pointing at the bars at the top and bottom of it, we can scroll back and forth through the library. When we have found an item we are interested in, we can click the mouse on its name, and the type in question will be fetched from the library onto the scratchpad. The picture icon of the type will be displayed here. There can be several items on the scratchpad at the same time, and the pad space is managed automatically by the system.

Figure 4-3: The type editor template.

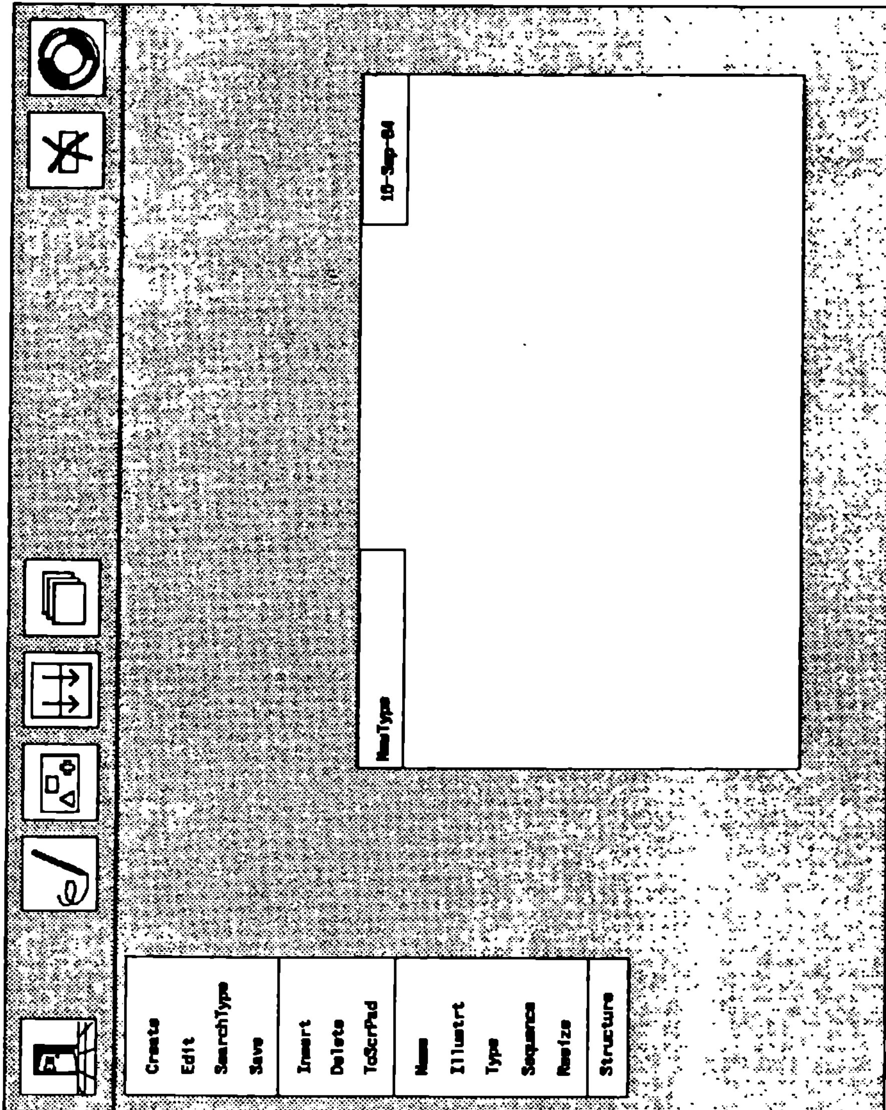
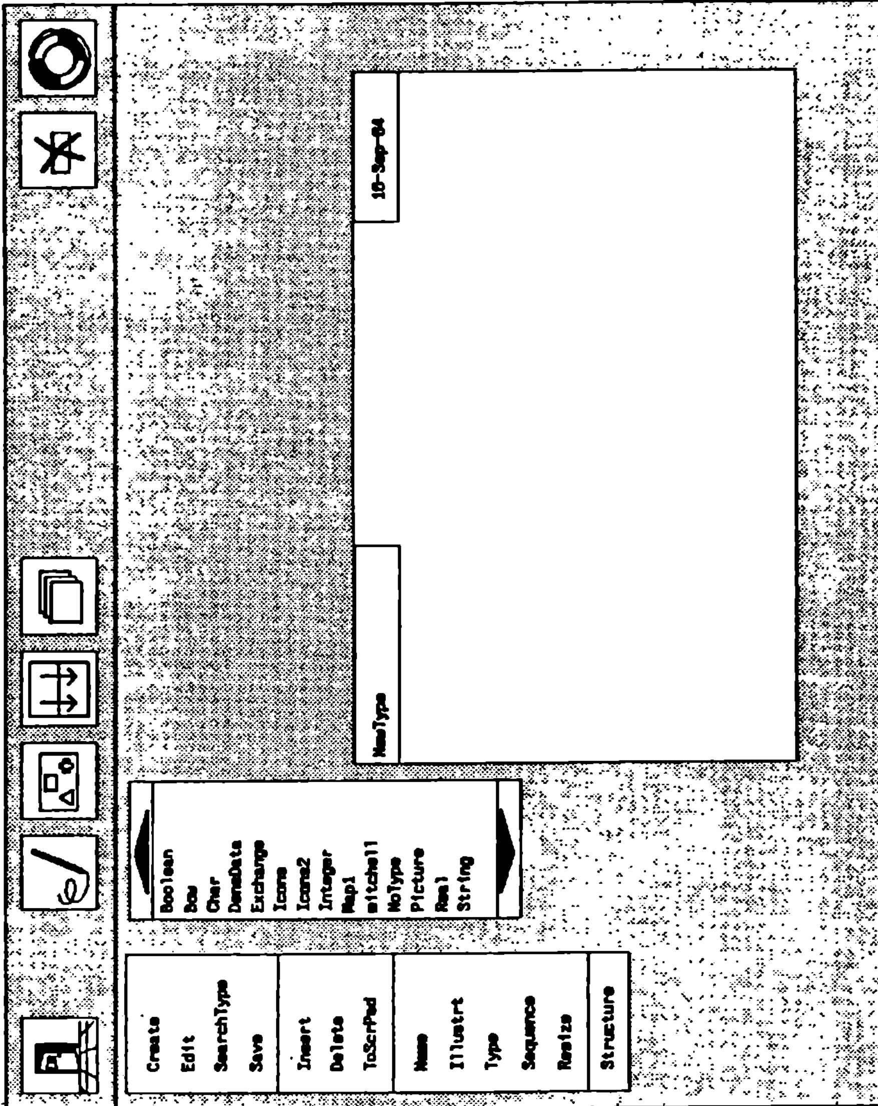


Figure 4-4: The scroll window.



Notice that, even though we access the library items through names (they are out of view, so we need an indirect reference), we pick from a menu. This means that the names need not be unique. We can fetch all items with the same name to the scratchpad and then inspect their structure to determine which one we are interested in.

4.4.3. Adding and manipulating elements

The **insert** command is used for inserting new data elements in the type. After selecting the *insert* command, the element is inserted anywhere in the type template simply by positioning the cursor at an appropriate location in the template (marking the lower left corner of the element picture) and clicking the mouse again. Now the text "NoName", indicating an unspecified, empty element, will be displayed in the selected position. Next, we can attach a type to the element (**type** command). To do this an icon of the element type must be present on the scratchpad. The library search facility described above is used for fetching types to the pad. To type an element, the element is first selected within the type template, then the desired element type is picked on the scratchpad. The icon of the type is now displayed at the position of the element, at a default size. The system will highlight first the type template, then the scratchpad to indicate where mouse actions are expected next.

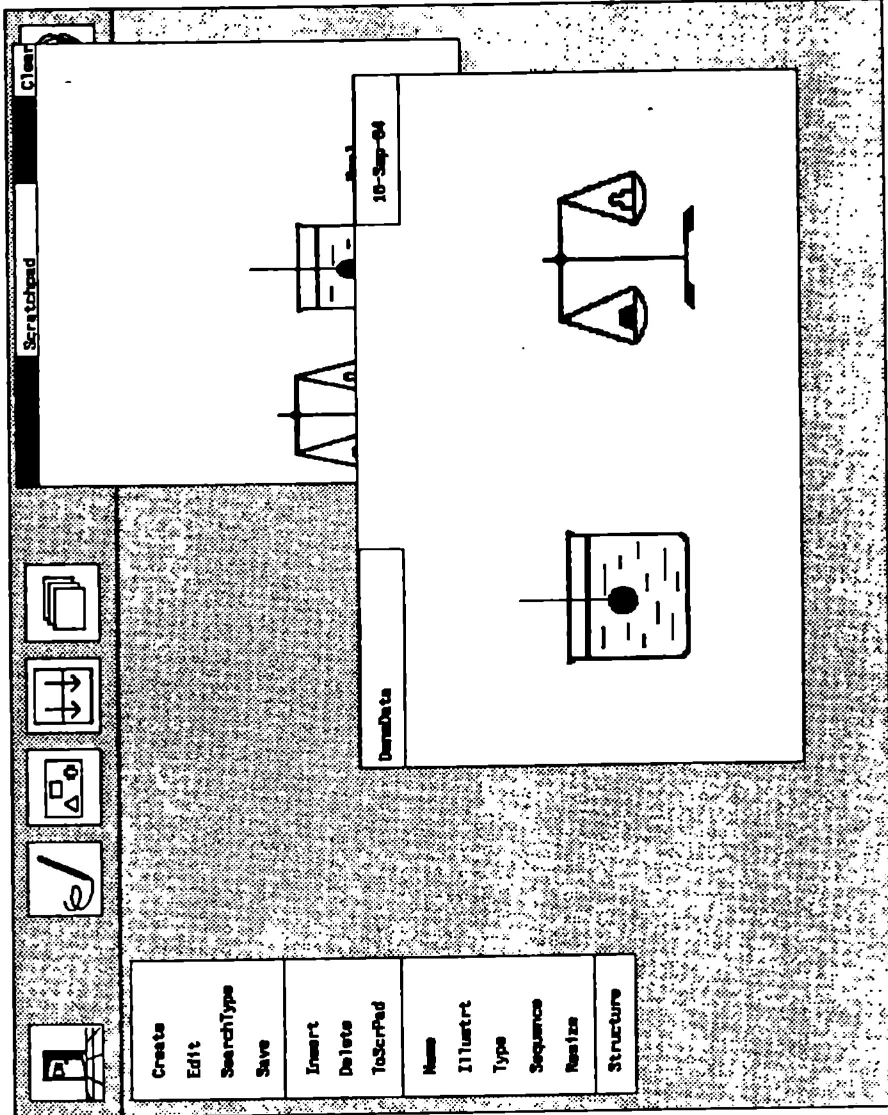
The new element can subsequently be given its own identifying picture and name. A drawing created with the picture editor can be inserted as the pictorial icon of an element. In the picture editor, the drawing is moved to the scratchpad. Entering the type editor, the **illustrate** command transfers the drawing pointed at on the scratchpad to the element selected, and the picture is associated with that element. Similarly, if the frame of the type template is pointed at, the picture becomes the icon for the current type.

The **name** command works like *illustrate*, but associates a name with the item pointed at instead of a picture. A prompt is displayed on the type element or in the name box of the type template, and the name can then be input on the keyboard.

Figure 4-5 shows a data type intended for use in a simple physics experiment, where the students are to determine the density of various materials. There is one icon representing the volume measurement (obtained by submerging the object in question in water) and one for the weight.

When an element has a picture associated with it, this picture overrides any other icon or name that might be used to stand for the element, such as the icon of its type. If the element does not have an icon, its name is displayed. If it does not have a name, its type icon is shown. If there is no type icon, the

Figure 4-5: Creating a type.



type name is given, defaulting to "NewType". Similar rules apply to types and functions.

A type element created in the above described manner can hold a single value of some type. The **sequence** command will change an element into a sequence, of unspecified length, of values of its type. The icon of a sequence element is automatically framed by a bold line to indicate this. Selecting *sequence* twice on an element returns it to a simple element. One cannot make a sequence out of the whole type, but a type can of course contain a single sequence element. We will say more about structured data in chapter 5.

The attributes of a type element can now be summarized. They are the *type*, the *name*, the *picture*, the *position* and *size*, and the *structure* (sequence or simple data element). After an element is inserted, we might want to change its attributes. If we would like to change the type, name, picture, or structure, we simply use the respective commands again.

If we want to change the position or size of the element, we can use the **resize** command. This will display a box that we can position and size as we please. The element picture will subsequently be resized to fit the new box. In this way, the user is completely free to compose a data layout consisting of

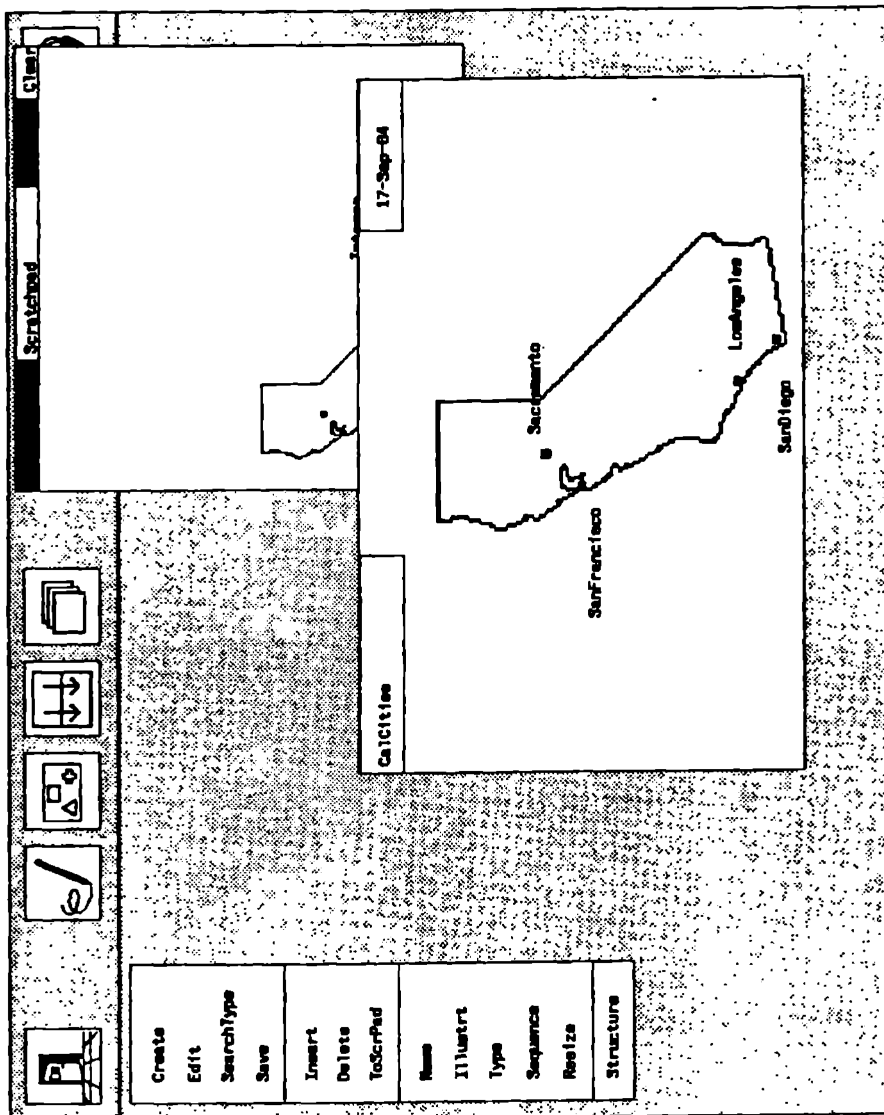
several independent pictures, but that together may make up a coherent and comprehensive depiction of a data structure. We can even add a background illustration by including an empty element (of type "NoType") whose picture fills the whole template. The data elements proper should be added on top of this background. For some applications this technique can give the program user important visual cues about the meaning of data (figure 4-6).

Finally, the **delete** command deletes the element pointed at from the type template being edited.

Notice how the task of building a new element is split up into its constituent subtasks. This simplifies thinking about the total task, and it also helps avoid stringent modes and forced subtask sequences. (It does not matter in what order we specify the element attributes.) We also see the uniformity of the commands: First create the element, then specify the attributes one by one.

Since all types are built up from components, each being of some other type, there have to be some built-in primitive types. As in most other programming systems the PiP system provides the standard types *boolean*, *integer*, *real*, *character*, and *string*. Since the system is picture-oriented, it is also very natural to include a primitive type *picture*. Our system allows the definition of simple enumerated picture types, much like the enumerated types of e.g. Pascal.

Figure 4-6: A type with background illustration.



4.4.4. Saving and restoring

The **save** command saves the current type permanently in the type library. If the type is already present in the library, the old version is deleted. Note that this equality is not determined by the type name, since the name is treated as any other attribute here. Rather, the system keeps track of the identity of types and functions, and only if an item is fetched from the library and then reentered is it considered to be "equal" to an existing item.

A type can be fetched from the library and modified, or just inspected, by first searching the library and then selecting the **edit** command. This command copies a type represented by its icon on the scratchpad into the type template. Any type already in the template is cleared. The **to scratchpad** command transfers a picture from a type to the scratchpad, so that it can be used to illustrate other types or functions as well. The type itself can not be transferred directly from the template to the scratchpad. This is done to force the programmer to *save* the type in the type library before starting to use it (e.g. in a function). The type can then easily be fetched from the library to the scratchpad by means of the *search type* command.

4.5. Defining functions at the object level

As we explained earlier, functions can be edited at two levels, the object level and the function level. There are actually two separate editors for this.

At the object level, a simple function can be programmed in terms of operations on the constituents of its input and output data structures. The programming paradigm is that of displaying pictures of the function's input and output data, and specifying the function by showing how the output is computed from the input. This specification is performed as a sequence of mouse pointing actions.

When entering the function editor we have, as for the other editors, a comprehensive menu of commands to build functions.

4.5.1. The function template

Figure 4-7 shows the display after the object level function editor has been entered and the **create** option has been selected. As with the other editors, we see a large, blank template. This template is split into two main parts, marked "input" and "output", respectively. In the input part we are to add a description of the function's input data structure, and correspondingly for the output part. Next, we can connect the input and the output by pointing actions that will describe the effect of the function. The template also has

small windows for the function name and the date it was last modified. The template is automatically dated, initially has the default name "NewFunc", with no input or output type specified.

4.5.2. The library scroll window

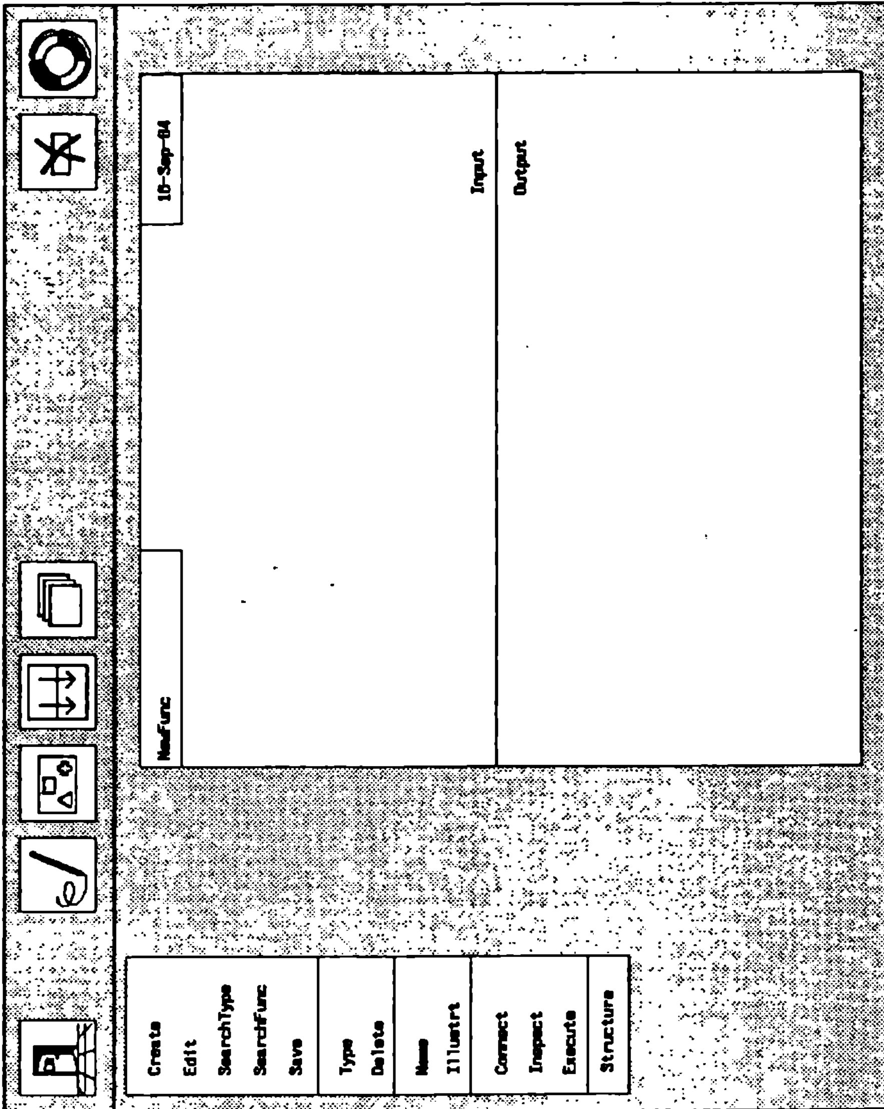
Just as for the type editor, a scroll window can be invoked to show the types and functions available in the libraries, and to fetch them to the scratchpad.

4.5.3. Typing a function

The **type** command is used for inserting types in the input and output parts of the function template. Only one type can be inserted in each of the two parts. If a compound data structure is desired, a type for it must first be created. This is in keeping with the FP model, where a function works on a single data object.

To insert a type, an icon of the type in question must be present on the scratchpad. The library search facility is used for fetching types to the pad. After selecting the desired type, it is inserted in the input or output part of the template simply by positioning the cursor within the appropriate part and clicking the mouse again. Now the type will be displayed in the selected window. The type is displayed at the first level of refinement (first level underneath the icon) since this is the level most often considered. The zoom

Figure 4-7: The object level function editor template.



facility can of course be used to change this. Any type already present in the window will be removed.

Figure 4-8 shows the function template filled in ready to program the density program. In figure 4-9 we have zoomed two of the data elements to reveal their type structure (which in this case is a primitive type).

The types can be changed by using the *type* command again. Changing a function's types will cause the current function definition to be erased, since it becomes meaningless. The input or output type can also be deleted by selecting **delete**.

A drawing created with the picture editor can be inserted as the pictorial icon of a function in the usual way (**illustrate**), and a name can be associated with it (**name**). When a function has a picture associated with it, this picture overrides the function name, so that the icon is displayed instead of the name when the function is used within other functions.

Figure 4-8: Input and output types inserted.

