

# Programming by Example

Daniel Conrad Halbert

Ph.D. in Computer Science

Department of Electrical Engineering and Computer Sciences  
Computer Science Division  
University of California, Berkeley

November, 1984

Sponsor: Xerox Corporation

Susan L. Graham, Committee Chair

*Abstract:* Programming by example is a way of programming a software system in its own user interface. The user of the system writes a program by giving an example of what the program should do. The system records the sequence of actions, and can perform it again. Programming by example allows a user to create programs without doing conventional programming.

Programming by example was incorporated into a simulation of an office information system. As the system evolved, it became clear that the basic concept of programming by example needed to be extended: certain aspects of program creation are better done by program modification than by using the programming-by-example mechanism. The final system includes a static program representation that is easy to understand, a mechanism for describing data, and a method of adding control structure.

A user operates on certain data while creating a program by example, but does not necessarily tell the system why that data was selected. The user can supply this missing information by using data descriptions, which are program operands that specify how to choose the data to be used when the program is run. The system automatically supplies reasonable default data descriptions while recording a program; if necessary the user can modify the descriptions later to reflect his or her true intentions.

Since programming by example is best at recording sequential user actions, rather than branching or iteration, control structure that alters program flow is specified by program editing. The operands in iterations and the predicates in conditional control structures are built from data descriptions.

Real users have learned to use the system quickly and with little difficulty. The techniques used in this system can be applied to other systems as well. This research has demonstrated that programming by example is a practical method for creating programs.

*This document is a slightly corrected version of a dissertation submitted to the Department of Electrical Engineering and Computer Sciences, Computer Science Division, University of California, Berkeley, in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science. A nearly identical version of this document was also published by the Office Systems Division, Systems Development Department, Xerox Corporation, as technical report OSD-T8402, December, 1984.*

*An online version of this document was recreated by the author in March, 1999. For more information, contact the author at [halbert@world.std.com](mailto:halbert@world.std.com) or [halbert@halwitz.org](mailto:halbert@halwitz.org).*

Copyright © 1984 by Daniel Conrad Halbert. All rights reserved.

# Table of Contents

Table of Contents .....	ii
List of figures .....	iv
Acknowledgments .....	v
Chapter 1 Introduction .....	1
1.1 Thesis .....	1
1.2 Motivation .....	1
1.3 Why is programming considered hard? .....	2
1.4 Programming by example .....	3
1.5 To reiterate — what programming by example is .....	4
1.6 Requirements for a programming-by-example system .....	4
1.7 Limitations of simple programming by example .....	5
1.8 Research history .....	6
Chapter 2 Work by others .....	7
2.1 Simple examples .....	7
2.2 Spreadsheet programs .....	7
2.3 Pygmalion .....	8
2.4 Programming by abstract demonstration .....	9
2.5 Tinker .....	9
2.6 Programming by Rehearsal .....	11
2.7 Exemplary programming .....	12
2.8 Juno .....	12
2.9 Dynamic program building .....	13
2.10 Query-by-Example .....	13
2.11 Programming by homomorphisms and descriptions .....	14
2.12 Inductive inference .....	15
Chapter 3 The example system .....	16
3.1 Selection of the system .....	16
3.2 The Xerox Star 8010 Information System .....	16
3.3 User programming in Star .....	20
3.4 SmallStar .....	20
3.5 Basic programming by example in SmallStar .....	22
3.6 Choice of recording mechanism .....	23
3.7 Internal form of the recorded program .....	24
3.8 Sample program .....	25
3.9 The SmallStar virtual machine .....	25
3.10 Evolution of SmallStar .....	26
Chapter 4 Data descriptions .....	29
4.1 Choosing data in programs .....	29
4.2 Finding out the user's intentions .....	30
4.3 Choose the data, or choose the container? .....	31
4.4 SmallStar data descriptions .....	31
4.5 Content and editing of data descriptions .....	34
4.6 References to data in early versions of SmallStar .....	39
4.7 An earlier concept: generalization .....	40
4.8 Defining generalization .....	43
4.9 Generalization becomes data description .....	44
4.10 Evolution of the program display .....	45
4.11 Problems with SmallStar data descriptions .....	47
4.12 Data descriptions and other programming-by-example systems .....	49

Chapter 5 Control structure .....	52
5.1 Control structure constructs.....	52
5.2 Control structure in other programming-by-example systems .....	52
5.3 Set iteration in early versions of SmallStar .....	53
5.4 Set iteration in SmallStar now .....	54
5.5 Conditionals and programming by example.....	56
5.6 Conditionals in SmallStar.....	57
5.7 Other control structure .....	61
Chapter 6 User trials.....	63
6.1 Purpose of trials .....	63
6.2 The trial users .....	64
6.3 Procedure.....	64
6.4 Observations .....	65
6.5 Results.....	66
Chapter 7 Discussion and conclusions .....	67
7.1 Summary of work; principal contributions.....	67
7.2 Syntactic and semantic programming by example .....	68
7.3 Creating the program: recording vs. editing.....	69
7.4 Real and contrived examples .....	70
7.5 Adding programming by example and SmallStar techniques to other systems.....	71
7.6 Programming by example and conventional programming .....	73
7.7 Conclusions .....	73
References .....	74

## List of figures

Figure 2-1. Recording a factorial program in Pygmalion.....	8
Figure 3-3. A selected paragraph and its property sheet.....	19
Figure 3-4. A SmallStar desktop.....	21
Figure 3-5. A program window containing a simple program.....	22
Figure 3-6. Pictographs used in program display.....	23
Figure 3-7. A sample program.....	25
Figure 3-8. Program display in the first version of SmallStar.....	27
Figure 3-9. Program display in the second version of SmallStar.....	27
Figure 4-1. The relation between operands and descriptions.....	32
Figure 4-2. Program to move a particular document out of a folder.....	33
Figure 4-3. A document description sheet.....	34
Figure 4-4. Program to move the first document out of a folder.....	35
Figure 4-5. A text frame description sheet.....	36
Figure 4-6. Documents for an order entry application.....	37
Figure 4-7. The original part number program, showing the incorrect row description.....	37
Figure 4-8. The description sheet for the incorrect row description.....	38
Figure 4-9. The corrected row description sheet.....	38
Figure 4-10. Program display in the second version of SmallStar.....	40
Figure 4-11. The spectrum of generalization.....	41
Figure 4-12. Program display using a data list.....	46
Figure 4-13. Program to delete all the text in a field in a table, displayed using a data list.....	47
Figure 4-14. An obscure program.....	47
Figure 4-15. A program with long description paths.....	48
Figure 4-16. A text string description sheet.....	51
Figure 5-1. Program to move one document out of a folder.....	55
Figure 5-2. Description sheet for any document in a folder.....	55
Figure 5-3. Program to move any document out of a folder.....	55
Figure 5-4. Adding the set iteration loop.....	56
Figure 5-5. Program to move all the documents out of a folder.....	56
Figure 5-6. A mailing form.....	58
Figure 5-7. Mailing form program, before addition of the conditional.....	58
Figure 5-9. The property sheet for the conditional.....	59

## Acknowledgments

Dave Smith and the ideas in his Ph.D. dissertation provided the inspiration for this research work, and started me with Xerox and programming by example. I would like to thank him for numerous enlightening and thought-provoking discussions, and most of all for his unflagging enthusiasm. My managers at Xerox, Ralph Kimball, Dave Smith, and Marney Beard have supported me intellectually, managerially, and fiscally. The members of the Advanced Development group at Xerox have provided valuable critical comments throughout this work. Bill Verplank, especially, has followed my work closely, has always been ready to discuss a new idea or look at another variation in the prototype, and has pointed out many parallels between programming by example and other concepts and systems.

Susan Graham, my research advisor at Berkeley, has followed this research from the start. Her questions and comments and her careful reading of this dissertation have been very valuable, and have especially kept me thinking about programming by example in a larger context. The other members of my committee have also helped a great deal: Richard Fateman was always thinking about applying programming by example to other areas, and Steve Palmer kept me honest about what I could and could not prove with experiments that tested the usefulness of programming by example.

The members of the System Concepts Laboratory at the Xerox Palo Alto Research Center created the wonderful Smalltalk environment which made coding the system a real pleasure, and contributed to many of the initial discussions about programming for non-programmers and programming by example. I would also like to thank the designers of the Xerox Star system, whose attention to simplicity and good design made Star an ideal system for trying programming by example.

Sara Bly and Marion Sturtevant, my readers at Xerox, provided excellent content and style criticisms; without their help this dissertation would have been quite a bit muddier. Janice Heiler at Xerox prepared this dissertation for reproduction as a technical report, and did a marvelous job, which included excellent copyediting and the redrawing of several blurry figures.

My carpooling partners, Eric Schmidt, Marc Meyer, Bob Hagmann, Sara Radicati, Eric Cooper, Ricki Blau, Susan Eggers, Gregg Foster, Mark Hill, and Helena Winkler were all wonderful company on the long ride between Berkeley and Palo Alto.

I dedicate this dissertation to my parents, Melvyn and Edith Halbert.

# Chapter 1

## Introduction

### 1.1 Thesis

Programming by example is a way of programming a software system in its own user interface. The user of the system writes a program by giving an example of what the program should do. The system remembers the sequence of actions, and can perform it again. Succinctly, programming by example is “Do What I Did.”

Programming by example can be a practical way of letting the user of a system create programs to automate the tasks he needs to do. This dissertation describes an enhanced programming-by-example mechanism which has been incorporated into a simulation of an office information system. To make programming by example a reasonable way of writing more than very simple programs, the basic concept was extended to include a data description mechanism, methods for adding control structure to programs, and a static program representation suitable for reading and editing. The resulting system is learnable and can be used to create practical programs.

### 1.2 Motivation

Most computer software used to be written with its ultimate users clearly in mind. A computer installation had its own programmers who would write user-specific software. Even if software from outside was used, it was frequently modified locally to suit the idiosyncrasies or particular applications of each installation. Computers were expensive, programmers less so.

But computers have become much cheaper, and are now commonplace at work and at home. There is a corresponding explosive growth in the amount of software available. Computers and software are becoming mass-market merchandise. Users frequently buy general-purpose software, expecting it to include enough functionality to cover their applications.

The user no longer has a programming staff to fall back on. How then can he make a general-purpose system into a special-purpose one that suits his needs? The system might provide the mechanism for the user to do his task, but if it requires seventeen general-purpose operations to do what could take one special-purpose operation, the user will be frustrated and annoyed. One solution is to let the user do his own programming. Certainly we should not expect the average user to write programs with the same ease as professional programmers. (See [Anderson] and [Cuff] for general discussions about the problems of introducing programming to the average person.) What we can do instead is to find ways to make programming easier, to couch it in terms with which the user is familiar, and to reduce the mental burden of programming.

Programming by the user has already appeared in a number of different ways. The degree of user programmability can vary from choosing among options the system provides, to systems which require programming by the user as an integral part of their functionality.

For user programming in its simplest form, systems provide options or modes from which the user can choose in order to customize the operation of the program. For instance, a program that prints might provide an option of single or double spacing. Though it might be stretching a point to call this real programming, the system is supplying a very simple, specialized programming language.

Other systems let the user create programs consisting of sequences of keystrokes or other simple user actions. Keystroke programming is particularly common in text editors; the sequences the user creates are frequently called *macros*. The well-known text editor TECO [Digital] takes this idea of macros to an extreme. The Wang Glossary word processing system [Wang] provides a simple programming language (*Decision Processing*) whose actions are the names of the keys. The advantage of this programming technique is that the user is programming in the user interface of the system. Since he already knows this user interface, he does not need to learn a whole new programming language, though he may need to learn new commands that relate specifically to programming and not to normal, “manual” operation of the system.

Fewer systems provide a separate programming language for automating their operations. The Xerox Star office system [Smith *et al* 1981] [Smith *et al* 1982] has experimentally provided a language called *Cusp* (*customer programming*) which has English-like syntax and is specifically designed for manipulating the data objects found in the Star system. The MACSYMA algebraic manipulation system [Mathlab] is programmable using a language which is a superset of its user-interface commands and syntax.

Finally, some systems are programmed by their users as a necessary and integral part of their operation. Spreadsheet programs are a good example. They do nothing without the user supplying a program by giving formulas for various spreadsheet cells.

The methods of user programming we have discussed above are all quite popular. Most software systems provide options or modes for tailoring their operation to suit particular users. Spreadsheets are an example of an instant and phenomenal success. Keystroke programming is widely used, and is usually the first thought when one thinks of adding programming to a user interface.

What makes these methods of user programming so popular? Why do we not instead expect the normal users of a system to write programs for it in some more conventional programming language? The answer lies in what makes programming difficult, and how the methods of user programming we have mentioned circumvent these difficulties.

### **1.3 Why is programming considered hard?**

Why is it that programming appears to be difficult, and something our average non-programming user does not want to attempt? There are two strong reasons: one is based on knowledge, the other on skill or talent.

The knowledge a programmer must have about a particular programming language is largely a mass of detail. He must know the exact syntax and semantics of the language he is using. These details differ from language to language; constructs that appear the same in two languages may have different meanings. A good programmer is good at remembering such details. A non-programmer or casual programmer has no desire to learn such details; they take time to learn, and one forgets this detailed knowledge unless it is used frequently.

Programmers must also learn idioms: there are idioms peculiar to particular programming languages, and there are standard programming idioms which one is trained to use (e.g. hash tables, iteration constructs, use of backpointers, etc.). Programmers know how difficult it is to write programs in a new language the first few times, even if one knows a number of other languages. It is easier to see how other people use the language by looking at existing programs than simply to refer to the language manual. Programmers also know that their knowledge about data and program structures takes a few years to learn, either through a formal computer science education, or through some sort of apprenticeship where they study and modify existing programs before writing their own.

This side of programming ability is one of knowledge and education. Another necessary ability is one of skill or talent. It is the ability to create abstract plans without feedback.

A programmer must keep the state of an intangible reality in his mind. The programmer does not see the effects of the program he is writing until he runs the program; he must imagine them instead. Even when his program is run and produces output, most of the actions and decisions the programmer has specified happen invisibly.

A non-programmer is not used to creating plans in such an abstract manner. Most people can follow and give directions with little trouble. They can read and write recipes, follow office procedures, and tell their friends how to drive to their houses. But all these actions occur in familiar physical reality. It is easy for someone to imagine the consequences of an action, because he is familiar with its physical realization, and can picture what would happen.

A person following or creating a plan that deals with ordinary reality has a chance to try out the plan, step-by-step, and modify it as he proceeds. He can do this in his mind's eye, or actually try out the steps of the plan. A person following a plan can observe its effects as he executes each step, and stop or modify the plan if it seems that something is going wrong.

A programmer instead must create his plan (a program) without being able to try it out as he writes it. He cannot experiment with parts of the plan beforehand, because in conventional, non-interactive programming systems there is no simple way to try out small pieces of a program. When he runs the entire program to test it, it is frequently difficult for him to execute it step-by-step, and usually impossible for him to modify it as he tries it. Thus he must be skilled in mentally simulating his plan and imagining its consequences before it ever runs on a computer. Not all people have this skill naturally, though many can learn to do it, with effort.

In order to make user programming easier, we must therefore attack these two problems: we must reduce the amount of detail the user must know before programming, and we must reduce the mental overhead necessary to create the plan that is the program.

## **1.4 Programming by example**

We need to remove these two obstacles that we have mentioned: one is lack of knowledge, the other is lack of skill. To help the user avoid learning a mass of new detail, we should let him rely mostly on what he was already learned and practices often. To help him create programs more easily, we should let him build them incrementally, so he can test them as he creates them. These two ideas are the essential features of programming by example.

A user must learn to operate any system he intends to use regularly. There is no substitute for practice and training if he is to become adept. However, once he does learn the system, it becomes second nature to him.

How can we let the user program with the knowledge he already has? The answer is simple: let him write programs in the world of the system with which is he is already familiar, using exactly the system operations he already knows. There may be a need for additional commands, for instance to add control structure to the program. But in general, the commands in his programming language will be the same as the commands he uses normally.

Letting the user try out his program as he is writing it is equally simple. We just record the user's commands as he operates on sample or real data, transcribing his commands to form a program. This recording does not interfere with or change the normal operation of any command. At any time during



recording, the user can see that the program has worked correctly up to that point (at least for his example), because he can see the results of his actions immediately.

If these two ideas were the entire substance of programming by example, there would be nothing left to say. But just letting the user record his actions and have them repeated exactly makes for dull programs. Creating a programming-by-example system becomes more interesting and more difficult when we try to provide ways for the user to specify the intention of his program or add control structure to it. That is the subject of my research.

### **1.5 To reiterate — what programming by example is**

- When the user writes a program in a programming-by-example system, the statements in his program are the same as the commands he would normally give the system. So he is *programming in the user interface* of the system.
- A program is written by remembering what the user does as he gives normal commands. Thus the user programs by *giving an example* of what he wants the program to do.

A system must provide both of these features if it is to be called a programming-by-example system. Certain systems provide programming in the user interface, but they are not programming-by-example systems because they lack the ability to record a program as it is being executed. For instance, many text editors provide what is commonly called a “macro” facility, in which sequences of ordinary commands can be stored and executed repeatedly. However, in most of these editors, the individual commands in the macro are not executed as the macro is being written. A similar example is the scripts one can write for the command languages of various operating system executives or shells.

### **1.6 Requirements for a programming-by-example system**

Since programming by example includes programming in the user interface of some system, it follows that in order to program a system by example, it must have a user interface. That is, the system must provide a visible world in which the user can manipulate and change data “manually”, using commands that take effect immediately. Such worlds are familiar to programmers: examples are command languages and the environments they manipulate, and text and graphic editors.

But rarely is such a world available in conjunction with a conventional programming language. With most languages such as, say, Pascal, there is no visible and easily manipulated world that exists independent of the program (though sometimes the debugger tries hard to give the impression of such a world). Instead, a Pascal program specifies its own world. The world is created when the program is run, and destroyed when it finishes. Beforehand, when the programmer is writing the program, the world exists only in his mind’s eye.

To create a programming-by-example system for writing Pascal programs, one would have to create a world of Pascal objects, and a user interface for it. The actions available in the user interface would presumably map closely into Pascal program constructs. The user would manipulate the objects in the world, and his actions would be transcribed to produce a Pascal program.

Some language systems do provide permanent worlds which exist across the invocation of programs. Lisp, APL, and Smalltalk all have global static spaces in which global names and the objects to which they are bound live indefinitely. Users can execute language constructs directly in the user interface to compute values and possibly change the state of the world.

However, many of the constructs in these language systems are not imperative; they do not change the state of the world. Instead, operations on objects are usually applicative: they create new objects, leaving the original objects undisturbed. Newly created objects are discarded immediately unless they are bound to some global name. Successive user actions must be linked together explicitly to take effect together. For instance, in Lisp, one may say:

```
(list 1 (+ 1 2))
```

The system responds with:

```
(1 3)
```

But in order to get the first element of this list, one must say:

```
(car (list 1 (+ 1 2)))
```

since the results of the first action have not been saved; the (1 3) generated by the system is ephemeral. Thus it is awkward for the user to build a program out of successive commands to the system, unless some mechanism is added to allow these applicative actions to be combined easily, such as automatically binding the intermediate results to system-generated variable names. (This is the technique used in the MACSYMA algebraic manipulation system [Mathlab]. The Tinker system for programming by example in Lisp [Lieberman & Hewitt] [Lieberman] saves intermediate values in a more sophisticated way; see section 2.6.)

The user actions in non-programming-language systems are usually imperative. For instance, most commands in a text editor alter the state of the world: they change the text, the current selection, the cursor position, and so forth. In the Xerox Star office system, used as the basis for the programming-by-example research discussed in this dissertation, most commands are also imperative. In these systems, since each user command changes the state of the system, several commands executed in sequence naturally form a straight-line program.

## 1.7 Limitations of simple programming by example

I have already stated my primary motivation for adding programming by example to a software system: I want to make it possible for normal, non-programming users to write programs. Programming by example makes this possible by making programming easier. It allows a user to write programs in the user interface of the system, with which he is already familiar, and it intertwines the acts of *demonstrating* what the program is supposed to do, *writing* the program, and *testing* it.

These are the strong points of programming by example. But there also disadvantages in using the technique, some of which can be overcome by extending the basic concept.

In a pure programming-by-example system, the user records a program which will repeat precisely what he did. Occasionally, running such invariant programs is useful, but most of the time, doing *exactly* the same thing is not what the user has in mind. The user would like to *parameterize* or *generalize* the program, so he can vary the data the program uses. He should be able to indicate to the system that certain data objects used in the example served merely as placeholders, and should be replaced by parameters.

Other times, the user may have chosen data for the example because the data has certain salient characteristics. For instance, he may choose the first item from some list, or a string that matches some pattern. The item chosen is not a constant, but it is not a parameter to the program either. The user should be able to provide a computation or a description that will choose data with the desired characteristics.

Pure programming by example also limits the programmer to straight-line programs. Again, this is a restriction on the usefulness of the programs that are recorded, since two of the wonderful things about computers are their abilities to repeat actions and to make tests which choose between different actions. So the programming-by-example system should be augmented to provide some kinds of control structure, such as iteration and the conditional execution of actions.

Programs created conventionally have a static representation which is written by the programmer. The programmer can look at the static representation to read or edit his program. But since a program written by example is not created by typing in some code, there may be no human-readable static representation, though obviously the system must keep some internal representation. Nevertheless, the programmer may want to look at the program he has created without just running it again, in order to edit it or just to review it. One way is to execute the program one step at a time, observing the effect of each step. But this will be unsatisfactory for longer programs or if the program needs to be edited; the user probably wants to be able to read the program, instead of just watching it. So some static representation of the program transcript is desirable.

For programs written in a typescript system, such as a command language or a text whose actions are invoked solely from the keyboard, the transcript language can be a simple record of the user's keystrokes. But for a system in which, for instance, commands are selected using a mouse, some other transcript language will have to be invented. The user does not need to be able to type in program text directly; readability is what is important.

## **1.8 Research history**

The story of the evolution of a research project is rarely told. Instead, the final result is presented as a *fait accompli*. But the development of research ideas, the blind alleys taken, and the gradual improvement in understanding of the basic issues form an interesting tale in themselves, and may help others to understand the motivations and conclusions of the dissertation. In each chapter, including this one, I will include information about how the research path developed, what other techniques were tried, and how new ideas came about.

The initial impetus for this research came from reading David Smith's dissertation: *Pygmalion: a Computer Program to Model and Stimulate Creative Thought* [Smith]. This dissertation describes a system, Pygmalion, which combines two powerful ideas: graphical programming (that is, programming not using a conventional textual programming language), and programming by example. I had originally read this dissertation in 1977 while searching for material about visual debuggers. In 1979 I started preliminary investigation into graphical programming, with Pygmalion being one of the main inspirations for studying this topic. Late in 1979 I visited Smith at Xerox in Palo Alto, and before I knew it, I had a summer job with the Office Products Division (now the Office Systems Division) of Xerox, in association with the Learning Research Group (now the System Concepts Laboratory) at the Xerox Palo Alto Research Center. My charter was to think about programming for the non-programmer. Six weeks of thinking led to a short memorandum discussing possible research directions [Halbert 1980]. At that point it became clear that programming by example was worth further study, and that it would be useful to build a prototype of a system that included programming by example. This dissertation describes the work that followed.