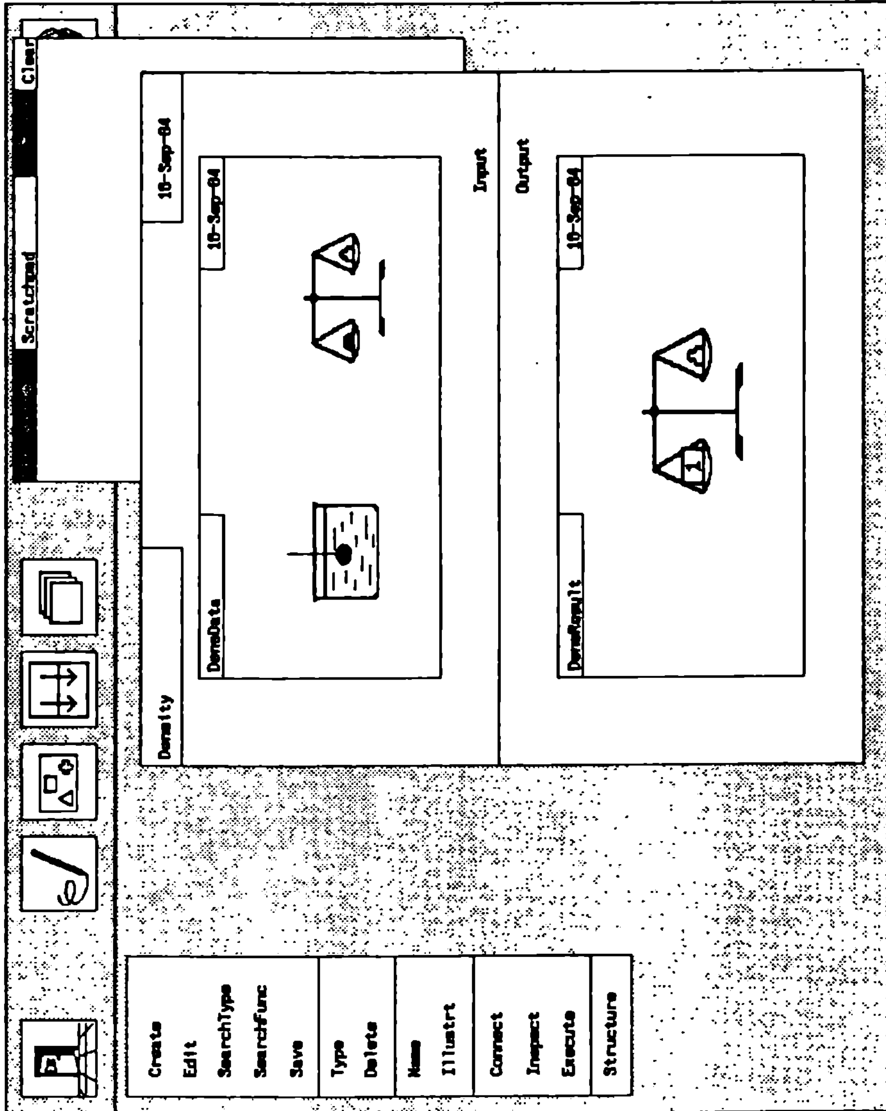
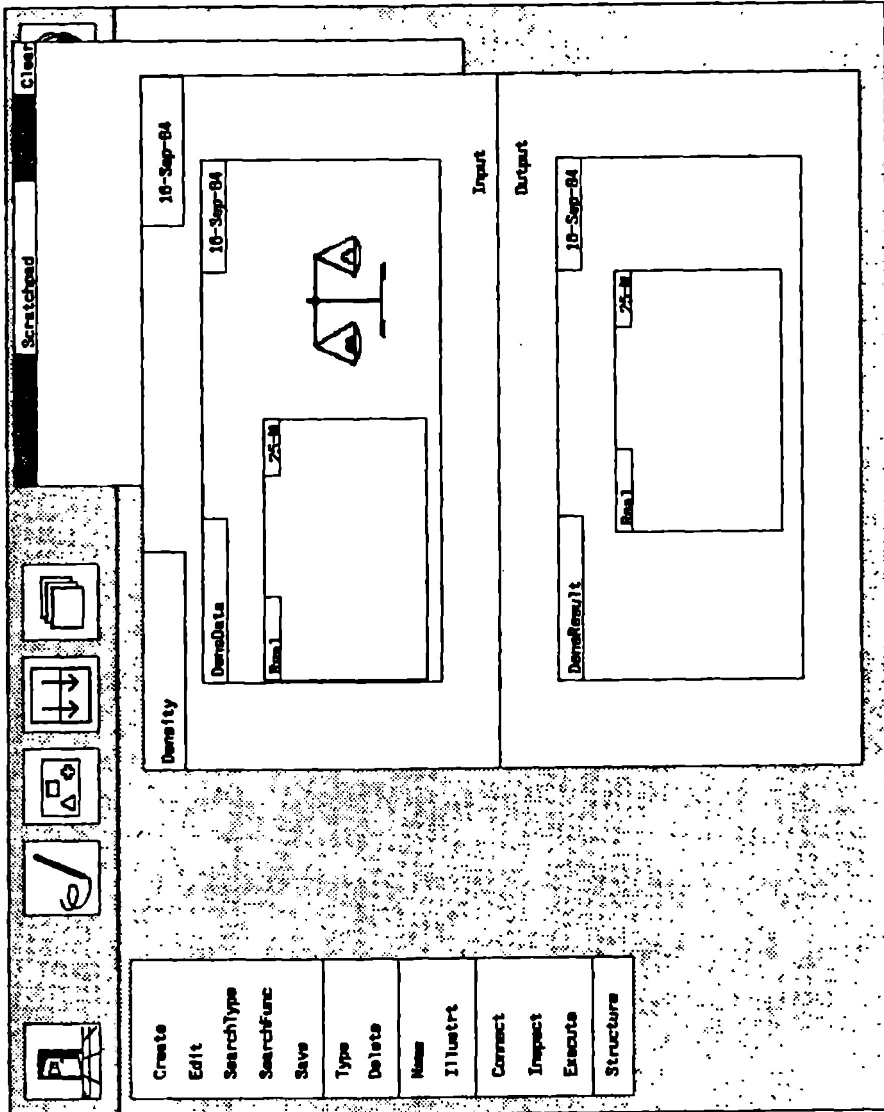


Figure 4-9: Zooming to show type structure.



4.5.4. Programming a function

Connect is the programming command within the object level function editor. When *connect* is pushed, elements in the input and output types of the function can be pointed at to indicate how the output is computed from the input. Primitive functions and functions defined in the function library can also be included in the specification of the function. In detail, the composition of an FP expression works in the following way:

- The current expression being built is kept in a stack until it is assigned to an output. The stack is displayed in the lower left part of the screen and is continuously updated.

- Each time an element of the input type is selected with the mouse, a reference to it is pushed onto the stack.

- When an element in the output type is selected, the topmost entry in the stack is popped and associated to the output element. This entry now describes how the output element is computed from the input. If there is already an expression associated to the output element, the old expression will be deleted before the new one is inserted.

- When a primitive or user-defined function is selected, this function is

combined with some of the topmost entries in the stack, depending on the arity of the function. For example, if the "+" function is selected, the two topmost entries are popped off the stack, and a new expression, representing the addition of the two entries, is pushed back onto the stack. All common primitive arithmetic and logic functions, as well as relations, are available in submenus that appear when an operator class (e.g. arithmetic) is selected. The *search function* command can be used to locate user-defined functions for insertion in the expression being built.

- When **done** is pushed, the output expression specified so far is accepted as the definition of the function being created, and the main menu again becomes active (no more input-output connections can be made until *connect* is again selected). A warning is issued if the function is not completely specified, i.e. if there are still output type elements without any expression associated with them. *Connect* can be repeatedly entered and exited to build up or modify the function in a stepwise manner. The system will instantly compile the FP expression corresponding to the output expressions specified, and this will be invoked when the function is executed. The FP expression generated can be displayed, if desired.

Figure 4-10: Programming a simple function (1).

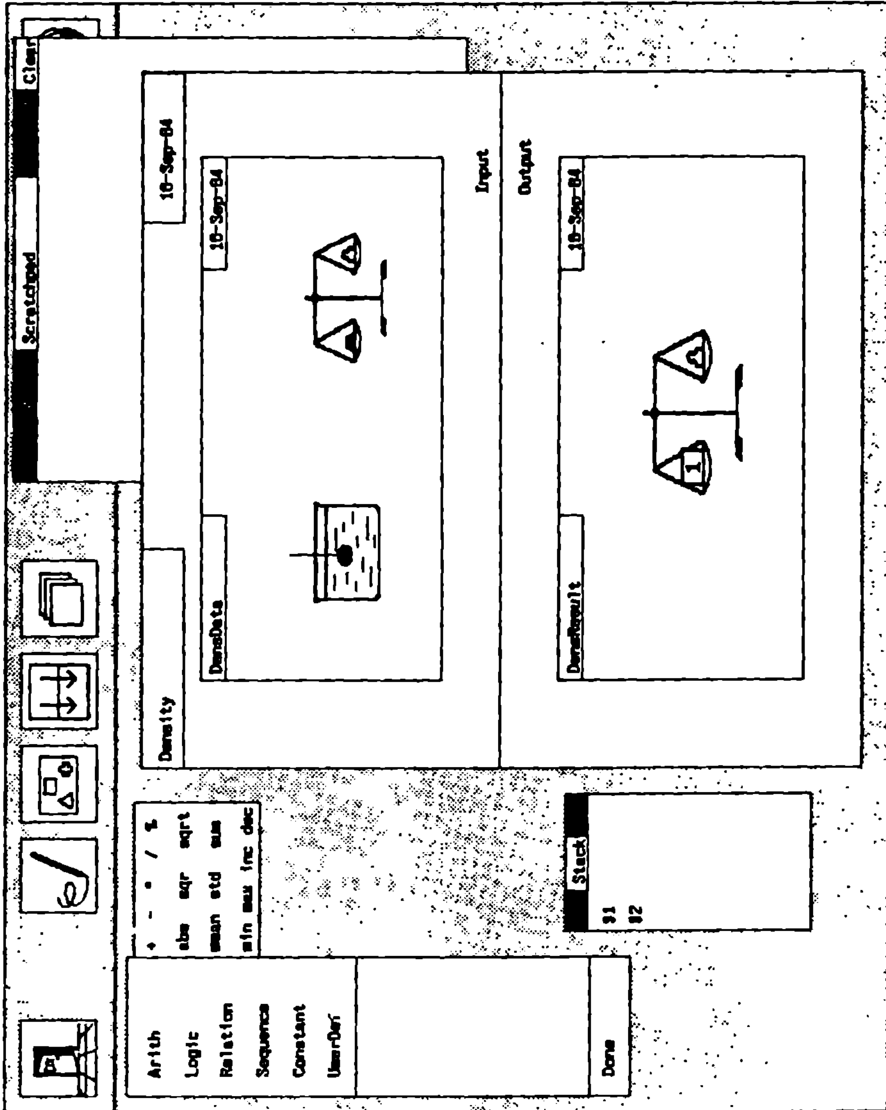


Figure 4-11: Programming a simple function (2).

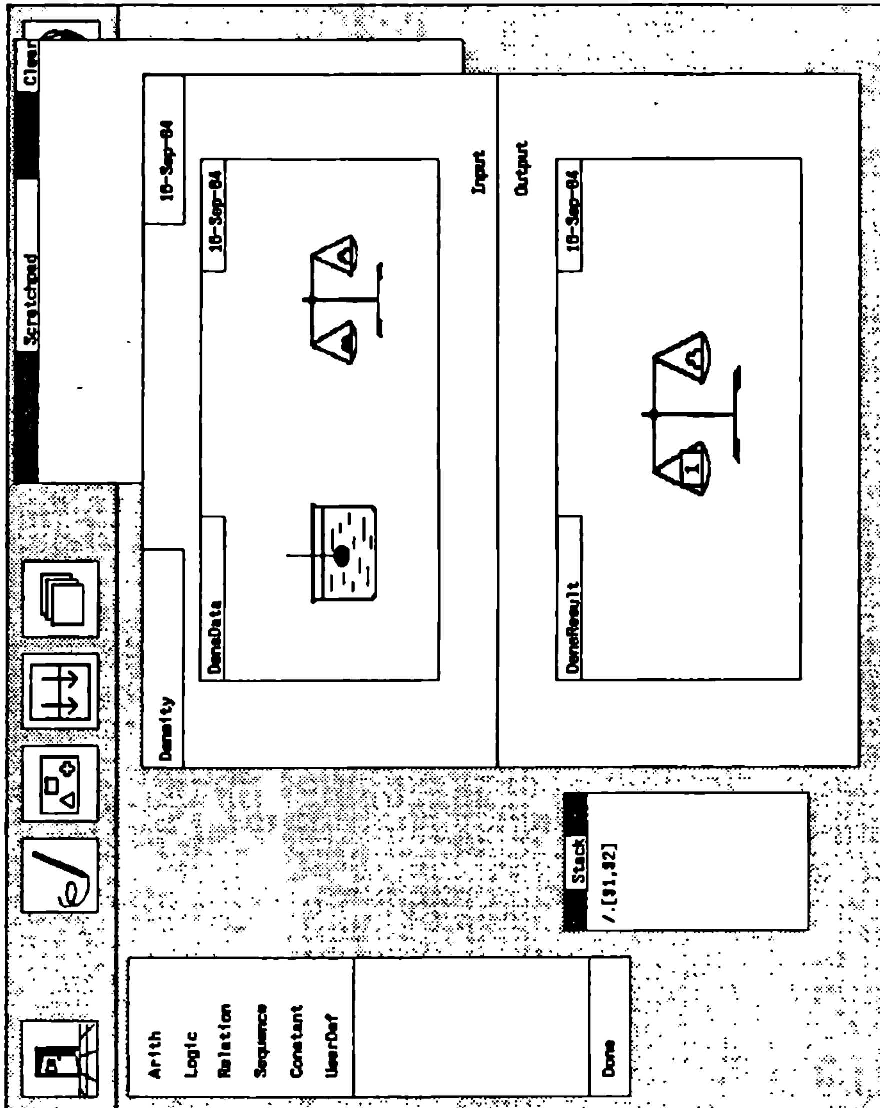
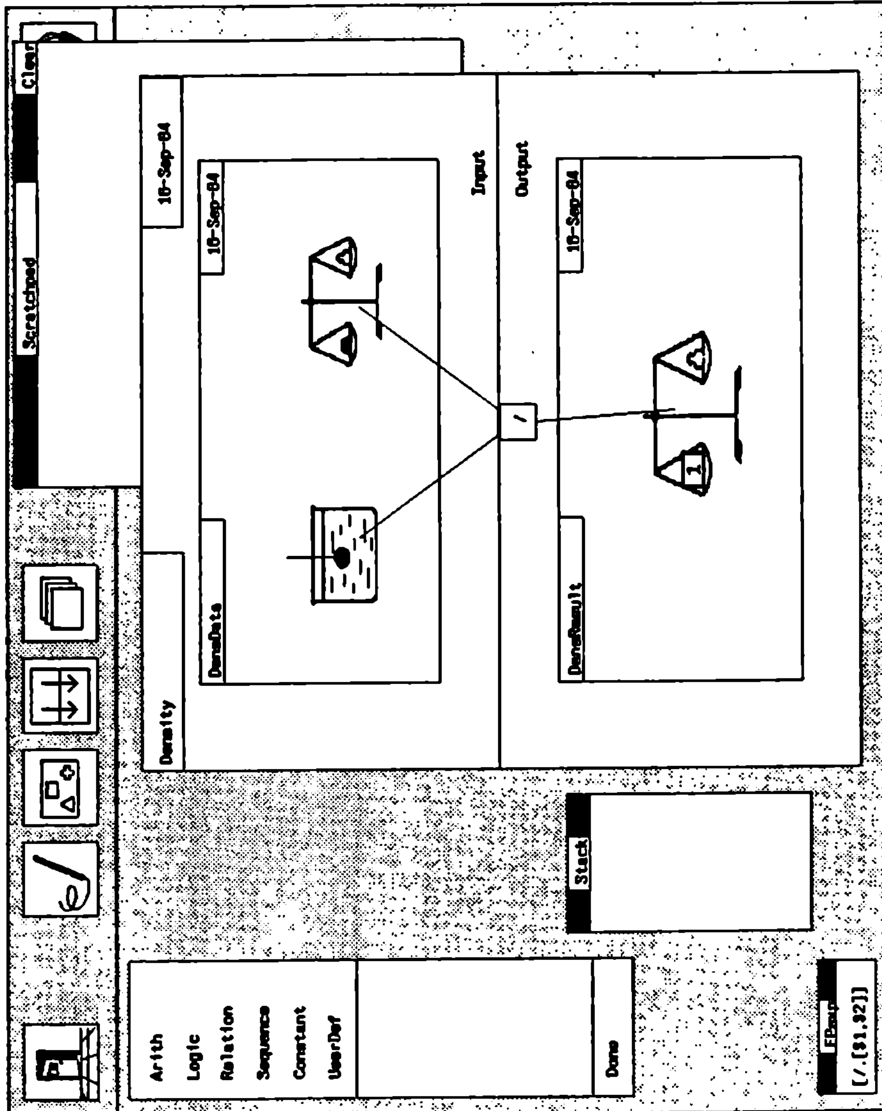


Figure 4-12: Programming a simple function (3).



Figures 4-10, 4-11, and 4-12 show snapshots of the programming of the simple density function. First, the two input elements are selected, and expressions representing them pushed onto the stack.[1] We have also selected **arithmetic** to find the division operator on the submenu. Next, figure 4-11, selecting division changes the stack into the arithmetic expression shown. Finally, we point at the output element, the stack is popped, and the final expression created (4-12). The system displays the computation of the output element as a flow graph.

We can make some remarks about this programming style:

- As we explained in chapter 3, this kind of interaction mimics the informal way we explain programs, by showing pictures of the data and defining the computation on them by pointing sequences similar to handwaving. This is what we called animated writing.

- Within a function, there is no order imposed on the computation of each individual component. This is contrary to traditional von Neumann programming, and it is very much in tune with the graphical paradigm

---

[1]In the prototype implemented, the stack entries are simply shown as the FP expression generated ($ is the selection operator). This is illegible to most intended users, so it should be replaced by arrows pointing at the actual elements. Also, the display of the resulting FP expression is included in the prototype only.

of building a function as an object on the screen, and the random access nature of pictures. It frees the user to define things in the sequence that feels most natural. It is made possible by the FP model: The strict functional semantics splits the data into an input and an output part, so there is no possibility of side-effects. The sequence of function argument evaluation is irrelevant. There is no possibility of reusing results from another subexpression within the same application.

- Since the result types of all functions and expressions are known at programming time, it is easy for the system to check type compatibility on the spot. This means that if the programmer tries to attach an expression to an output component that is not of the same type, or to apply a built-in function to input data of the wrong type, the system can immediately refuse to do so, providing instantaneous feedback and elimination of errors.

4.5.5. Inspecting a function

Usually, when a function is displayed in the object level function editor, the correspondences between input and output data are not shown. How each output element is computed can be inspected by means of the **inspect** command. When an output element is selected, its relation to the input is

drawn. Only one output element can be shown at a time in this way in order not to make the screen too crowded. The display forms a dataflow graph connecting the input and output data via operators, essentially showing a drawing of the pointing actions used to program the function (figure 4-13).

We previously denounced dataflow graphs as a primary means for program display. Here, we utilize their ability to convey simple computations by showing small graphs in a carefully controlled manner, as secondary displays superimposed on the main data display.

4.5.6. Executing a function

Program execution is available at the touch of a button. The **execute** command will execute the function displayed once. All input data are prompted for one by one. The resulting output data are displayed on or adjacent to the corresponding output elements. Figure 4-14 shows the screen after executing the density function.

Notice that program execution is available at the individual function level. Any function can be executed as easily as a whole program. In fact, program execution is so convenient, and available instantly after a function has been defined, that it is reasonable to get into the habit of checking out an execution as part of getting familiar with the program. This is what we have referred to as *reading by execution*.

Figure 4-13: Inspecting a function.

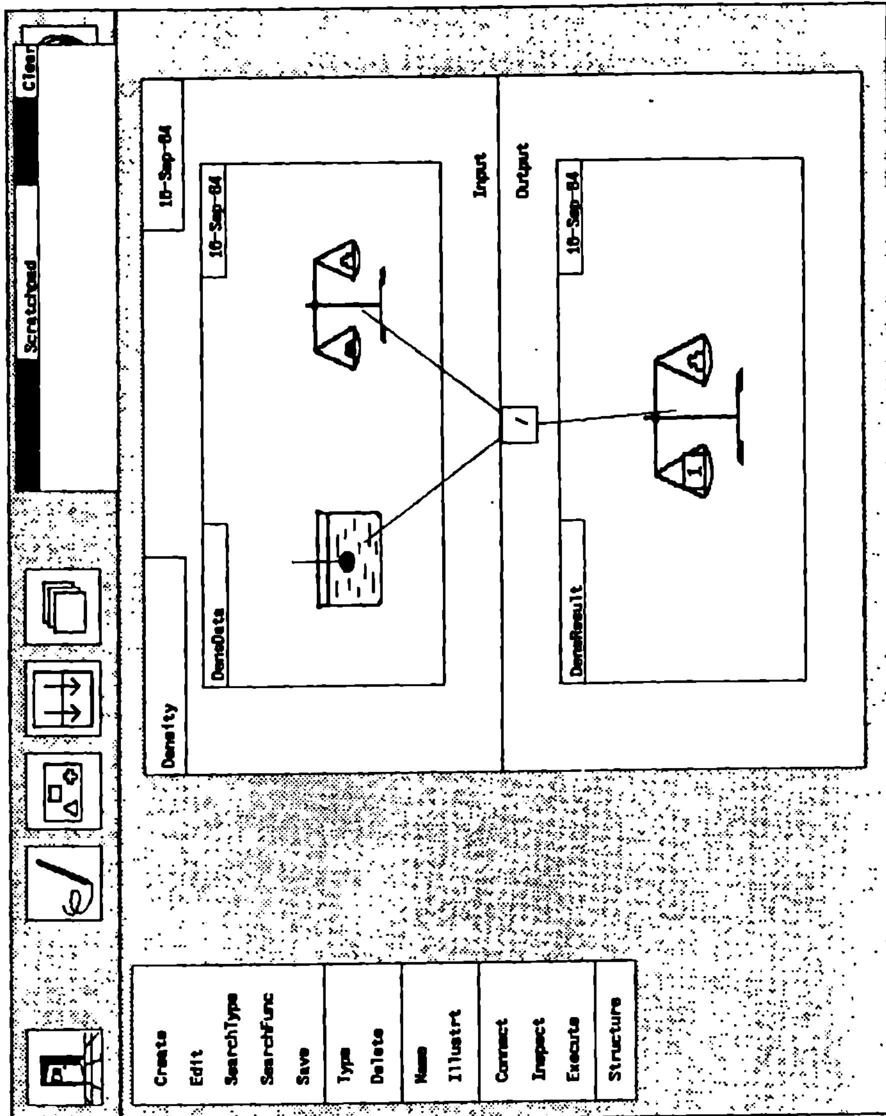
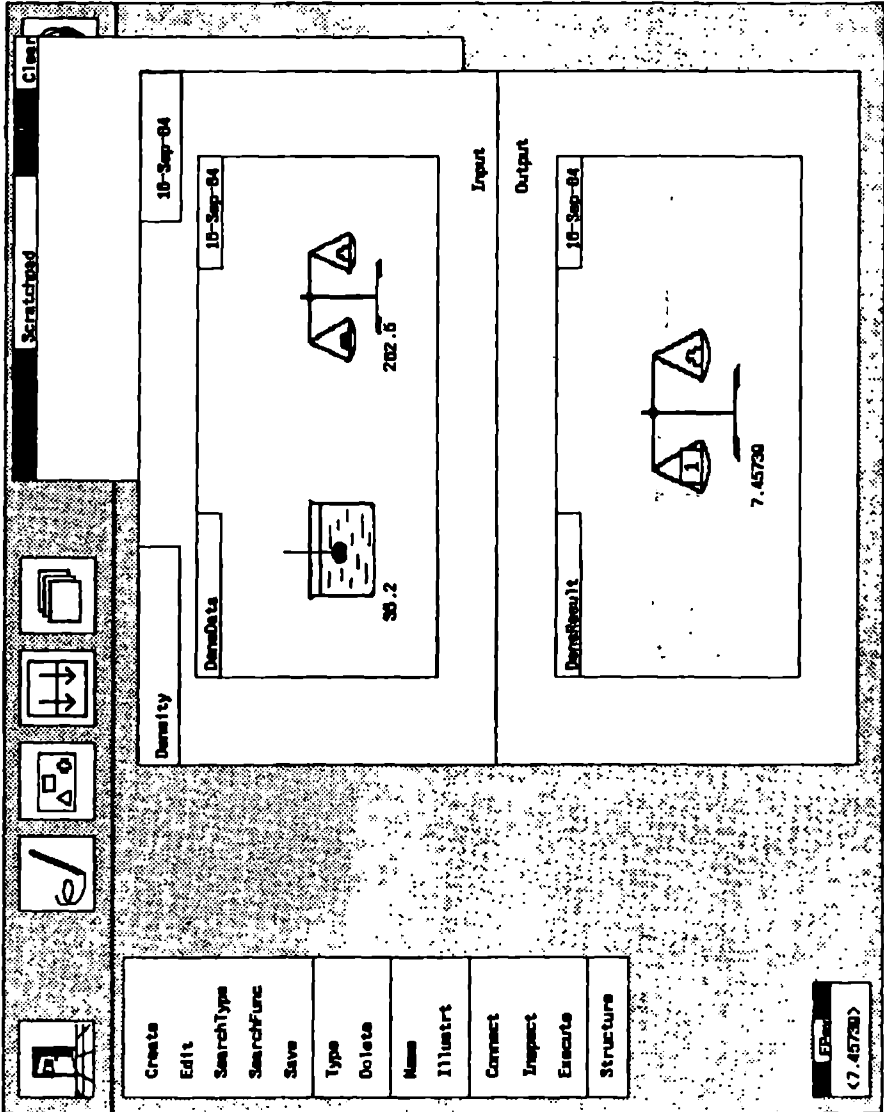


Figure 4-14: Executing a function.



4.5.7. Saving and restoring

As for types, there is an **edit** command that moves a function from the scratchpad into the template, and a **save** command that saves it in the library. Again, there is no command that directly moves the current function to the scratchpad, forcing the user to save the function before using it.

4.6. Defining functions at the function level

At the function level, already defined functions are treated as atomic entities that are combined in various ways to form a new, compound function. Typically, a program consists of a library of simple low-level functions created with the object-level editor, and several layers of function definitions built on top of each other using the function level editor. Indeed, the system can be tailored to provide a programming environment for a particular application domain by including a library of functions and types for that domain. With the proper set of functions, the programmer might never need to descend to the object level of function definitions.

As we remarked earlier, the split between the object and the function level of function editing corresponds roughly to the division between functions and functional forms in FP. At the object level, we were concerned with defining a function in terms of its effect on data objects. Here, we use functional forms to operate on functions. Thus, we are at this level more interested in

displaying function structures than we were at the object level. Similarly, since we do not have to think about the constituents of data objects, but only have to make sure we combine functions that are type-compatible, we are correspondingly less interested in displaying the data structure.

This means that we could think of using some kind of dataflow graph to represent the functions. We could do this at the object level because the graphs were so simple. Here, we have more complex graphs, both in terms of the number of nodes, and in terms of the kinds of nodes, since we want each functional form to have a specific graphical representation. We have therefore designed our own "structured flowgraph" that allows more graphical information to be displayed for each node, both system and user supplied, yet is more compact than regular flowgraphs.

4.6.1. The function template

When entering the function level editor and selecting the **create** command, the display looks as in figure 4-15. The template is much like the one we saw at the object level, but there is an additional function window to the left of the input and output windows. This window is used to show a picture of the function proper, in terms of a structured flowgraph. The input and output data structures are still displayed as for the object level, but their role is now somewhat reduced. Function composition happens in the function window,

and the data displays are used during program execution to prompt for and show data.

4.6.2. Typing a function

As for the object level editor, types can be inserted in the input and output windows to define the argument and result types (**type**).

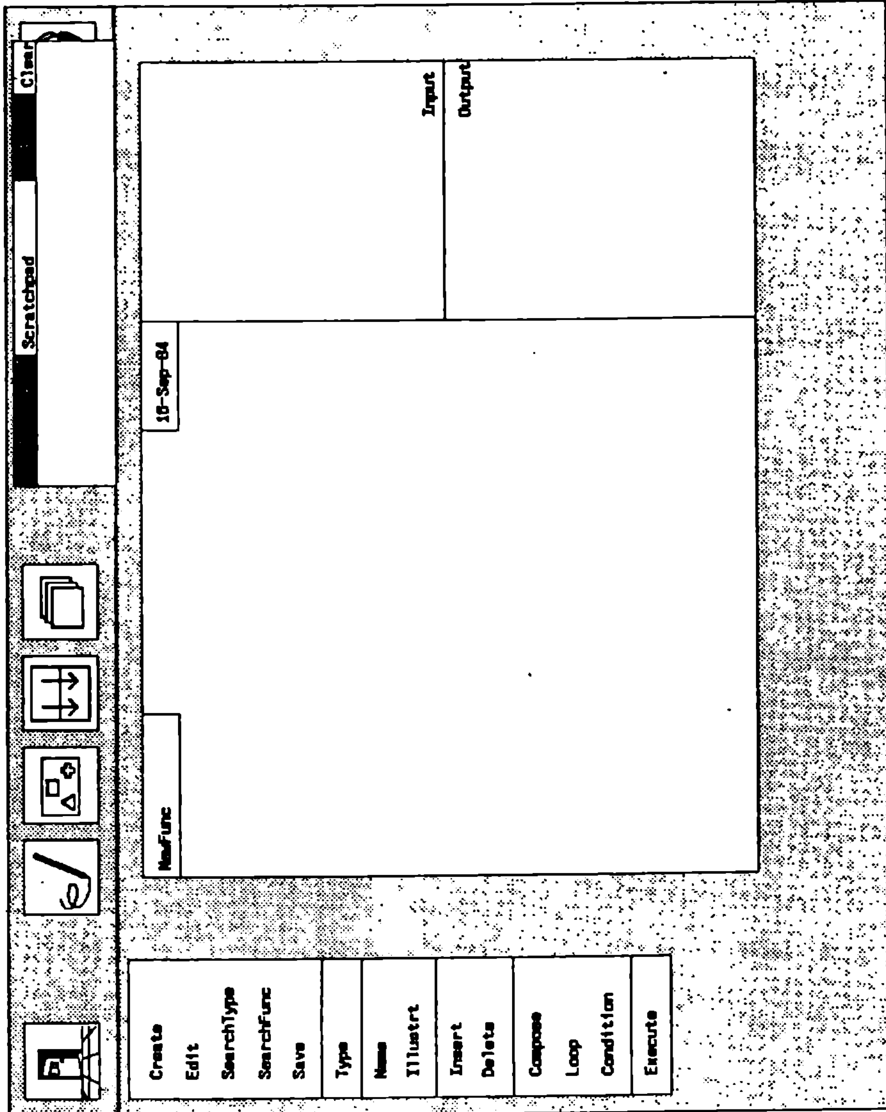
4.6.3. Programming a function

A functional form is an operator with functions as its arguments. For example, the conditional form takes three arguments: the predicate and the two alternative functions. Programming at the function level consists of invoking functional forms, inserting them at the appropriate places in the function being created, and then filling their argument slots with functions from the library (**insert**).

In detail, the programming proceeds as follows:

- A functional form is selected from the menu, and is inserted by selecting the function frame onto which it is to be composed. By this we mean that if the expression $f \circ g$ is currently displayed, and we insert the expression h onto g, then the expression $f \circ h \circ g$ will result. If we insert a functional onto white space, it will be appended to the "rightmost" function, i.e. it will become the function closest to the input data.

Figure 4-15: The function level template.



- The insertion will create a functional frame corresponding to the functional. Each functional has its characteristic frame with one or more slots that can be filled with functions or more functionals. For example, the *loop* functional has two empty slots, one for the condition and one for the loop body (fig. 4-16; condition is left slot). The conditional has three slots (fig. 4-17; condition is middle slot, true branch to the left).

- All functionals carry an implicit *compose*, in that e.g. the expression $(\rightarrow;)$ (*conditional*) added onto f yields $(\rightarrow;)\circ f$.

- Composition is the "natural" functional and is the only one that utilizes the depth dimension for display. It is shown by one expression being on top of another. The other functionals have their characteristic frame shapes. Invoking a *compose* by itself creates an empty frame.

- Regular functions can be fetched from the library and inserted into the empty slots. Their top-level icons will be displayed in the slots together with the function name.

- Type checking is performed every time two functions are combined.

Figure 4-16: The loop frame.

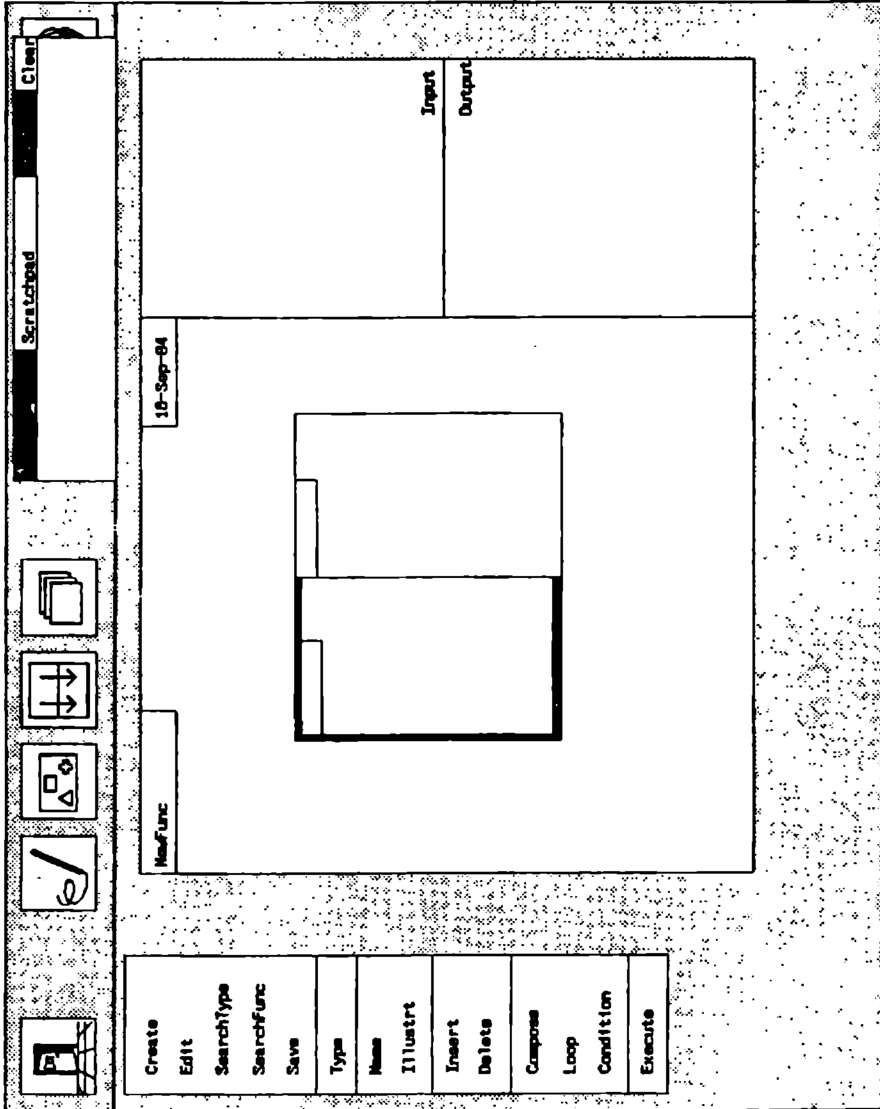
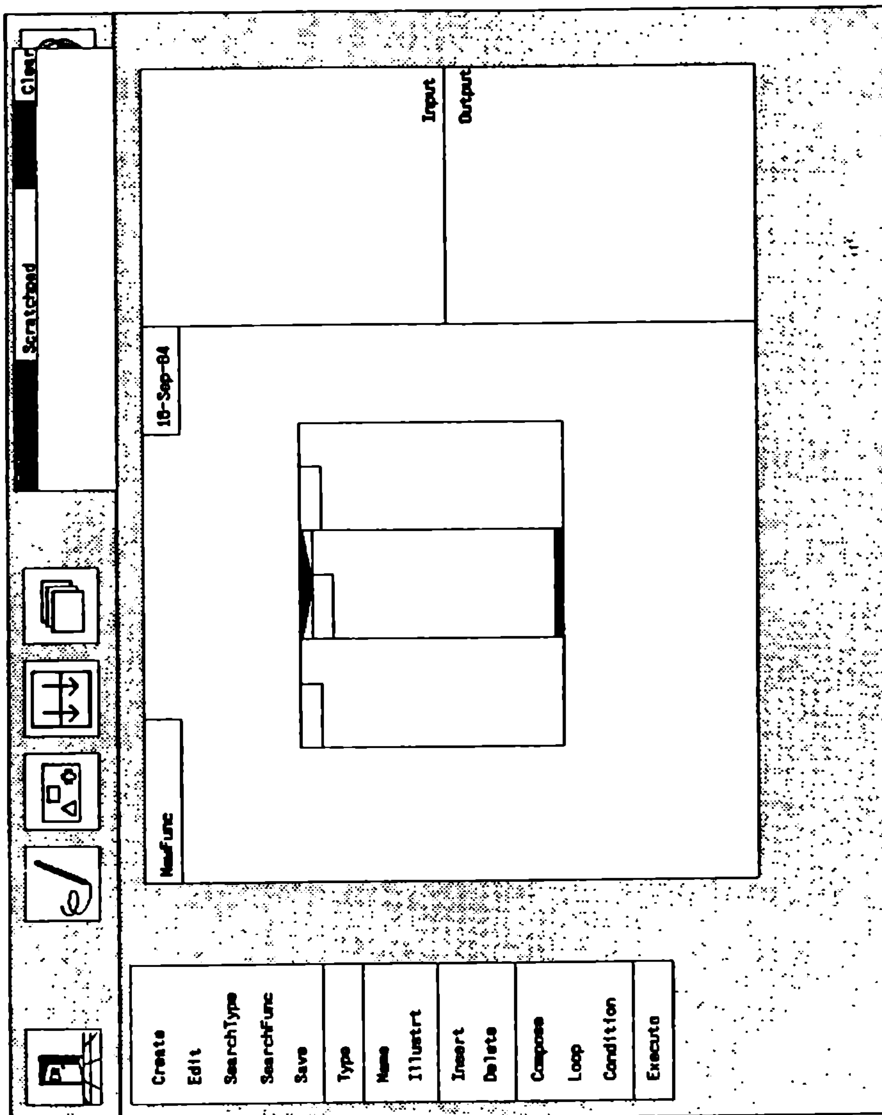


Figure 4-17: The conditional frame.



The system cannot, however, flatly refuse to combine incompatible pieces, since an erroneous intermediate state may be necessary to achieve a correct result. (E.g., the expression h inserted to create $f \circ h \circ g$ may be incompatible with f, but a subsequent insertion of i may render $f \circ i \circ h \circ g$ legitimate.) The system will flag this by inserting an empty *compose* frame, indicating that there is something missing. The function cannot, of course, be executed before all empty slots are filled.

- The **delete** command is used to remove a single function or functional from the compound function being edited. The removal will leave an empty slot that can subsequently be filled with the *insert* command. If the types on either side of the empty slot are compatible, the slot is removed.

- As the function is being built, the system will determine its position within the template, and make adjustments as it is being added to. I.e., the picture is not as freely formed as the pictures of data types. This is a consequence of flow structure being much more abstract than data layout, so that it is natural for the system to exercise more control of the format here.

Figure 4-18 shows an example. We have put a loop around our density function and added a function that reads in the data sets. The picture corresponds to the Pascal program in Program 4-1.

```
GetData;
while NotDone do
   begin Density; GetData end;
```

Program 4-1: Pascal version of fig. 4-18.

The function application starts at the bottom of the stack of pictures. The size and position of a function picture determines which substack it belongs to.

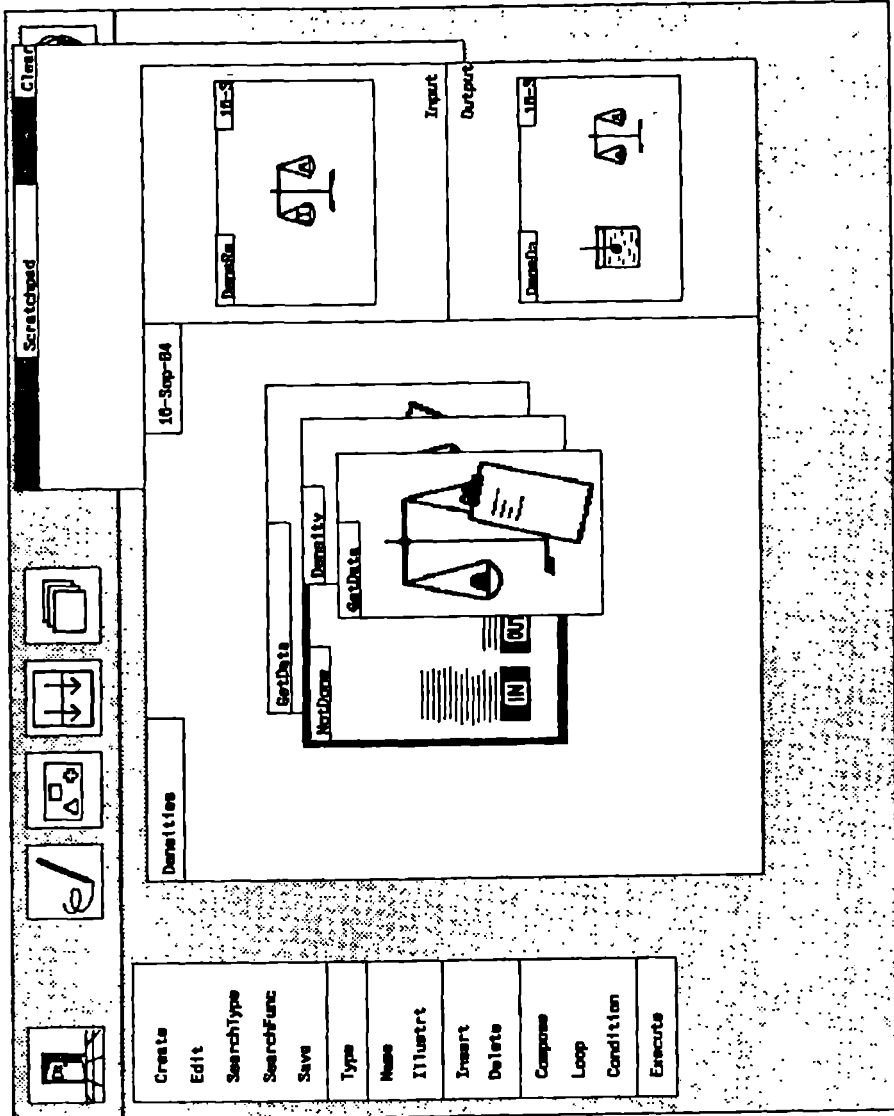
We notice how the function is being built in a stepwise manner from existing components, much like that of an erector set. The framework of functionals is first built, and is then filled in with functions.

4.6.4. Executing a function

A touch at the **execute** command will start a single execution of the function. As at the object level, data are prompted for and displayed in the input and output windows. In addition, the function is traced by highlighting the components involved in the function window.

The **step** command allows more finely grained execution and animation at a

Figure 4-18: Composing a function.



pace determined by the programmer. Each time *step* is pushed, one function component is executed. Its icon is highlighted on the display, and its input and output data are displayed in the data windows. At each step, the *zoom* command can be used to obtain the definition of the function about to be executed, and *it* can be stepped through recursively in the same way.

The **break** command allows insertion of breakpoints, so that a function can be animated up to a certain point in the above manner.

The system serializes the application of functions in a standard way, e.g. the *construct* operator will be evaluated from left to right.

Serialization is convenient since it simplifies the display, and it makes it easier for the user to insert breakpoints. We also observe that, even though FP computes with much parallelism, all concurrent branches are independent (no side effects), so that serialization is trivial to obtain without changing the semantics. When we set a breakpoint, we put a stop on one function node in the dataflow network, and all nodes dependent on the output of this node will be held up, too. All nodes the breakpoint node depends on will have been evaluated when the break occurs. The other nodes may or may not be evaluated, depending on the serialization, but this is of no significance when inspecting the breakpoint node.

The serial display is used only for stepwise animation. When the program is executed, all parallel expressions are conceptually executed in parallel and highlighted to show concurrency.

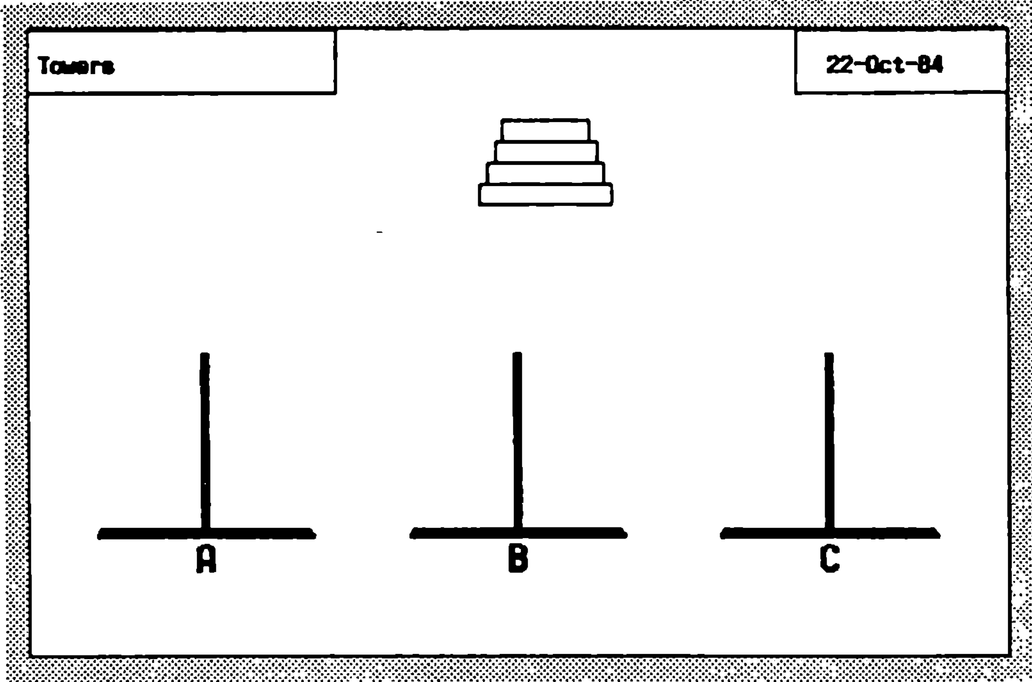
4.7. Another example

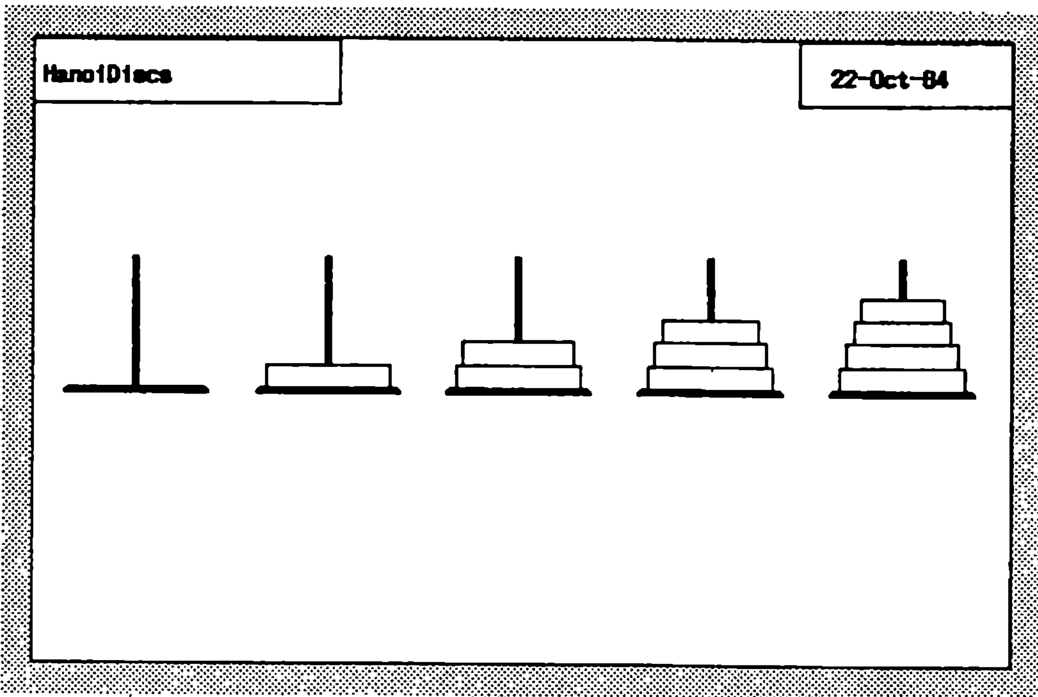
Our next example is an implementation of the well-known Towers of Hanoi problem, providing an opportunity to show the use of enumerated picture data types.

Figure 4-19 shows the two data types we have created for this example, excerpted from the type editor. The *Towers* type will hold the current disc contents of the three pegs, along with a count of the number of discs we have to move (represented by the stack of discs above the pegs). The disc count is an element of type integer. The three pegs are of type *HanoiDiscs*, an enumerated picture type.

HanoiDiscs is defined as a regular type, but when inserting elements of type *picture*, it is interpreted as an enumerated picture type with the pictures of the elements as the constants of the type. The sequence of insertions of the constants determines the order of the type. The built-in operators *inc* and *dec* can be used to get the next higher and lower picture constant when programming with picture types. Here, five pictures are used to represent the

Figure 4-19: Types for Towers of Hanoi.





content of a peg of up to four discs.[2]

Figure 4-20 shows the program that recursively solves the Towers of Hanoi problem, moving a certain number of discs *n* from peg A to B via C. The program first checks whether there is only one disc left. If so, the disc is simply moved from A to B. Otherwise, *n-1* discs are moved from A to C via B, one disc is moved directly from A to B, and the *n-1* discs are moved from C to B via A.

The function *MoveACTop* (truncated to *MoveAC* in fig. 4-20) is shown in figure 4-21. It is slightly different than the usual formulation in a textual language. First, it explicitly swaps its arguments, since there is no parameter name substitution that can do this. Secondly, it *adds* one disc back to the number to move, since there is no stack that saves the previous data. The one data object passed from function to function acts as a global pool of data.

Figure 4-22 shows the object-level definition of a swapping function (the *inspect* facility shows the computation of only one output element at a time), and figure 4-23 shows the function that subtracts one disc from the number to move. Note how we can include constants in the expression.

---

[2]The alert reader will realize that these pictures are not enough to cover all peg configurations, since the discs are of different sizes. The simplified solution illustrates our points, though.

Figure 4-20: A program for Towers of Hanoi.

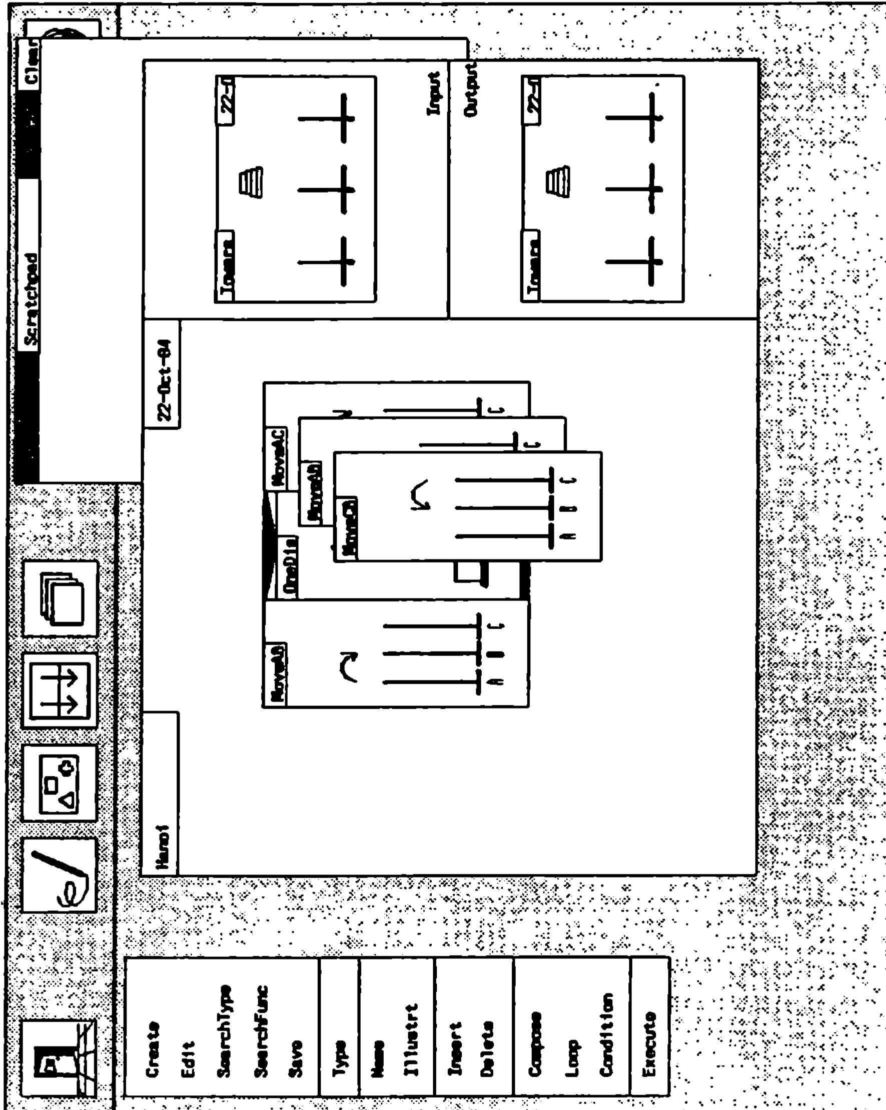


Figure 4-21: Moving the *n-1* discs.