## Chapter 2

### Work by others

The idea of programming by example cannot be traced to a single seminal bit of innovation. The idea has been used successfully a number of times, and is simple enough that it is reinvented from time to time. In this chapter I will review briefly other work in programming by example. Certain aspects of some of the systems mentioned here will be discussed in more detail later.

#### 2.1 Simple examples

Industrial robots have long been programmed by example [Tanner] (also see [Vonnegut]). The programmer of the robot can either physically move the robot's "arms" (*leadthrough*) or else command the robot from a control panel (*walkthrough*), and have the robot record the sequence of movements and actions. The machine moves the workpiece, its drill bits, cutting tools, spray guns, arc welders, and so forth as it follows the program.

To create a program on a programmable calculator, the user asks the calculator to start remembering keystrokes, and then pushes the keys that would be pressed if the same calculations were being done by hand. Thus the programming is done using the normal user interface of the calculator. No calculations are actually done during program recording, so this is not real programming by example. Although one early programmable calculator did use programming by example [Kahan], no current calculators have adopted the idea.

Radia Perlman, visiting at the Xerox Palo Alto Research Center from the MIT LOGO laboratory, created a programming-by-example system for young children, called *TORTIS* (described in [Smith]). Using a simple control box with a few buttons on it, a child could manipulate a *turtle* (a small robot that runs on the floor), making it go forward, turn, and raise and lower a pen which was used to draw pictures. Sequences of these actions could be recorded by example. Children as young as four years old were able to understand the concept of writing a program by example, and frequently wrote error-free programs the first time, because of the by-example nature of the programming.

The screen text editor EMACS [Stallman] contains a simple programming-by-example system for sequences of editor commands called *keyboard macros*. The user can ask that a sequence of his actions be recorded as they are being executed. The sequence is bound to a single command key for later playback. Other text editors provide similar "macro" capabilities, though not all actually perform actions as the program is being recorded, and hence do not provide true programming by example.

#### 2.2 Spreadsheet programs

Electronic spreadsheet programs, used mostly on microcomputers, have become very popular during the past few years. They are widely used by non-programmers and are easy to learn and use. One reason is the immediate feedback they produce, both during set-up and use. When the user of a spreadsheet enters a formula for a spreadsheet cell, he may beforehand have entered some data in the cells used in the formula. The formula is usually applied immediately after type-in, and so the user can see the result and determine if it makes sense. This reflects the immediate program testing aspect of programming by example.

In some spreadsheet programs, the user need not even type in the coordinates of the cells used in a formula. He may instead, point at the cells with the cursor while entering the formula. Thus, there is a

close association in his mind between the actual data and the computation to be done. The user is coming very close to writing a formula by giving an example.

### 2.3 Pygmalion

David Smith's *Pygmalion* system [Smith] combines programming by example and *graphical programming* (expressing programs using two-dimensional graphical objects, not just linear strings of text). Pygmalion has provided the inspiration for much other research in both of these areas.

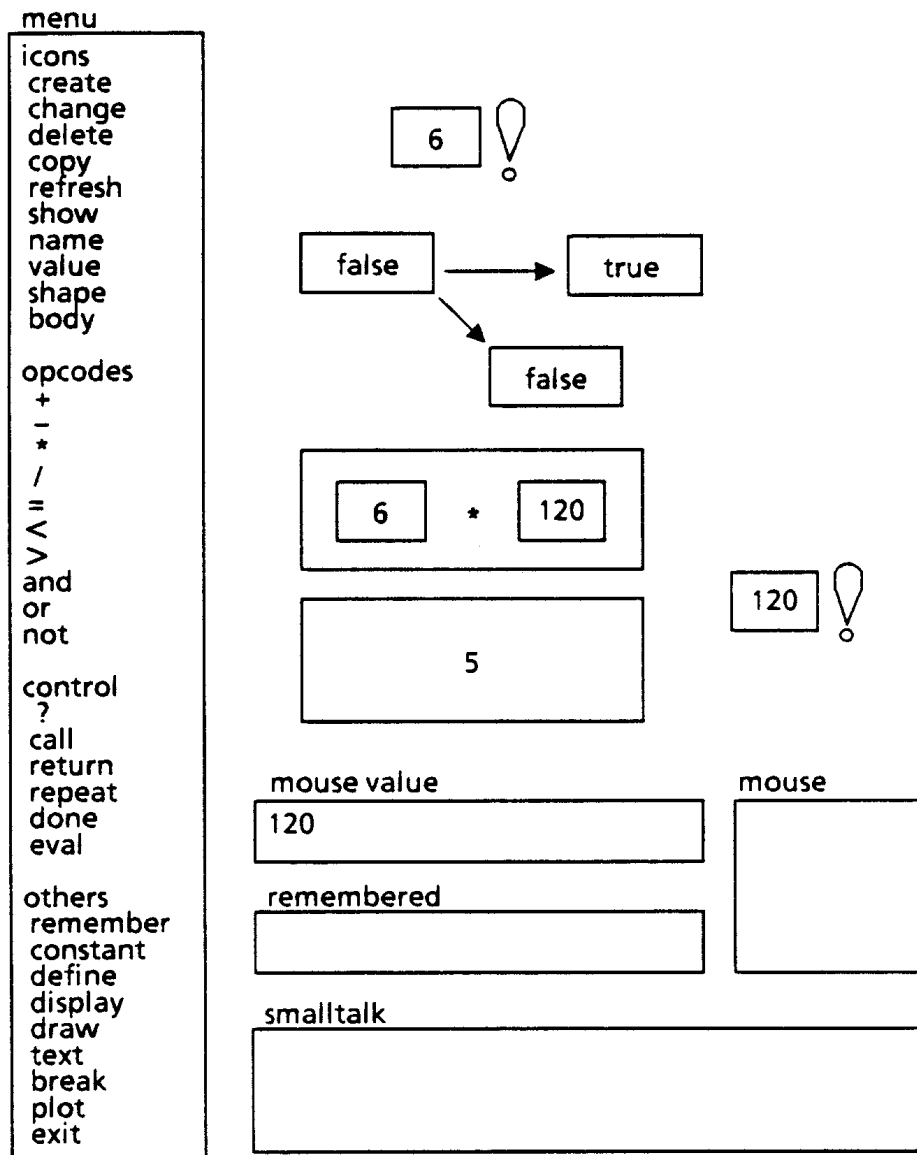


Figure 2-1. Recording a factorial program in Pygmalion.

The Pygmalion user has a world of iconically represented operations and operands he can manipulate by hand. He can move data values into and out of icons, ask operator icons to do their appointed tasks, and create new icons and specify their shapes. The operator icons provided are those of a typical

programming language: the usual arithmetic, relational, and boolean operators. For instance, the user can move two numbers to a multiplication icon, and then have the multiplication icon multiply the numbers and replace itself with an icon containing the result.

All programming is done by example. The user's icon manipulations are recorded and stored in an icon for playback. The system has conditional statements and an infinite loop construct with a separate "exit loop" statement.

Figure 2-1 shows a step in the creation of a recursive program to compute factorials. The box with an '!' next to it is the icon for the factorial program. The user is about to multiply 6 by 5!, which has already been computed to be 120 by the factorial icon next to the multiplication icon. The three boxes connected by two arrows form one icon, which contains the conditional test for termination of the recursion. Only the result (false) is shown because the predicate has already been evaluated.

## 2.4 Programming by abstract demonstration

Gael Curry built a graphical programming system using programming by example that is similar to Pygmalion [Curry], though he calls it programming by *demonstration*. A user of the system can manipulate an iconic world like Pygmalion's, but the user is supplied with a fixed set of kinds of icons (memory cells and operators), and cannot create new kinds of icons. The basic data values are numbers. The operations provided are almost the same as those in Pygmalion, though an operator icon produces a result memory cell instead of replacing itself with the result of its operation.

Curry's system is based on a programming methodology called *programming by abstract demonstration*, or *PAD*. The user can write programs by example which contain only constants, using concrete sample data. However, the user can also record programs using abstract data values. A datum can be completely unspecified and abstract, or the user can specify that a datum will fall within some numerical subrange.

The PAD system can execute programs abstractly as well as concretely, producing abstract, approximate results. For instance, adding a numerical range and a number produces a result that is a numerical range. If a loop is executed abstractly, the system iterates until the state produced by the loop becomes stable (if possible).

Curry provided abstract execution for his programming-by-example system because he felt it would be easier to show that a program written by example would work for an entire class of inputs if the sample data were made abstract. The user can look at what a program does by executing it abstractly, without supplying real data. In Pygmalion, the user must supply concrete sample data to a program and run it in order to examine it.

## 2.5 Tinker

Henry Lieberman has designed a programming system for Lisp called *Tinker* that includes aspects of programming by example [Lieberman & Hewitt] [Lieberman]. The system is somewhat different from the others we have described because the user is aware he is writing a conventional Lisp function, but tests it as he is writing it by supplying sample values for the variables. Unlike some other systems we have mentioned, Tinker does not emphasize programming for non-programmers. A user of Tinker should know how to program in Lisp, though he need not be very experienced.

A function is constructed in Tinker by supplying data, applying functions to it, and building up larger expressions out of these pieces. The pieces are selected and programs are built using display menus of

commands and program fragments. Tinker keeps track of how the pieces are being fitted together, and builds a conventional Lisp function incrementally.

For example, suppose the user wants to build a function, called 2nd, to retrieve the second member of a list. He supplies a sample datum, say (1 2 3), as the argument for 2nd. Tinker creates a variable name, 2nd-Arg-1, for the datum, associates the two, and then puts the pair in a menu. Then the user selects (1 2 3) with its associated expression 2nd-Arg-1 from the menu, and applies cdr to it, yielding (2 3). This result is also inserted into the menu. Tinker now displays the function 2nd as

```
(defun 2nd (2nd-Arg-1)
  (cdr 2nd-Arg-1))
```

Now the user selects (2 3) and its associated expression, (cdr 2nd-Arg-1) from the menu, and applies car to it, yielding 2. The function definition is now complete, and Tinker displays:

```
(defun 2nd (2nd-Arg-1)
  (car (cdr 2nd-Arg-1)))
```

The user inserts conditionally-executed statements into his program by doing other examples with different sample data. At some point, the user will do something different than what he did in a previous example. At that point, the system asks the user for a predicate which will distinguish the two cases, and inserts a test of the predicate into the program. One branch of the test will execute the old code; the other branch will execute code about to be supplied.

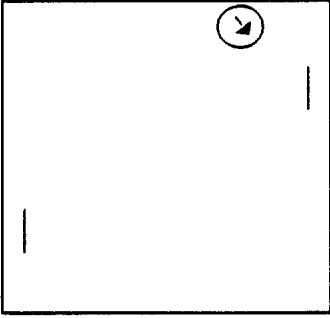
<p><b>Tinker EDIT menu</b></p> <p>TYPEIN and EVAL          TYPEIN, but DON'T EVAL          NEW EXAMPLE for function          give something a NAME          fill in an ARGUMENT          EVALUATE something          Make a CONDITIONAL          Edit TEXT          Edit DEFINITION          Step BACK          UNFOLD something          COPY something          DELETE something          UNDELETE thing deleted          UNDO the last command          LEAVE Tinker          RETURN a value</p>	<pre>(DEFUN PONG-STEP ()   (MOVE-PADDLES (GET-POINT))   (BOUNCE-BALL    (SEND BALL ' :XCOR)    (SEND BALL ' :YCOR)    (SEND LEFT-PADDLE ' :YCOR)    (SEND RIGHT-PADDLE ' :YCOR)    (SEND BALL     ' :FORWARD     PONG-SPEED))</pre>	<p>Laura: 2, Henry: 3</p>  <p>Graphics</p>
<p><b>Defining (BOUNCE-BALL 200 350 -200 150):</b></p> <p>Result: 200, Code: BALL-X          Result: 350, Code: BALL-Y          Result: -200, Code: LEFT-Y          Result: 150, Code: RIGHT-Y          Result: ?, Code: (SEND BALL (QUOTE :BOUNCE-OFF) 0)</p>		
<p>Type something to evaluate:          (SEND BALL ' :BOUNCE-OFF 0)</p>		

Figure 2-2. Creating a pong video game with Tinker

Lieberman has found that the by-example nature of programming in Tinker is particularly effective when building programs that produce graphical output. The effect of each graphic operation can be seen

immediately, and errors can be corrected quickly. In figure 2-2, the programmer is in the middle of constructing a pong video game. The top middle window shows the program as it is being constructed, and the second window from the bottom is a menu of program fragments and their results.

## 2.6 Programming by Rehearsal

Laura Gould and William Finzer, working in the Systems Concepts Laboratory at the Xerox Palo Alto Research Center, have built a programming system called the *Rehearsal World*, which uses a programming paradigm known as *Programming by Rehearsal* [Gould & Finzer]. The system was created for curriculum designers, to allow them to program their own interactive educational activities.

The system uses the theater as its organizing metaphor. There are predefined *performers*, which perform actions when given specific *cues*. Related cues for each kind of performer are grouped together into *cue sheets*. Performers are visible, graphical objects. Prototypical instances of each kind of performer can be copied by the user for incorporation into his own programs. Related performers are grouped together into *troupes*.

Figure 2-3 shows the creation of a spelling game in progress. The game is shown in the window labelled *Jumble1*. The box labelled *JUMBLE* is a button performer that, when pressed, will send a cue to the text performer above it that rescrambles the letters there. The box in the window *Jumble1Wings* is called *List1*. A cue sheet for *List1* is shown at the bottom of the figure. The *ControlTroupe* window contains performers useful for implementing control structure.

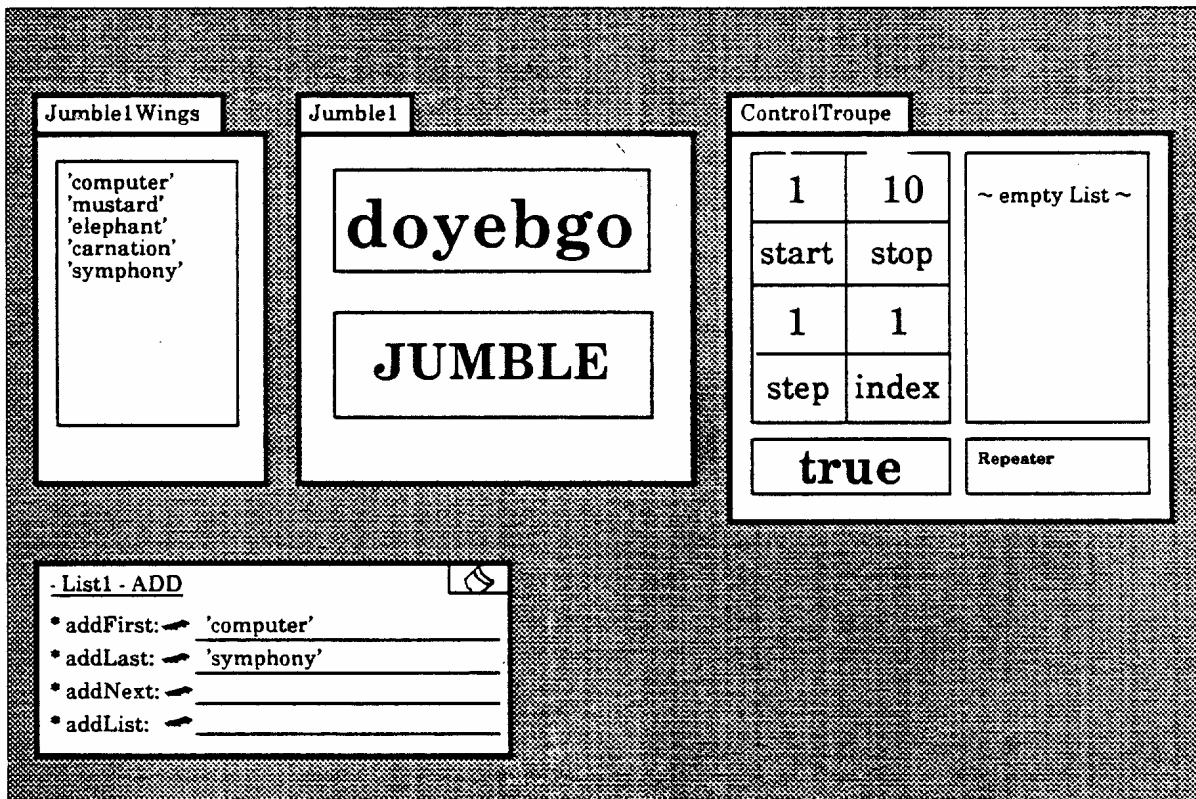


Figure 2-3. Building a jumbled word game in the Rehearsal World.

The Rehearsal World uses programming by example to record the invocation of cues. In the figure, the four eyelash symbols to the left of the blanks in the cue sheet are *eyecons*. When an eyecon is pressed,



it opens to watch the user do something for one step, and records the code for the user's action in the blank space next to it. For instance, in order to make the JUMBLE button do something, the user opened an eyecon in a cue sheet for the JUMBLE button, and then invoked the cue *setJumbled* in a cue sheet for the text performer above the button (these cue sheets are not shown in the figure).

The Rehearsal World supplies a very simple kind of programming by example, treating all objects used in an example as constants, and not providing for control structure created by example. The user does not have to use programming by example in order to create code. Nevertheless, many users find the eyecon a very useful and natural way to write code. The Rehearsal World has been used successfully by several curriculum designers with little or no programming experience to create quite complicated educational games.

## 2.7 Exemplary programming

Don Waterman, William Faught, and others at the Rand Corporation have done work on programming by example, though they call it *exemplary programming*, or *EP*. They have built two systems, EP-1 [Waterman] and EP-2 [Faught 1980a] [Faught 1980b], both of which can be used to automate sequences of commands to programs that use simple typed input. In both EP-1 and EP-2, the user generalizes his programs by saying explicitly, before creating the typescript (called the *agent*), which of the data items used in the example are to be considered as parameters in the recorded program. The system queries the user for the names of the parameters, and what values they will have in the given example. After the user has finished giving the example for the agent, the system substitutes parameter variables for the uses of the example data.

In EP-2, the user can also insert loops and conditionals in his programs, but must do so using the *mind channel*, a special mechanism that circumvents the recording mechanism. The user must have some knowledge of Interlisp in order to use the mind channel.

## 2.8 Juno

*Juno*, built by Greg Nelson [Nelson] (also described in [Gould & Finzer]), is an experimental geometric programming system. A Juno user lays out points on a screen, and then draws line segments, shapes, and figures, using those points. Juno records the user's actions as he does them, and so builds a program by example.

More interestingly, Juno lets the user describe constraints about sets of points. There are only a few basic constraints: a pair of points must lie on a horizontal or vertical line, the line segment defined by a pair of points must be the same length as another line segment, or, two lines segments must be parallel. More complicated constraints are built out of these simple ones. When the user specifies constraints, they are included in the program being recorded.

Complicated figures can be constructed by combining and repeating smaller programs. Juno has a powerful constraint satisfaction mechanism, capable of solving the non-linear as well as linear equations that are produced by combining constraints.

If the user edits a figure, the corresponding program is also edited: deleting a point, for instance, will delete the parts of the program that describe the point and the constraints related to it. The figure will then be redrawn by executing the new version of the program. The program text is visible to the user and can also be edited directly.

Juno has proven useful and instructive, both as an artistic tool and as a system for doing research on constraints.

## 2.9 Dynamic program building

Peter Brown has experimented with a programming-by-example technique called *dynamic program building* [Brown], and has built a system called *Build*. In *Build*, the programmer types in statements in a BASIC-like language. The system is always trying to execute the program. It stops and waits for the user to type in more statements whenever it encounters a *loose end*, which is a point where no statements have yet been entered. An empty program contains a loose end, so the system waits for a statement to be typed, executes it immediately, and then waits for another statement.

When constructs such as **while** loops or **if-then-else** statements are entered, they initially contain loose ends. If a user types in a **while** loop, with its predicate, and the predicate is true, the system will stop at the loose end in the body of the **while**. The user can then enter statements one at a time, each of which will be executed in turn. When the user terminates the loose end, the **while** will start looping, and will continue until the predicate evaluates to false. Then the program will stop at the loose end after the **while**, and the user will be able to enter more statements.

If the user wishes to change part of the program he has already typed in, he moves to that point and indicates he wants to edit. *Build* will insert a loose end at that point, and re-execute the program from the beginning. The program will stop at the loose end, and the user can then delete or insert statements.

This scheme is intriguing, but poses some problems. If the program is to be re-executed from the beginning, the system must restore global static state, such as the contents of files, to what it was before the program ran the first time. In addition, executing parts of the program, particularly loops, many times over as the program is being edited might be quite time-consuming. Nevertheless, dynamic program building is a novel idea that might work very well in certain contexts.

## 2.10 Query-by-Example

Moshe Zloof and his colleagues have been working for several years on a system called *Query-by-Example (QBE)* [Zloof 1977]. Though the name sounds similar to programming by example, Zloof's approach is actually quite different, because QBE does not build algorithms by example.

The system operates on relational data bases. The user initially sees the headings of a relation, with no data shown below them. To fetch items from the data base, the user creates a sample, but possibly fictitious entry in the table. For instance, if the user wants to find everyone who works for the manager Smith, he adds the following to the table:

<u>Name</u>	<u>Salary</u>	<u>Manager</u>
P. <u>Jones</u>		Smith

Jones is an example of an employee who works for Smith. He may or may not actually exist. P. is a command that asks the system to print everyone who fits the given query. Jones is underlined by the user because it is an example (a variable). Using an example name such as Jones is not actually necessary when giving such a simple query.

To find everyone who works for a manager who works for Martin, the user says:

<u>Name</u>	<u>Salary</u>	<u>Manager</u>
P. <u>Doe</u>		<u>Doe</u>
		Martin

Here, Doe is used to link the two parts of the query.

Predicates may also be given in queries. To find out whose salary is more than \$20,000, the user gives the query:

<u>Name</u>	<u>Salary</u>	<u>Manager</u>
P.	>20000	Doe

We have described so far only a smattering of what the QBE system can do. More complicated predicates can be given, several relations can be used in a query, data base entries can be inserted (command I.), updated (U.), deleted (D.), and so forth.

In my programming-by-example work, we use the word “example” to mean an example execution of an algorithm. The procedure itself is as important as the result. QBE uses “example” to mean the example result of an algorithmic procedure whose details are unimportant to the user. The user does not write an algorithm by example for the data base operations; instead he gives a constraint on what the system may return when it is given a query. This constraint is given with the help of an example result.

Zloof and his associates have integrated QBE into *Office-by-Example (OBE)* and the *System for Business Automation (SBA)* [Zloof & de Jong] [de Jong & Byrd] [Zloof 1981]. The user can write programs to update data bases, send mail to others, print reports, and similar actions. These programs contain data base queries that use QBE, use the underlined variable mechanism of QBE to parameterize form letters, and so on. But the procedural parts of these programs are not written by example.