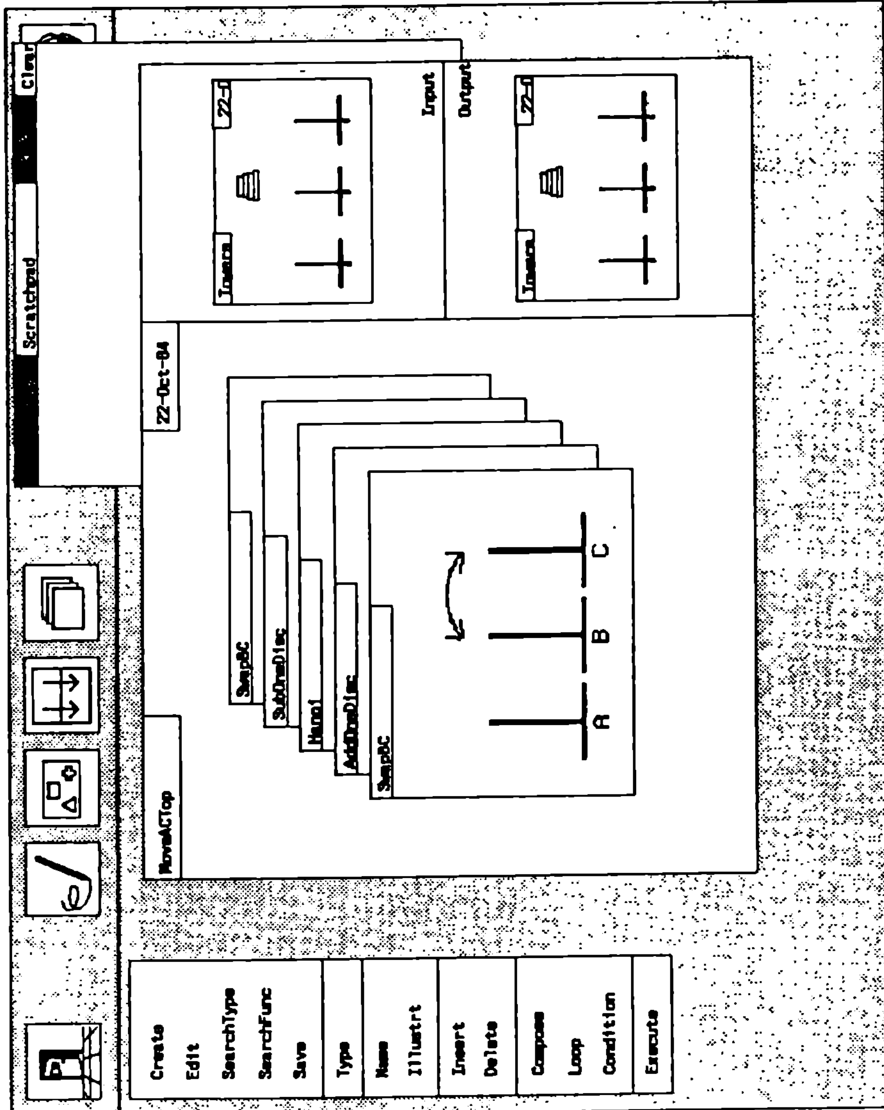


Figure 4-22: Swapping two towers.

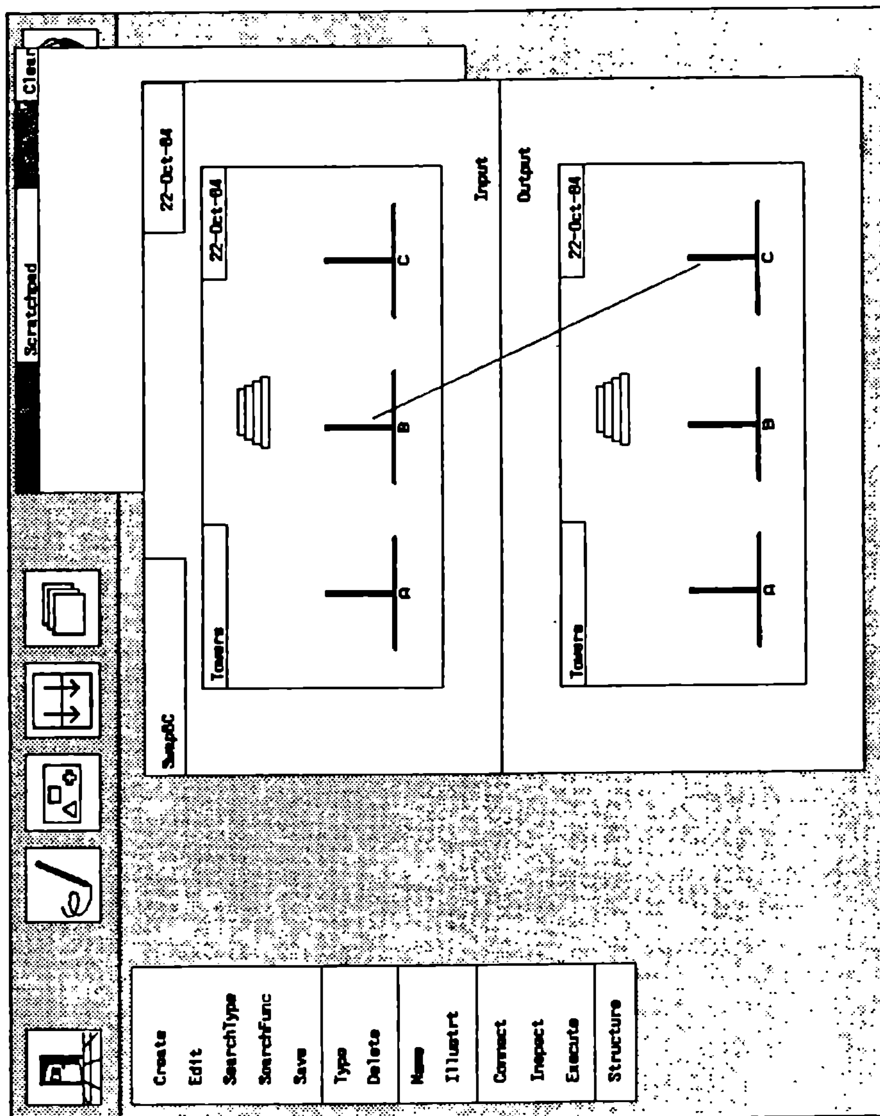


Figure 4-23: Subtracting a disc.

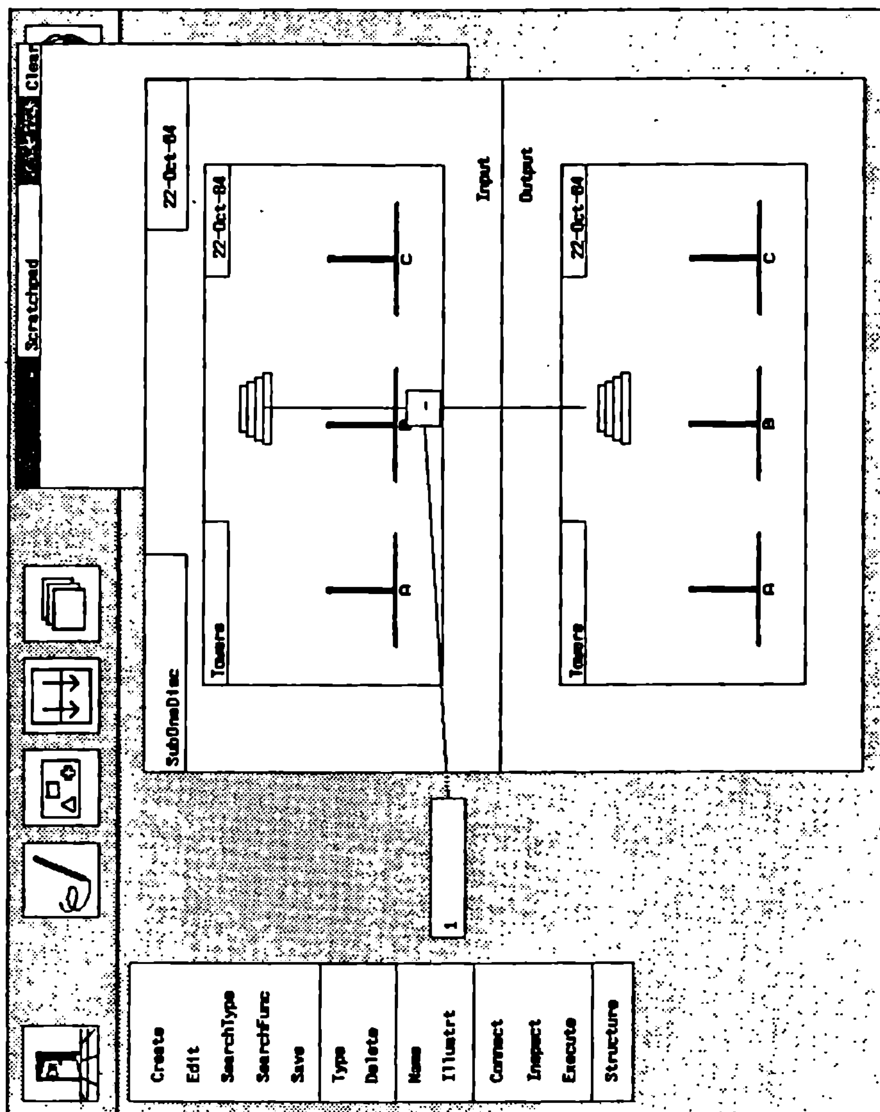


Figure 4-24 shows how one disc is moved from A to B. The *dec* operator will select the picture preceding the current picture value of peg A. Similarly, the new B peg is computed from the old one via the *inc* operator (not shown).

When the program is executed, the system will ask for initial data. Pictures are prompted for by zooming a picture of the picture type on top of the picture element (fig. 4-25). The picture value can then be selected by clicking the mouse on the desired picture constant. During execution, the picture values are displayed completely filling the space occupied by each element, i.e. in the space normally showing the element picture (fig. 4-26). In this way, stepping through the function will give an animated sequence of how the discs move among the pegs.

4.8. The typing mechanism

Functional Programming, as originally defined in [Backus 78], is untyped. As we explained earlier, we have included a typing mechanism in our system for two reasons: First, it allows the programmer to express more semantics as interesting pictures within the framework of the system. Second, we wish to benefit in the usual ways from type definition and checking.

There have been a few attempts at extending Backus' FP with a typing mechanism [Guttag 81, Frank 81]. The method used views functions and

Figure 4-24: Moving a disc from A to B.

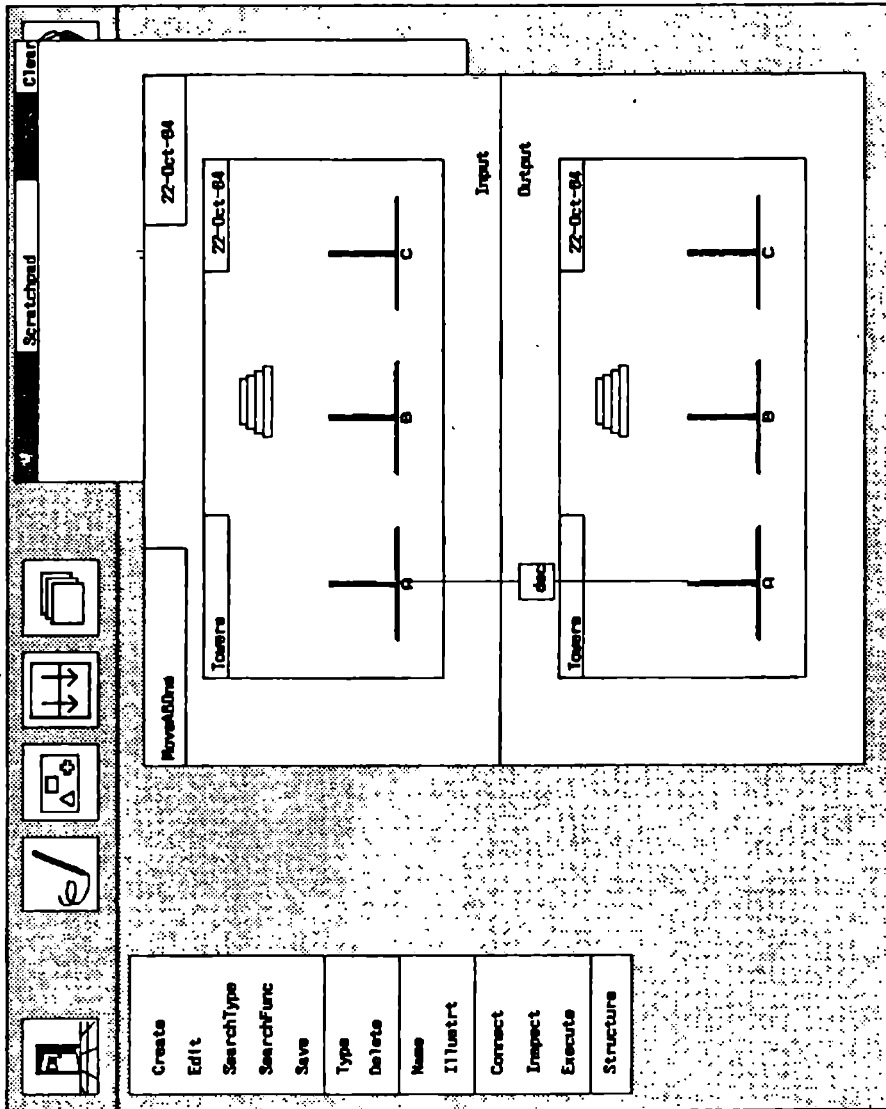


Figure 4-25: Prompting for a picture value.

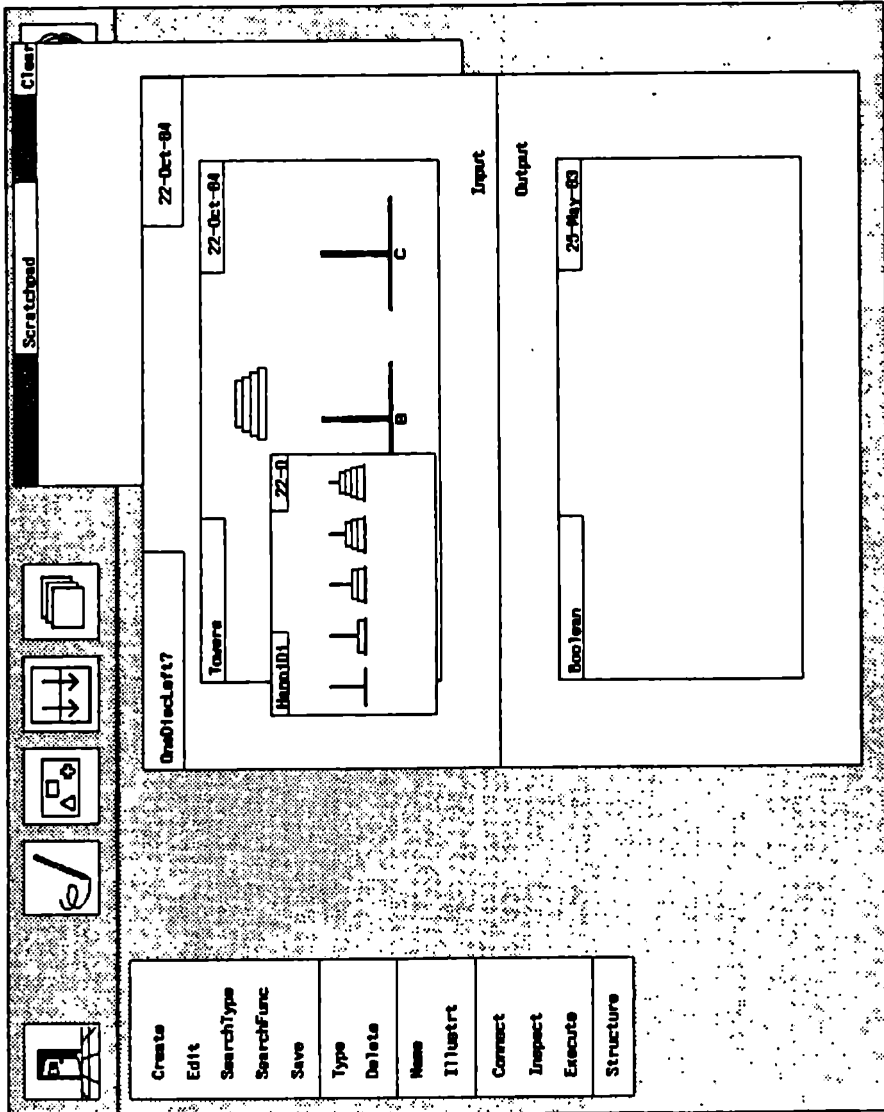
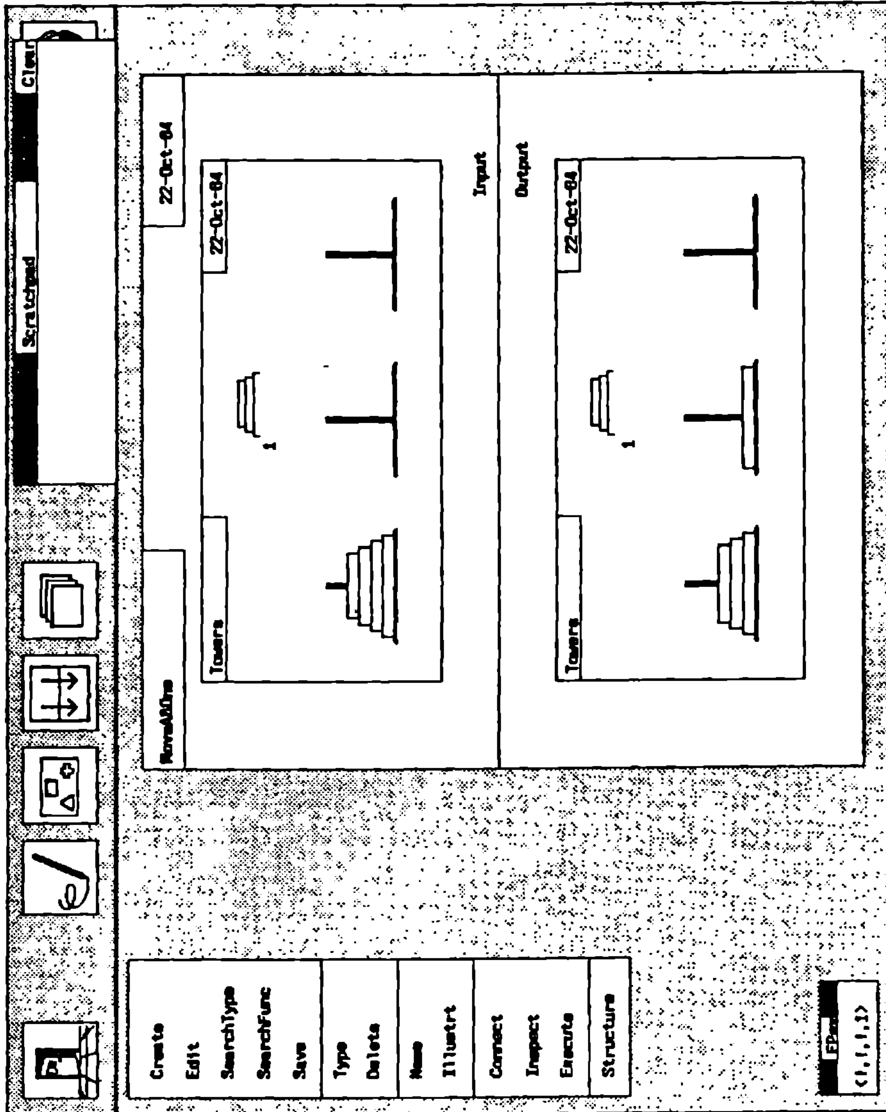


Figure 4-26: Animating the disc moves.



functional forms as type transformers. For example, the *apndl* function transforms a sequence, composed of an element of type T and a sequence of elements of type T, into a sequence of elements of type T. Similarly, the *apply to all* functional form has as a result a sequence of elements of the range (output) type of its argument function. When we program a function, we specify its domain (input) type and range type. Type checking of a function definition can now be performed by computing the range type from the domain type, using the transformation rules, and comparing it to the given range type. At each step, we also check that the intermediate result is compatible with the domain of the next function to apply.

The technique of type transformers has the advantage that it unifies generic functions, like *apndl*, and functions that always result in objects of the same type (modeled as a constant transformation). Thus, this framework can be used to handle built-in functions, that are often generic, and user-defined functions, that are usually specific.

In our present system, we do not provide the programmer with the capability of defining generic functions. Several generic functions are built in, however, e.g. arithmetic and list operations. The underlying typing mechanism therefore handles the general case. In detail, the type checks we described earlier are performed as follows:

- At the object level, pointing actions in the input type create expressions on a stack, and pointing actions in the output type compile these expressions into the function expression being built (cf. system description, above).

- Pointing at elements in the input type merely creates new stack entries, so no type checking is done here.

- When a function is applied to an expression on the stack, type compatibility is checked. For a user-defined function, this means exact match. For a built-in function, special rules depending on the function are followed, so that generic functions can be accommodated.

- At the same time, the resulting type of the application is computed. For a user-defined function this is the output type of the function. For a built-in function, special rules again apply, so that the resulting type can be a function of the input type.

- When an expression is inserted in the resulting function expression by pointing at elements in the output type, type compatibility is again enforced. Since in all cases a user-defined function is created, an exact match is required.

- At the function level, functions are connected together as in a dataflow graph. Since all function components are user-defined, an exact type match is required for each function combination.

- For composition, this means that the output type of the first function must be identical to the input type of the second. For the conditional, in addition the input types of all three functions involved must be identical, the output type of the condition must be *boolean*, and the output types of the branches must be identical. For the loop, the input types of both functions must be identical, the output type of the condition must be *boolean*, and the output type of the loop body must be the same as its input type.

Thus, the system could easily be extended to allow for generic user-defined functions as well, but this would entail a substantial addition to the user interface that would complicate the system beyond what we feel is appropriate for this work.

Note that the built-in functions are all available in separate menus during programming. This sets them apart from the user-defined functions that are found in the function library. Restricting the generic property to the built-ins thereby becomes far less artificial. One can of course argue that the split

between the two kinds of functions is artificial to begin with, and that it destroys some of the uniformity and simplicity of FP. Unfortunately, however, we cannot have a typed version of FP without either including an elaborate, general generics scheme (as in Ada), or making a split (as is done in most von Neumann languages). The solution we have chosen presents the programmer with a simple system that still provides access to the most important generic operators, and in a way that minimizes confusion.

Chapter Five

Programming by example

So far, our data structures have been quite simple. We have worked with collections of data elements, like Pascal *records*, each of which has again been a collection of smaller elements, or atomic pieces of data. In this chapter we will present the mechanisms available in PiP that allow more interesting structures, like arrays and trees, to be defined, and we shall see how this leads us to a version of programming by example.

5.1. Defining structured types

5.1.1. Sequence structuring

The basic data structure of FP is the recursive sequence, and it is natural to make this available at the programming level. An FP sequence is a generalization of the array. For example, the sequence *<1,2,3,4>* is an array of four elements that can be accessed much like an array in standard languages by means of the selection operators in FP. The generalization allows the elements to be heterogeneous. This works in two directions: First,

the elements need not be of the same type.[1] Thus, a sequence can be interpreted as a *record*, and this is what we have utilized to implement our typed objects with heterogeneous elements. Second, an element may itself be a sequence. If all the atomic elements in such a recursive sequence are of the same type, we have available *trees* of this type. For example, the sequence *<<1,2>,3>* may be depicted as shown in figure 5-1.

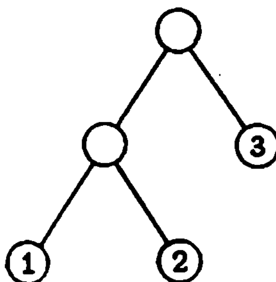


Figure 5-1: A sequence drawn as a tree.

If we wish to make this kind of structuring available in PiP, we have to decide how it is to be depicted. To do this, we have to identify what the components of the structuring mechanism are and give these a graphical representation. An FP object is composed of *atomic objects* that are connected via two mechanisms, *sequence*, represented by the ",", and

---

[1] Of course, in FP there are no types, but objects like *<23,T,nonsense>* map into mixed-type sequences in a typed version of FP.

recursion, represented by the "<>". Thus, if we determine how to depict atoms, sequence, and recursion, we should be able to depict any object structure. The way this is done in PiP is as follows:

- An element of a type may be specified as being a *sequence.* This means that the actual object value corresponding to the type element can be any recursive sequence of objects of the element type. We refer to the element type as the *base type* of the sequence. The structure can only hold components of the base type. (Note that the base type may itself be a structure of other elements, but it is considered "atomic" at this level of type definition).

- For such a sequence element, four additional attributes can be attached. They are:

  - A *node picture* that tells how an element of a corresponding sequence object is to be depicted. This picture is used for elements that themselves are composed of other elements or sequences of the base type. This corresponds to internal tree nodes.

  - A *leaf picture* that tells how elements that are not sequences, i.e. that are of the base type of the sequence, are to be displayed.

- A *sequence direction* that tells in what direction (right, left, up, down) the element pictures are to be juxtaposed to show sequence.

- A *recursion direction* that tells in what direction recursion is to be shown.

- A display algorithm, to be described shortly, is employed to construct pictures of arbitrary objects using the four attributes above as building blocks.

To see how this works, consider figures 5-2 and 5-3. In figure 5-2, the drawing of the bottle is attached as the leaf picture to the element in question, and the sequence direction is set to right. A sequence containing five elements of the base type, $<b_1,b_2,b_3,b_4,b_5>$, will then be drawn as shown. This is the way we display arrays. Note that we only need two of the four attributes, the leaf picture and the sequence direction, to display linear structures. Another example is shown in figure 5-3. Here we attach node and leaf pictures of the familiar kind used in tree structure illustrations, sequence direction right, and recursion direction down. An element of the form $<<l_1,l_2>,l_3>$ will then be displayed as a tree.