## 2.11 Programming by homomorphisms and descriptions

Giuseppe Attardi and Maria Simi [Attardi & Simi] have combined aspects of Query-by-Example and the System for Business Automation (described in section 2.10) with programming-by-example techniques, to create the *PHD* system (*programming by homomorphisms and descriptions*).

The procedural aspects are created by example, as in Tinker, Pygmalion, and similar systems already mentioned. However, data selection and iteration constructs are done using a declarative technique which resembles the QBE query mechanism.

To describe an iteration set, a set of data items is specified using QBE. An operation is performed on the whole set by applying some simple or complex operation to an example member of the set, and then the operation is used to construct a homomorphism which operates on the whole set.

Attardi and Simi show the construction of the following program, among others, in their paper:

*Compute the total amount due from a collection of invoices, summing up the amounts from the invoices which have not been paid.*

We will summarize it here as a simple example. Assume there is a data base of invoices; each invoice contains, among other fields, a **Paid** field and a **Total** field.

The user first creates a skeletal function **Total-due**, which returns one value, the sum described above; on the screen a labelled box with a result field appears. The user then creates a QBE query for all unpaid invoices, by asking for all invoices in which the **Paid** field contains **NO**. He then invokes the **query** command, and ends up with a list of invoices.

The user then uses the **map over** command, giving as an argument the list of invoices, because he is going to define a mapping for the homomorphism which will compute the sum. As an example, he fetches the value in the **Total** field from the first invoice and indicates this is the function to perform on the rest of the list. He ends up with a list of numbers, which are the values of the **Total** fields of all the invoices.

Finally, the user shows how to combine these values, by asking the system to perform the  $+$  operation over all of them. The  $+$  operation conveniently takes multiple operands, not just two. (Spreadsheets usually provide a similar operation, allowing the user to calculate the sum of a whole range of cells.) The user ends up with the sum he desired, copies this value to the result field of the Total due function, and then he is done.

Attardi and Simi claim that using homomorphisms for programming by example reduces the need for explicit conditional and iteration constructs, and provides a convenient way to generalize programs. In the example, above, for instance, retrieving all the relevant invoices was done without a loop or a conditional, and fetching the Total field value from those invoices and applying the  $+$  operation over the resulting list were both done without resort to explicit loop control structure.

## 2.12 Inductive inference

A somewhat different approach to programming by example is taken by some other workers in this area. Their methods involve synthesizing a program from a number of examples, using *inductive inference* techniques.

Alan Biermann and Ramachandran Krishnaswamy describe such a system, which they call an *autoprogrammer* [Biermann & Krishnaswamy]. At conditional forks in the program structure, the user tells the system why he is doing the following steps (e.g. **note  $I > 0$** ). Several similar steps in the examples can be inferred to be several executions of the same step in the synthesized program. Biermann and Krishnaswamy prove that their method produces correct programs, if the user has gone through all the paths in the intended program, and also prove that all possible programs can be synthesized, given suitable examples. Variables and constants are noted explicitly. Their system has a nice graphical user interface.

Michael Bauer describes a similar system, but emphasizes its artificial intelligence aspects, noting that it depends on knowledge about variables, program inputs, instructions, and procedures [Bauer]. Ian Witten describes a simple desk-calculator system which does not include conditionals, but which predicts the remaining portion of a program based on an unfinished example and previous examples [Witten]. Program parameters are detected by noting differences between the constants used in the various examples.

Robert Nix describes a radical approach to inductive inference in a system which provides what he calls *editing by example* [Nix]. The system generates programs to do text transformations. The user gives the system one or more examples of some text before and after some editing operations. Based on these examples, the system constructs a generalized program to do the transformation. Note that the system does *not* look at the editing operations themselves, only at the example input and output. Input and output examples are given separately and do not always need to occur in pairs. By using various heuristics, the system can synthesize some desired programs using just a few input examples and perhaps only one output example for each program. Synthesized programs are presented to the user so that he can verify their correctness.

This approach is very different from the algorithmic methods of programming by example used in systems such as Pygmalion, since it pays no attention to the user's actions. Nevertheless, Nix finds it a feasible way of creating text-editing programs, and believes further work should be done on similar practical applications of inductive inference.

## Chapter 3

### The example system

The Xerox Star 8010 office information system was used as the starting point for doing research into programming by example. This chapter will describe the reasons for choosing Star, the salient characteristics of the system, and the simulation of it that was constructed for use in the research.

#### 3.1 Selection of the system

In order to study programming by example, I needed a system that provided the user with a world he could manipulate through a user interface, without programming. There are a number of common software systems that fill this requirement, some of which have already been mentioned: text editors, graphic editors, operating system command languages, electronic mail systems, programming language systems such as Lisp, APL, and Smalltalk, and so on. But in order to make sure I was looking at programming by example in a general way, I wanted to choose a system that was fairly general in what it tried to do, and one that provided a lot of varied functionality. I also wanted to use a system that had a graphical user interface, because I felt such an interface would make programming by the user easier, and because it would be more fun, both for me and for the user.

Clearly, the most general systems available were programming language systems. Since the initial research was done in association with the then Learning Research Group at the Xerox Palo Alto Research Center, an obvious choice would have been to add some kind of programming by example to the Smalltalk system. But the Smalltalk system has applicative aspects, as mentioned in section 1.6, which would have meant augmenting the user interface in some way. I also felt that such an effort would largely duplicate that of Henry Lieberman's Tinker system for Lisp (see section 2.5).

Another possibility was to design and implement quickly a system such as a graphic editor or a mail processing and filing system, and then extend it to have programming by example. Because of the short amount of time available initially (a summer), this seemed infeasible. There were almost no such pre-existing systems written in Smalltalk.

Fortunately, one such system was available, which satisfied all these desires nicely. David Smith and Steve Hayes, both of Xerox, had written a prototype of the Star system in Smalltalk-76. The prototype had been done several years before as an exercise in quick prototype development, and provided only the icon and desktop illusions of Star, but the code was well written and looked as if it would be easy to extend. I named this previously nameless prototype *SmallStar* (or sometimes, *WhiteDwarf*), and set to work adding more Star functionality and some way of creating programs by example.

#### 3.2 The Xerox Star 8010 Information System

The Xerox Star is a multi-user general-purpose office information system. Each user has his own processor, large-format bit-map display screen, keyboard (figure 3-1), and mouse. Users share one or more file servers, printers, and communications gateways. The user and server processors are all connected with an Ethernet local network. Users can edit text and graphics in documents, file documents, share them with other users, have them printed, send and receive documents to and from other users as electronic mail, do records processing, maintain data bases, fill out forms, create new forms, and write applications programs that process Star data objects. I will describe the features of Star that are most relevant to this research; there is much more I will not mention.

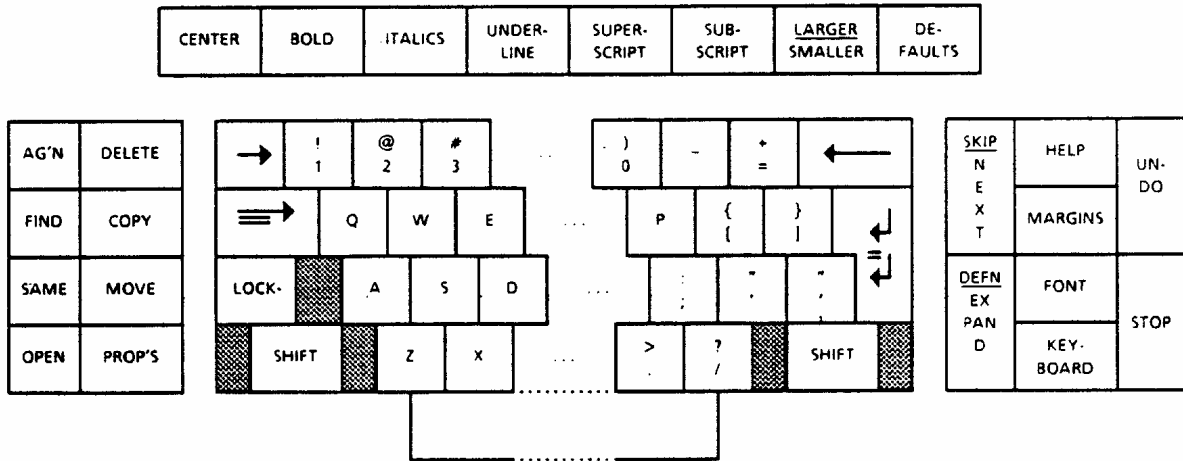


Figure 3-1. The Star keyboard.

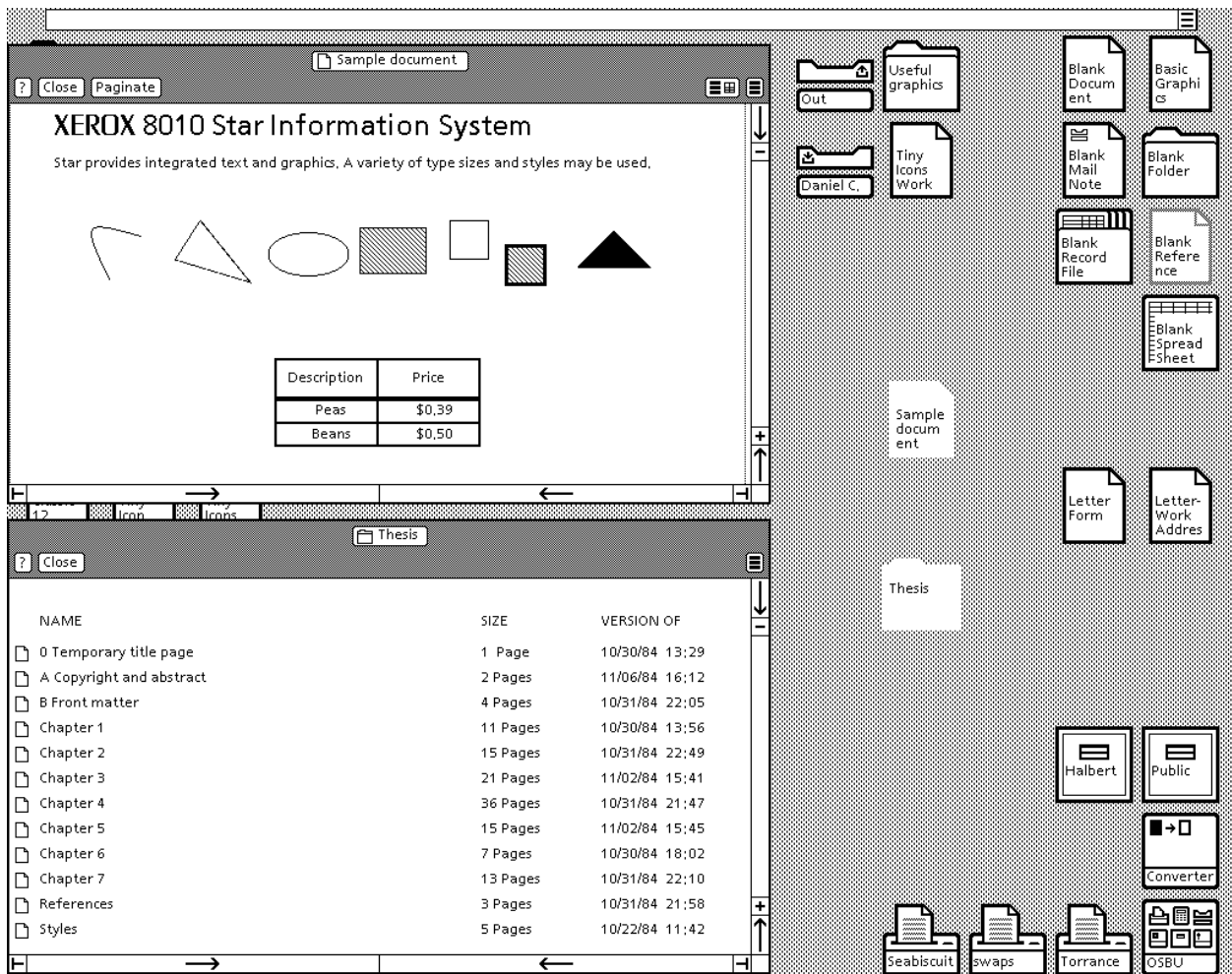


Figure 3-2. A Star desktop.

On his screen the user sees what is called a *desktop*. On the desktop are small graphic icons that represent and resemble *documents, folders, file drawers, printers, in- and out-boxes, records files*, and so on. (Refer to figure 3-2 throughout the following description.). Icons either represent places where data is kept, or are used to invoke some function. Every icon has a name, displayed inside the icon picture, which may be changed by the user through the *property sheet* mechanism (see below). Two or more icons may have the same name even if their contents differ; icon names need not be unique.

Documents contain text interspersed with rectangular objects called *frames*. Frames may contain text, graphics, other frames, or may be tables. Frames also have names which are visible to the user through property sheets. The names of all the frames inside a particular document must be unique. *Fields* are named blank spaces embedded in text and bounded by special characters that are meant to be filled in by the user (for instance, they may make up the variable parts of a form letter). The slots in tables are also fields. The **NEXT** key on the keyboard allows the user to step through the fields in a document successively.

Folders are *containers*: they hold other icons, such as documents and other folders. File drawers are also containers, like folders, but represent directories of files stored on a remote file server; they can be shared among several users.

Icons such as printers and mailboxes represent function. There are also less metaphorical function icons, such as a *converter* icon which performs conversions between non-Star and Star document formats.

Along the top of the desktop is a one-line message area which displays errors and informative messages. To the right of the message area is a small box containing three heavy horizontal lines: this symbol indicates a *pop-up menu*, and this particular one is the *desktop pop-up menu*. If the mouse is moved into the pop-up menu box and a mouse button is pushed, a menu of commands appears; the user can invoke a particular command by pointing to it with the mouse and releasing the mouse button.

Star is a *direct manipulation* system [Shneiderman]. The user manipulates Star objects directly, pointing at them with the mouse. To this end, the mouse has two buttons: **Point** and **Adjust**. **Point** is used for *selecting* objects. **Adjust** is used to adjust what is included in the selection; for instance, it is used to lengthen or shorten a text selection, or to select more or fewer icons.

The user can select an icon by moving the cursor to the icon and pressing **Point**; the icon changes from white to black to indicate it has been selected. Icons on the desktop show their names, but not their contents. To show an icon's contents, the user selects the icon and presses the **OPEN** key on the keyboard. A large window is created on the screen. For documents, the window shows a page of the text and graphics in the document; for folders, the window shows a pictorial list of the icons in the folder; similarly, for other kinds of icons, the window shows their contents. The user can then edit text, remove data and icons, and in general manipulate the icon's contents. The icons in a folder or file drawer window may be selected just like icons on the desktop.

Near the top of a window is a *header* giving the name of the icon, and listing several commands that may be invoked using the mouse (**Close** to close the window, **?** for help, etc.). More than one icon may be opened at a time. Icon windows do not overlap each other, but they do cover up icons. When a window is opened or closed, the adjacent windows are adjusted in size as necessary.

Four keyboard commands can be applied to nearly all Star objects. These universal commands are **MOVE**, **COPY**, **DELETE**, and **PROPERTIES**. If an icon is selected and the **MOVE** key is pressed, the icon is lifted off the desktop into the mouse cursor. The cursor can be moved somewhere else, and then the icon is set down by pushing a mouse button. (There is an invisible grid for icons on the desktop, so one icon will not overlap another.) **COPY** is similar to **MOVE**, except that the original icon is not disturbed; instead,

a copy of it is put into the cursor. **DELETE** merely removes the selected icon. Similar obvious and analogous actions occur when the universal commands are applied to objects other than icons, such as text, graphics objects, tables, and so forth.

If an icon is moved or copied onto a container icon, instead of onto a blank spot on the desktop, it is put into that container. Moving an icon to the out-box mails it. Moving an icon to a printer causes it to be printed.

The **PROPERTIES** command causes another kind of window called a *property sheet* to appear. A property sheet describes salient characteristics of the object that is selected, and allows some or all of those characteristics to be changed. Figure 3-3 shows a selected paragraph and the paragraph property sheet for it. For icons, the properties listed include the name of the icon, the last time it was changed, and so on. For text, the properties include such characteristics as the font used, and whether the characters are boldface or italicized.

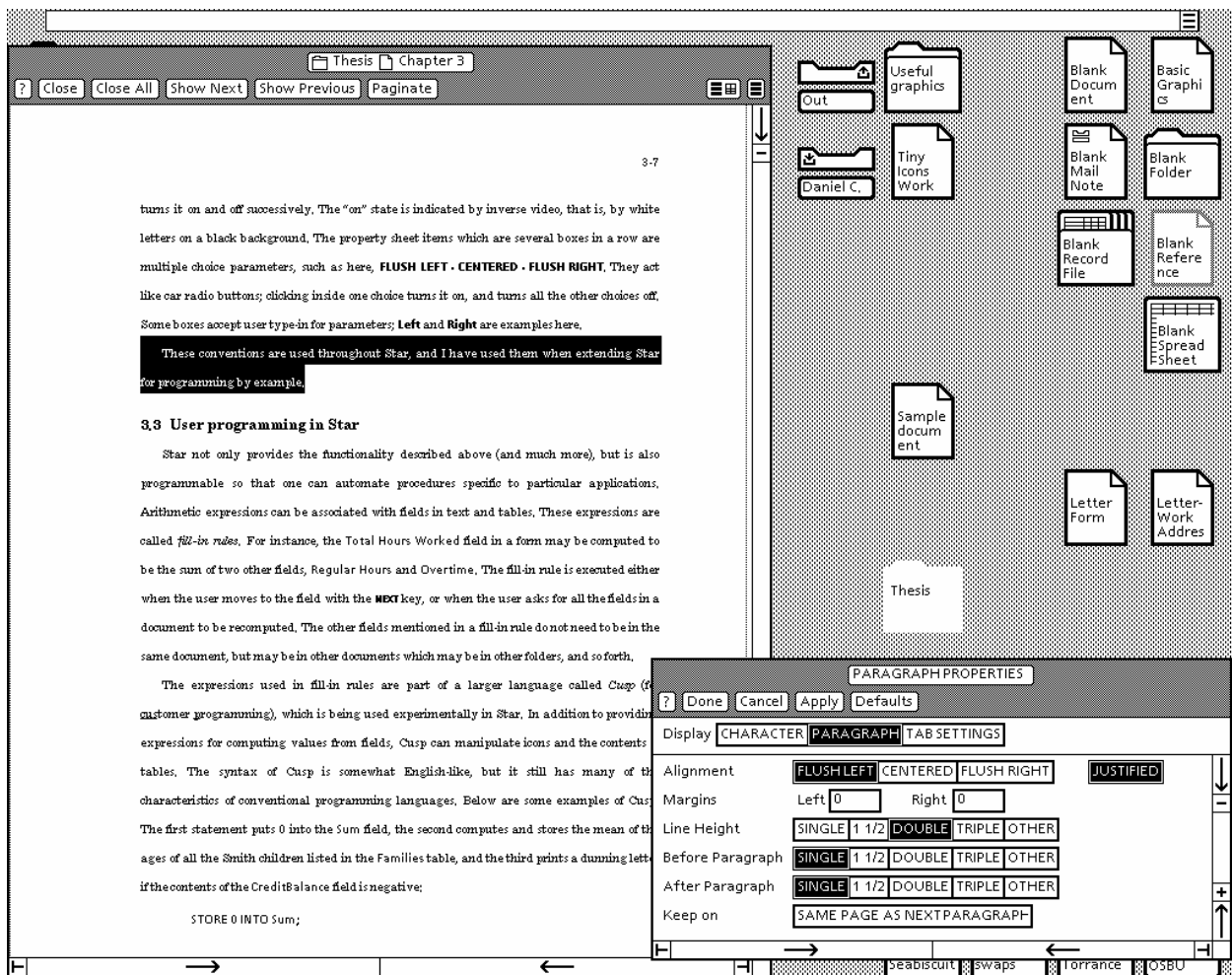


Figure 3-3. A selected paragraph and its property sheet.

The property sheet shown illustrates a number of Star user interface conventions. Text strings not in boxes are labels, (examples here are **Alignment** and **Keep on**), or information the user may not change. Single boxes such as the items **JUSTIFIED** and **SAME PAGE AS NEXT PARAGRAPH** are Boolean parameters. Clicking inside such a box with the mouse button **Point** turns it on and off successively. The



“on” state is indicated by inverse video, that is, by white letters on a black background. The property sheet items which are several boxes in a row are multiple choice parameters, such as here, **FLUSH LEFT - CENTERED - FLUSH RIGHT**. They act like car radio buttons; clicking inside one choice turns it on, and turns all the other choices off. Some boxes accept user type-in for parameters; **Left** and **Right** are examples here.

These conventions are used throughout Star, and I have used them when extending Star for programming by example.

### 3.3 User programming in Star

Star not only provides the functionality described above (and much more), but is also programmable so that one can automate procedures specific to particular applications. Arithmetic expressions can be associated with fields in text and tables. These expressions are called *fill-in rules*. For instance, the **Total Hours Worked** field in a form may be computed to be the sum of two other fields, **Regular Hours** and **Overtime**. The fill-in rule is executed either when the user moves to the field with the **NEXT** key, or when the user asks for all the fields in a document to be recomputed. The other fields mentioned in a fill-in rule do not need to be in the same document, but may be in other documents which may be in other folders, and so forth.

The expressions used in fill-in rules are part of a larger language called *Cusp* (for customer programming), which is being used experimentally in Star. In addition to providing expressions for computing values from fields, Cusp can manipulate icons and the contents of tables. The syntax of Cusp is somewhat English-like, but it still has many of the characteristics of conventional programming languages. Below are some examples of Cusp. The first statement puts 0 into the **Sum** field, the second computes and stores the mean of the ages of all the Smith children listed in the **Families** table, and the third prints a dunning letter if the contents of the **CreditBalance** field is negative:

```
STORE 0 INTO Sum;
STORE MEAN[Families[Row CALL IT Parent
  WITH Parent.LastName = "Smith"].Children.Age INTO
  AverageSmithChild;
IF CreditBalance < 0 THEN
  MOVE THE Document WHOSE NAME IS "PleasePay" TO THE Printer
  WHOSE NAME is "Gutenberg";
```

Cusp programs exist either in fill-in rules, or in special frames called *Cusp buttons*. If the user pushes **Point** while over a Cusp button, the code in the button is executed.

### 3.4 SmallStar

As mentioned, this research in programming by example was not done in the actual Star system, but instead in a simulation of Star called SmallStar. SmallStar presents the user with a desktop that looks very similar to a real Star desktop (refer to figure 3-4). SmallStar windows and property sheets are simplified versions of the corresponding real Star objects. Several kinds of icons in SmallStar are weak simulations of the corresponding real Star icons. For instance, SmallStar's in- and out-boxes and printers do not actually perform any function, and file drawers in SmallStar hold icons stored on the local disk, not on remote file servers.

SmallStar has a few icons which Star does not have. There is a *trash can* icon, into which deleted documents are automatically placed. The user can retrieve mistakenly deleted documents from the trash

can, and empty it out from time to time. There is a simple desk calculator which provides the standard four arithmetic functions, plus relational and boolean operations which are used to help write programs. (This use of the calculator will be explained in more detail in section 5.6.) Finally, there are program icons used to write programs by example, and program buttons, which are special kinds of frames that exist inside documents, and are otherwise exactly like program icons. Program buttons are similar in concept to Cusp buttons in real Star.

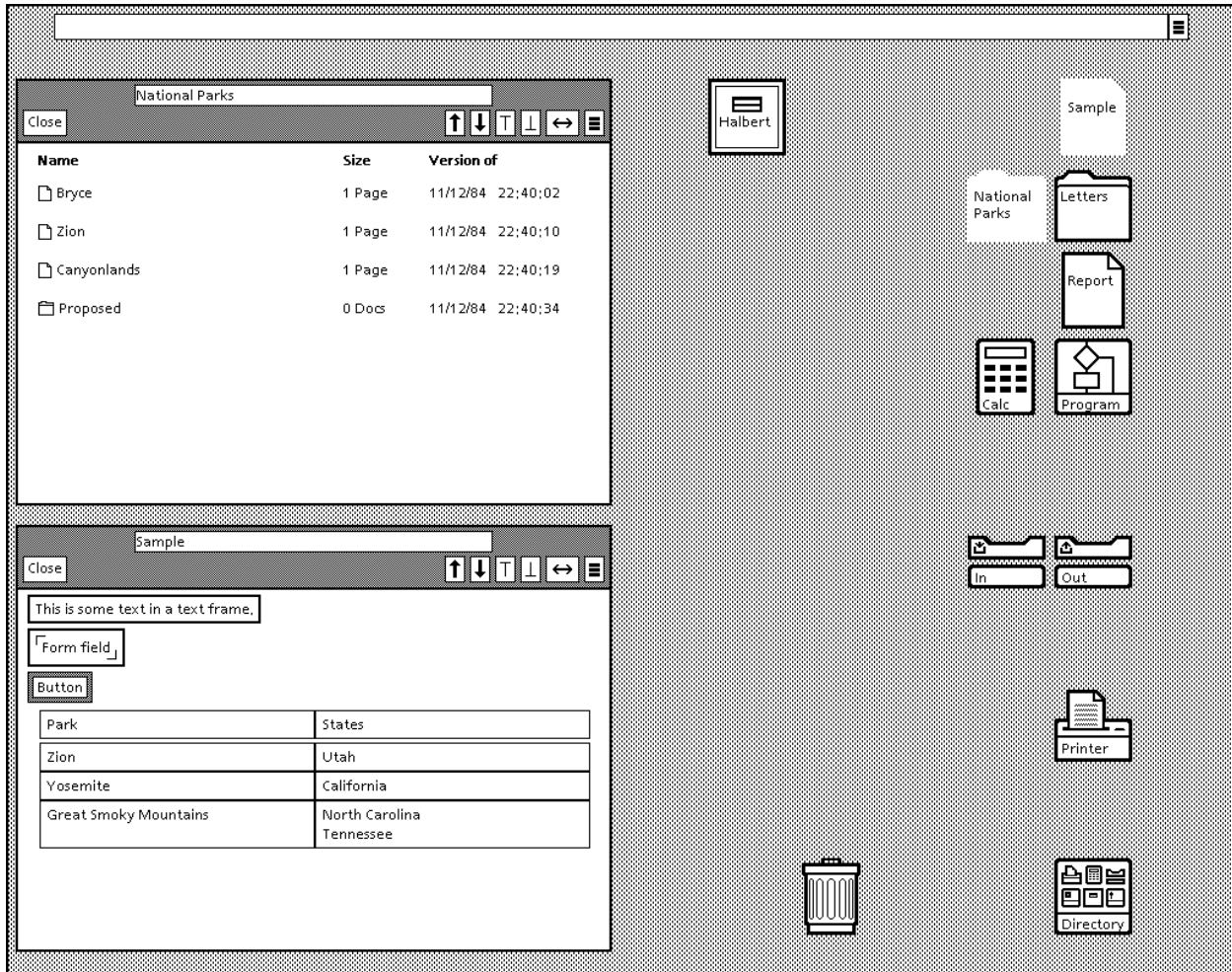


Figure 3-4. A SmallStar desktop.

Closed SmallStar icons have pop-up menus, a feature Star does not have. If the user selects an icon using the **Point** mouse button and continues holding down the button for a short while, a pop-up menu will appear that contains commands specific to that icon.

The contents of documents are simpler in SmallStar also. In Star, the text of a document may contain embedded text and graphics frames, fields, and tables. In SmallStar, there is no top-level text; instead, a document consists of a sequence of frames. SmallStar does not have graphics, but it does have text frames, *form fields* (text frames that each contain a field), and tables. Frames and tables have names as in Star.

Despite these deficiencies, the SmallStar system is still interesting in its own right. It can manipulate icons in exactly the same way as Star, and its documents provide enough functionality to be able to do interesting text editing and forms creation.

### 3.5 Basic programming by example in SmallStar

To write a program by example in SmallStar, the user asks the system to record his actions. He can do so either by opening a program icon or button and pressing the **Start Recording** command in its window, or by using the **Start Recording** command found in the pop-up menu available from a closed program icon or button. (Figure 3-5 shows a program icon window containing a very simple program.) The user then goes through the actions of the task he wants to be recorded. As he does them, the system creates a program by making a transcript of his actions. When the user is done, he invokes the **Stop Recording** command. He may then run the program using the **Run** command. Both **Run** and **Stop Recording** are available in the pop-up menu or in the window.

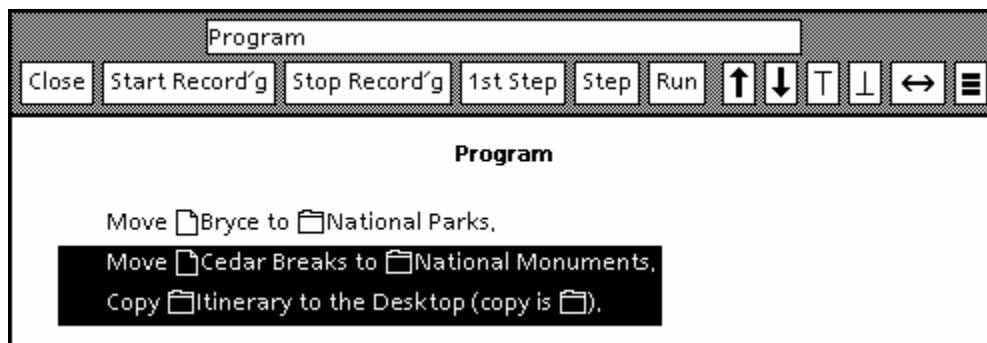


Figure 3-5. A program window containing a simple program.

SmallStar not only records a program internally, but also presents the program to the user in a human-readable form. The program is visible in the program icon when it is open. If the program icon is open while the user records a program, each user action is added to the program incrementally and displayed. When the user does a **Stop Recording**, the program is cleaned up slightly, and adjacent, related commands are merged if possible.

The language in which the transcript is presented is English-like, and is designed to be as readable as possible. It was initially inspired by Cusp, but has since strayed. It tries to be immediately understandable, imposing little non-natural-language syntax or semantics on the user. The user is not expected to write programs in the language. He cannot type in a program using it, and he cannot edit the individual characters in a program statement. Small pictographs are used to indicate icons and other Star objects; the pictographs used are listed in figure 3-6.

The user can edit the program in simple ways. He can record new steps in the program. If there is already a program in the program icon, starting to record will add steps after the last step in the program. Single statements and ranges of statements may be selected (in figure 3-5, the last two steps of the program are selected). If the user first selects a statement in the program and then presses **Start Recording**, statements will be added before the selected statement. Statements may also be selected and deleted. I will discuss more sophisticated editing, such as operand editing and adding control structure, in later chapters.

Deleting statements, or adding new statements to a program may cause problems. Each statement in a program produces a change in the SmallStar world. So the action performed by one statement may be

impossible to do unless a previous statement has been executed to set up the correct state of the world. If the user tries to insert statements in the middle of a program without setting the world to some meaningful state, then he may be unable to record the actions he wants to, or the actions he records may become impossible to execute when the program is run from the beginning. Similarly, if the user deletes statements in the program, then statements after his deletions may become meaningless.

















	document		text selection
	folder		table
	file drawer		column of a table
	printer		row of a table
	in- and out-box		field at the intersection of a row and a column
	calculator		text frame
	program		form field
			labelled parameter (e.g., Result or Entry box in the calculator)
			program button

Figure 3-6. Pictographs used in program display.

The surest way to prevent such problems is to require the user to run his program up to the point of the desired edit by single-stepping or by indicating a breakpoint, and then have him re-record all the rest of the program. The latter requirement is annoying, especially if a change is made near the beginning of the program. Both requirements are unnecessary if the changes made do not interact with the rest of the program. The system could analyze the user's changes to see whether they conflict with the rest of the program, but this analysis might be difficult. It is easier to let the user make the edits he desires without such restrictions. Instead, the system can provide helpful error messages and make it easy for the user to run the program up to the point where the editing will take place.

For this reason, and as an aid to program debugging, program icons also allow programs to be executed one statement at a time. The **First Step** command starts the program over again from the beginning, and executes the first statement. The **Step** command is used to execute each subsequent statement. An arrow in the left margin of the program points to the next statement to be executed.

If the program encounters an error while running, it stops, displays an error message, and indicates the statement that caused the error.

This recording mechanism is designed to have a low overhead cost for the user. It should be as easy as possible for the user to decide to record a temporary program to do some repetitive task. The user should feel free to write a program whenever it suits him to do so, and so it should not appear to be a chore. Hence, for instance, the recording commands are available in the closed program icon pop-up menu, so it is not even necessary to open a program icon to start recording a program.

### 3.6 Choice of recording mechanism

The recording mechanism just described must be explicitly invoked by the user. If the user, for instance, decides that something he has already done was worth recording, he will have to repeat his actions for the benefit of the recording mechanism. To avoid this problem, one could record programs

using a *history* mechanism. If the system had such a mechanism, it would keep track of everything the user does. At any time the user could review the history and select parts of it to be replayed or stored as a program. The UNIX system command language processor *csh* [Joy] and the Interlisp system [Xerox] both have well-known and heavily used history mechanisms, though they are not used for creating programs by example.

I did not choose to use a history mechanism for several reasons. One is that the amount of history to be stored is potentially unbounded. The other is that the user must review this history to find the actions he wishes to make into a program. For systems in which the transcript is just a record of the commands the user typed and saw echoed, it is easy for the user to review the transcript, since it is just what he typed in and what the system typed out. But for systems like Star, in which there is no typescript of what the user did, the user would have to review his actions as translated into the transcript language. The user might confuse statements in the transcript, and indicate the wrong range of statements to be saved as a program. For both of these reasons I felt the simple **Start** and **Stop Recording** mechanism was clearer and more explicit, and chose it over a history mechanism.

### 3.7 Internal form of the recorded program

A program could be recorded merely by recording the user's literal actions: his keystrokes and mouse manipulations. This is certainly the simplest way of recording a program by example. But in many cases, a literal recording of the user's actions would be too concrete. For example, the program to move a document from the desktop into a folder could be represented internally as:

```
Move mouse cursor to point (430,540);
Press left mouse button;
Press MOVE key;
Move mouse cursor to point (300,300);
Press left mouse button;
```

This is unsatisfactory. Suppose, for instance, the user moved the document or folder to some other spot on the desktop and then tried to run the program again. The program would stop working because the mouse positions would no longer correspond to icons. In addition, if this kind of transcript were shown to the user, it would convey very little information about what the program does, except that a **MOVE** was taking place.


Clearly, the program should be recorded at some higher level of abstraction. In SmallStar, a program is represented internally as an n-ary tree. Each leaf in the tree is a user *command*, which consists of the name of an action, and a list of the descriptions of data objects the action uses (see chapter 4 for a thorough description of these descriptions). These actions in commands correspond closely to atomic user actions, though a single command may consist of several user operations. For example, the two-step actions of **MOVE** or **COPY** (picking up and then putting down an object) are recorded as two separate commands initially, and are displayed on two lines in the transcript. The two parts of the command are later merged. For instance, when the user selects the document Doc and then presses the **MOVE** key, the transcript shows:

```
Move  Doc to ...
```

After the user sets the icon down, the system adds:

```
... the Desktop.
```

After the user does a **Stop Recording**, the system merges these two lines into the single command:

Move  Doc to the Desktop.

User text type-in is also merged. The system may initially record the user typing “Th”, then “e”. This is merged into the single type-in “The”.

The program is recorded in tree form, though initially it is just a linear list of commands, because later editing of the program to add control structure is easily done if a tree form is used.

### 3.8 Sample program

It is difficult to convey the experience of using the program recording mechanism merely by describing it here. However, I will attempt to give the flavor of the system.

Suppose the user wanted to write a short program to copy the document called Meetings-Today, save the copy in the folder Meetings, and then print Meetings-Today. He would do the following sequence of keystrokes and mouse-button clicks:

**Start recording**

press **Point** over Meetings-Today

**COPY**

press **Point** over Meetings

press **Point** over Meetings-Today

**MOVE**

press **Point** over Printer

**Stop recording**

The program produced is shown here in figure 3-7. This entire program is recorded in a matter of a few seconds, in far less time than it would take to type it in.

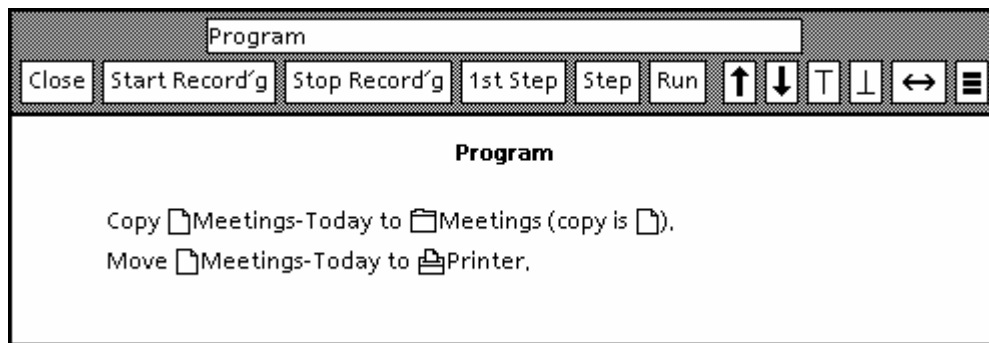


Figure 3-7. A sample program.

### 3.9 The SmallStar virtual machine

This section and the next one describe the internal structure and the history of the implementation of SmallStar. It is not necessary to understand the implementation details presented here in order to comprehend the user's view of SmallStar, but the details may be of interest to the reader. Some knowledge of Smalltalk will be helpful in understanding this section.

SmallStar uses the object-oriented programming paradigm offered by Smalltalk. SmallStar, for instance, has a Smalltalk class named **Icon**, which provides general behavior for icons, and subclasses for more specific types of icons and their behaviors. For example, there is a subclass of **Icon** named

**Container**, which in turn has subclasses **Folder** and **FileDrawer**. User actions are transformed into messages to be sent to instances of classes such as these. For instance, to select and move a document on the desktop, the user's **Point** mouse click is translated into a **SELECT** message sent to the icon, which is an instance of class **Document**. The user then presses the **MOVE** key, and **MOVE** is sent to the icon. The icon removes itself from the desktop and places itself in the cursor. When the user clicks down with **Point** somewhere on the desktop, the message **INSERTAT: pt**, with an argument **pt** corresponding to the location of the cursor on the desktop, is sent to the (sole) instance of the class **Desktop**, which handles all such desktop functionality, and the icon is then displayed on the desktop at the chosen location. To summarize, if **doc** is the document icon, and **desktop** is the desktop, and the user sets down the icon in the third row and third column of the desktop, then the Smalltalk code executed is:

```
doc SELECT.  
doc MOVE.  
desktop INSERTAT: 3@3.
```

Every single user action maps into a capitalized message name (a procedure name). Care has been taken to separate these procedures from the ones that actually do the work of displaying icons, moving them, and so forth. These capitalized procedures also are decoupled from the mechanics of the user interface, so that they are not concerned with keystrokes and mouse button clicks. Therefore, a complete transcript of the user's actions can be collected by saving these messages along with their arguments and their receivers.

The complete set of these message names that exists in SmallStar define, in essence, the SmallStar virtual machine. User actions are translated into virtual machine commands, but the commands can also be executed directly without going through the user interface. Thus, SmallStar can be operated programmatically.

### 3.10 Evolution of SmallStar

In this section I will describe the first implementation of programming by example in SmallStar, and then describe how it and SmallStar changed in subsequent versions.

The initial development of an programming-by-example mechanism for SmallStar proceeded very quickly. As discussed in the previous section, SmallStar had already been built with the idea of a Star virtual machine in mind. The problem was to capture the virtual machine commands. To do this, I made all virtual machine commands pass through a single locus of control in SmallStar. Commands were recorded in a linear list at this locus if recording was turned on. A few commands were not meant to be recorded (e.g. those that controlled program recording), and they were marked as such. The first version of SmallStar simulated essentially only the icon illusion of Star, so that the only programs one could write involved moving, copying, and deleting icons.

All programs were initially recorded in an invisible program icon. After the program was recorded, it could be copied into a newly created visible program icon. Program icons at that time looked like tape cassettes, because I was trying to stretch the idea of program recording to fit a tape recorder metaphor.

There was a one-to-one correspondence between each user action and each step in the transcript. The transcript language was vaguely English- and Cusp-like. As the user did each action, the transcript step it produced was echoed in a one-line area just below the regular desktop message area. The user could open a cassette icon after a program was recorded, and see the entire program transcript, though the transcript was not editable in any way. Figure 3-8 shows a simple program that copies the **Report** document on the desktop and then moves it into the **Memos** folder. Needless to say, this program is not terribly easy to read. The **variable table** lists every data object touched or created. Here, “variable” [3] was created as

the result of the copy. The To [1] and To [4] lines in the transcript are the setting-down of an icon after it has been moved or copied into the cursor. The motivation behind the variable table will be discussed in more detail in chapter 4.

```

variable table

[1] the Desktop
[2] Report (Document)
[3] Report (Document)
[4] Memos (Folder)

program

Select [2]
Copy [2] (into cursor as [3])
  To [1]
Move [3] (into cursor as [3])
  To [4]

```

Figure 3-8. Program display in the first version of SmallStar

After this initial attempt, finished in September 1980 [Halbert 1980b], the prototype was cleaned up and improved in various ways. The internal form of a program was changed from a simple linear list to an *n*-ary tree, so that control structure could be added to the recorded program (see chapter 5). An *ad hoc* parsing technique was used to combine the atomic actions of multi-part commands such as **MOVE** and **COPY** into a single recorded command. The display of the program was improved somewhat, so that now the transcript of the program mentioned above looked like figure 3-9. Variable table entries were now indexed by simple generated names (A, B, C, etc.), instead of by number (e.g. [1]). The work up to this point (spring 1981) was described in my Master's report [Halbert 1981].

<b>Data</b>		
<b>Name</b>	<b>Type</b>	<b>Value</b>
A	a Document	Report
B	a Document	Report
C	Folder	Memos

**Program**

```

Select A;
Define B to be (Copy A to the Desktop);
Copy B to C;

```

Figure 3-9. Program display in the second version of SmallStar.



At this time it was clear there was more work to be done on programming by example, but I felt that I was exhausting the limited possibilities of writing programs that just manipulated Star icons. So I added a number of other Star features to SmallStar, notably text editing, fields, tables, and property sheets. The goal was to be able to write a simple order-entry application by example. Such an application had already been implemented in Cusp on another prototype of Star, and I wanted to match that level of sophistication.

After these changes, the idea of a single locus of control in the implementation at which all recording could take place became impractical. The arguments supplied for each recorded command had to be processed in a certain way (to be discussed later) in order to be recorded, a task best done in the procedure where the command was executed, because of that procedure's knowledge of what it was doing. In addition, there was much less of a one-to-one correspondence between each user action and what was to be recorded. So the idea of a single locus of control was discarded in favor of doing the recording inside each procedure implementing a SmallStar virtual machine operation. This change, while necessary, was unfortunate, because I lost the ability to confine the knowledge of how to do recording to a single place in the code.

Initially, I included the selecting of data objects as one of the operations I recorded in the program transcript. **Select** statements can be seen in figures 3-8 and 3-9. However, I found that Select statements were always redundant to the operation of the program. One would see

```
...  
Select X.  
Move X to the desktop.  
...
```

and similar constructs. The data object being used was always mentioned twice: once by being selected, and then again in the operation that followed. This redundancy made programs longer but no clearer. If the object was not used in a subsequent operation, then the **Select** was useless anyway. In addition, changing the size of an existing selection caused extra **Select** operations to be recorded. This was particularly noticeable if the user was doing text selections. For instance, the user might select a single character, say "T", and then extend the selection to, say a whole word, "The". This would produce two **Select** statements, only the second of which was useful. My initial approach was to record all select operations, and discard superfluous ones when the user pressed **Stop Recording**, but I soon decided to omit selection operations altogether. Selecting an object does not change the state of the SmallStar world, and so discarding the **Select** operations had no effect on the proper operation of programs.

As I progressed through various versions of SmallStar, the Smalltalk-80 system being developed by the then Software Concepts Group at Xerox PARC was becoming stable. Since I was hitting a number of implementation limits in Smalltalk-76, I spent a few months converting the code to Smalltalk-80, gaining quite a bit of performance because of the conversion. In addition, a number of visual cosmetic changes were made to make SmallStar appear much more like the real Star system.

Most of the subsequent changes in SmallStar came about because of changes in my thinking of how to record and handle the operands used in a program written by example. This evolution of ideas is discussed in Chapter 4.

## Chapter 4

### Data descriptions

When a user creates a program by example, he does not necessarily tell the system why he chose the data he used in the program. His intentions may be concealed from the system. In this chapter I will discuss this problem, and present the idea of *data descriptions* as a way of allowing the user to specify his intentions. I will also give a history of the earlier concepts that lead up to the present idea.

#### 4.1 Choosing data in programs

Computer programs do computation. They take input, give output, and in between they compute things. If we look at the process of computation more closely, we can divide it into first getting the data on which the computation is to be done, and then doing the necessary computing.

When a computation is done with some data object, the programmer has decided to use that object for a certain reason. There are several possibilities:

- The object is a fixed quantity, to be used as a constant in the resulting program.
- The object is the result of some previous computation.
- The object is merely a placeholder; it represents a parameter to the program. Each time the program is run, some (possibly different) value will be read as input by the program, or will be supplied by a calling program.
- The object has been chosen from among other possibilities, based on certain characteristics it has, or based on its relationship to other data. The program looked for an object, in the right context, that matched some characteristics or that satisfied some constraints.