Briefly, the algorithm, upon encountering a sequence, splits the available

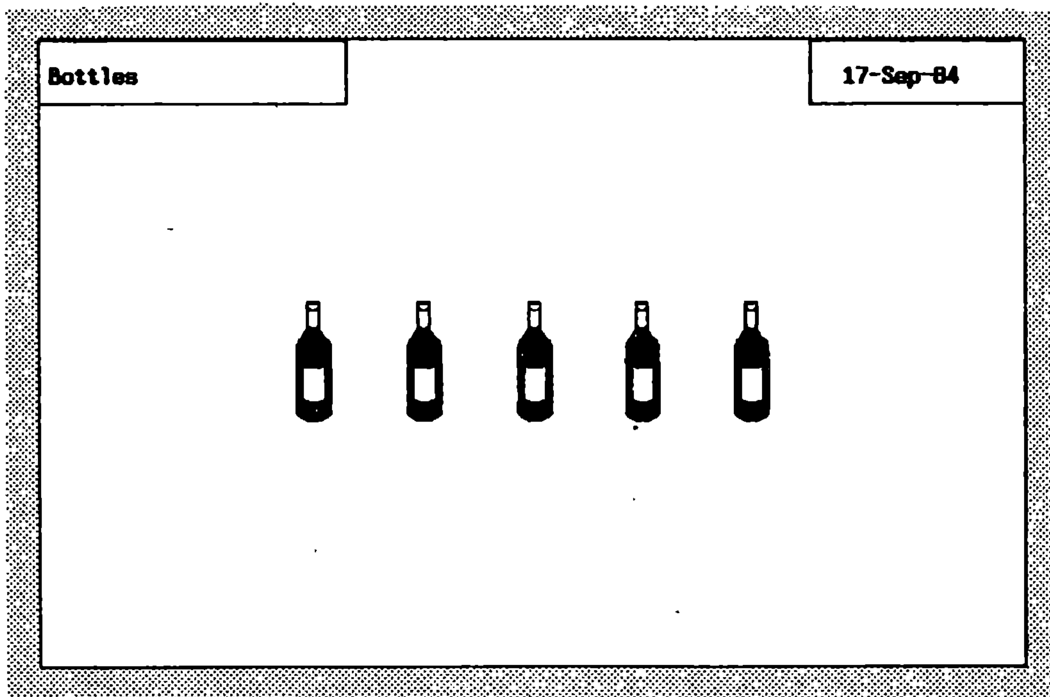


Figure 5-2: Sequence structuring.

space into adjacent boxes in the sequence direction, displaying one element in each. In the case of recursion, the space is similarly split in the recursion direction, the node picture displayed in one part and the next level of recursion displayed in the other part of the picture. Algorithm 5-1 shows the procedure in detail.

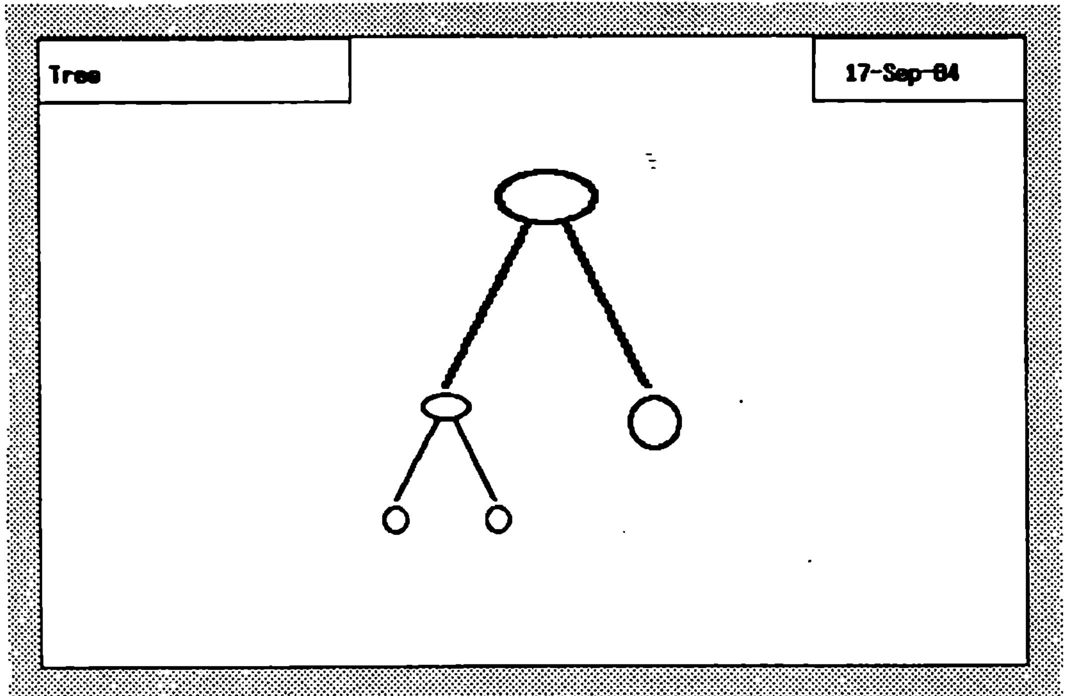


Figure 5-3: Sequence structuring.

```
Display(S, B, E) =
   -- Display object S in rectangular area B,
   -- according to the element description E.
begin
   if S is of E.basetype then display E.leaf in B;
   else -- S = <s1,...,sn>
      split B in 2 pieces b1 and b2 in E.recursion direction;
      display E.node in b1;
      split b2 in n pieces c1,...,cn in E.sequence direction;
      for each si in S do Display(si, ci, E);
   endif;
end;
```

Algorithm 5-1: Sequence structuring.

The four element attributes and the base type of the element are referred to

in an obvious manner. By **display** X **in** Y, where X is a picture and Y is a rectangular area, we intend that X is resized to fit Y and then shown within Y. By **split** Y **in** i **pieces** I **in** Z **direction** we mean that the rectangular area Y is split in i equal-sized areas. The splits run orthogonally to the direction Z, so that the pieces are juxtaposed in the Z direction, and are given names I in the same direction. Thus, if **E.recursion** is *down*, **b1** is the upper and **b2** the lower half. If **E.sequence** is *right*, the elements are displayed from left to right.

The four sequence display attributes have default values, so that in most cases they may not need to be specified. The pictures default to the picture icon of the element base type. The default sequence direction is to the right. The default recursion direction is *none*. This corresponds to the display not being split in the recursion direction at all. This is what is desired in most cases, where a simple, linear array is to be displayed. Indeed, a tree structured sequence is really not very useful, since no data can be stored in the internal nodes of the tree. This makes e.g. a search tree impossible, rendering such a structure rather vacuous. The section on recursive structuring, below, shows how more useful tree structures can be obtained.

The type editor **structure** command is used to attach the sequence attributes, via four commands on the submenu.

The display algorithm is not applied automatically to any structured element. Rather, structural display is a display function along with zooming, window overlap and on-line assistance. It can be invoked in the type and function editors by the **show** subcommand of *structure*. When *show* is pushed and a sequence element is next selected, this element will be displayed according to the algorithm above. This can of course be applied recursively to the elements of the sequence. When applied to a non-sequence element, the type-icon of the element type is shown. Recursion can be turned off by **don't show**.

Recursive display is most meaningful in the function editor when it can be applied to an actual data object. If no data are available, as in the type editor, a small example data object will be substituted by the system.

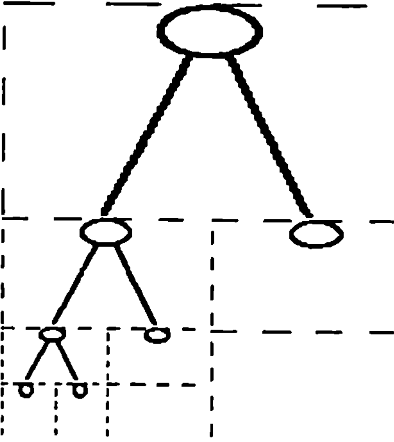
There are several possible variations to the algorithm above. The only natural way to make the sequence split is probably to create equal-sized pieces, but there are several alternatives for the recursion split. The algorithm above splits according to some fixed factor, e.g. 0.5 (giving two equal-sized pieces). We could also ensure that each level of recursion has the same space by first finding the number of levels l, splitting the area to begin with in l pieces, and then replacing the split statement by "move one level down". Figure 5-4 shows four different displays of the same structure, with

a) a recursive split with factor 0.5, b) a recursive split with factor 0.707, c) a linear split, and d) a Fibonacci split. The advantage of the recursive splits is that a structure of any size can be displayed without the upper-level nodes getting too small (the size of a node is independent of the size of its subtree). This is important, since these are usually the nodes we are interested in. Correspondingly, the linear and Fibonacci splits show the total tree structure better. In our PiP system, we have used a 0.5 recursive split.

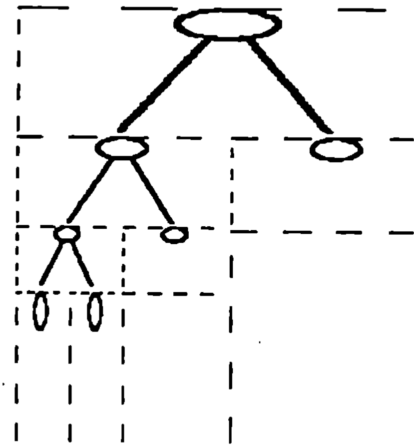
5.1.2. Recursive structuring

As we saw above, the kind of recursive structures we can build with sequence structuring is not very useful in practice, so this mechanism is mostly used to build arrays. But the recursive definition of types as sequences of elements of other types gives us another way to define recursive structures. If a type contains an element of the type itself (or there is some such element somewhere down the type hierarchy) we have potentially recursive data objects. In terms of FP, these objects are like the recursive objects above, but each internal node can now carry data, since the type description can provide for other elements than the recursive ones at each level. To display these structures, we could use the zoom mechanism to open up successively deeper levels of recursion, but we can also use essentially the same algorithm as above. The following enhancements are needed:

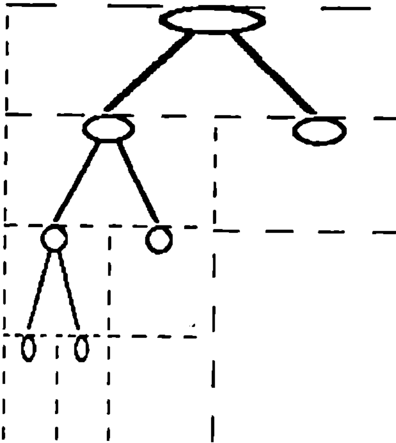
Figure 5-4: Display alternatives.



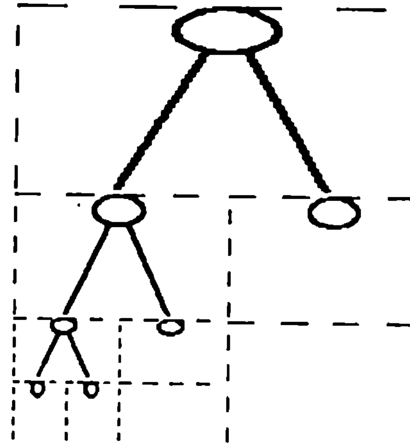
a)



b)



c)



d)

- Mixed recursion is now possible since there can be nodes in the tree of different types. The element description therefore cannot be passed from one level to the next, but must be found at each level.

- The test for termination now must check whether the object to be displayed is of a type that is not recursive or that the programmer does not wish to see recursively. Since the recursive type is a structured type (in the FP sense, a sequence) recursion along the main line will not stop at this criterion, but must stop when the object in question is empty (the empty sequence, Φ). We are usually not interested in displaying an empty node, but rather would like a node with empty components ("children" in tree terms) to be indicated as a leaf node. We therefore display a leaf node in the recursion direction if all elements in that direction (see the following) are empty.

- In each type, there may now be several elements that are to be shown recursively (mixed recursion), each with its own display attributes (pictures and directions). The splitting actions therefore become more complicated. The recursion split is performed as shown in figure 5-5. First, the elements to be displayed recursively are examined to establish in which directions recursion is to be made. If recursion is in a single

direction, the algorithm is as before. If recursion is in more than one direction, a multiple split has to be done, as shown.

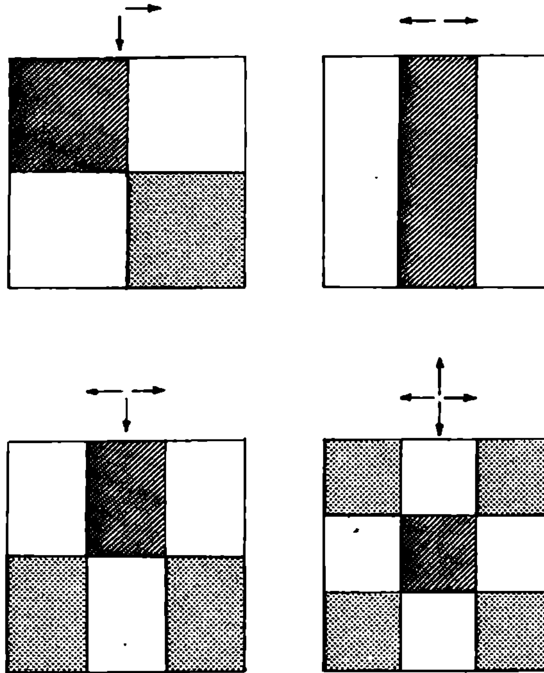
- The sequence split is now done within the space for each recursion direction. All elements with the same recursion direction are displayed sequentially within the space for that direction. To avoid confusion, only one sequence direction is allowed for each type, and all elements will be displayed according to this direction.

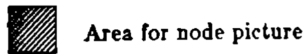
- The *structure* command is used for attaching recursive attributes and displaying recursive structuring as well, thereby unifying and simplifying the concepts for the user. There may be several elements in a type that can have a recursive display, and it is left up to the user to point out which ones it is desirable to display recursively. The display algorithm will therefore only pursue recursion on those elements the user explicitly has selected with the *show* subcommand. The order in which recursion is specified in each direction determines the order for the sequence part of the algorithm.

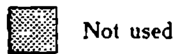
The modified algorithm is shown in algorithm 5-2.

Figure 5-5: Recursive splitting.

Directions:







```
Display2(S, B) =
   -- Display object S in rectangular area B.
begin
   let E be the element description of S;
   let L be the list of descriptions of elements of S;
   if no element in L is to be displayed structured then
      display E.leaf in B;
   else
      let r be the number of recursion directions of L;
      split B in r+1 pieces b0,...,br according to fig. 5-5;
      display E.node in b0;
      for each recursion direction di of L do
         if all elements of S
            with recursion direction di are empty then
            display E.leaf in bi;
         else
            let n be the number of elements
               with recursion direction di;
            split bi in n pieces c1,...,cn in E.sequence direction;
            for each element sj of S with recursion direction di do
               Display2(sj, cj);
         endif;
      enddo;
   endif;
end;
```

Algorithm 5-2: Recursive structuring.

Figure 5-6 shows a simple example of recursive structuring. The *BinTree* type describes a node of a binary tree. It contains some data, an integer key field, and left and right subtrees. The subtrees are defined as elements of type *BinTree* itself, as can be seen in the zoomed picture, figure 5-7. Attaching node and leaf pictures, and recursion direction *down*, we can display three-node example subtrees as shown in figure 5-8.

Figure 5-9 shows another example, using the same *BinTree* type, but with the

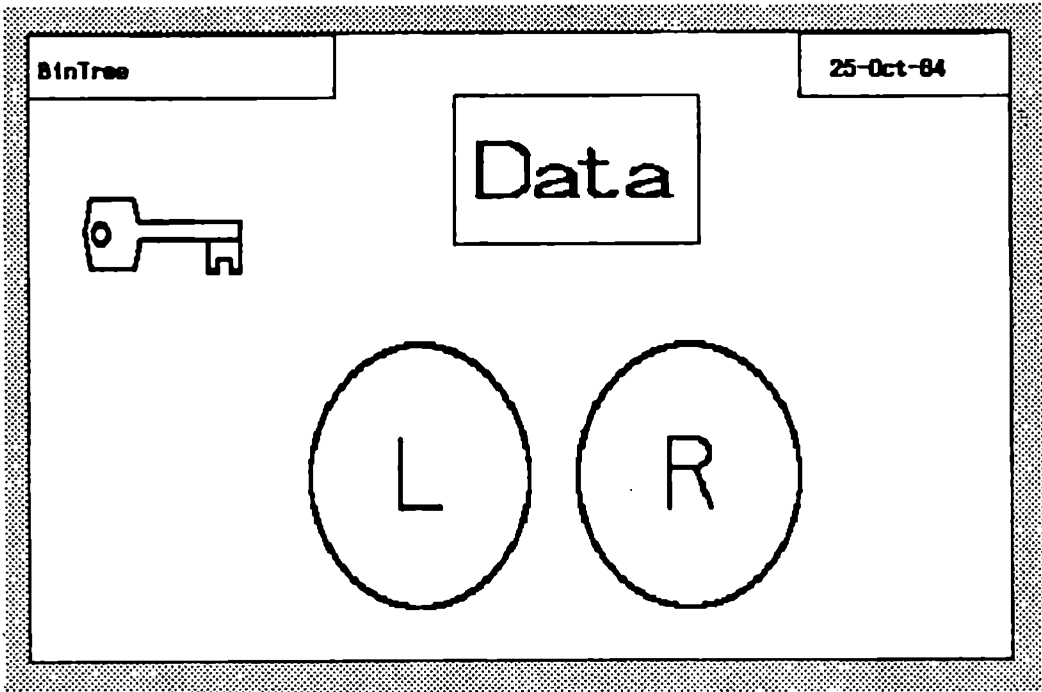


Figure 5-6: Recursive structuring: a binary tree.

two recursive elements acting as child and sibling pointers to allow any fan-out. We have simply changed the recursion directions, to down for the child and right for the sibling. Figure 5-9 shows a type CSTree that contains one element of type *BinTree*, and we have specified recursion on both the child and sibling elements of *BinTree*. We have shown the same structure with two different sets of pictures, resulting in one regular tree node diagram, and one LISP-style diagram.

As with the sequence structuring algorithm, the display attributes for

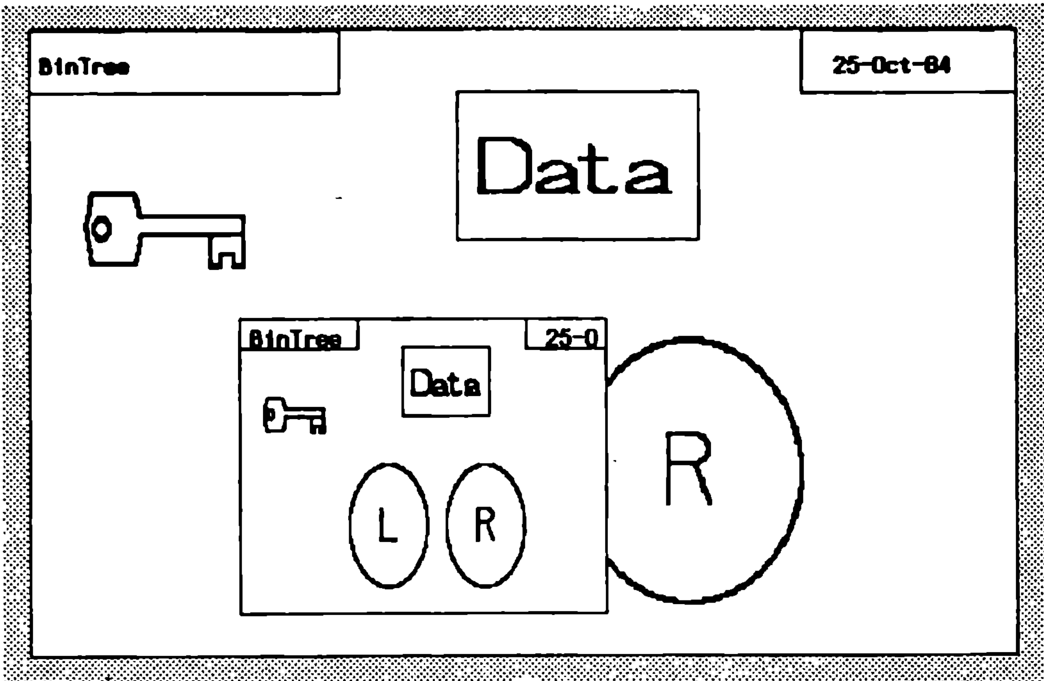


Figure 5-7: Zooming the subtree structure.

recursive structuring have defaults that simplify the use of recursive data structures. They are the same as for sequence structuring, except that the recursion direction defaults to down, since this is most often used.

Note that this algorithm is completely general, in that it can be applied to any type element. If the element is of a type that does not contain any recursive elements, such as a basic (built-in) type, the recursion will simply stop, and the type icon of the element (which is the default picture) will be shown. Also, we see that it is a straight generalization of the previous

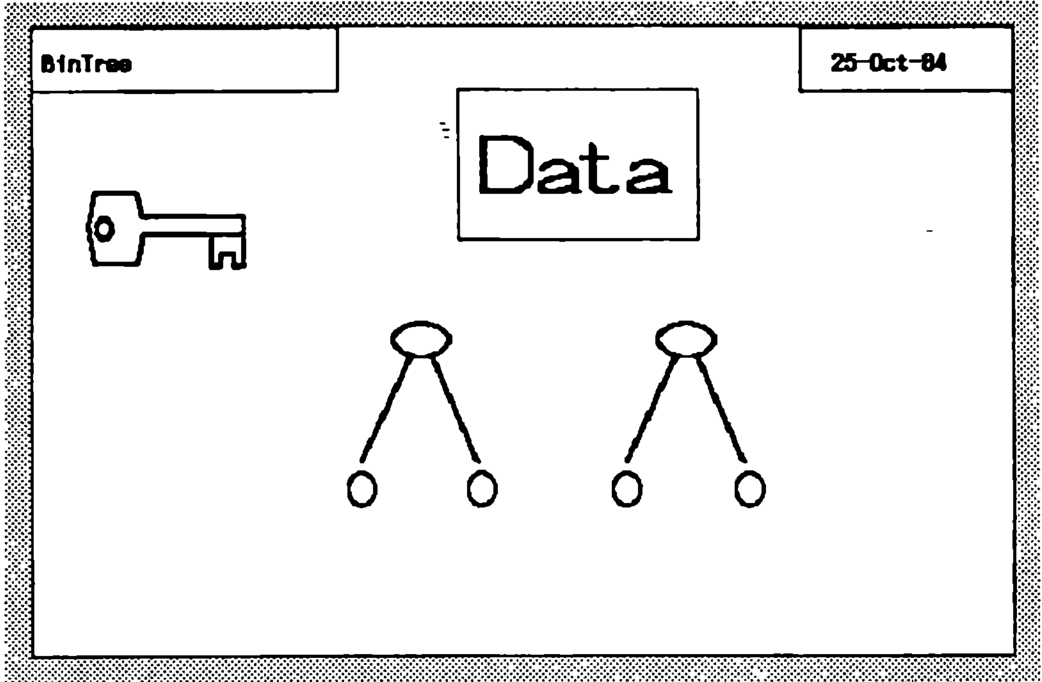
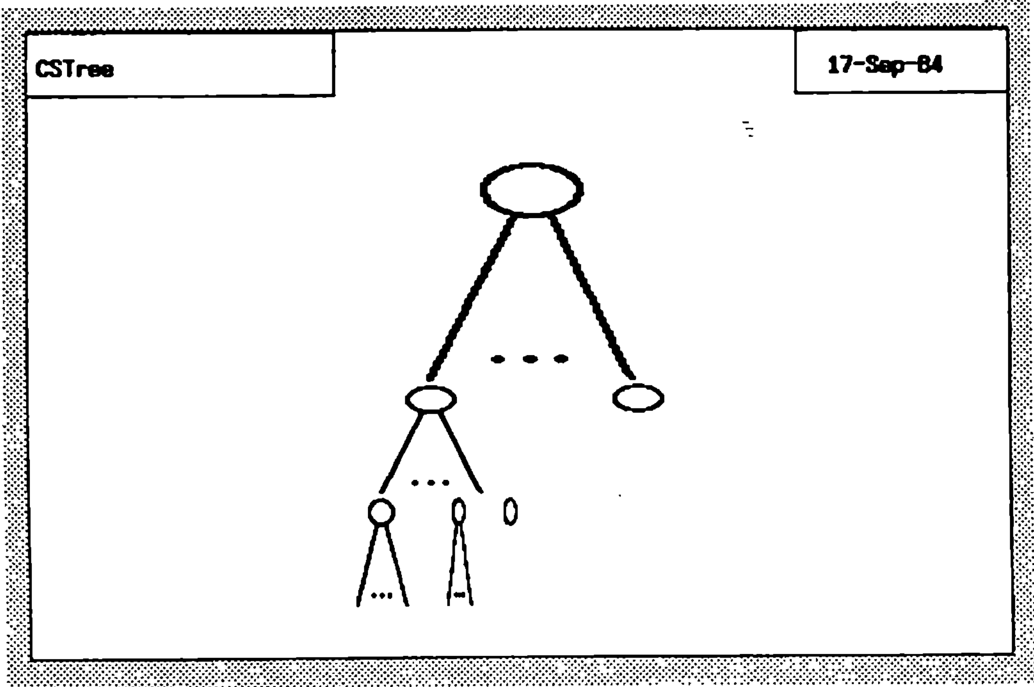
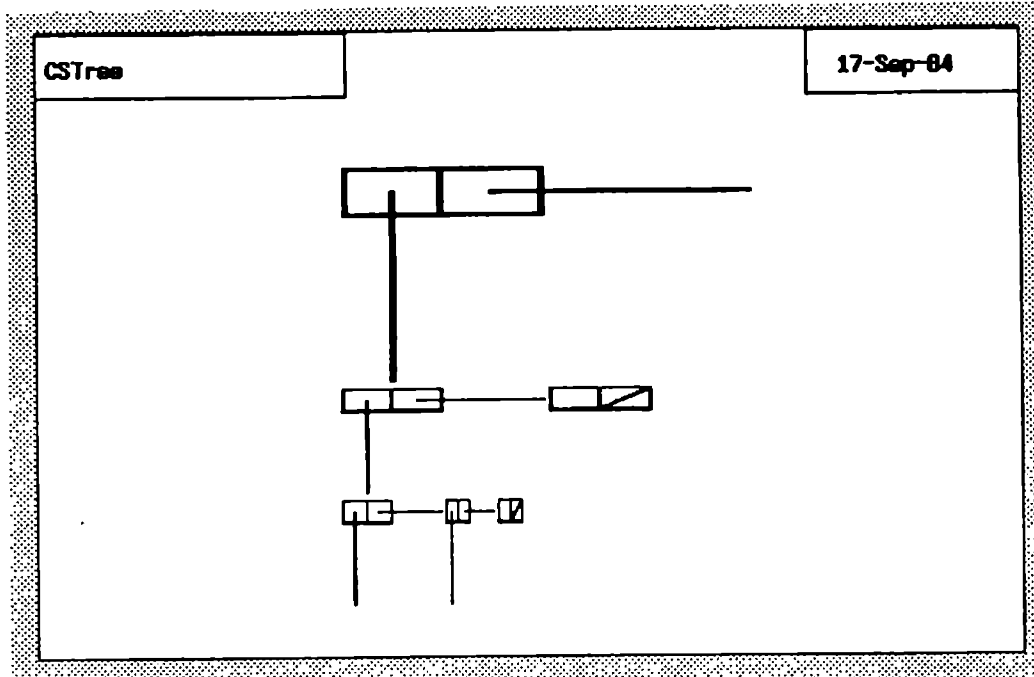


Figure 5-8: Recursive subtree display.