It is the last possibility that causes the most trouble in programming by example. In a program written in a conventional programming language, the programmer writes code to choose data. There are explicit computations in the program that do searches, fetch objects out of tables, and so forth.

However, in a program written by example, another possibility arises. As in a conventional program, the user could record an explicit computation to choose some data. But he could also select data manually, the same way he would if he were doing, by hand, the task he is trying to program. Instead of programming a search, he may do the search himself, by eye. Instead of indexing into a table, he may just pick the right element.

The problem is that this manual selection of data leaves parts of the program incompletely specified. The user has done some computation to choose the data, but the computation is all in his head, and has not been communicated to the system. Yet this seems natural to the user, since the most natural way to program a task by example is to repeat exactly what one would do to perform the task manually.

Suppose, for instance, the user of a file system interface wants to write a program to delete all his BASIC programs. If he were using a typescript system, such as one of a number of operating system executives, he would probably type something like

```
delete *.BAS
```

to the executive. Here there is no problem. The file system interface provides a pattern matching mechanism: the user has taken advantage of it in a natural way. On the other hand, consider a direct manipulation system, such as Star. The files might be listed separately, as for example:

A.BAS  
LETTER.TXT  
A1.BAS

While recording the example, the user would select A.BAS and A1.BAS, and then delete them. Here the user has performed a search in the list for the files he wants, and has never communicated to the system the rule he was using for conducting the search. The user's *intentions*, that is, why he chose the data he did, remain a mystery to the system.

#### 4.2 Finding out the user's intentions

There are several ways to find out the user's intentions. One is that the system can require the user to supply explicitly the computation used to select the data, just as in a conventional program. The user could do the computation step-by-step, or perhaps use some mechanism provided in the user interface of his system, such as a query or search mechanism. If the user does not explicitly perform a computation to select a data value, the system assumes the value is a constant in the program. For example, suppose a text editor provided a simple method of programming by example, and the user wanted to write a program to delete a line of text that matched some pattern and replace it with another. The user might program an explicit search for the line to be deleted, using a search command provided by the editor, delete the line, and then type in some constant text which would always be used when the program was run again. The user cannot omit the search command. If the user had not included the search in the program, the program would probably have deleted the line of text at the current position of the cursor, which is not what the user had in mind. And if the user had chosen the line to be deleted by looking for it by eye, rather than by using a search, and used explicit context-sensitive positioning commands (e.g., *go up 17 lines*), then the resulting program would also do the wrong thing.

Another method for determining the user's intentions is to use inductive inference techniques. The user does several examples, not just one. When recording a program, the system notes the differences between the examples, and draws conclusions about what the user meant. For instance, Ian Witten's desk calculator system uses a simple inference technique (see section 2.12). The user repeats the example several times. If a number changes from one example to the next, then it must be a parameter to the program. Suppose, for instance, the user enters

$$2 + 3 \div 4$$

as one example, and then enters

$$2 + 3 \div 7$$

as another. The system will construct

$$2 + 3 \div \underline{\text{variable}}$$

as the resulting program. When the program is run again, the system will prompt the user for the value of the variable.

This sort of inductive inference technique is at the simple end of a class of possibilities using artificial intelligence techniques. Witten's system makes inferences based on previous very similar examples. But one can imagine AI systems with more knowledge of the user's needs, the problem domain, and the programming system. Such systems could make more sophisticated guesses about the user's intentions.

For instance, in the example given previously, the user is trying to delete all his BASIC programs. In the example program he records, he deletes A.BAS and A1.BAS. If the system is trying to detect a pattern in this example, it can choose certain filename patterns which match the names of the files that were selected. Possible reasonable patterns might be A\*.BAS, A\*.\*, and \*.BAS. Based on the user's previous actions, and some knowledge about what is reasonable, the system would choose one of these. Of course, it might choose incorrectly.

This example illustrates the basic problem with such AI techniques. The system can guess what the user means, but it can never be sure. It cannot read the user's mind. The richer the problem domain, the more likely it is there will be a variety of reasonable alternatives. Nevertheless, such techniques are quite useful if they end up saving the user time and bother in writing his program.

There is a middle ground. If the system is unsure as to what the user's intentions are, it can ask the user about one of several alternatives, or offer some way of letting the user correct the system if it guesses wrong. This is the method I have chosen to use in SmallStar.

### **4.3 Choose the data, or choose the container?**

The methods of specifying what data to use mentioned in the previous section are complicated by the need to talk about the choice of the container of the data object, as well as the choice of the object itself. A certain object may be contained in another, which is inside another object, and so forth. At each point in the path to the final object, a choice is made about how to select the object at that point.

Sometimes the division between object and container is sharp, but sometimes it is not; it depends on the system. In conventional programming languages, there is a clear distinction between a name and its binding, that is, between a variable and its contents. The user interfaces of some systems that are programmable by example provide data objects which clearly correspond to such programming language variables. For instance, the Pygmalion system (see section 2.3) has *icons* which can contain numbers (not to be confused with Star icons). If we take the number 42 from one Pygmalion icon and copy it into another icon, it is reasonable to assume we are concerned with whatever value is in the first icon, not with the number 42 specifically. The choice of 42 is not important, but the choice of the icon that contains the number is. Therefore, the programming-by-example mechanism in Pygmalion concentrates on data selection methods for icons, not their contents.

On the other hand, in the Star system, if we write a program to print a document that is in a folder, it is not clear whether we want the program always to print that particular document, or want it to print whatever document happens to be in that folder. Since either choice is plausible, the system must provided the ability to specify the choice of both the document and the folder.

### **4.4 SmallStar data descriptions**

To let the user specify his intentions about why he chose the data he did, I have added the concept of *data description* to programming by example. SmallStar keeps an explicit *description* for each and every data object used in a program recorded by example. These descriptions are built as the program is being recorded. They describe the data objects to be used during the later execution of the program by giving the characteristics of the desired object. Or alternately, one could say they supply *constraints* that narrow down the possible choices for an object. For instance, a data description for a document might say that the desired icon is the newest document on the desktop with the name "Parts Contract". A description for a text string might specify a string that matches the pattern "\*Canyon\*", or might ask for the first 5 characters in a text frame. There are no constants in a program. *Every* object used has a corresponding data description.

Each step in a program consists of a command and some operands. These operands are pointers to data descriptions (see figure 4-1). As a program is being recorded, every newly referenced object is assigned a reasonable default description by the system. During recording, or afterwards, the user can go back and modify these default descriptions if they turn out to be incorrect.

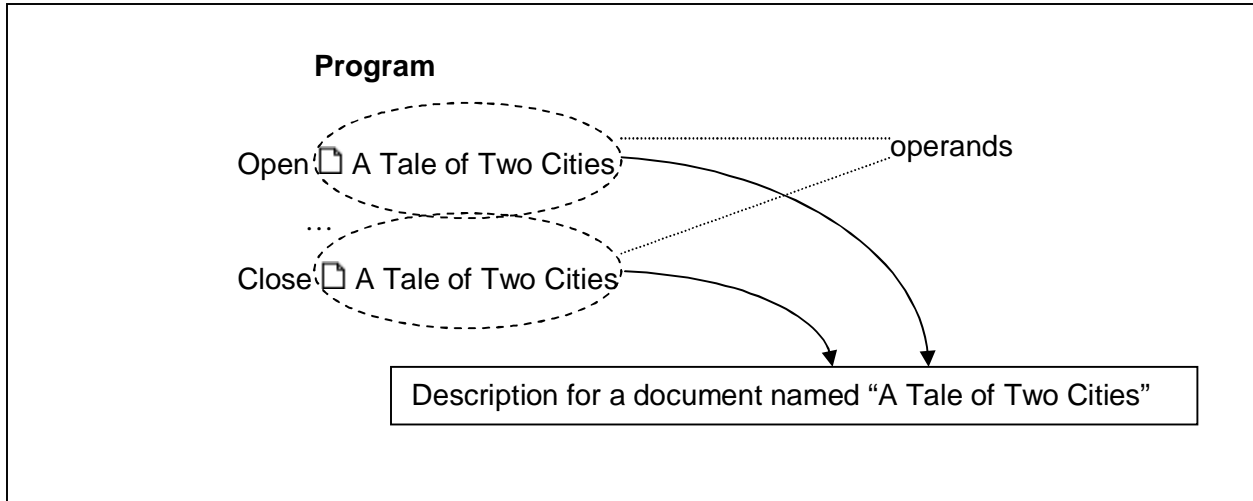


Figure 4-1. The relation between operands and descriptions

In addition to this specification of how to choose an object, a data description contains a slot for temporarily binding an object to the description once the object has been chosen. When a program is asked to run, it first clears all the binding slots in its data descriptions. As each command in the program is executed, the data descriptions corresponding to its operands are used to find the relevant data objects. If a description has not yet been used in the program and therefore does not yet have a value bound to it, a search is made for a data object matching the given specification. If more than one object matches the description, one is chosen arbitrarily. (Alternatively, the system could ask the user to choose one.) Once an object is found, it is temporarily bound to its description for the duration of the execution of the program. Later uses of that description will shortcut the search process and use the object already found.

(In reality, a description generates a set of all the objects that match the description. Normally, only the first element of the set is used, and so the user is not aware the entire set has been generated. The entire set is used when the program does set iteration; this is discussed in section 5.4.)

Thus in SmallStar I have mixed some of the techniques described in the previous section. The system tries to guess the user's intentions by using reasonable default data descriptions, but it makes its guesses in a fairly superficial way. Much of the time, it does not expect to guess correctly. Instead, the system makes it very easy for the user to modify the guesses the system has made, by editing the data descriptions. So programming by example has become a mix of program recording and program editing.

Descriptions are created for every object referenced by the user during program recording, whether the objects are referenced directly or indirectly. For instance, if the user selects a document in a folder, the system builds a description for the folder as well as the document. Every item on the path to an object is included, and the descriptions on a path are linked. Thus, in this example, the description for the document is linked back to the one for the folder.

More precisely, the description-building mechanism works in the following way. User commands in the program transcript consist of the name of the action, and the operands it uses. The operands are

pointers to descriptions. While a program is being recorded, the system keeps a table whose keys are the actual objects used in the program, and whose entries are the corresponding descriptions. When the system records a user command, it looks up every object used in the command in this table. The system uses an existing description if one has already been created for the object. Otherwise, it creates a new description and enters it in the table. The system then does a recursion and looks at the objects that contain that object, if there are any, looks them up in the table, and continues similarly until it reaches a top-level object.

For instance, if the user deletes some text in a field in a table, the system will record a **DELETE**, find or create a description for the text string, then recursively find or create descriptions for the field the text is in, for the row and the column the field is in, for the table the row and column are in, for the document the table is in, and so forth until it reaches something that is contained in nothing but the desktop.

Illustrated in figure 4-2 is a simple program that moves the document named **Treaty** out of the **Negotiations** folder onto the desktop. Notice that command operands may be selected. Each operand corresponds to a description; that is, each operand represents the *use* of a description. In the figure, one use of the description for **Negotiations** is selected. The other uses of the description have been outlined in gray to show the user where else this description is being used. The way a description is displayed indicates its current state. In this case, the description specifies a search for a folder whose name is “**Negotiations**”. This description is *not* a constant, which will always be bound to exactly the folder **Negotiations** used in the example. For instance, if the user deletes the **Negotiations** used in the example and creates a new folder **Negotiations**, then the new one will be used.



Figure 4-2. Program to move a particular document out of a folder.

I mentioned previously that if an object is used more than once in a program, new descriptions are not created for each use; instead, the same description is used several times. This is a fundamental heuristic, and is a matter of convenience. If the user refers to an object more than once, the system assumes the uses are connected in intent; that is, it does not assume that another use of the same object is accidental. If the user edits the description for an operand in one program step, he will have conveniently changed the description for the uses of that operand in all steps.

This heuristic works very well. Rarely does a user refer to the same object for a completely different purpose. However, exceptions can occur, and so there is a way to defeat the heuristic. Suppose, for instance, a user writes a program to print two documents on two printers, and happens to use the same printer because one of the printers was broken at the time the example was recorded. If the user selects one use of the printer in the program, both uses will be highlighted. The user, instead, needs two different descriptions, and so he uses the **Make Description Different** command in the program window pop-up menu, which makes a copy of the description for the selected operand and changes the operand to use the new description.

Conversely, two different operands may be merged to use the same description. The user selects an operand, presses **COPY**, and then drops what is put in the cursor onto another operand. The second operand is changed to point to the same description as that of the first operand.

#### 4.5 Content and editing of data descriptions

A data description specifies a particular way of looking for a data object. To see and edit a data description, the user selects a use of the description in the program and presses the **PROPERTIES** key. He may do this while a program is being recorded, since each step in a program is added to the transcript as it is recorded, or he may choose to do this after recording is finished. If a description is examined during programming, recording is suspended for the duration.

In either case, pressing the **PROPERTIES** key opens up a window that resembles a property sheet. The user may then look at the detailed description, and may change it to suit his needs.

A number of different search methods are provided for each kind of data object. These methods are available as a menu of choices in the description sheets. In addition to the menu of choices, the necessary parameters and sub-choices for each choice are displayed. Those that are inapplicable for the current choice are grayed out. Here, for instance, in figure 4-3 is the description sheet for the document Treaty used in the example program above. All description sheets for documents resemble this one.

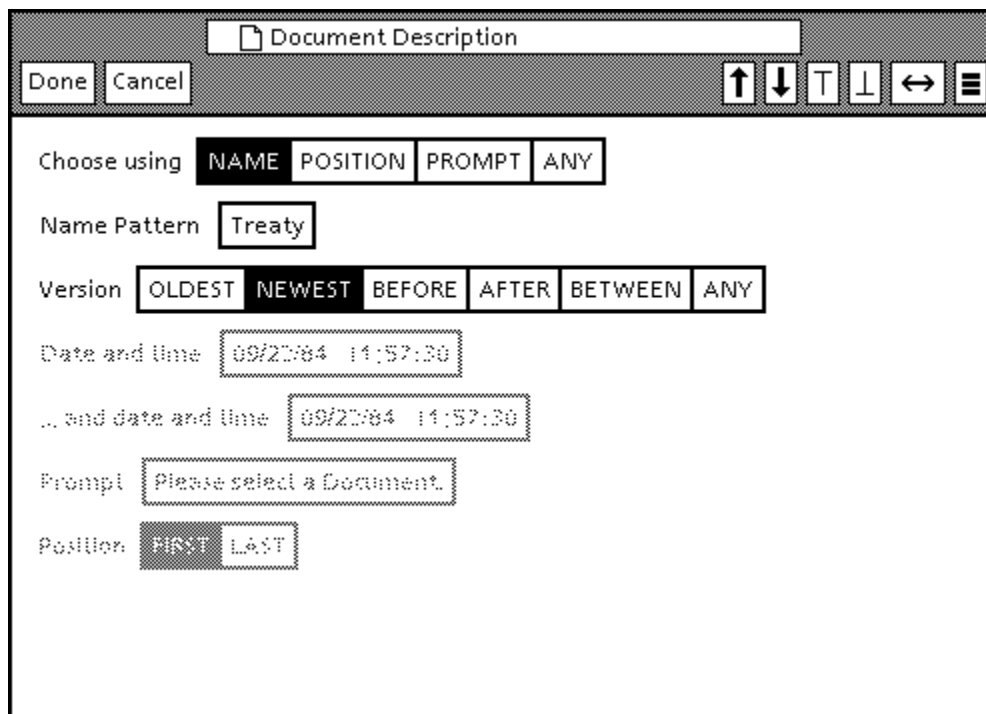


Figure 4-3. A document description sheet.

The main description choices given are listed in the **Choose using** item. The choices are **NAME**, **POSITION**, **PROMPT**, and **ANY**.

If **NAME** is chosen, the system looks for a document whose name matches the pattern given in the **Name Pattern** parameter. The pattern may be a simple text string, or a pattern such as **Chapter\***, which matches any document whose name begins with “Chapter”. Since icon names need not be unique, the

system must have some reason to choose between two icons that match the same name pattern, and so the **Version** choice further narrows the document choice based on the date and time when the document contents were last changed. Some of the items in the **Version** choice require supplying one or two dates and times, and so the parameters below **Version** are used for this purpose.

If **POSITION** is chosen, all the items become gray except for **Position** at the bottom. The system looks for a document in the position specified (**FIRST** or **LAST**). Position on the desktop is not well defined, since the desktop grid is not labelled with axes, so this choice is displayed only when looking for a document in a container, such as a folder.

**PROMPT** requests the system to ask the user when the program is run to choose a document. The systems prints out the request given in the **Prompt** item; a default prompt is supplied, which the user may change.

**ANY** specifies that the user does not care which item is picked. For picking a single document, this choice is not very useful, but in section 5.4, I will show how this choice is used to indicate set iteration.

Of course, the choices given here do not give all possible ways of choosing a document. Many possibilities are missing. For instance, the **POSITION** choice only includes **FIRST** and **LAST**, and does not include the more general possibility of the  $n$ -th document. There is also no way to choose a document based on any characteristic of its contents.

Nevertheless, the choices given were selected to cover many of the common reasons for choosing a document. My intent is not to cover all possible cases, but merely a large fraction of reasonable possibilities. I will discuss this limitation further in section 4.11.

As an example of the use of description sheets, consider the simple program given in figure 4-2. The program as recorded looks for the document *Treaty* in the *Negotiations* folder, and moves it onto the desktop. Suppose the user wanted to change this program so that it moved the first document in *Negotiations* to the desktop. The user would select the use of the description for *Treaty* in the program. He would then bring up the description sheet for *Treaty* (see figure 4-3 above), and change the **Choose using** item from **NAME** to **POSITION**. He then wants to set the Position item to **FIRST**, but this is already its current state, so he need do nothing. After he presses **Done** in the description sheet header, the description is changed, and the program transcript is updated to show this change. The resulting program is shown in figure 4-4.



Figure 4-4. Program to move the first document out of a folder.

Description sheets with similar structures and choices exist for all the other kinds of data objects the user may choose. Some description sheets are straightforward. Figure 4-5 shows the description sheet for



a text frame. A text frame may be chosen, among other ways, by giving a pattern for its name or by giving a pattern for its contents (here, choose the frame containing the text string “snake”).

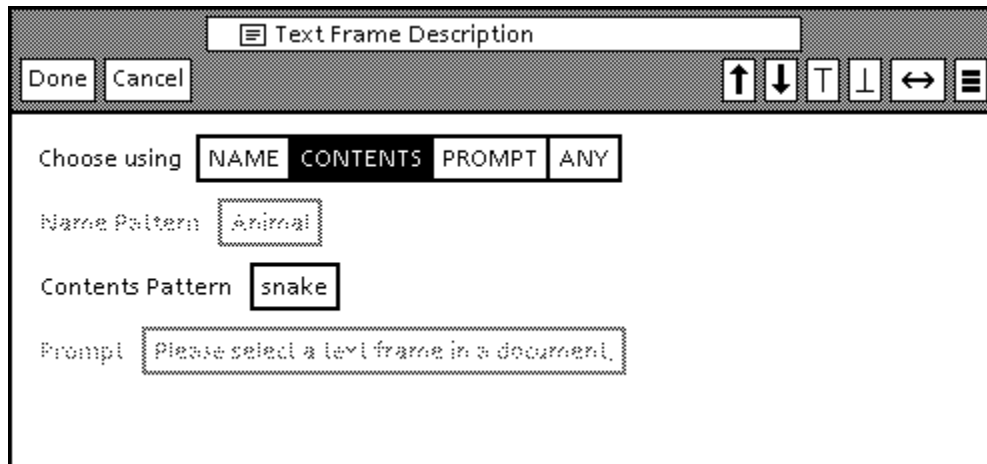


Figure 4-5. A text frame description sheet.

Other description sheets provide more sophisticated choices. So far, we have seen only choice and literal parameters. But there are cases where the user would like to give a description as a parameter to another description.

Consider the following example (figure 4-6). The user is designing an order-entry application. He has a table of parts information (**Parts**) in the **Parts Data** document, and a customer order blank (**Order Blank**) in the document **Order Form**. By telephone, a customer orders 5 handsaws, but does not know the part number for a handsaw. The user wishes to write a program to look up the part number, given that its name is already in **Order Blank**. He records a program to copy the part number from the row for “handsaw” in **Parts** to **Order Blank**. This program, as recorded, is shown in figure 4-7. But the system chooses an incorrect default description for the row chosen in **Parts**; it guesses that the user wanted to choose the row based on its position (**row #2**). This will not work. If the user tried to look up the part number for hawks, the program would again return the part number for handsaws.

Instead, the user wants the program to choose the row in **Parts** whose **Name** field is the same as the **Name** field in **Order Blank**. That is, he wants the description of the row in **Parts** to require that the text in its **Name** field match the text in the **Name** field in **Order Blank**. To do this, he can insert a description for the **Name** field in **Order Blank** into the row description sheet.

The incorrect row description is selected in figure 4-7. If the user brings up the sheet for this description, he sees what is illustrated in figure 4-8. He does not want to choose the row based on **ROW POSITION**; he wants instead to choose it based on its **CONTENTS OF FIELDS**. After indicating this choice, he must give patterns for each field in the row. For each field in the row, the description sheet offers the choice of using a literal pattern (**Text**) or a description (**Description**). The first and last patterns do not matter, and so the user chooses **Text** from them and enters “\*” (meaning “match anything”) in the parameters for those fields. For the second field, he cannot use **Text** “handsaw”, for that will cause the description to produce the same incorrect result. So, instead, he turns on the boolean parameter **DESCRIPTION**, which grays the **Text** parameter, and makes the **Description** parameter accessible. Inside the **Description** parameter is a non-functional placeholder text string description, indicated by

≡ -fill this in-

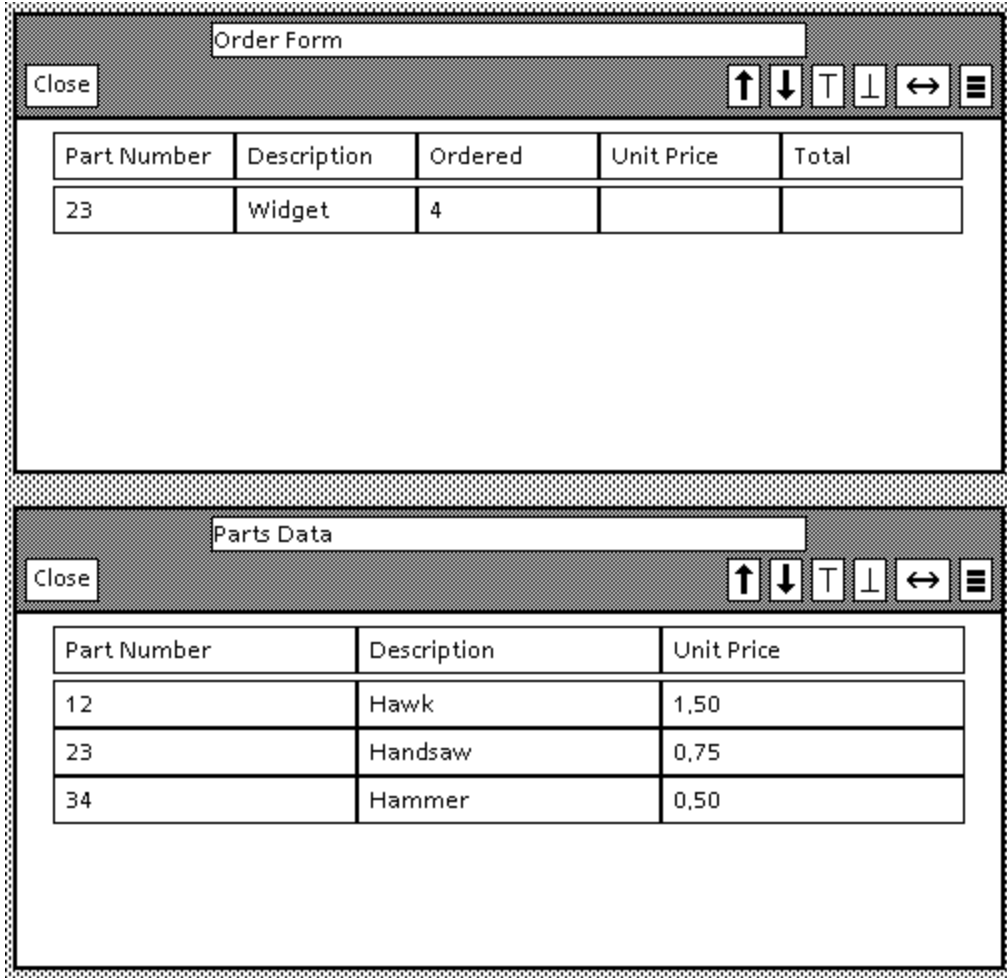


Figure 4-6. Documents for an order entry application.

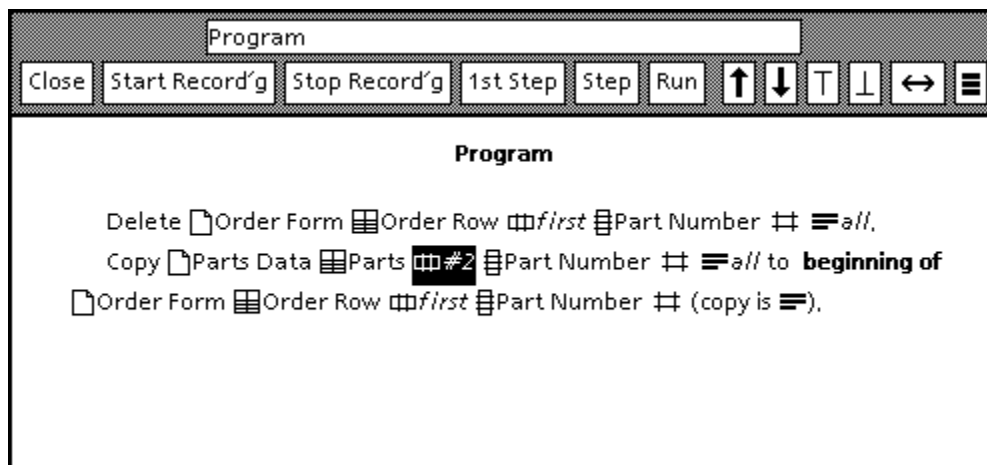


Figure 4-7. The original part number program, showing the incorrect row description.

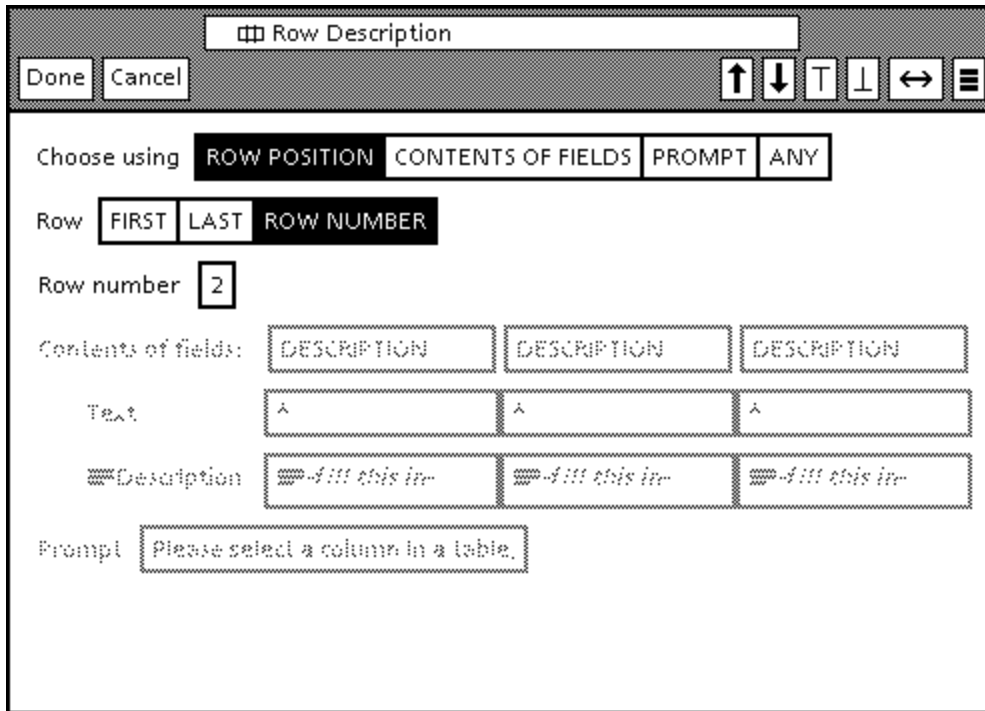


Figure 4-8. The description sheet for the incorrect row description.

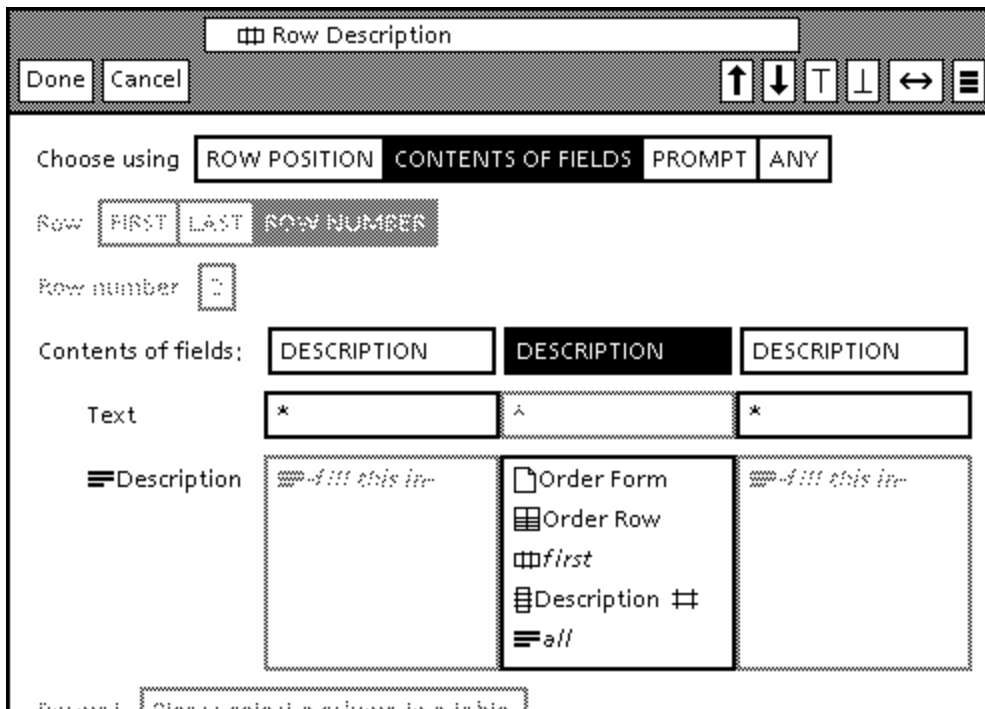


Figure 4-9. The corrected row description sheet.

The user wants to replace this description with one referring to the Name field in Order Blank. To do this he selects the object for which he wants to create a description (in this case, the text "handsaw" in

the order blank), and invokes the command **Create Description**, which is available in the pop-up menu of either the program window or the row description sheet. The command creates a complete description for the text selection, including everything on the path to the text, and puts this description in the mouse cursor. The user then drops this description into the field in the row description sheet. The resulting description sheet is shown in figure 4-9. The complete path for the description of the text selection is shown in the appropriate field parameter. If any part of the path is wrong, the user can select it and edit it using its description sheet.


Notice that while it is possible to create data descriptions while not recording a program, the initial creation of a description must be done with a real Star object, as in program recording. Creation of data descriptions is done by example whether or not a program is being recorded.

#### 4.6 References to data in early versions of SmallStar

In the very first implementation of programming by example in the SmallStar prototype, programs contained no data descriptions or variables, only constants. Programs consisted of actions and operands, but the operands were direct pointers to the data objects used in the program. The original implementation provided programming by example only for icon manipulations, so the operands were always icons. Each time a program was run, it did exactly the same thing. In fact, programs held on to their operands so tenaciously that many things could be done to the operand icons and yet the program would still run. For instance, an icon used by a program could be deleted, but since the program still held a pointer to the object, and since Smalltalk uses a reference-counting scheme for storage reclamation, the object would continue to exist, and would reappear if moved or copied. Similarly, renaming an icon would have no effect on its use in a program.

I quickly discovered this scheme did not work; it failed if the recorded program used a newly created copy of an object. Every **COPY** operation creates a new object, and so each time a program that contains a **COPY** is run, it will create a different new object. However, if the program refers to the copy, it will refer to the original copy created during the recording of the example, and not to the desired new copy.

Consider a program which includes the statement:

Copy  Clone to the desktop.

This statement creates a copy of the document **Clone**. The new copy of **Clone** also has the name **Clone**, but in order to distinguish it, let us call it **Clone<sub>2</sub>**, and call the original **Clone<sub>1</sub>**. Suppose that later in the program, the contents of **Clone<sub>2</sub>** are edited by the user, and eventually the document is printed.

Now consider what happens when the program is run. Yet another copy of **Clone<sub>1</sub>** is created, which we will call **Clone<sub>3</sub>**. However, the program will edit and print **Clone<sub>2</sub>**, not **Clone<sub>3</sub>**, rather than correctly using the result of the **COPY** operation, since the program contains a hard pointer to **Clone<sub>2</sub>**.

Always using hard pointers to refer to objects is therefore incorrect. At the very least, some kind of slots to hold newly created objects must be used. A newly created object would be put in such a slot, and subsequent references to the created object should refer to the slot, so that they can refer to the created object. Such a slot is, of course, a variable, in the conventional programming language sense.

This technique of using a simple variable was used in the next version of the prototype. In order to make all references to data objects uniform, every operand used in a program statement, whether a constant or a newly copied object, was an index into a *variable table*. (See section 3.10, figures 3-8 and 3-9. Figure 3-8 is reproduced here as figure 4-10 for convenience.) These indexes into the variable table were initially just numbers ([1], [2], etc.), but later simple names (A, B, etc.) were used to improve the readability of the program. The entries in the variable table were still hard pointers to objects, but the indirection provided by the variable table allowed programs containing **COPY** operations to work correctly. Slots in the variable table were filled in with pointers to objects as the program was being recorded. Usually these pointers were never again disturbed, but if a **COPY** was done, the old value in the appropriate slot in the variable table would be replaced with a pointer to the newly created object. So except for copied objects, the “variables” in the variable table were just constants.

Data		
Name	Type	Value
A	a Document	Report
B	a Document	Report
C	Folder	Memos

**Program**

```
Select A;
Define B to be (Copy A to the Desktop);
Copy B to C;
```

Figure 4-10. Program display in the second version of SmallStar.

The creation of the variable table naturally spawned the reuse heuristic described in section 4.4. The variable table was displayed to the user, and for clarity and conciseness it did not contain an entry for every use of every object. Instead, each object had one and only one entry. New entries were added to the table as the program was being recorded only if an object not previously referenced was used as an operand. Since neither a program nor its variable table could be edited, using this heuristic actually had no effect on whether the program accurately reflected the user's intent or not; it merely made the program more readable.

Though the variable table enabled **COPY** to work correctly, the data objects used in a program were still constants. There was no way for the program to operate on data other than that used in the original example. To eliminate this restriction, I started working on ways of *generalizing* programs. This topic is discussed in the next section.

#### 4.7 An earlier concept: generalization

In this section I will discuss the idea of program *generalization*. I attached great importance to the term initially, but have since deemphasized its use. However, it was the study of this notion that led to the current idea of data descriptions. The evolution of the idea of generalization is covered in this and the

next section. An earlier discussion of generalization and its use in SmallStar can be found in my Master's report [Halbert 1981].

Despite the existence of variable tables described in the previous section, programs were still constant. They always performed the same operations on the same data. It was clear that some way of *generalizing* a program so that it would operate on data other than that used in the example was needed.

For most other programming-by-example systems, generalizing the example program means specifying which data are constants, and which are parameters to the program. Some programming-by-example systems provide very weak forms of generalization. For instance, in simple “keystroke macro” systems found in various text editors, there is no parameter to the program other than the current position of the cursor. Since most text editor commands are position-relative, moving the cursor will cause the program to operate on different data.

Other systems provide program parameters, but ask the user to specify them explicitly. In the Tinker system (see section 2.5), the user is creating Lisp functions, and specifies the arguments to each function as he starts to define the function. He then gives the arguments sample values and proceeds to create the function by example. In the EP system (section 2.7), the user gives names for the program parameters, and then gives the parameters the values they will have in the example program. After the example is given, the system substitutes the parameters for the sample values in the typescript.

In Ian Witten's desk-calculator system (see sections 2.12 and 4.2), program parameters are not noted explicitly by the user; instead, an inference scheme detects differences between program examples and determines which values in the program must be parameters. In the Pygmalion system (section 2.3), programs are inherently parameterized because the data objects held by the program are references to value containers and not values themselves (explained in section 4.3).

In all these systems, the only kind of program generalization provided is parameterization. When I initially thought about programming by example, this seemed to me to be inadequate, since it did not provide a way for the program to choose data based on some constraints. I envisioned a spectrum of generalization, with constants at one end, and variables at the other (figure 4-11). Most programming-by-example systems handled only the two ends of the spectrum. But I wanted to use the middle of the spectrum as well. If the user used a certain data object in a program example, but did not mean for it to be a constant, there were nevertheless some characteristics about that object that caused the user to choose it. When program was run again and a different object was used, the new object used would be chosen because it was similar to the example object in some respect. I said that the object used in the example had to be “generalized” in some way.

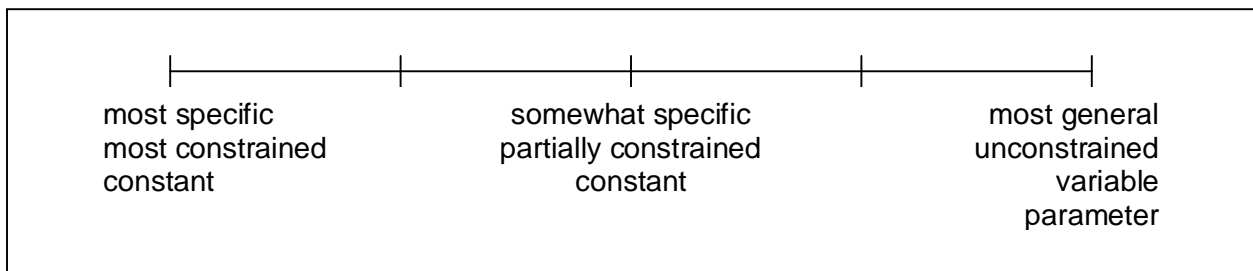


Figure 4-11. The spectrum of generalization.

The problem here is to discover in what way the example object is similar to the desired object. In other words, the system wants to ask the user, “Why did you choose the data you did?” I set out to provide a way for the user to answer this question, and implemented a generalization facility in SmallStar.

The early SmallStar prototypes simulated only the icon manipulation functionality of Star. Data generalization was done as the program was being recorded. The variable table scheme described previously continued to be used. In the very first implementation of generalization, the user selected the icon he wished to generalize with a special, third mouse button. (Xerox research mice have three buttons; Star product mice have only two.) The mouse button brought up a pop-up menu with several choices on it. The choices were:

**same location**  
**same row**  
**same name**  
**ask for a similar object**

The user chose one of these kinds of generalization, causing the system to record a statement which would find an icon with the same characteristics when the program was run again. For instance, if the user selected the document **Moonstone** with the **same name** generalization, the system would record the statement:

Define B to the Document whose name is "Moonstone".

assuming that B was the next unused variable name in the variable table. Subsequent references to Moonstone would use variable B.

The generalization **same location** added a **Define** statement that looked for an icon at the same place on the desktop, assuming that the icon was on the desktop. **Same row** looked for an icon in a container at the same position as the selected icon (for instance, the third icon in a folder). **Ask for a similar object** prompted the user at run-time for an icon of the same type (in the case of **Moonstone**, for a Document).

**Define** statements were inserted at the point in the program where an icon was first generalized. Previous references to the icon therefore would not use the generalized value. For instance, in the example above, previous references to **Moonstone** would be treated as constant references. So it was important that the user generalize an icon before he used it.

Subsequent changes to this mechanism were slight. The use of an extra mouse button was discarded, since it did not fit with Star. Instead, the user would select the icon to be generalized in the normal way, using the **Point** mouse button, and then choose a kind of generalization from the desktop pop-up menu. **Same location** and **same row** were merged into a single choice, **same place**, which chose desktop location or position in a container as appropriate.

This generalization mechanism is the one described in my Master's report [Halbert 1981]. The most serious problem that occurred while using this mechanism was that the user frequently forgot to do a generalization when necessary. An icon choice had to be generalized before the icon was used, but many users (myself included) often forgot to do so.

There are several ways of solving this problem. One is to remind the user, whenever he uses an icon, that a generalization is possible. A generalization menu would appear after using an icon. The user would choose a generalization if desired; if not, he would either indicate he did not want to generalize, or else he would ignore the menu entirely and it would quietly go away.

Another method is to remove the restriction that an icon is generalized in the program only after a **Define** statement has been added using the generalization menu. Instead, the **Define** statement would be moved up in the program, to a point before the first use of the icon. Generalizing an icon choice late in the program, after one or more uses of the icon, would therefore still work. But this method has the

disadvantage that the generalization may not work for previous uses of the icon. Suppose the user selected an icon, moved it from one folder to another, and then generalized it based on its position. Did the user mean the position in the first or the second folder?

A third method is to provide a completely *ex post facto* generalization method. All generalizations would be specified after the program was recorded. The user would edit the uses of an icon in a listing of the program, and indicate what generalizations he desired. For consistency with Star, the obvious way to do this editing would be through a property-sheet-like mechanism. This method also has the advantage that the user does not need to consider what generalizations are necessary while the program is being recorded. Instead, he can concentrate on the task he is trying to program.

Besides this difficulty of when to generalize, it was also clear that this version of SmallStar did not provide enough different kinds of generalizations. Choices more complicated than the few listed in the generalization menu were needed. In addition, some choices really required parameters, such as one allowing the user to provide his own prompt for the generalization **ask for a similar object**, instead of being forced to use the prompt the system provided. I also had plans for extending SmallStar to simulate more Star functionality than simple icon manipulation; the possible generalizations for text selections, rows and columns in tables, and so forth could not all be handled using such simple generalization menus.

These desires to improve the generalization mechanism and extend the functionality of SmallStar eventually led to the transformation of the generalization mechanism into the present data description mechanism. This evolution is described in the next two sections.

#### 4.8 Defining generalization

Though I used the idea of generalization extensively in the versions of SmallStar that led up to my Master's report, I was bothered by being unable to come up with a satisfactory explanation of what the word "generalization" meant. This definition remained elusive; I had no theory of generalization.

The best definition seemed to be that a generalization was any run-time binding of data to a variable. The most general form of generalization was parameterization, where the value was passed in to the program. The other kinds of generalization were more constrained, and represent searches in various spaces for data that matched given characteristics.

I hoped to find parallels between programs written conventionally and programs created by example with generalizations. But only parameterization had a parallel with conventional programming. There were no special constructs in conventional programming that corresponded to other kinds of generalization. If a programmer had written a conventional program to do what was implied by a SmallStar generalization, he would have included explicit code to do the requisite lookup or search. For instance, if a programmer were to write conventional code to do the **same row** generalization, looking for, say, the third icon in the **Contracts** folder, he might write `B := Contracts[3]`. If he wanted to do same name to find a document named **Remodeling** in **Contracts**, he might write a loop that did a linear search for **Remodeling** (shown here in some mythical programming language):

```
R := nil;
for each Document D in Contracts do
  if D.name = "Remodeling" then
    R := D; exit loop;
  end if;
end for each;
```



These kinds of examples are not called “generalizations” when they are written in conventional programming languages. In fact, they have no special name; they are just ways of getting at data; the word “generalization” seems inappropriate. Because of this lack of distinctiveness, it was unclear that there were clearly defined program constructs that could be called generalizations. The `Define` statements used for SmallStar generalizations replace pieces of code like the ones given here. But they are more a convenience than anything else; they are just constructs that add to the power of the language, allowing various kinds of searching and indexing.

In retrospect, this confusion arose because I was using generalization for two different purposes. The generalization mechanism provided program parameterization, using **ask for a similar object**. But it also provided a descriptive way to specify data searches. It was a mistake to call these searches “generalizations”. A program can become generalized by substituting data searches for data constants, but the searches themselves are not generalizations.

#### 4.9 Generalization becomes data description

This concern for defining exactly what “generalization” meant was overshadowed by another issue. The early versions of SmallStar described in the previous section still used hard pointers to refer to constants in programs. This was inappropriate for a programming-by-example mechanism in the real Star system. Hard pointers to Star data objects are sometimes available for referring to objects during a single session, but they become invalid after a session with Star is finished. Furthermore, using hard pointers makes the transfer of programs between users more difficult. A program sent by one user to another would have to carry its constants with it, since one user cannot refer to objects on another user's desktop. Finally, the concept of a hard pointer is alien to the Star user. Users choose objects by name or by position. Two icons of the same type and with the same name are distinguished by their contents and the date and time when they were last changed.

So it seemed more appropriate to address all Star objects by some more indirect method, such as name or position. This naturally led to the idea of using the generalization mechanism which already existed to address all objects referenced in a program, and not just those that the user explicitly chose to generalize. The ways of generalizing an object now become ways of describing it. But rather than putting explicit `Define`-like constructs into the program transcript, each description could be associated with an entry in the variable table.

Since such a description is an imprecise reference to an object, the description must include the complete path needed to address the object. For instance, the system must be able to distinguish between the document `Chapter 4` on the desktop and the document `Chapter 4` in the `Thesis` folder.