Figure 5-9: Recursive structuring.





algorithm, so that, with the proper interpretation, it incorporates the latter and can be applied to sequences as well. This has the implication that we do not need to be distinctive about what kind of element we attach the structured display attributes to. We simply say that they can be attached to any element. For an element of a recursive type, and for a sequence element, they will be used as described. Otherwise they are ignored.

Hence, we have achieved an important unification of two related, but different, mechanisms in the user interface. From the user's point of view, structuring attributes are general properties. From the system's point of view, no confusion can arise, since it is always known whether we wish to see a structured display of a sequence or of a simple element with potential recursion.

5.2. Programming with structured data

5.2.1. Executing with structured display

The display mechanisms for structured data must be applied to *objects* of such data. This means that it is only during program execution that we can benefit from this kind of display. We have therefore not contributed to the task of function programming, we have just made it easier to show input and output data in an illuminating way during execution. As such, the

mechanisms are certainly very useful, but we would have liked to give the programmer the opportunity to work with the same kind of pictures during programming as well. This leads us to the concept of programming by example.

5.2.2. Programming by example

When we worked on types with only simple data elements, there was a very close correspondence between the picture of an element, which actually describes a whole family of possible data, and a piece of data that is an instance of the element during execution. This is an important mechanism, since it is this that gives the programmer the feeling of working on concrete data instead of referring to remote entities via proxies. It is made possible by laying out the structure of the data and drawing pictures that can be identified with them. In the same way that ordinary pictures are interpreted as windows into a part of reality rather than indirect references to objects out of view, the concrete layout of data is interpreted as the data themselves rather than a description of them.

It is the static nature of simple data that allows us to draw concrete, static pictures of them. We would certainly like this to work for dynamically structured data as well. This means that the structured pictures we developed above have to be available at the programming level too, since we want to display types in terms of layout of objects.

What does it mean when we start displaying data structure in the type slots of object-level programming? When we supply a type description as input type, we say that the program should accept data of a certain structure only, and then process them as further described. For static data, we can specify the structure fully as a type. For dynamic data, we can go one step further by elaborating on an input element, and making the system show the structure of it: If the input data not only satisfies the type description, but also is of a form compatible with the data structure shown, then we can go on processing it. The natural interpretation of a data structure elaboration of an element is that the data structure displayed is a *specification of acceptable input data*. Further, the data shown impose a *minimum requirement* on the actual data during computation. For example, if we expand a sequence element to have two components during programming, we can program in the usual way, pointing to any of these two components as parts of expressions. This means that an object during execution also must have *at least* two components, otherwise the result would be undefined.

Similarly, an output type specification indicates that the output is going to be of a certain structure. Elaborating an output element says that the output data should have at least this much structure.

The step to programming by example is now a short one: If the function is

independent of the structure of the input, or it always requires the same input structure, we specify this function as we have done before, possibly with the added requirement of an input data structure. But what if the function depends on the structure of its input? For example, a recursive tree traversal algorithm will perform differently on a leaf node and an internal tree node. We can easily handle this situation by allowing *function variants*. By letting the same function be defined several times, each with different input data structure requirements, we get a collection of functions that together handle all interesting variations of input data. The function variants define *example data* since each variant handles a whole family of possible objects compatible with the example provided.

Thus, we are talking about a *structural* form of programming by example. Programming with structured types takes the form of telling the system, "if the data look like this, then do this". Looking back at chapter 3, where we observed how people like to illustrate their programs, we see how well this corresponds to the pictures we draw and how we explain programs: We draw examples of typical data structures and describe how the program works on them, leaving the generalization up to the listener.

Generalization is the hard part of programming by example. How can we generalize from example data structures in a way that seems obvious and

deterministic to the user? The key to a useful generalization scheme was already mentioned above: Each specified data structure represents a minimum requirement. All data objects containing at least this structure can potentially be subjected to the corresponding function variant. Among all admissible variants for an object, the one requiring the most structure will be selected. We will describe in detail below how programming by example works in PiP.

Function variants and matching criteria. In PiP, several functions can have the same name. Functions with the same name are taken as function variants. That is, when a function with a certain name is to be applied, all functions with that name are considered, and the one found most compatible with the input data is applied. The compatibility check is performed as follows:

- The input type of the function must be the same as the type of the data. This is guaranteed by the function editors, which will not accept composition of incompatible functions.

- Each of the elements of the type in question is examined. If an element has a data structure description associated with it, the corresponding object element is also examined to determine whether it is compatible with the description.

- For a simple data element, the description can only indicate an empty or non-empty element. A non-empty object element is compatible in both cases, whereas an empty object element requires an empty element description to match.

- For a structured data element, the description indicates a tree structure. Any object element that contains the tree as a sub-structure is compatible.

- For a sequence structure, this specializes to the number of sequence components used. If the object element is a sequence with at least this number of components, it will match.

- Among all the function variants that match a given object, the one with the best match is chosen. The "best match" here means the maximal matching data structure description. "Maximal" is in turn interpreted as the maximum number of tree nodes (or components of a sequence). This means that a variant with a complex data structure description can readily be defined as a special case overriding the simpler, general case.

- If several elements have data structure descriptions attached, a

compromise may have to be found between best matches of the different elements. The globally maximal match is found in this case. Note that a situation with more than one element with data structure description attached, and especially if the global maximum does not coincide with the local maxima, probably indicates a badly programmed function that should be split up.

All function variants must of course have the same input and output types, in order to allow type checking at edit time.

The **expand** subcommand of *structure* is used to attach data structure requirements to functions. If *expand* is used on an element, a singular (one node) data structure is first shown. Then, successive expansions can be used to build the desired structure. Expanding a sequence gives one more component. Expanding a sequence component or a recursive element gives one more level of recursion. The zoom facility may be used to obtain the structure of an element so that the right sub-element can be expanded. Inversely, the **contract** subcommand will cause a structure to shrink. The element pointed at will disappear.

Code generation from structured data examples. Once the proper data structures are set up in the input and output parts of the function

template, programming proceeds as before. The correspondence between example elements and elements in the actual data object is direct. That is, the nodes or components of the actual data are matched directly to the example data. For example, pointing at the second component of an example sequence generates code that always selects the second component of the actual data. Pointing at the right son of the left son of the root of an example tree generates code that does the same selections on the actual data.

This choice is deliberately very simple. Any "clever" choices, like mapping the last component of an example sequence to a function finding the last element of any data sequence, will necessarily confuse the programmer. If the example data consist of a two-component sequence, pointing to the second element sometimes means "second" and sometimes it means "last". So, whatever interpretation the system applies, it will be wrong some of the time. But it is far better to work with a system that may be wrong sometimes because it is consistent in a case where the user is not, than with a system that fails when the user is consistent.

5.3. An example

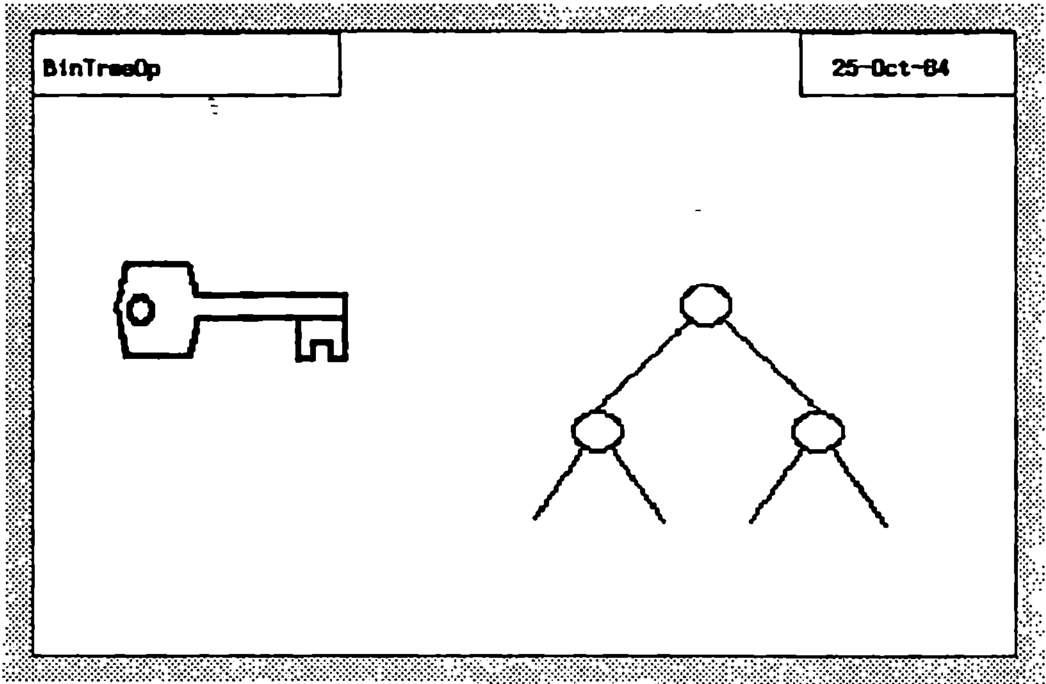
As an instance of programming by example in PiP, consider again the binary tree node defined in figure 5-6. We will program a function that searches for a given key in a tree of such nodes. Programming by example will be needed, since the progress of the search will depend on whether the node examined is a leaf or internal tree node.

Figure 5-10 shows two new data types we will use. The input to the search problem is described by the type *BinTreeOp*, consisting of a key to search for, and a tree of type *BinTree*. (The tree shown is the icon of the *BinTree* element.) The result of the search will be in terms of type *BinTreeData*, containing the key and the data of the node found. If a node with the given key is not found, the returned key will be -1 (all keys assumed nonnegative).

Figure 5-11 shows the search function. It simply checks whether the root node of the input tree has the desired key. If so, the data are extracted from the root (*BinGetData*). Otherwise, the search continues to the next level, moving down to the left or right child (*BinGoDown*).

The *BinGetData* function (fig. 5-12) extracts the data from a zoomed picture of the root node. The *Bin=* predicate similarly compares the input key and the key of the zoomed root. *BinGoDown* (fig. 5-13) is further composed of

Figure 5-10: Types for binary search.



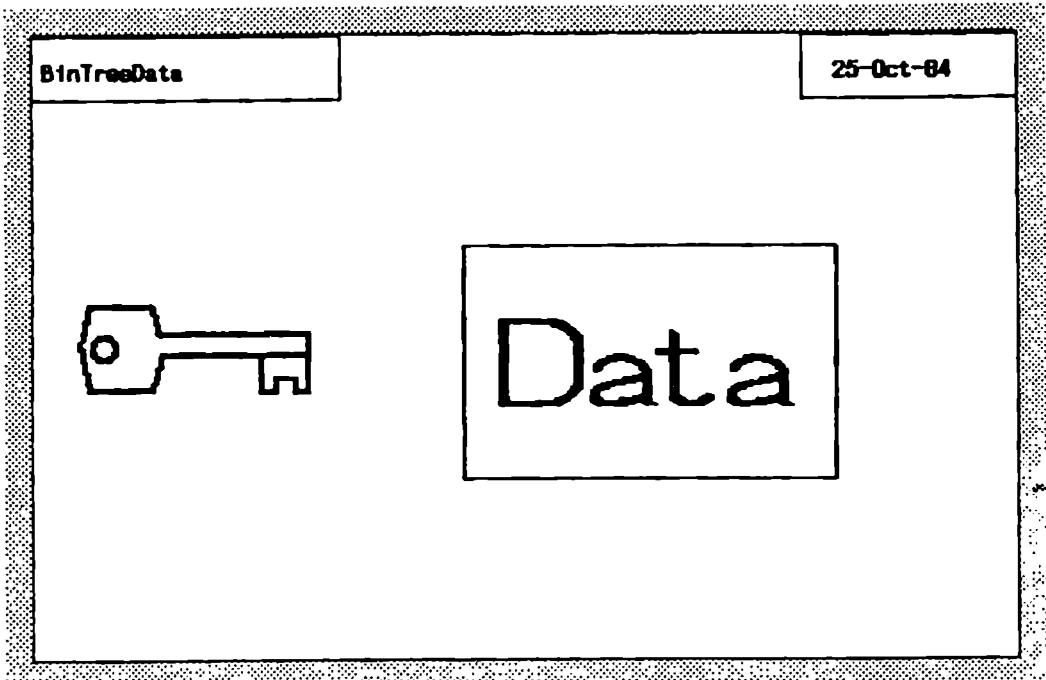
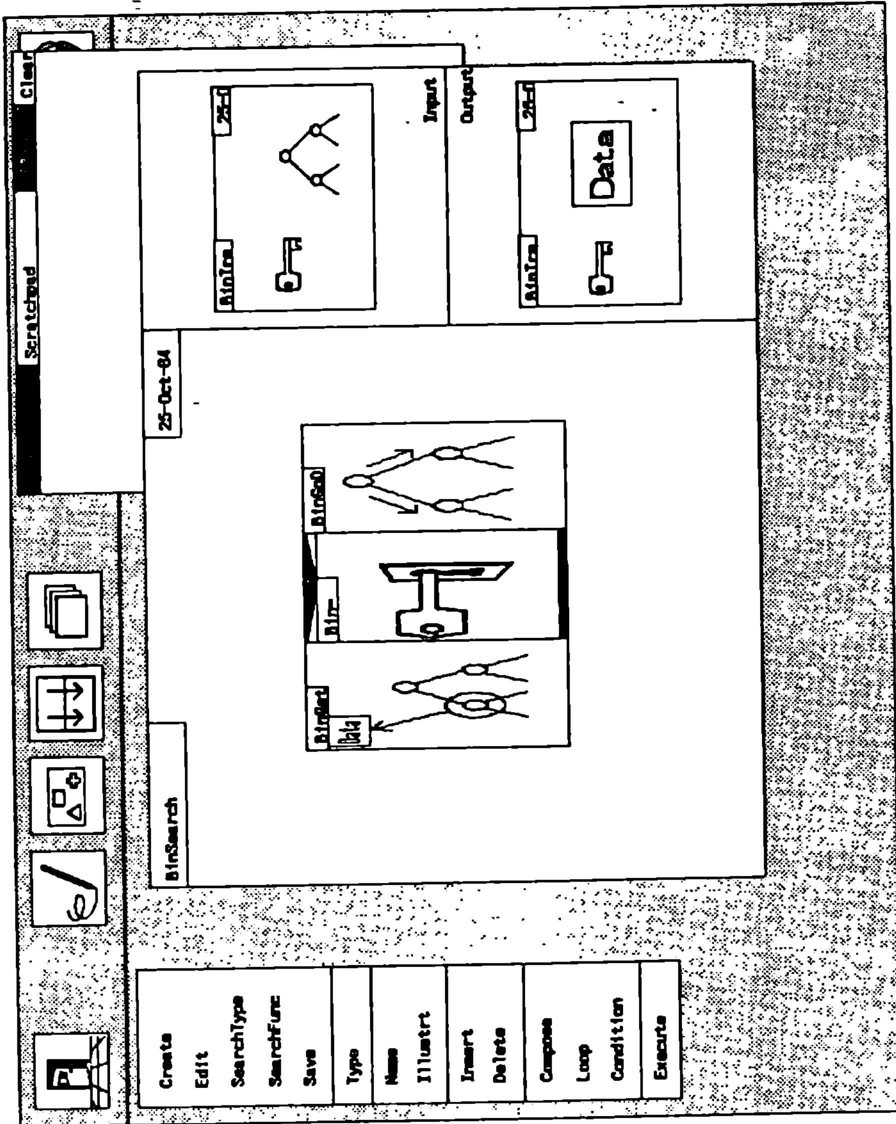


Figure 5-11: Binary search function.



several functions. Depending on whether the root node of the input tree is less or greater than the key, the search proceeds to the left or right (functions *BinLSearch* and *BinRSearch*). Next, the *BinSearch* function is applied recursively.

Finally, *BinLSearch* and *BinRSearch* are where programming by example can be used. For if the input tree is actually a leaf node, the key was not found, and we have to return the value -1. Otherwise, if the input tree is not a leaf, we shall select the left (right) subtree. We program this as two function variants for each of *BinLSearch* and *BinRSearch*.

Figure 5-14 shows the first variant of *BinLSearch*. We have expanded the tree element to showing a single leaf node. In this case we put the constant -1 in the output key, and also in the key of the output tree node, since the search will then stop upon the next application of BinSearch. (The other elements of the output tree node are irrelevant and can just be copied from the input.)

Next, in figure 5-15 we have expanded the input tree to show one level of recursion for the other variant of *BinLSearch*. Here, the left subtree can be pointed at directly and copied to the tree element of the output, effectively moving down to the left child. The key field should also be copied. When

Figure 5-12: Extracting node data.

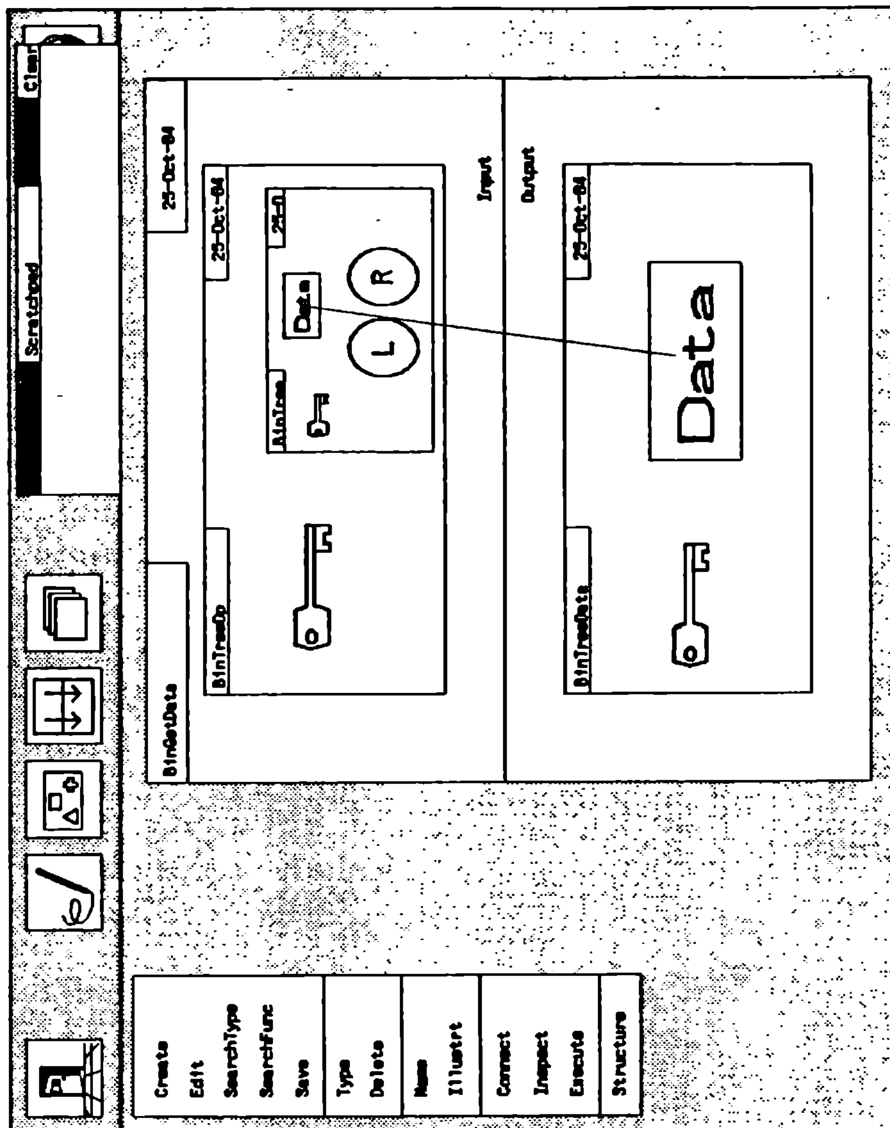


Figure 5-13: Moving down to a subtree.

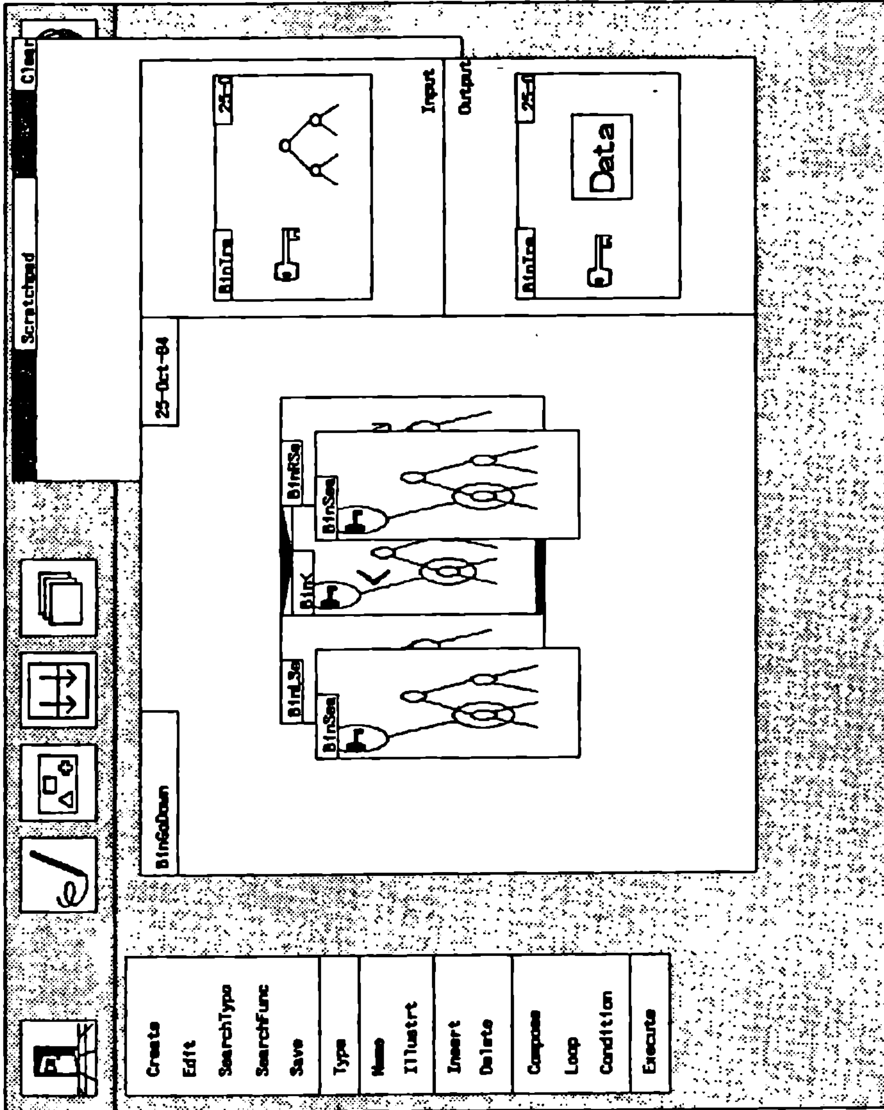
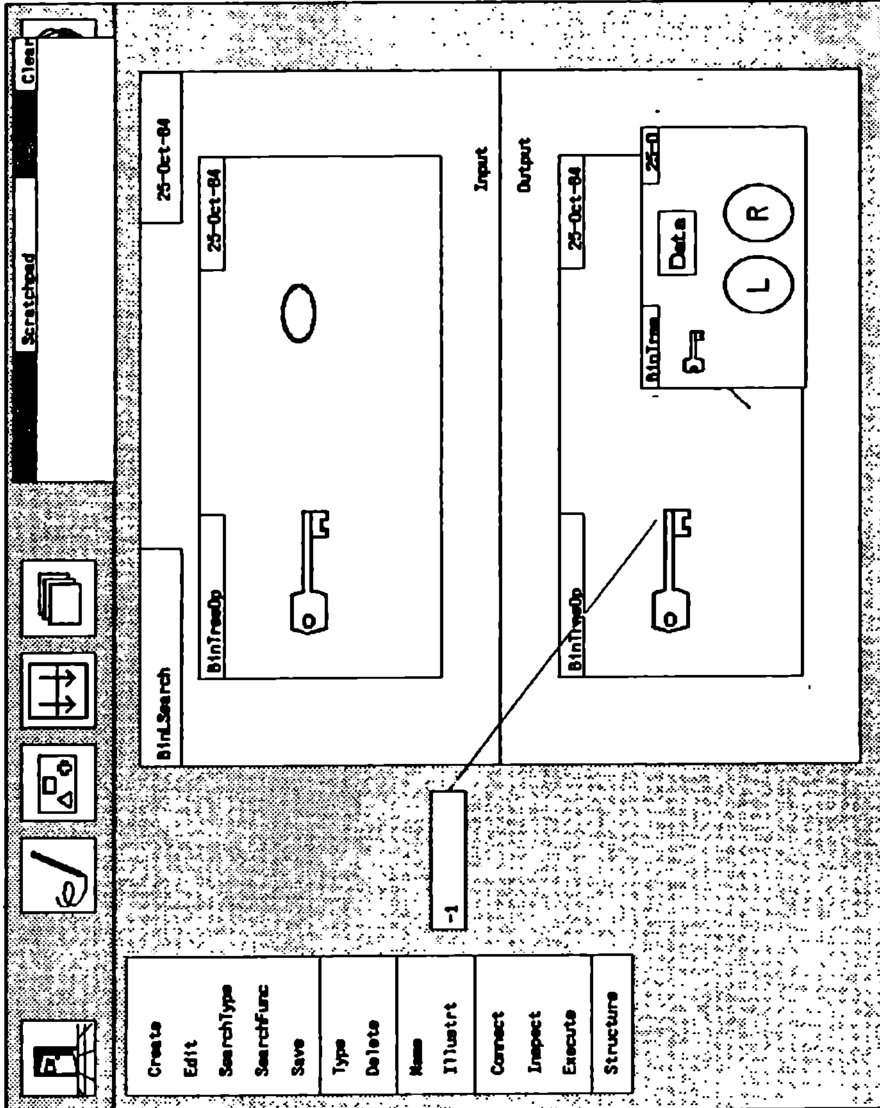