Every time the user refers to an object not previously used while recording a program, the system needs to create a description for that object. The system could ask the user for a description, or it could make some reasonable guess. I chose the latter alternative, for the same reasons I wanted an *ex post facto* generalization mechanism. The user may not know during program recording exactly why he chose an object, and in any case asking him about every object he touches would cause annoying interruptions during program recording, when he is concentrating on the algorithm and not the choice of data.

When the program is run again, the description is used to find the object. The description is not used until the object is needed. A plausible alternative would be to find all the data at the start of program execution, but this does not work in some cases. The desired object may not exist yet, or may not be in the right place. For instance, the description for a document might look for the first document in some folder. But the desired document may not be the first until some preceding document is deleted or moved out.

All this, of course, is the data description mechanism currently in SmallStar, described in section 4.4. Once the decision to use this sort of mechanism was made, the basic idea of data description did not change. But the presentation of the mechanism to the user did change; its evolution is discussed in the next section.

What happened to generalization? It is still there, somewhat disguised. When a program is recorded, all references to data are generalized immediately into descriptions. The generalization is done implicitly by the system, instead of explicitly by the user. But since every object is represented by a description, one can no longer point to the program and say that this datum is generalized, while that one is not. This distinction has been lost.

For a long time after adopting the idea of data descriptions, I continued to refer to descriptions as generalizations, and started to use the terminology of “data description” only recently. “Generalization” was confusing, since one data reference was not more or less general than another, but I stuck to the term until it became clear that it did not describe well what was happening.

Since program parameterization is provided by data descriptions (using **PROMPT**), I am using the data description mechanism for parameterization as well as description. I used the “generalization” mechanism for the same two purposes (as I discussed at the end of section 4.8). But the mix is a natural one, and is not confusing to the user, though it does make describing the semantics of data descriptions somewhat more difficult.

#### **4.10 Evolution of the program display**

The initial presentation of programs after data descriptions were implemented was quite similar to the display presentation used for programs with generalizations. A program icon showed a variable table, which was renamed the data list, and a program transcript. Figure 4-12 shows a one-line program, which moves the document **Treaty** from the **Negotiations** folder to the **Signed Treaties** folder. The **Negotiations** folder was already open when this program was recorded.

The data list was a list of all the data objects used in a program whether directly or indirectly. Each entry in the data list corresponded to a description, and was given a generated name, such as **F2** for a folder or **Tb1** for a table. The name of an entry stayed constant even though its description might change. Each entry displayed two aspects of itself. One was the path needed to reach it, expressed in terms of entry names. For instance, in the example, document **D1** can be found inside folder **F1**. The other aspect shown was the current description for the data list entry. In the example, the description for **D1** specifies a search in the folder **F1** for a document named “**Treaty**”. The program transcript used the data list names for operands, as was done in the previous versions of SmallStar when the data list was a variable table.

Data List	
Path	Description
📁 F1	📁 Negotiations
📁 F1 📄 D1	📄 Treaty
📁 F2	📁 Signed Treaties

**Program**

Move 📄 D1 to 📁 F2.

Figure 4-12. Program display using a data list.

The data list for some programs has more entries than would have been in the variable table for the same program, because the data list includes objects that have been referenced only indirectly, as part of the path to an object referenced directly. The variable table for the same program would not have included objects referenced indirectly. For instance, in the example, folder F1 is in the data list, even though it is not directly used as an operand in the **Move** statement.

The length of the data list quickly became a problem. For instance, figure 4-13 shows the size of the data list for a single-statement program that deletes all the text in a field in the table **Chinaware**, in row 1, column **Dinner Plates**. The path to the text to be deleted is long, and so the data list contains a large number of entries. For programs that refer to several other objects with equally long paths, the data list becomes horrendously long.

In an effort to suppress some of this display, the next technique I tried was to make items in the data list selectively visible. By default, the data list showed only the entries that corresponded to objects used directly in the program. For the example given here, only the data list entry for **T1** would be shown. By using various commands in the program window pop-up menu, the user could ask that the entire data list be displayed, or could select a particular entry in the data list, and ask that only the entries for the path leading up to the selected entry be displayed. For instance, if the user selected **R1**, and asked for the path entries to be displayed, the entries for **D1**, **Tb1** and **R1** would be shown.

But this technique made programs even harder to read. Figure 4-14 shows another simple program. It is impossible to tell what is being deleted from where without asking for more entries in the data list to be displayed.

Readability also suffered because of the generated names **T1**, **D1**, and so forth. These names meant nothing to the user, and only served as cross-references to the entries in the data list.

Because of these problems, I merged the descriptions given in the data list into the program transcript itself, discarding the generated names, and removing the data list display, since it was now redundant. There are no longer any variable names in a program. In conventional programming languages, variable names are used because they allow the same object to be referred to in multiple places. Sharing the descriptions of objects and displaying the descriptions in multiple places substitutes for using names.

After this change, program display reached its current format, shown in the figures in sections 4.4 and 4.5. There are still some problems with program display, but they will be discussed in the next section.

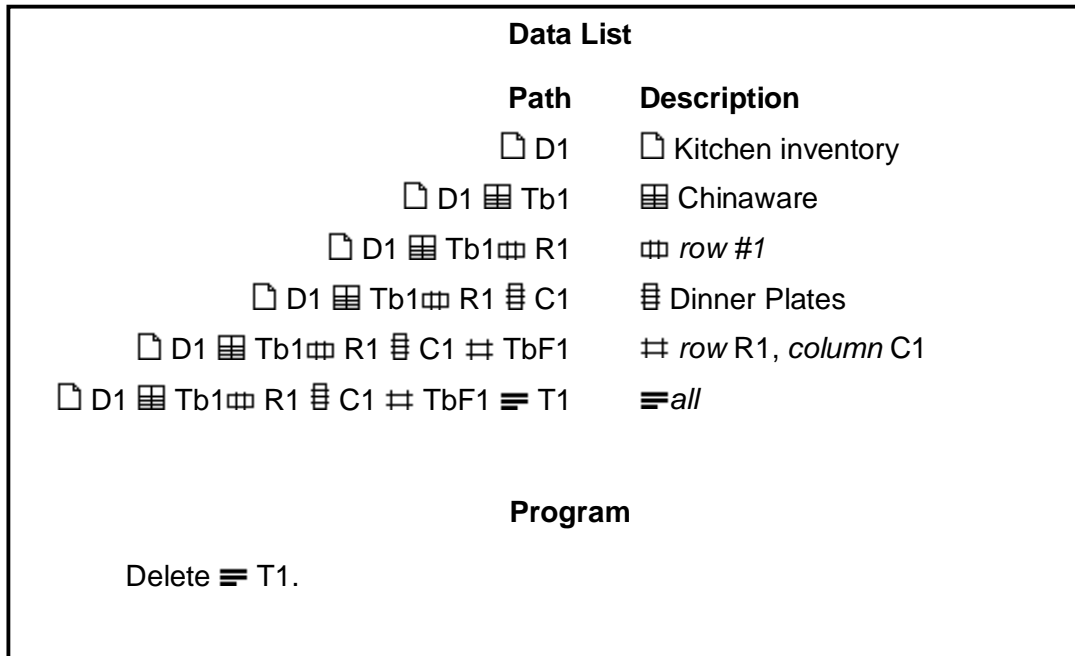


Figure 4-13. Program to delete all the text in a field in a table, displayed using a data list.

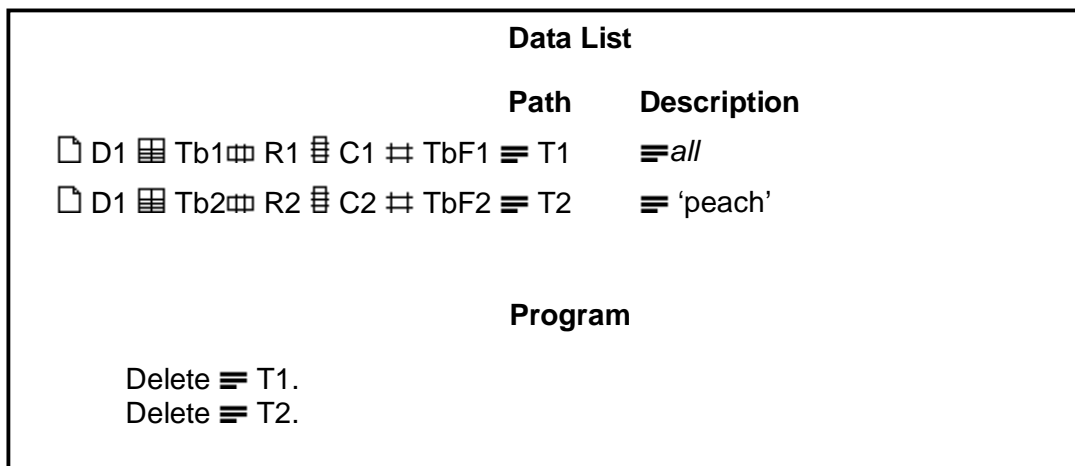


Figure 4-14. An obscure program.

#### 4.11 Problems with SmallStar data descriptions

Some problems still remain with the use of data descriptions in SmallStar. Some of these are peculiar to Star or SmallStar, and do not represent problems with data description in general, while others are problems that could appear in other systems that used data descriptions with programming by example.

Long paths to objects pose a readability problem in SmallStar. Figure 4-14 above already hinted at the problem by showing long paths in the data list. Consider the example in figure 4-15, for instance, which shows the text in a text frame being copied into a field in a table. The casual observer's first reaction to this example is to suggest that some of the detail that is shown be suppressed. But this poses other

problems. Suppressing the display of any of the path components removes information the reader of the program needs to have in order to identify the operands. The most extreme suppression of detail is suggested by figure 4-14, and it indicates what difficulty this can cause. In addition, all the descriptions in a path are of relatively equal importance, since it is plausible that the user will want to edit any one of them in order to fix an incorrect default description.

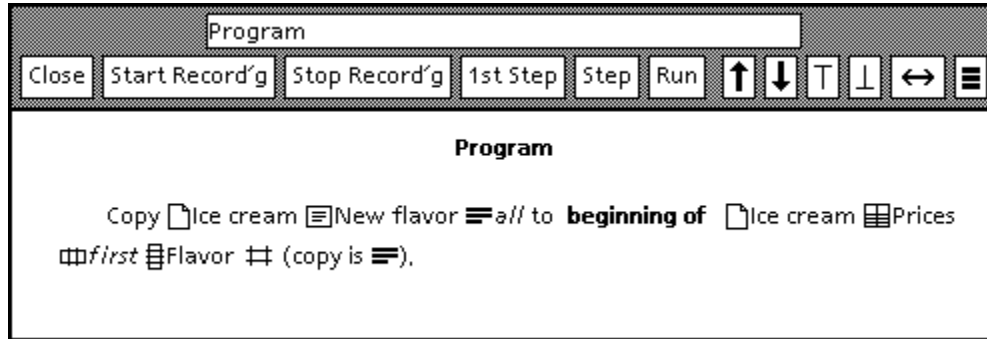


Figure 4-15. A program with long description paths.

Some more attractive layout of the operands in the program statement would help. As an experiment, I tried putting each description in the path on a separate line, nicely indented. Program readability improved, but the resulting program display used up far too much screen space. Some compromise program display format, which used up less screen space but still separated the path components clearly would probably be successful.

The problem of long paths is inherent in Star. Objects can be highly structured and nested. The path to an object deep in a structure is long, and the number of ways to describe how to get there is large. The direct manipulation nature of Star allows the user to circumvent these descriptions by pointing directly to objects. But the user's intentions in pointing to an object reflect choices from many alternatives.

The implementation of the heuristic which determines the sharing of descriptions in a program (described in section 4.4) poses another, different sort of problem. The heuristic is that all references to the same data object in a program will share the same description. The implementation does this by keeping a table of objects already referenced and their corresponding descriptions, and looking for the operands of newly recorded program statements in the table. (I will call this table the *object-description table*, for convenience.) If the object is already in the object-description table, then the description that has already been created is used. However, this way of implementing the heuristic requires that hard pointers to objects be saved, though only for as long as the program is being recorded.

Currently the object-description table is discarded after the user stops recording. A problem arises if the user adds new steps to the program. New descriptions for the operands in the new steps will be created, even if descriptions have previously been created for those operands. Since the old object-description pairs are no longer in the table, the system cannot find the old descriptions. The user must therefore manually make the new operands refer to the old descriptions after recording the new program steps.

For example, suppose the records a program which moves a document out of a folder. Later, he adds a step to the program which moves a different document out of the same folder. The program will have two different descriptions for the same folder, since the two parts of the program were recorded at different times. If the user wants the operands in the two steps to refer to the same description, he must select one of them, press **COPY**, and drop the result onto the other operand.

Various plausible solutions do not work. The object-description table cannot be kept indefinitely. I already mentioned in section 4.9 the impossibility of storing hard pointers indefinitely; it was this problem that led to data descriptions. If the new steps are recorded during a later session than the one in which the program was originally recorded, or if the original program has been moved or copied to a different desktop, then the hard pointers cannot be kept.

Another possible solution is to look for an existing description that matches a newly created one. But this solution also has problems. The new description for the object may not be the same as the old one, because the object may have moved, been renamed, or otherwise been changed. The original description for a document **Floor Plan** may have looked for it in the **Building** folder, but if **Floor Plan** were moved onto the desktop in some program step before the newly recorded ones, then the new description would specify looking on the desktop, not in the folder.

It would be possible to require the user to run the program up to the point of insertion, as mentioned in section 3.5. The object-description table could then be set up in preparation for recording new statements. But this sort of requirement was already rejected because it was inconvenient, and required setting up data to be as it was at the beginning of the program.

Currently, no better solution to the problem has been found, so the user must be aware of the problem, and make descriptions be the same when necessary.

Another problem arises with the method used to specify data description. All the supplied ways of describing a choice of a data object are given on the data description sheet, in menu form. The sheet lists a number of reasonable possibilities, but is not by any means an exhaustive list of alternate ways of choosing a data object. If the user would like to choose data in some way not provided on the description sheet, he is out of luck. Each alternative on the description sheet is backed by an internal algorithm for doing the choice; there is no way for the user to specify a new alternative and its corresponding algorithm.

It would be possible to provide such a “trap door” into the description mechanism, but then some way must be provided for programming the algorithm for the new alternative. If the system is powerful enough, the algorithm could be programmed by example. If not, it could be programmed in the implementation language of the system or some other programming language, though probably not by the ordinary system user. Data description alternatives could be tailored to the system installation.

However, the idea of such a trap door mechanism, accessible only to experts, does not fit well with programming by example and the whole concept of programming by the user. Instead, the system designer should select the alternatives available on description sheets carefully, and make sure they are comprehensive enough to satisfy the needs of nearly all users.

## **4.12 Data descriptions and other programming-by-example systems**

Data descriptions, which are a fundamental component of programming by example in SmallStar, do not appear in other existing programming-by-example systems. However, data descriptions resemble certain mechanisms used in other systems for specifying data or constraints on data.

The Juno geometric programming system (see section 2.8) allows the user to lay out points on a screen and then draw line segments, shapes, and figures using those points. The user's actions are recorded to create a program. A Juno program includes constraints about the points used in the program. However, unlike SmallStar, the constraints specify relationships between data objects (points), and are not used to describe the objects themselves. In fact, points have no inherent characteristics of their own, and become interesting only when they are combined with other points and used to draw shapes and figures.

A Juno user specifies constraints as he builds a program. Unlike SmallStar's method of providing default data descriptions, Juno does not try to suggest or guess possible constraints. For example, though two points may happen to lie on a horizontal line, they are not constrained to do so unless the user specifies such a constraint explicitly. Automatic constraint generation or suggestion might be useful to the Juno user, but it might also be a nuisance: Juno has a simple user interface for specifying constraints, and they are quickly and easily given.

The data base queries found in Query-by-Example (see section 2.10) are also similar to data descriptions, especially in the way they are used in the PHD programming-by-example system (section 2.11). The PHD system and other QBE systems store data in relational data base tables. Each row in a table is a tuple. Using Query-by-Example, the user constructs queries to select sets of tuples from these tables. The queries specify various constraints on the values of tuples. For instance, in a table of employees, the user may wish to choose everyone whose manager is Smith. The table will have a column for *Manager*. The user would construct a query which selects only those tuples whose *Manager* value is "Smith".

The column headings in a relational table resemble the characteristics of SmallStar objects listed in a description sheet (for instance, see figure 4-3, a description sheet for a document). A document description could therefore also be expressed as a PHD-like QBE query. The icons in a folder might be listed in a relational table as:

<u>Name</u>	<u>Version</u>	<u>Position</u>
Bryce	14 Oct 1984 2:23:34	1
Zion	12 Jun 1983 1:12:17	2
Canyonlands	16 Dec 1984 9:45:00	3

To choose the document Bryce in the folder, the user would give the query:

<u>Name</u>	<u>Version</u>	<u>Position</u>
Bryce		

PHD queries resemble data descriptions. Description sheets and PHD queries are both used to state constraints on data. But the menu of alternatives used in data descriptions provides a more flexible constraint mechanism and a better user interface. In the example given above, to select the last document in the folder, the user might say something like:

<u>Name</u>	<u>Version</u>	<u>Position</u>
		<b>MAXIMUM</b>

However, the user must know which the special operator to use (in this case, **MAXIMUM**) in the *Position* column in order to retrieve the tuple with the largest position value. The description sheet, on the other hand, makes this choice visible at all times.

The description sheet shown in figure 4-16 chooses a text string. The sheet includes a switch which selects whether or not to ignore letter case; such a choice would have to be somehow encoded in a QBE query. The choice **PROMPT**, which is present in nearly all description sheets, has no equivalent in the PHD query mechanism; such description choices, which are not based on data values, cannot be expressed in QBE.

Text String Description

Done Cancel

↑ ↓ T ⊥ ↔ ≡

Choose using **PATTERN** TEXT POSITION ALL PROMPT

Contents pattern Wind Cave

Letter case **MATCH CASE** IGNORE CASE

Position in text 19

Length of selection 9

Prompt Please select some text.

Figure 4-16. A text string description sheet.

The PHD system also differs from SmallStar in the way constraints are first given. In SmallStar, the system creates a default data description when the user first refers to an object. Later, the user can edit the description if it is incorrect. In PHD, the user always gives an explicit query to select data. If PHD were more like SmallStar, it would let the user select the data without giving a query, and then propose a plausible query which would have selected the data the user chose. Such a query generator is a reasonable addition to the PHD system.



## Chapter 5

### Control structure

Programming by example produces straight-line programs in a natural way. A program is a transcript of sequential actions done by the user. But much of the great power of a computer comes from its ability to repeat actions many times and to choose what to do based on the result of some test. If such power can be added to programming by example, then more powerful programs can be written, and programming by example will become better at automating the user's tasks.

#### 5.1 Control structure constructs

Frequently, certain statements in a program are executed only if some condition is true. Most programming languages have constructs like the familiar **if-then-else**. I will call such constructs *conditionals*.

When a computer program repeats the same actions over and over (a *loop*), it usually does so because the operands of the actions are changing in some way. Sometimes the loop is repeated until some condition is satisfied (here called a *while-loop*), and sometimes successive values from some set are substituted for an operand until the set is exhausted (*set iteration*).

There are many other kinds of control structure, but I will concentrate on set iteration and conditionals.

#### 5.2 Control structure in other programming-by-example systems

Not all programming-by-example systems supply control structure constructs. The simpler ones generally avoid them, but the more complete programming systems make some attempt at providing control structure.

The Pygmalion system (section 2.3) provides a way of constructing conditionals. To create a conditional, the user records actions which generate a boolean truth value, and then places the value into a graphical **if-then-else** construct. The system then looks at the truth value in the conditional and, depending on whether it was **true** or **false**, records subsequent user actions into either the **then** or **else** branch of the conditional. When the user indicates he has finished recording actions in the conditional branch, the system then reverts to recording actions in the main-line of the program. To record the actions for the other branch of the conditional, the user must run the program again, changing the example data so that the opposite truth value is generated. Thus recording all the alternate paths in a program requires giving a number of different examples.

Pygmalion also supplies an infinite loop construct, and a separate action that means “exit loop”. Together these constructs can be used to construct while-loops. Pygmalion has no set iteration construct.

In the Tinker system (section 2.5), conditionals are not inserted explicitly by the user. Instead, the system notes when two examples for the same program differ in their actions at some point. At the point at which the user does something different, he is asked to supply a predicate to distinguish the two cases. Once two program paths have diverged because of a conditional, they do not rejoin. Tinker does not provide a way of generating looping constructs by example, but the user is free to use the control structures provided by Lisp.

The Programming by Rehearsal system (see section 2.6 and figure 2-3) provides conditionals and set iteration through the *true-false performer* and the *list performer* available in its **ControlTroupe**. But these control structure mechanisms are not directly connected with the programming by example provided in the Rehearsal World.

Spreadsheet programs (see section 2.2) can do a sort of set iteration, since they provide ways of replicating the formula for a particular cell or set of cells. Parts of the formula may be varied in step with the replication. For instance, if the formula for cell C13 is A2+B13, then the user may replicate this formula into cells C14 through C20, asking that B13 be varied accordingly, but that A2 be held constant. Thus, the formula for C14 would be A2+B14, for C15 it would be A2+B15, and so forth. Thus the set iteration is accomplished by multiple calculations and rules, and not by control structure.

The PHD system (see section 2.11) uses the idea of set iteration heavily. The user chooses a set of data using a Query-by-Example (section 2.10) mechanism, records a program by example using one member of the set, and then, using the program, creates a homomorphism that operates over the whole set. The designers of the PHD system claim that this technique reduces the need for conditional constructs, since much of the time, conditionals are used to choose the data on which to operate.

### 5.3 Set iteration in early versions of SmallStar

The version of SmallStar described in [Halbert 1981] did not provide any sort of conditionals, but did supply a simple mechanism for specifying set iteration. In the program window pop-up menu, there were two commands, **for each do ...** and **... end for each**, used to start and end a loop, respectively. Between invoking these two commands, the user would record the actions he wanted to be inside the loop. Loops could be nested. The commands provided only one kind of iteration set, namely, all the icons of a given type in some container (all the documents in a folder, all the file drawers on the desktop, and so forth).

The mechanism is best illustrated with an example. Suppose the user wanted to write a program to move all the documents in a folder to the desktop. He would start recording, select the folder, open it, select any of the documents inside the folder as the example on which to generalize, and then invoke **for each do ...**. The system would start building a set iteration loop. Then the user would then move the document out of the folder and invoke **... end for each**. The system would close the loop, and execute the statements in the loop for the rest of the members of the iteration set, which in this case is the rest of the documents in the folder. Finally the user would close the folder and stop recording. Here is the program that would be shown in the program window (*A* is a variable name generated for the folder):

```
Select A;
Open A;
For each Document B in A do
  Move B to the desktop;
end;
Close A;
```

When the user closed a loop with **... end for each**, the system repeated the loop for the rest of the members of the iteration set. It is also quite plausible that the system might not have done this, and instead, repeated the loop for the whole iteration set only when the program was run. Executing the loop for all the members of the iteration set during program recording is slightly more faithful to the intent of the example, but if that will take a long time, the user may not want to wait for it. For SmallStar, either choice is reasonable.