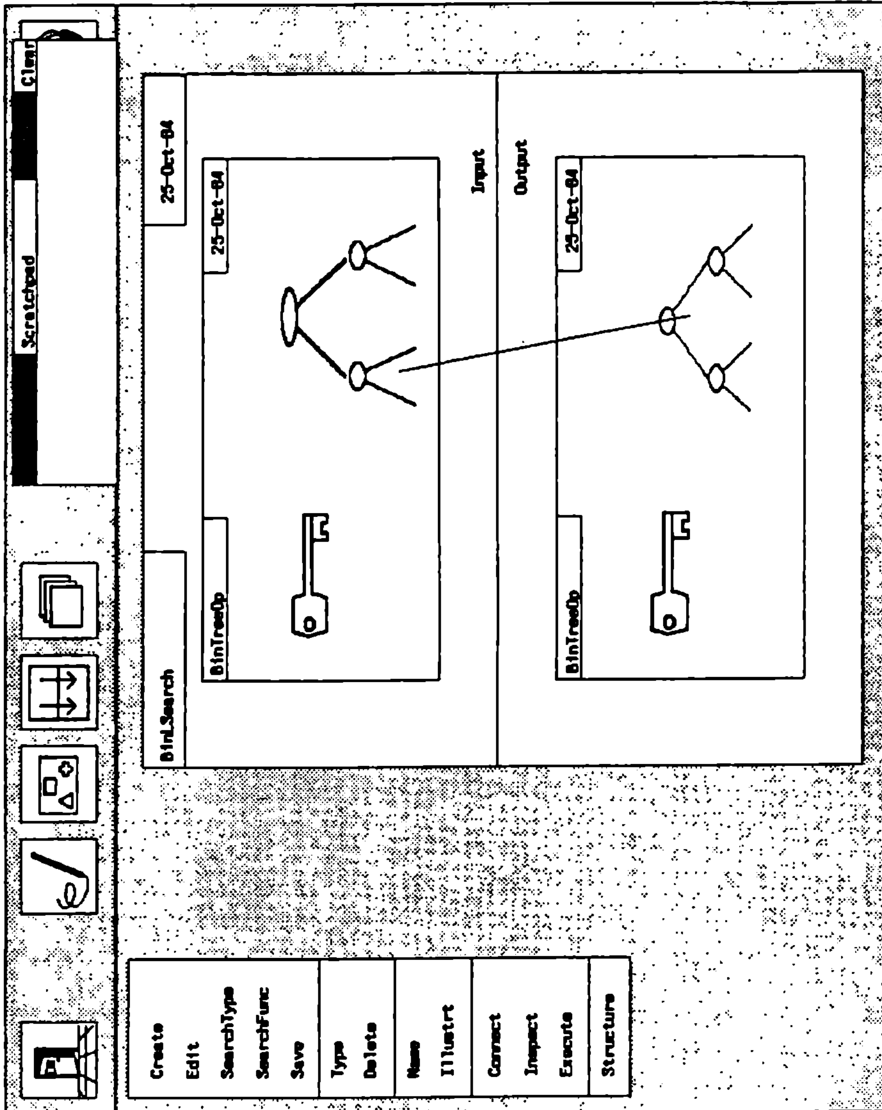
Figure 5-14: Leaf node variant of left search.



executing the *BinLSearch* function, any actual tree with at least the structure shown (a non-leaf root node) will match the data structure required by this variant, and the left subtree will be extracted. The leaf variant of figure 5-14 will also match, but since the other variant has more structure it will be selected in all cases except when the input data contain a leaf.

Splitting up the definition of *BinLSearch* clarifies the case analysis and allows the user to focus on one situation at a time. The cases are programmed using displays of example data, providing a concrete structure to think about, but at the same time covering all possible input data structures.

Figure 5-15: General variant of left search.



Chapter Six

Discussion and conclusions

6.1. Casual programming

When designing and discussing a programming system it is certainly important that we define the intended audience of the system, and determine how and for what purposes these people are supposed to use it. We have previously indicated that we have primarily addressed naive or casual users. But what kind of programming will these users be interested in? In this section we describe more precisely the class of applications we have in mind for our system, and comment on the characteristics a system for such use should possess.

6.1.1. The need for a casual programming tool

In chapter 2 we mentioned that there is a growing need for end-user programming tools in the electronic office environment. This is to provide the flexibility required by rapidly changing and individualized routines. But this is only one part of a general need for employing the powerful microcomputers to do specialized information processing tasks.

For the last four years, the author has himself enjoyed the pleasure of having at his disposal a microcomputer in the home. Apart from some standard applications, of which text processing dominates, the most important use of this computer has been to quickly create small programs for specific problems. A typical example is a program that computes the interest on a bank account. Several accounts were involved, and the interest rate and method of computation were unknown, causing several recomputations on each account. Thus, it was not convenient to use a pocket calculator, so a small Pascal program was developed.

The characteristics of these programs are:

- They are short, typically less than 200 lines long.
- Most have been used only once (i.e. one session).
- A relatively long time has passed between the writing of each program (a few weeks to several months).
- No standard application was available to solve the problem.

Other people, whether computer professionals or not, experience the same needs. Recently, we received some promotional material for computer

software describing a similar example: A crop-dusting pilot had spent two weeks hacking together a system that provided him with the fairly rudimentary computations he needed.

Writing even small programs with traditional tools can be quite time-consuming for casual programmers, and writing a dozen 100-line programs a year is certainly not enough to get in good programming shape. Thus, it would not have been worth the effort of writing the above programs at all, had it not been for the fact that the author because of his profession and interests had the programming language and operating system at his fingertips, or that the pilot needed his program so badly. Using an off-the-shelf package would either have been too complicated (for someone unfamiliar with the package), too expensive (if it had to be purchased), or not possible at all. It nevertheless remains that for a small group of computer-proficient people, the microcomputer represents an extremely handy tool for solving relatively simple problems that are still too hard for pen, paper, and pocket calculator, and that cannot be covered by an available applications package.

We would like this power to be available to a much wider class of users. The term *casual programming* refers to the creation of small applications by programmers who do not count computer programming as one of their main activities or interests.

6.1.2. Properties of a casual programming tool

To facilitate casual programming, new programming environments will have to be built. These environments must be *simple*, provide *concrete metaphors*, and generally be much more *supportive* than current environments intended for professionals or students of programming. The systems should be more *active*, since the programmer cannot be assumed to be completely in command of the situation at all times. It is reasonable to make a system present much information without being asked to do so. For example, the system may start explaining things if it is clear that the user has a problem. Since the users cannot be expected to remember details about a system between each time they use it, systems will have to be self-documentary and present their options via menus. Learning the various aspects of systems should be accomplished through the use and *exploration* of it, rather than by consulting separate documentation. We thus borrow some of the features of computer games, the kind of software that best has succeeded in capturing the attention of the user.

A further important feature of a useful tool for casual programming is *standardization*. No matter how simple and supportive a system is, there will still be a core of knowledge about computation that any programmer will have to acquire. Our task is to design a tool that makes it as convenient as

possible to obtain and maintain this knowledge. A standard emerged from a long process of evolution ensures the quality of the interface and minimizes the effort needed to move between systems.

A useful analog is the automobile, for which a generic set of physical controls and a new vocabulary has been developed. It took a few decades to settle on a convenient standard interface for automobiles. This interface, although minor variations exist between different cars, is sufficiently standardized so that once one masters the fairly simple skill of driving, one can handle any car after a brief exploration of its controls. Given that the abstract computational model we wish to present to people as a computer is a much more complex device, it will probably take longer for a similar standard to emerge here. The system we have developed is but one step in the long sequence of ideas, trials, failures, consolidations and new ideas that must be followed.

It is interesting to note that our philosophy for building casual programming environments is quite the opposite of the school promoted in, e.g., [Dijkstra 80], which advocates "a crusade against anthropomorphic terminology" (p 105), and claims that "to get rid of our operational models [is] computer science's major task" (p 103). On the contrary, we rejoice with [Krueger 83] in realizing that "with responsive electronics, all our anthropomorphisms

can become real" (p 222). On the other hand, the specific system we have developed does replace the traditional operational model with an algebraic one, and we have been careful not to carry real world pictures farther than to data illustrations, providing only abstract pictures for the algorithmic framework.

6.1.3. PiP is for casual programming

Programming in pictures and casual programming are certainly independent ideas, but they turn out to have a large and interesting intersection.

We have talked earlier about how pictures convey concrete metaphors, and provide a good medium, e.g., for animation and aesthetics, and for creating displays with a rich and well-structured information content. Pictures therefore seem an obvious candidate as a means for achieving the goals of a system for casual programming. The system for programming in pictures we have developed was designed with casual use in mind, and exhibits the simplicity and support a casual programmer needs.

On the other hand, pictures do not seem to be able to contribute a whole lot towards expressing systems programs. A pictorial debugging tool can certainly be of immense value, but as a way to compose the programs themselves the utility seems at the moment to be limited: First, since systems

programs concern abstract data with no "real life" connection, the argument about embedding real life semantics in programs as pictures is no longer convincing. The user will want to restrict the display to machine-oriented drawings, like boxes and arrows. Second, limiting the definition of a program piece to what can fit on a screen is certainly good software methodology, but if each piece cannot do very much it can be an impractical constraint. Unfortunately, most graphical program representations seen up till now are less economical with respect to screen space than text. Third, professional programmers are capable of thinking in abstract terms and will therefore not want to spend time drawing pictures they really do not need. Finally, several empirical investigations (cf. ch. 2) confirm that the less novice a programmer is, the less influenced he/she is by the user interface. This is not to say that pictures are useless for systems programming, but it seems hard to obtain the great benefit we saw for casual programming.

6.2. Correctness issues

All programmers make mistakes, so the degree to which a programming system can minimize the number of errors in a program after a certain amount of effort has gone into it is an important issue. In practical programming, a system can contribute to program correctness by preventing some class of errors from occurring and aiding in the discovery and correction of others. The system at hand makes several contributions in this respect.

6.2.1. Syntactical errors

With the usual text editor approach to program composition, quite a large number of syntactical errors are usually committed during the first formulation of a program. A few passes through the compiler is usually adequate to remove these easily, mostly thanks to good use of a typed symbol table.[1] Syntax-directed editors improve the situation by preventing the errors from being committed in the first place. This is achieved by selecting program structure templates from a menu instead of typing them, or at least having the code syntax-checked on the spot. If the editor also maintains a symbol table, the checking can include user-defined names as well. Our system takes this trend to its conclusion. Since every item used during programming is picked from a menu and inserted in a given framework, there is really no syntax in the usual sense. The program is simply built from existing components, and the question as to whether it conforms to some format specification simply does not arise. This means that not only is the user prevented from making syntactical errors, he/she does not even have to think about the concept.

---

[1]There are still systems (FORTRAN, C) where a typographical error may not show up until the linker finds an undefined symbol!

6.2.2. Semantic errors

At the next level of errors, we encounter the typing issue. Type checking can discover many simple slips of the mind that would otherwise be hard to find. Again, the common approach is to let the compiler, as an activity separate from editing, take care of this. In our system, type checking is an integral part of editing, making it impossible to connect pieces that are not compatible. This of course speeds up error correction, but the fact that the system can give instant responses to questions of whether two components are compatible should also encourage a style of interaction where the machine plays a more active role in supporting the user. This allows the programmer to spend more time thinking about higher-level issues that the system cannot check.

Part of programming in pictures is the idea of displaying all (and only) relevant objects. For example, in a scoped environment, the programmer would see only those data that are accessible from the current code, and could therefore not make errors regarding data access. This is not very visible in our system, since the scoping of FP is trivial.

6.2.3. Logic errors

At the higher semantic level, or program logic level, only the programmer is in a position to determine whether a given program is correct. But there is still a lot a programming environment can do to make this task easier. In traditional environments, the editor displays only a minor part of program attributes, so a debugger is usually provided to shed more light on the program. In our system, we systematically display *all* important aspects of programs, and by unifying these displays, we also show how they are related. Merely showing all this information without any effort on the programmer's part significantly contributes to the task of locating errors. It should also increase the probability that errors are discovered without extensive testing, since the progress of program execution is usually displayed whether it has been asked for or not.

Debuggers mostly display current data values. Indeed, we argued in chapter 3 that data provide the concrete basis around which the program revolves. Our system embodies this view and is therefore well equipped to provide the proper support by showing how data change through intermediate values.

The simple algebra developed for FP [Williams 82] gives us several benefits. In addition to making it simple to generate expressions from the user's input and transforming these into efficient code, it contributes to program

reliability by providing the programmer with a simple, clear, and well-defined model that does not contain any of the fine points and surprises that so often cause mysterious errors when using ordinary programming languages. It could also form the basis for future extensions to incorporate machine-aided program verification.

6.3. Software engineering issues

We have in this work explained how a system for programming in pictures can help the programmer in thinking about programs, but without explicitly relating this to the various phases of program development. The system we have designed can aid the user at several levels of program perception, and throughout the development cycle. We will here make some further comments about this.

6.3.1. Design

The design stage cannot be influenced directly by a system that only addresses implementation, but the flavor of the tools that are to be used later can certainly still have an impact on the design work. For our system, the influence on design comes mainly at two levels.

First, the FP computational model encourages systematic decomposition into small components, and promotes the view of a program consisting of data

that flow between computational nodes, rather than modules that pass control among each other and nibble on their own data. Our user interface reinforces this view by presenting it in terms of concrete pictures.

Second, the use in the programming environment of iconic pictures that stand for complex structures and convey high-level semantics can be transferred to the design environment and yield many of the same benefits we found for programming. Usually, program components and key mechanisms are identified at the design stage. Now, key pictures and picture elements that will later appear in the program can be identified and established at this stage too.

In addition, the existence of a typing mechanism allows the interfaces between components to be well defined in a systematic manner supported by the system. Our implementation promotes the definition of types as separate entities, thereby stressing their role as the glue between functions. Thus, the designer sees the system not only as a collection of computational nodes, but as a graph where the arcs also carry substantial significance and have to be defined carefully.

6.3.2. Prototyping

To facilitate prototyping, a system has to provide mechanisms that make it easy to write simple versions of programs and then to modify and enrich these. Our system aids the programmer in several ways.

The structure and semantics constraints built into the editor allows a fairly relaxed editing style, since the editor will take care of ensuring the well-formedness of the program. Thus, programs can easily be sketched, and in such a way that even rudimentary sketches can be executed and their overall functionality checked.

The functional model is extremely simple, and encourages small, simple components that are used by many parts of the total program. Modifying and enhancing a program is thereby made attractive. One can expect a development style akin to the productive "structured growth" paradigm used in the LISP community [Sandewall 78].

We mentioned earlier that our system embodies the view of a program as a hierarchical composition of abstractions by unifying the access path and the abstraction path. Thus, the system helps preserve the design decisions by ruling out accesses across system components that are incompatible with the original design. In traditional systems, these kinds of accesses have a tendency to obscure the design as the program evolves.

6.3.3. Testing

We have already mentioned how program animation should be an integral part of a programming system, both for program creation and inspection. Our system significantly aids in the testing process by making program execution available at the fingertips of the user, and by displaying as much and as many aspects of the program as possible. The system is further emphasizing the display of data, the key to an informative debugging aid.

6.3.4. Maintenance

As for prototyping, a system that encourages small components and makes it easy to modify them can simplify maintenance as well.

The problem of quickly obtaining an understanding of how a program works is particularly apparent during maintenance. The person assigned to make a modification is frequently not the original author, and even if he/she is, the details of the program may be long forgotten. Our system can help here on several fronts:

- Program details are displayed as a dataflow graph superimposed on pictures of data. This representation shows program function in a much more direct fashion than conventional text, and should therefore be faster to parse and understand. The availability of execution further contributes to a more complete description that leaves far less to be deduced by the reader than is presently required.

- Higher-level program structure is shown as structured diagrams that convey information about program composition much more clearly and concretely than regular syntax. The diagrams are simple and emphasize only the overall structure of the function, suppressing irrelevant detail.

- Pictorial icons are used throughout the system. It is simple for the programmer to attach semantic, explanatory information to all program components, information that helps the reader to understand what the components stand for. Once understood, our ability to recognize pictures and attach meaning to them helps in navigating through the program and relating the various components.

6.4. User reactions

This thesis does not include an empirical investigation of how users have responded to the system developed. To achieve such results that are at all reliable in a scientific sense, a major experimental effort will have to be launched, and there was not room for such an undertaking within the framework of this thesis. We have, though, collected several reactions from people who have stopped by our office and taken a look at the system, and we have of course reflected on the merits of our design ourselves:

- The casual style of programming the system allows was quite positively

received. In particular, the syntax-free specification of programs by simple pointing actions, and the lack of constraints on the sequence of these specifications, seem to have created some of the handwaving style we had in mind.

- Users reacted positively to the object-level display of a function as a flowgraph superimposed on pictures of data. This form of display combines a good representation of simple data flow, the flowgraph, with a good way to illustrate data, into a picture that shows all major aspects of a program without becoming confusing.

- Our systematic assignment of picture dimensions to program aspects also seems to be a useful idea. In particular, the zooming mechanism that reveals underlying structure was well received and experienced as a very natural and powerful feature.

- Creating pictures with our picture editor proved to demand much patience and a certain amount of artistic skills. A better graphics editor would probably have improved the situation somewhat, but we still have a feeling that most people would quickly get tired of drawing their own images to the last bit. In a practical system, it would therefore be

necessary to have access to a picture library. This could be organized as a collection of large "sheets" that could be scanned and pictures and picture elements picked up by the editor. It should also be possible to contribute pictures to the sheets, and a community of programmers should be able to share the same sheets. After some time of usage, every programmer would have access to a rich library of good illustrations and only occasionally have to resort to original work.

- The FP computational model has been a mixed blessing. For programs with simple data structures, like the ones presented in this thesis, its simple semantics is certainly convenient. For larger programs, with complex structures of heterogeneous data, it is more difficult to use. The programmer ends up spending more time writing auxiliary functions that extract the interesting data and put the results back where they belong, than the actual computation. There is a small chance that this is a result of the users still thinking in von Neumann terms, but some of the examples we have attempted seem very hard to simplify. The FP model was chosen for its many benefits for graphical display. It might very well be that we would have ended up with another model (e.g. an object-oriented one), had we looked at the way people think (concrete objects) instead.

- At present, input data values have to be given by the user every time a program is executed. For large data sets or repetitive execution on the same data, this is of course unacceptable. Our design had initially a fifth tool, a data editor, for the creation of data objects that could be used during execution. The inclusion of such a tool would certainly be useful, for example providing a large scratchpad where data objects can reside, represented by icons. Underlying the scratchpad, however, we would like to have a database organization to store large amounts of user data. In the next section we will comment more on merging current trends in personal database systems with a casual programming system.

6.5. Further work

We started this chapter by defining our long-term goal of a tool for casual programming. At the same time, it became clear that developing such a tool can only be the result of a long evolutionary process, with contributions from many fields and researchers. We hope that we have provided a useful piece of the mosaic. As we conclude this thesis, there are a few issues that relate directly to what we have done, and that we see as natural extensions of our work.

From a programming language point of view, a major shortcoming of our

system for programming in pictures is its inability to handle data abstractions graphically in a general and powerful way. We commented on this in section 3.1.1 where we conjectured that this difficult problem may have been one reason for the lack of work on data-oriented displays. As we pointed out, the core of the problem is that the mapping between the implementation and the specification of an abstraction is usually only implicitly given as a simple data type definition along with a collection of operators. This means that the graphical interpretation of the abstraction must be programmed explicitly. Furthermore, the inverse function, the mapping of user pointing actions to the underlying structure and the operations on it, must be defined. It seems far from trivial to design a framework for this that will not inflict substantially more work on the programmer than is necessary to program abstractions in conventional languages.

From a user interface point of view, it is interesting to observe the work being done to facilitate end-user programming within the office environment (cf. section 2.4). These efforts grow out of the database community, and attempt to make it simple to query the database by providing graphical interfaces and simple conceptual models. Users are also given a creative role by being able to change the structure of the database (manipulate metadata). As these systems get to contain more functionality and computational power, they

approach full-fledged programming systems for end users. Casual programming and end-user database programming seem headed for the same goal, and it would be useful to attempt a merger of the two views.

Finally, looking at our research methodology, we have to apply self-criticism. We remarked in the introduction how human-computer interfaces now seem to be catching the interest of mainstream computer science. But if our focus of attention changes, so are our methods likely to be ready for revision as well. When we break out of the protective shell of language theory and design and start reasoning about human reactions in general, we have to look to other fields that deal with similar questions, like psychology and sociology, and learn from their methods. These mainly consist of statistically sound experiments. The research in this thesis includes no empirical experiment that supports its claims about the usefulness of pictures in programs. Of course, we have with the help of our colleagues weeded out many unhappy design decisions, but it is most unfortunate that our resources precluded a thorough statistical investigation. It must be clear that for the part of computer science that addresses human concerns, statistical experiments must accompany the claims before they can be assumed valid.

6.6. Conclusions

What have we learnt from our experience with programming in pictures? With our implementation in mind, how do our thoughts and ideas in the early chapters of the thesis fare?

One fact that just has become more clear is the importance of choosing a good computational model. The concepts of the model shine through the graphical interface and influence the way the user thinks about programming. We have seen how the simplicity of FP has made the programming of small programs neat, and how its lack of some powerful features can make some programming awkward.

But if the model determines the way programmers reason, what has become just as clear is how pictures can help this thought process go smoother. The power of pictures to provide concrete handles and to show all the details that are usually hidden was well demonstrated in our system. Combined with an interaction style that supports the user by suggesting a menu of options, we created a system that we believe significantly reduces the amount of abstract knowledge and thinking required by the programmer.

This leads us to another major observation. The advantages of a pictorial programming system seems considerable for casual users, but less important

for professional programmers. We have argued that there is a need for a tool for casual programming, and have made a system that exhibits some of the properties of such a tool. We think casual programming is a very interesting application area for graphical interaction.

We stressed the importance of data structure display and showed that it is possible to design a system where the data structure is the centerpiece around which the algorithm revolves. We further showed that animation can be an integrated part of program representation based on these displays.

Our philosophy that a program must be displayed as *one* object instead of a series of multiple views lead us to a careful exploitation of picture dimensions. This was very useful, in particular zooming seems to be a powerful technique worth exploring.

We feel the approach of our work has been very successful. Instead of working from within the established programming domain and adapting to a new medium, we looked first at how people work with pictures and made a connection to programming, being only as formal as absolutely necessary. This has given us a system that is very different from other attempts in the same direction. We feel our solution responds better to the users' needs.

Appendix A
Summary of the FP model

The FP programming model is fully described in [Backus 78]. We here give only a brief summary of the model.

An FP system consists of:

1. A set O of *objects*. O contains a set of ***atoms***, and all finite ***sequences*** $<x_1,x_2,...,x_n>$ of objects x_i, including the empty sequence Φ. For example, if the atoms include the integers, $<3,<1,4,1>>$ is a sequence of length 2. The atoms include T and F for predicate evaluation, and a special atom $\perp$ (read "bottom") indicating an undefined value. Φ is also among the atoms, it is thus the only object that is both a sequence and an atom. Sequences are "bottom-preserving", i.e. $<x_1,x_2,...,x_n> = \perp$ if $x_i = \perp$ for some $1 \leq i \leq n$.

2. A set F of *functions* that map objects into objects. A function is either ***primitive***, i.e. built-in, ***defined***, or it is a ***functional form*** (see below). All functions in F are bottom-preserving, i.e. $f{:}\perp = \perp$ for all $f \in F$.

3. An operation, ***application***. $f{:}x$ is the result (an object) of applying $f \in F$ to $x \in O$.

4. A set $\mathbf{F}$ of *functional forms* that are combinations of functions in F. For example, if $f,g \in F$, then $f \circ g$ is a functional form, the ***composition*** of f and g.

5. A set D of ***definitions*** that define and name functions in F, of the form ***Def*** $l \equiv r$, indicating that the symbol l is to denote the function given by r.

We have taken the set of atoms to be all integer and real numbers, character strings, T, F, and a set of fixed-size bitmaps interpreted as pictures. Some useful primitive functions are:

Selection: $i{:}x \equiv$ if $x = <x_1, \ldots, x_n>$ and $n \geq i$ then x_i else $\perp$.

Tail: $tl{:}x \equiv$ if $x = <x_1>$ then Φ, else if $x = <x_1, \ldots, x_n>$ and $n \geq 2$ then $<x_2, \ldots, x_n>$, else $\perp$.

Identity: $id{:}x \equiv x$.

Equals: $eq{:}x \equiv$ if $x{=}{<}y,z{>}$ and $y{=}z$ then T,

else if $x{=}{<}y,z{>}$ and $y{\neq}z$ then F, else $\perp$.

Null: $null{:}x \equiv$ if $x{=}\Phi$ then T,

else if $x{\neq}\perp$ then F, else $\perp$.

Arithmetic: $+{:}x \equiv$ if $x{=}{<}y,z{>}$ and y,z *are numbers*

then $y{+}z$, else $\perp$.

etc.

Distribute left: $distl{:}x \equiv$ if $x{=}{<}y,\Phi{>}$ then Φ,

else if $x{=}{<}y,{<}z_1,\ldots,z_n{>}{>}$ then ${<}{<}y,z_1{>},\ldots,{<}y,z_n{>}{>}$,

else $\perp$.

Append left: $apndl{:}x \equiv$ if $x{=}{<}y,\Phi{>}$ then ${<}y{>}$,

else if $x{=}{<}y,{<}z_1,\ldots,z_n{>}{>}$ then ${<}y,z_1,\ldots,z_n{>}$,

else $\perp$.

Here are some functional forms:

Composition: $f \circ g{:}x \equiv f{:}(g{:}x)$.

Construction: $[f_1,...,f_n]{:}x \equiv \langle f_1{:}x,...,f_n{:}x \rangle$.

Condition: $(p \rightarrow f;g) \equiv$ if $p{:}x=T$ then $f{:}x$,

else if $p{:}x=F$ then $g{:}x$, else $\bot$.

Loop: $(while\ p\ f){:}x \equiv$ if $p{:}x=T$ then $(while\ p\ f){:}(f{:}x)$,

else if $p{:}x=F$ then x, else $\bot$.

Apply to all: $\alpha f{:}x \equiv$ if $x=\Phi$ then Φ,

else if $x=\langle x_1,...,x_n \rangle$ then $\langle f{:}x_1,...,f{:}x_n \rangle$,

else $\bot$.

Insert: $/f{:}x \equiv$ if $x=\langle x_1 \rangle$ then x_1,

else if $x=\langle x_1,...,x_n \rangle$ and $n \geq 2$

then $f{:}\langle x_1, /f{:}\langle x_2,...,x_n \rangle\rangle$,

else $\bot$.

Constant: $\underline{x}{:}y \equiv$ if $y=\bot$ then $\bot$, else x,

where x is an object parameter.

Binary to unary: $(bu\ f\ x):y \equiv f:<x,y>$,

where x is an object parameter.

Here is a simple FP program that computes the factorial:

$$\mathbf{Def}\ eq0 \equiv eq \circ [id, \underline{0}]$$

$$\mathbf{Def}\ sub1 \equiv - \circ [id, \underline{1}]$$

$$\mathbf{Def}\ ! \equiv eq0 \rightarrow \underline{1}; * \circ [id, ! \circ sub1]$$

Appendix B
Code generation

When the programmer points at pictures of the data structures, the system will generate the corresponding FP expressions. This is done in the following way:

Object-level functions

In the object-level function editor, the function template contains a picture of the input data and a picture of the output data. The programmer performs three types of pointing actions:

1. **Input data.** When the user points at a piece of input data, an expression is generated to *select* that piece of data. Assume, for example, that the input data type has four elements. The system enumerates them 1-4, in some arbitrary way. A data object will have the structure $<x_1,x_2,x_3,x_4>$. Pointing at the second element will generate the expression *2* (in Backus' notation, the operator extracting the second element). The expression generated is pushed onto a stack for later use. If the element is zoomed, a sequence of selection operators

are *composed*. For example, if the second element is zoomed, and the third element of the zoomed picture is pointed at, the expression *3*o*2* is generated.

2. **Operators.** When the programmer selects an operator, either from one of the built-in menus or from the function library, an expression representing the operator is generated and combined with one or more of the expressions on the stack. The arity of the function determines how many of the elements on the stack should be included. The expressions from the stack are combined via a ***construction*** operator to make a sequence. The resulting expression is pushed back onto the stack. For example, if the second and third input elements were first pointed at, giving the two corresponding selection expressions on the stack, selecting the addition operator would create the expression *+*o*[3,2]*. All user-defined functions have an arity of one. Data types should be used to build composite arguments.

3. **Output data.** The pointing actions successively build an FP expression that describes how the output is computed. When an element of the output type is selected, an expression is removed from the top of the stack and inserted as the generating expression for the

element in question. A *construction* operator is used to combine the expressions for the different elements. For example, if the output data type has three elements, an initial (empty) expression *[, ,]* is set up. Next, if the second output element is pointed at, and the stack contains the expression E, the expression *[,E,]* will result. If the output type is zoomed, extra levels of construction operators will be inserted. For example, if the second element is zoomed and the first element of the zoomed picture is selected, *[,[E,],]* will be generated (assuming that the type of the zoomed element has two elements). The brackets of the constructions correspond directly to the brackets of the FP sequences output by the function being built.

Programming with structured data involves the same mechanisms. Much like zooming, the elaboration of input or output data structure gives rise to compositions of selection operations and nested constructions. Function variants use the ***null*** predicate to determine the structure of incoming data.

It is clear that code generation according to the above scheme cannot always result in the most efficient code. For example, an expression of the type *[1∘1, 2∘1]* is usually less efficient than *[1,2]∘1*. A function that swaps two elements will generate *[2,1]* instead of using the more efficient ***reverse*** operator. For this reason, a separate optimization pass should be applied to

the final expression generated. FP lends itself quite well to such optimization, and several algebraic laws have been discovered that can be utilized [Islam 81, Kieburtz 81, Williams 82].

Function-level functions

The function-level code generation is trivial, since the programmer directly specifies the functional forms to be created.

Appendix C
Implementation status

The prototype implementation of our Programming in Pictures system has been under development for the past 2 years, as an activity parallel to the thesis work. The system runs on a Sun 100 Workstation and consists at present (November, 1984) of about 10,000 lines of C code, building on the SunCore graphics library provided by Sun Microsystems. A system diagram was given in chapter 2.

The system does not provide all the features described in this thesis. Roughly, we have implemented chapter 4, but not chapter 5. A few things are still missing from what is described in chapter 4: The help facility and type checking are not implemented, and animation is still lacking single-stepping and execution breakpoints. Parameter prompting is done by displaying a window with explanatory text rather than by the region highlighting suggested.

References

[Amble 83] Amble, T.
Functional Analysis and Design Technique.
Technical Report Working Document 142650.01-3, RUNIT (Computing Centre at the University of Trondheim, Norway), Aug., 1983.

[Backus 78] Backus, J.W.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
Comm. ACM 21(8), Aug., 1978.

[Baer 80] Baer, J.-L.
Computer Systems Architecture.
Computer Science Press, 1980.

[Bannon 83] Bannon, L., A. Cypher, S. Greenspan, and M.L. Monty.
Evaluation and Analysis of Users' Activity Organization.
In *Proc. CHI '83, Human Factors in Computing Systems,* pages 54-57. ACM, Dec., 1983.

[Bauer 79] Bauer, M.A.
Programming by Examples.
Artificial Intelligence 12(1):1-21, May, 1979.

[Berry 85] Berry, O. and D. Jefferson.
Critical Path Analysis of Distributed Simulation.
In *Distributed Simulation 1985, SCS Multiconference.* Soc. for Computer Simulation, San Diego, Calif., Jan., 1985.
(To appear).

[Bewley 83] Bewley, W.L., T.L. Roberts, D. Schroit, and W.L.Verplank.
Human Factors Testing in the Design of Xerox's 8010 'Star' Office Workstation.
In *Proc. CHI '83, Human Factors in Computing Systems,* pages 72-77. ACM, Dec., 1983.

[Biermann 76] Biermann, A.W. and R. Krishnaswamy.
Constructing Programs from Example Computations.
IEEE Trans. on Software Eng. SE-2(3), Sep., 1976.

[Birtwistle 84] Birtwistle, G., B. Wywill, D. Levinson, and R. Neal.
Visualising a Simulation Using Animated Pictures.
In *Proc. Conf. on Simulation in Strongly Typed Languages*, pages 57-61. Soc. for Computer Simulation, La Jolla, Calif., Feb., 1984.
Simulation Series, Vol. 13, No. 2.

[Bo 82] Bø, K.
Guest Editor's Introduction: Human-Computer Interaction.
Computer 15(11), Nov., 1982.

[Brown 83] Brown, E.J.
On the Application of Rothon Diagrams to Data Abstraction.
SIGPLAN Notices 18(12), Dec., 1983.

[Brown 84] Brown, M.H. and R. Sedgewick.
A System for Algorithm Animation.
In *Proc. SIGGRAPH '84*, pages 177-186. ACM, 1984.
(Computer Graphics, 18, 3, July 1984).

[Butler 83] Butler, T.W.
Computer Response Time and User Performance.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 58-62. ACM, Dec., 1983.

[Card 80] Card, S.K., T.P. Moran, and A. Newell.
The Keystroke-Level Model for User Performance Time with Interactive Systems.
Comm. ACM 23(7):396-410, July, 1980.

[Cardelli 83] Cardelli, L.
Two-Dimensional Syntax for Functional Languages.
In P. Degano and E. Sandewall (editors), *Integrated Interactive Computing Systems*, pages 139-151. North-Holland, 1983.

[Carey 82] Carey, T.
User Differences in Interface Design.
Computer 15(11), Nov., 1982.

[Carroll 82] Carroll, J.M. and J.C. Thomas.
Metaphor and the Cognitive Representation of Computing Systems.
IEEE Trans. on Systems, Man and Cybernetics SMC-12(2):107-116, Mar./Apr., 1982.

[Davis 82] Davis, A.L. and R.M. Keller.
Data Flow Program Graphs.
Computer 15(2), Feb., 1982.

[deBalbine 78] de Balbine, G.
MTR - A Tool for Displaying the Global Structure of Software Systems.
In *AFIPS Conf. Proc. 47*, pages 571-580. 1978.

[DeRemer 76] DeRemer, F. and H.H. Kron.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Trans. Software Engineering SE-2(2):80-86, June, 1976.

[Dewar 84] Dewar, A. and B. Unger.
Graphical Tracing and Debugging of Simulations.
In *Proc. Conf. on Simulation in Strongly Typed Languages*, pages 68-73. Soc. for Computer Simulation, La Jolla, Calif., Feb., 1984.
Simulation Series, Vol. 13, No. 2.

[Dijkstra 80] Dijkstra, E.W.
My Hopes of Computing Science.
Software Engineering.
Academic Press, 1980, pages 95-109.

[Dionne 78] Dionne, M.S. and A.K. Mackworth.
ANTICS: A System for Animating LISP Programs.
Computer Graphics and Image Processing 7:105-119, 1978.

[Ellis 80] Ellis, C.A. and G.J. Nutt.
Office Information Systems and Computer Science.
ACM Computing Surveys 12(1), Mar., 1980.

[Frank 81] Frank, G.A.
Specification of Data Structures for FP Programs.
In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, pages 221-228. ACM, 1981.

[Frei 78] Frei, Weller and Williams.
A Graphics-Based Programming-Support System.
In *Proc. SIGGRAPH '78*, pages 43-49. ACM, 1978.
(Computer Graphics, 12, 3, Aug. 1978).

[Frome 83] Frome, F.S.
Incorporating the Human Factor in Color CAD Systems.
In *Proc. ACM IEEE 20th Design Automation Conf.*, pages 189-195. June, 1983.

[Goldberg 83] Goldberg, A. and D. Robson.
Smalltalk-80, the Language and Its Implementation.
Addison-Wesley, 1983.

[Gomez 83] Gomez, L.M., D.E. Egan, E.A. Wheeler, D.K. Sharma, and A.M. Gruchacz.
How Interface Design Determines Who Has Difficulty Learning to Use a Text Editor.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 176-181. ACM, Dec., 1983.

[Gould 83] Gould, J.D. and C. Lewis.
Designing for Usability - Key Principles and What Designers Think.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 50-53. ACM, Dec., 1983.

[Guttag 81] Guttag, J., J. Horning and J. Williams.
FP with Data Abstraction and Strong Typing.
In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, pages 11-24. ACM, 1981.

[Halbert 81] Halbert, D.C.
An Example of Programming by Example.
Master's thesis, Univ. of Calif., Berkeley, June, 1981.

[Hebalkar 79] Hebalkar, P.G. and S.N. Zilles.
TELL: A System for Graphically Representing Software Designs.
In *Proc. Spring COMPCON '79*, pages 244-249. IEEE Computer Society, 1979.

[Herot 80a] Herot, C.F., R. Carling, M. Friedell, and D. Kramlich.
A Prototype Spatial Data Management System.
In *Proc. SIGGRAPH '80*, pages 63-70. ACM, 1980.
(Computer Graphics, 14, July, 1980).

[Herot 80b] Herot, C.F.
Spatial Management of Data.
ACM Trans. on Database Systems 5(4):493-513, Dec., 1980.

[Howden 82] Howden, W.E.
Contemporary Software Development Environments.
Comm. ACM 25(5):318, May, 1982.

[Islam 81] Islam, N., T.J. Myers, and P. Broome.
A Simple Optimizer for FP-Like Languages.
In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, pages 33-39. ACM, 1981.

[Jacob 83] Jacob, R.J.K.
Executable Specifications for a Human-Computer Interface.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 28-34. ACM, Dec., 1983.

[Jones 78] Jones, P.F.
Four Principles of Man-Computer Dialogue.
Computer-Aided Design 10:197-202, May, 1978.

[Karhan 83] Karhan, C.J., C.A. Riley and M.S. Schoeffler.
Designing and Evaluating Standard Instructions for Public Telephones.
The Bell System Techn. J. 62(6):1827-1847, July-Aug., 1983.

[Karlsson 82] Karlsson, K. and K. Petersson (eds.).
Notes from: The Aspenas Symposium on Functional Languages and Computer Architecture.
SIGPLAN Notices 17(11), Nov., 1982.

[Keller 81] Keller, R.M. and W.-C.J. Yen.
A Graphical Approach to Software Development Using Function Graphs.
In *Proc. Spring COMPCON '81*, pages 156-161. IEEE, 1981.

[Kelley 83] Kelley, J.F.
An Empirical Methodology for Writing User-Friendly Natural Language Computer Applications.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 193-196. ACM, Dec., 1983.

[Kieburtz 81] Kieburtz, R.B. and J. Shultis.
Transformations of FP Program Schemes.
In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, pages 41-48. ACM, 1981.

[Kieras 83] Kieras, D. and P.G. Polson.
A Generalized Transition Network Representation for Interactive Systems.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 103-106. ACM, Dec., 1983.

[Kramlich 83] Kramlich, D., G.P. Brown, R.T. Carling, and C.F. Herot.
Program Visualization: Graphics Support for Software Development.
In *Proc. ACM IEEE 20th Design Automation Conf.*, pages 143-149. June, 1983.

[Krueger 83] Krueger, M.
Artificial Reality.
Addison-Wesley, 1983.

[Langlois 84] Langlois, L.
Simulation Visualization with SIMSEA, a General Purpose Animation Language.
In *Proc. Conf. on Simulation in Strongly Typed Languages*, pages 62-67. Soc. for Computer Simulation, La Jolla, Calif., Feb., 1984.
Simulation Series, Vol. 13, No. 2.

[Lawson 78] Lawson Jr., H.W., M. Bertran, and J. Sanagustin.
The Formal Definition of Human/Machine Communication.
Software - Practice and Experience 8:52-58, Jan.-Feb., 1978.

[Lippman 81] Lippman, A.
And Seeing Through Your Hand.
Proc. of the Society for Information Display 22(2):103-107, 1981.

[MacDonald 82] MacDonald, A.
Visual Programming.
Datamation 28(11):132-140, Oct., 1982.

[Magnenat-Thalmann 84]
Magnenat-Thalmann, N. and D. Thalmann.
Animated Types and Actor Types in Computer Simulation and Animation.
In *Proc. Conf. on Simulation in Strongly Typed Languages*, pages 51-56. Soc. for Computer Simulation, La Jolla, Calif., Feb., 1984.
Simulation Series, Vol. 13, No. 2.

[Maling 81] Maling, K. and K.A. Duke.
ALEX: A Conversational, Hierarchical Logic Design System.
J. of Digital Systems 5(4):361-382, 1981.

[Malone 81] Malone, T.W.
What Makes Computer Games Fun?
Byte Magazine 6(12):258-277, Dec., 1981.

[Mayer 81] Mayer, R.E.
The Psychology of How Novices Learn Computer Programming.
ACM Comput. Surveys 13(1):121-141, Mar., 1981.

[McNurlin 81a] McNurlin, B.C.
'Programming' by End Users.
EDP Analyzer 19(5), May, 1981.

[McNurlin 81b] McNurlin, B.C.
Supporting End User Programming.
EDP Analyzer 19(6), June, 1981.

[McNurlin 82] McNurlin, B.C.
Query Systems for End Users.
EDP Analyzer 20(9), Sep., 1982.

[Medina-Mora 82]
Medina-Mora, R.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, 1982.
No. CMU-CS-82-113.

[Meyrowitz 82] Meyrowitz, N. and A. van Dam.
Interactive Editing Systems: Part II.
ACM Comput. Surveys 14(3):353-415, Sep., 1982.

[Miara 83] Miara, R.J., J.A. Musselman, J.A. Navarro, and B. Shneiderman.
Program Indentation and Comprehensibility.
Comm. ACM 26(11), Nov., 1983.

[Molzberger 83] Molzberger, P.
Aesthetics and Programming.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 247-250. ACM, Dec., 1983.

[Mooers 83] Mooers, C.D.
Changes That Users Demanded in the Human Interface to the Hermes Message System.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 88-92. ACM, Dec., 1983.

[Moran 81] Moran, T.P.
Guest Editor's Introduction: An Applied Psychology of the User.
ACM Comput. Surveys 13(1):1-11, Mar., 1981.

[Moriconi 84] Moriconi, M. and A.L. Lansky.
Representation and Refinement of Visual Specifications.
Technical Report CSL-143, SRI International, Mar., 1984.

[Morris 81] Morris, J.M. and M.D. Schwartz.
The Design of a Language-Directed Editor for Block-Structured Languages.
In *Proc. ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 28-33. ACM, 1981.
(SIGPLAN Notices, 16, 6, June 1981).

[Morse 79] Morse, A.
Some Principles for the Effective Display of Data.
In *Proc. SIGGRAPH '79.* ACM, 1979.
(Computer Graphics, 13, 2, Aug. 1979).

[Murrell 83] Murrell, S.
Computer Communication System Design Affects Group Decision Making.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 63-67. ACM, Dec., 1983.

[Nakatani 83] Nakatani, L.H. and J.A. Rohrlich.
Soft Machines: A Philosophy of User-Computer Interface Design.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 19-23. ACM, Dec., 1983.

[Negroponte 81] Negroponte, N.
Media Room.
Proc. Soc. for Information Display 22(2):109-113, 1981.

[Newman 80] Newman, W.
Towards the Integrated Interactive System.
In C.E. Vandoni (editor), *Proc. Eurographics-80.* North-Holland, 1980.

[Ng 79] Ng, N.
A Graphical Editor for Programming Using Structured Charts.
In *Proc. Spring COMPCON '79*, pages 238-243. IEEE Computer Society, 1979.

[Norman 83] Norman, D.A.
Design Principles for Human-Computer Interfaces.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 1-10. ACM, Dec., 1983.

[Plattner 81] Plattner, B. and J. Nievergelt.
Monitoring Program Execution: A Survey.
Computer 14(11), Nov., 1981.

[Powell 83] Powell, M.L. and M.A. Linton.
Visual Abstraction in an Interactive Programming Environment.
In *Proc. SIGPLAN '83 Symp. on Prog. Lang. Issues in Software Systems*, pages 14-21. ACM, 1983.

[Price 83] Price, L.A. and C.A. Cordova.
Use of Mouse Buttons.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 262-266. ACM, Dec., 1983.

[Reiss 84] Reiss, S.P.
Graphical Program Development with PECAN Program Development Systems.
In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* ACM, 1984.
(SIGPLAN Notices, 19, 5, May 1984).

[Renfors 83] Renfors, M., B. Sikstrom and L. Wanhammar.
An Integrated CAD System for VLSI Implementation of Digital Filters.
In *Proc. VLSI '83*, pages 423-432. North-Holland, 1983.

[Revett 83] Revett, M.C. and P.A. Ivey.
ASTRA: A CAD System to Support a Structured Approach to IC Design.
In *Proc. VLSI '83*, pages 413-422. North-Holland, 1983.

[Roach 83] Roach, J.W. and M. Nickson.
Formal Specifications for Modeling and Developing Human/Computer Interfaces.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 35-39. ACM, Dec., 1983.

[Sandewall 78] Sandewall, E.
Programming in an Interactive Environment: The LISP Experience.
ACM Comput. Surveys 10(1), Mar., 1978.

[Sheil 81] Sheil, B.A.
The Psychological Study of Programming.
ACM Comput. Surveys 13(1):101-120, Mar., 1981.

[Shneiderman 77]
Shneiderman, B., R. Mayer, D. McKay, and P. Heller.
Experimental Investigations of the Utility of Detailed Flowcharts in Programming.
Comm. ACM 20(6):373-381, June, 1977.

[Shrobe 83] Shrobe, H.E.
AI Meets CAD.
In *Proc. VLSI '83*, pages 387-399. North-Holland, 1983.

[Sime 77] Sime, M.E., T.R.G. Green, and D.J. Guest.
Scope Marking in Computer Conditionals - A Psychological Evaluation.
Int. J. Man-Mach. Stud. 9:107-118, 1977.

[Sondheimer 82] Sondheimer, N.K. and N. Relles.
Human Factors and User Assistance in Interactive Computing Systems: An Introduction.
IEEE Trans. Systems, Man and Cybernetics SMC-12(2):102-107, Mar.-Apr., 1982.

[Summers 77] Summers, P.D.
A Methodology for Lisp Program Construction from Examples.
J. ACM 24(1):161-175, Jan., 1977.

[Teitelbaum 81a] Teitelbaum, T. and T. Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Comm. ACM 24(9), Sep., 1981.

[Teitelbaum 81b] Teitelbaum, T., T. Reps, and S. Horwitz.
The Why and Wherefore of the Cornell Program Synthesizer.
In *Proc. ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 8-16. ACM, 1981.
(SIGPLAN Notices, 16, 6, June 1981).

[Teitelbaum 83] Teitelbaum, R.C. and R.E. Granda.
The Effects of Positional Constancy on Searching Menus for Information.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 150-153. ACM, Dec., 1983.

[Teitelman 77] Teitelman, W.
A Display Oriented Programmer's Assistant.
Technical Report CSL-77-3, Xerox PARC, Mar., 1977.

[Tesler 81] Tesler, L.
The Smalltalk Environment.
Byte Magazine , Aug., 1981.

[Tesler 83] Tesler, L.
Object-Oriented User Interfaces and Object-Oriented Languages.
In *Proc. ACM Conf. on Personal and Small Computers*, pages 3-5. 1983.
(SIGPC Notes, 6, 2, 1983).

[Topmiller 78] Topmiller, D.A. and N.M. Aume.
Computer-Graphic Design for Human Performance.
In *Proc. Annual Reliability and Maintainability Symp.*, pages 385-388. IEEE, 1978.

[Unger 84] Unger, B., G. Birtwistle, J. Cleary, D. Hill, G. Lomow, R. Neal, M. Peterson, I. Witten, and B. Wyvill.
Jade: A Simulation and Software Prototyping Environment.
In *Proc. Conf. on Simulation in Strongly Typed Languages*, pages 77-83. Soc. for Computer Simulation, La Jolla, Calif., Feb., 1984.
Simulation Series, Vol. 13, No. 2.

[Warman 81] Warman, E.A.
Man in a Machine Environment.
In Sata, T. and E. Warman (editors), *Man-Machine Communication in CAD/CAM*, pages 1-13. North-Holland, Amsterdam, 1981.

[Williams 82] Williams, J.
On the Development of the Algebra of Functional Programs.
ACM Trans. Progr. Lang. and Systems 4(4), Oct., 1982.

[Williams 83] Williams, G.
The Lisa Computer System.
Byte Magazine , Feb., 1983.

[Williams 84] Williams, G.
The Apple Macintosh Computer.
Byte Magazine , Feb., 1984.

[Williamson 84] Williamson, R.
SODOS - A Software Documentation Support Environment.
PhD thesis, Univ. Southern California, Oct., 1984.

[Wixon 83] Wixon, D., J. Whiteside, M. Good, and S. Jones.
Building a User-Defined Interface.
In *Proc. CHI '83, Human Factors in Computing Systems*, pages 24-27. ACM, Dec., 1983.

[Wulf 81] Wulf, W.A., M. Shaw, P.N. Hilfinger, and L. Flon.
Fundamental Structures of Computer Science.
Addison-Wesley, 1981.

[Yavelberg 82] Yavelberg, I.S.
Human Performance Engineering Considerations for Very Large Computer Based Systems: The End User.
The Bell System Techn. J. 61(5):765-797, May/June, 1982.

[Zloof 77] Zloof, M.M.
Query-by-Example: A Database Language.
IBM Sys. J. 16(4):324-343, 1977.

[Zloof 82] Zloof, M.M.
Office-by-Example: A Business Language that Unifies Data and Word Processing and Electronic Mail.
IBM Sys. J. 21(3):272-304, 1982.