This method of set iteration was easy to understand. But it was not clear how well the method would work when more ways of specifying the iteration set were provided. I considered the description of an iteration set to be a form of generalization, and choosing from among multiple kinds of generalization caused problems when done during program recording. As explained in section 4.7, an *ex post facto* generalization mechanism seemed best. If the generalization were to be done after program recording, as a program editing operation, then it also seemed reasonable to add the control structure by editing the program after it was recorded, rather than trying to record the control structure in-line. This means that control constructs themselves are not added in a by-example way, though the statements they contain are programmed by example.

## 5.4 Set iteration in SmallStar now

In the current version of SmallStar, set iteration control structure is now always added to a program after it has been recorded. Iteration sets are specified by editing descriptions through their description sheets. The user records the program and includes the statements that will be in the body of loop, but initially records them into the main-line of the program. The user then selects the statements that should be in the body of the loop, wraps them in a loop, and then specifies which description is to be used for the iteration set. The details of this procedure are described below in an example.

In chapter 4 I described data descriptions, and implied that a data description searches for a single data object that matches the given description. In fact, this is not really true. The data description actually generates the set of *all* the objects that match the description, but in most cases only the first object in the set is used. The description is ordered in some reasonable way. For instance, a request to find the newest version of the document **Fishing Treaty** will actually generate a set of all the documents with that name, ordered from newest to oldest.

Most description sheets provide **ANY** as one of the possible ways of choosing a data object (see, for instance, the description sheet shown in figure 4-3). For single objects, this choice is not terribly useful. For instance, on a description sheet for a document in some folder, **ANY** means any document in that folder. But since the description would actually generate a set of all documents that match the given constraints, **ANY** would actually generate a set of all the documents in the folder. The resulting set can then be usefully used in a set iteration.

As an example, consider the simple move program discussed in section 4.4. The program moves the document **Treaty** out of the **Negotiations** folder. Suppose the user instead wished to move all the documents out of the **Negotiations** folder. (In Star it is possible to select consecutive icons in a folder and move them simultaneously, but this feature has not been implemented in SmallStar.) The user would start out by recording a program that just moved one document out of the folder. The program is given in figure 5-1. Then, the user would edit the description for the document, selecting **ANY** document in the folder (figure 5-2). The resulting program is shown in figure 5-3.

The user wants to repeat the **Move** statement for all documents in the folder. He first wraps the **Move** in a set iteration loop, by selecting the **Move** statement, and then invoking the command **Repeat ...** from the program window pop-up menu. The program then looks like figure 5-4. The set iteration loop is present, but the set to iterate over has not yet been specified, hence, there is a blank space labelled **-fill this in-**. The user selects the **any** document description, presses **MOVE** or **COPY** to produce another reference to that description, and drops the reference into the blank, yielding the program shown in figure 5-5. When this program is run, it will create a set containing all the documents in the folder, and then, one by one, move the documents out of the folder onto the desktop.



Figure 5-1. Program to move one document out of a folder

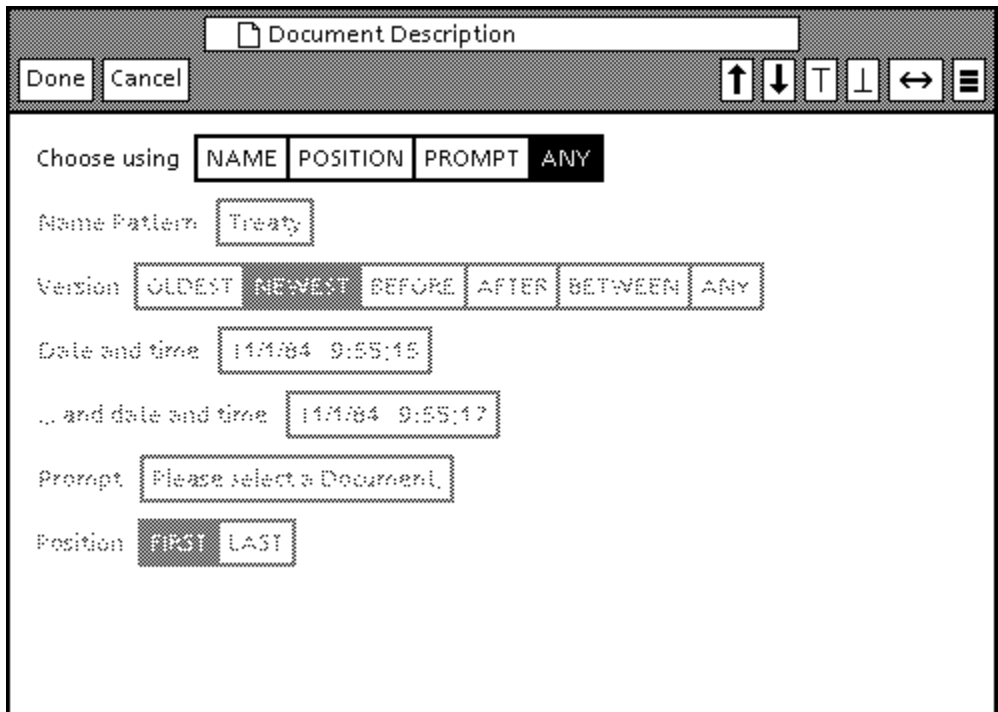


Figure 5-2. Description sheet for any document in a folder



Figure 5-3. Program to move any document out of a folder.

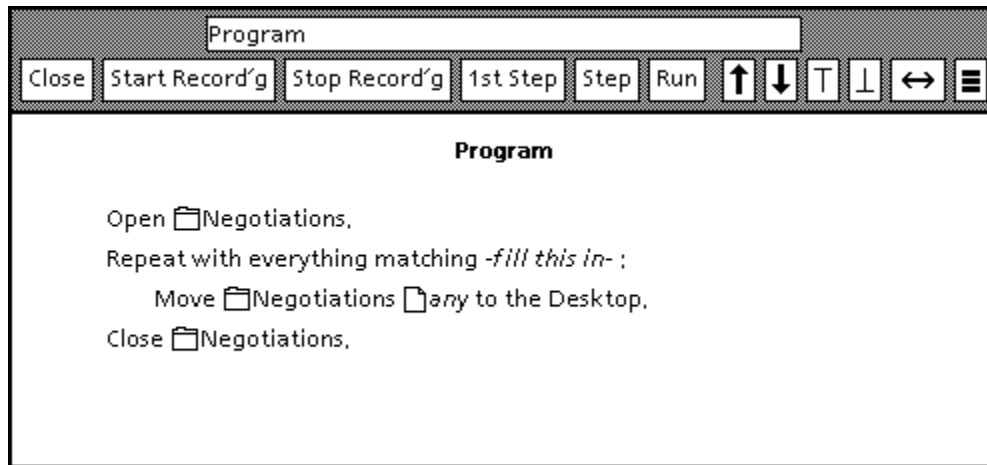


Figure 5-4. Adding the set iteration loop.

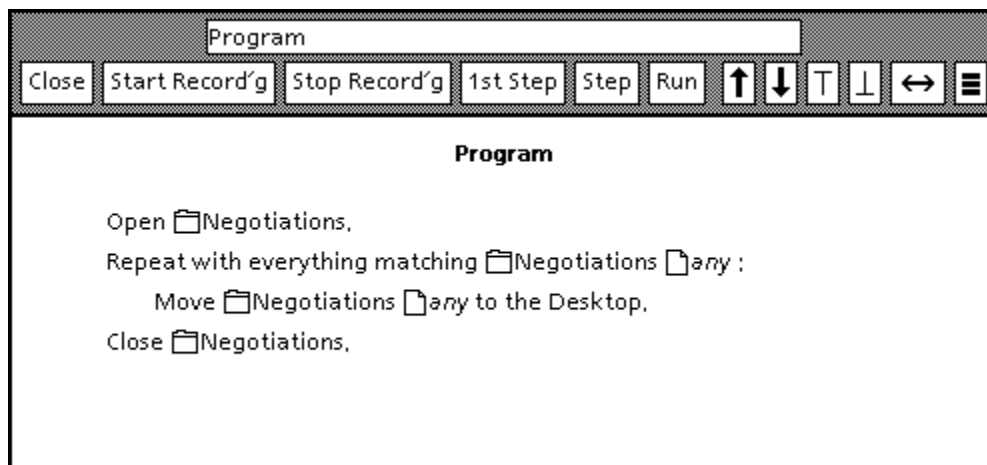


Figure 5-5. Program to move all the documents out of a folder.

No variable names are introduced in the program using this set iteration technique. There is nothing like the *D* in *For each document D in Negotiations do ...*. Sharing descriptions substitutes for this use of variable names.

This method of doing set iteration is quite convenient, though it does require that the user read and understand the program to decide which statements should be in the loop body, and which description will be used to generate the iteration set.

## 5.5 Conditionals and programming by example

Conditionals are inherently difficult to provide in a programming-by-example system. Programming by example naturally produces straight-line programs. When the user goes through an example that is being recorded, he is tracing a single thread of control through the program. But a conditional is a fork in the program graph, and only one fork at a time can be taken. For example, if an **if-then-else** construct is provided, when does the system record both branches? Recording one and then the other consecutively (say, **then** first and **else** second) may not work, since the code for **then** may change the state of the user's world enough so that the **else** code could not be recorded. To avoid this problem the system could

save the state of the world before recording the **then** code, and restore it before the **else** code was recorded. But this also may not work. The predicate evaluated to either **true** or **false**, thus selecting **then** or **else**. It may not be possible to record the code for the other branch of the **if**, given the result of the predicate.

One solution is to record only the branch selected by the predicate. This is what is done in Pygmalion and Tinker, as described in section 5.2. The program is incomplete until the other branch is recorded, and another example must be given that will cause the predicate to produce the opposite value.

Another possibility is to allow only **if-then** constructs, or some kind of **case** construct in which each case tests an arbitrary predicate. The user records an **if** statement or a case whose predicate must be true when it is given, so that the **then** code can be recorded. When the user stops recording, the program will not contain any unspecified paths in its control flow, though it may still be incomplete because it does not include conditionals for all the necessary cases. It may be impossible to record all conditionals in a single pass of program recording, since it may be impossible to make all their predicates be true at once (for instance, some may be negations of others). So the user must be able to edit an already recorded program to add new conditional statements.

Regardless of what method is chosen, it will probably be necessary to give several examples to cover all program paths. That is the penalty of programming by example.

## 5.6 Conditionals in SmallStar

Because of the problems mentioned in the previous section, I did not attempt to provide a conditional mechanism in early versions of SmallStar. In addition, Star itself, without extensions, has no way of expressing predicates which could be used in conditional statements. So I had to add a way of specifying predicates as well as a control structure construct for conditionals.

In the current version of SmallStar, the control structure for conditionals is added after the program has been recorded, just as with set iteration, as explained in section 5.4. In fact, I originally thought of adding control structure by program editing only for the set iteration construct, and only subsequently realized it could be applied to conditionals as well.

Conditionals are added in a way that is quite similar to the method used for set iteration. The user records a program first, and includes the statements that will be in the body of the conditional. After the program is recorded, the user selects the statements to be executed conditionally, and wraps them in a conditional. The user then edits the conditional to specify the predicate to be tested. If the predicate is a simple comparison, it can be expressed directly in the conditional. If it is a more complicated test, the user can record extra statements just before the conditional to compute the value of the predicate, using the relational and boolean operations provided by the calculator. Since the user can add extra statements to a program after it has been recorded, more conditionals and the statements they control can be added to an existing program, if there are more conditions to test.

I will illustrate this mechanism with an example. Suppose a customer's order is sent by first-class mail if it is under one pound, but by fourth-class mail otherwise. A mailing instructions form has already been made up, which assumes that fourth-class mail is to be used (figure 5-6). The program given here is probably part of some larger program. The program should open the mailing form, look at the **Weight** field, compare it with 1, and change "Fourth" to "First" if necessary. Then it should close the form and drop it on a printer.

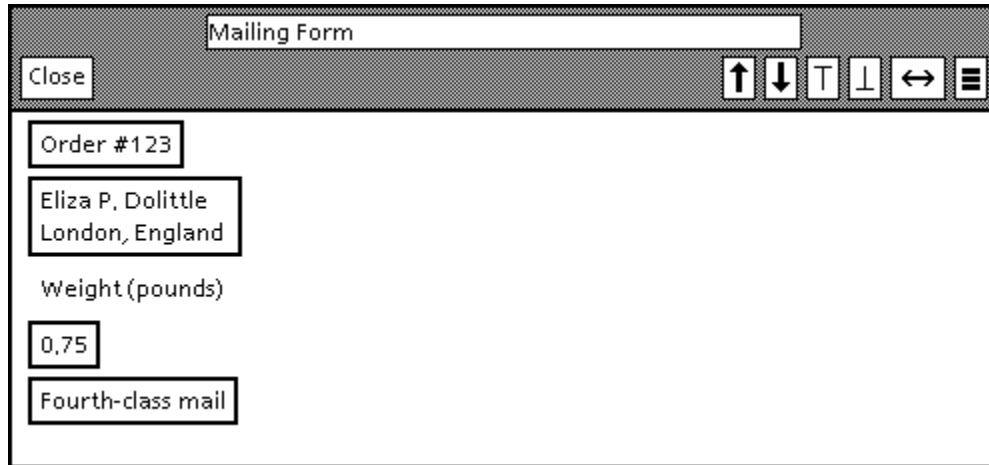


Figure 5-6. A mailing form.

First the user records the program, assuming that the condition is true. The initial program is shown in figure 5-7. The user selects the statements that will be in the body of the conditional, as shown in the figure. The user then wraps the selected statements in an if statement, using the If ... command in the program window pop-up menu. The resulting program, with its incomplete conditional, is shown in figure 5-8.

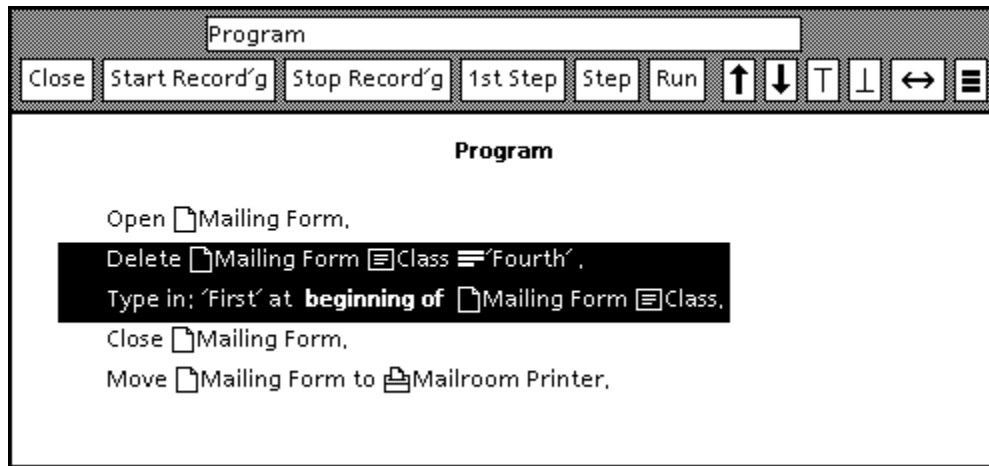


Figure 5-7. Mailing form program, before addition of the conditional.

To edit the conditional, the user selects the equal sign (=) in the program and presses **PROPERTIES**. Using the property sheet, the user changes the comparison to a <, and puts 1 in as the second operand. Then he selects all the text in the Weight field, and invokes **Make Description** from the program window pop-up menu to create a description for the selected text. The description is dropped into the property sheet as the first operand. The resulting property sheet is shown in figure 5-9. After the user presses **Done** in the property sheet, the program is changed to reflect the predicate, and is shown in figure 5-10.

The user could also have produced the predicate using SmallStar's calculator (shown in figure 5-11.). The calculator has relational and boolean operations as well as simple arithmetic operations. The user can compute a boolean value by inserting extra steps that perform calculator operations before the if

statement, and then editing the if statement so it tests the truth value generated in the calculator. This program is shown in figure 5-12. 'Yes' and 'No' are Star's way of saying true and false.

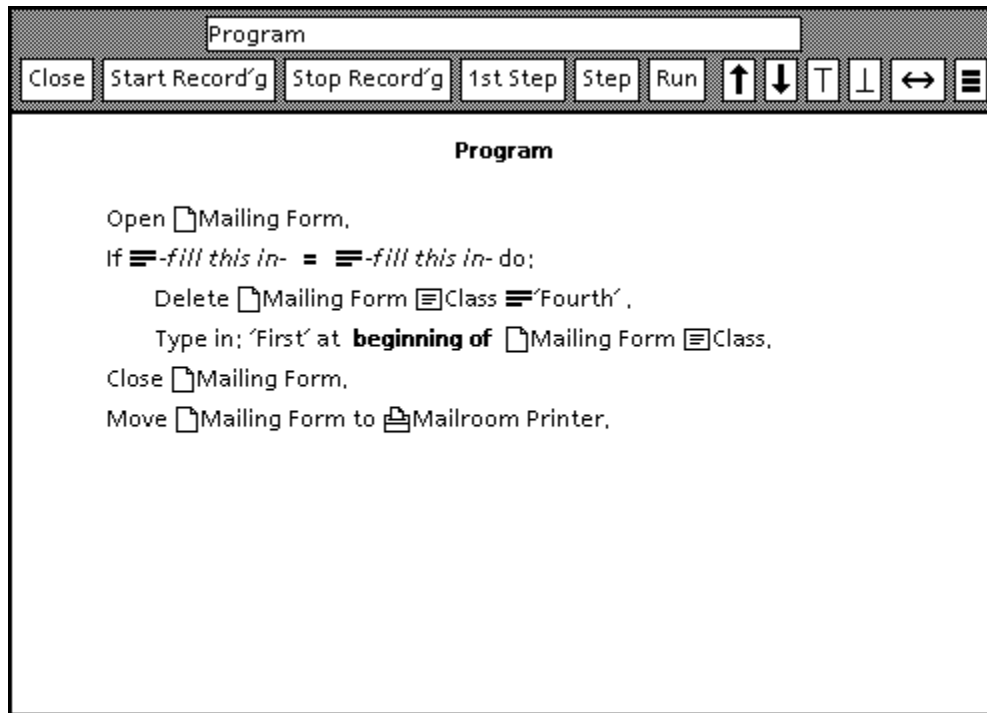


Figure 5-8. An incomplete conditional.

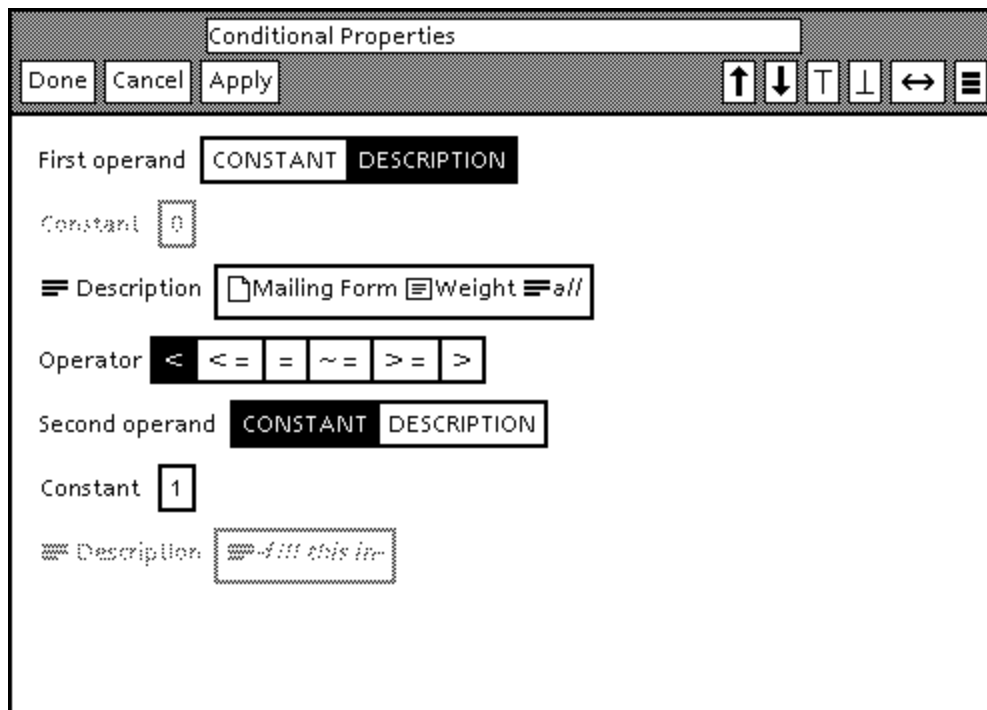


Figure 5-9. The property sheet for the conditional.



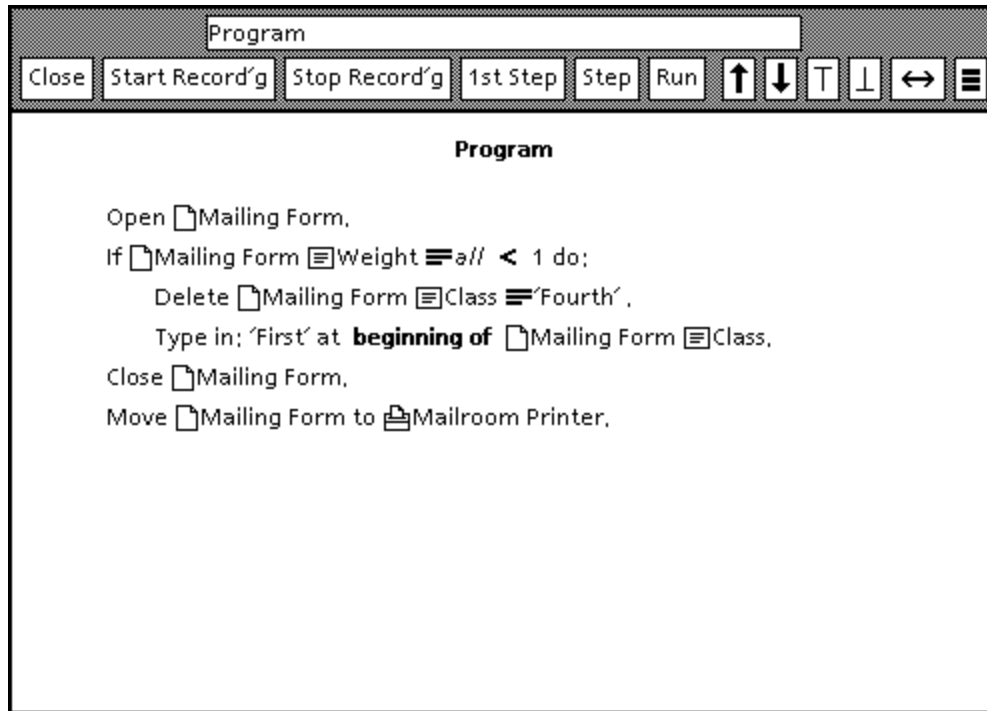


Figure 5-10. Completed mailing program.

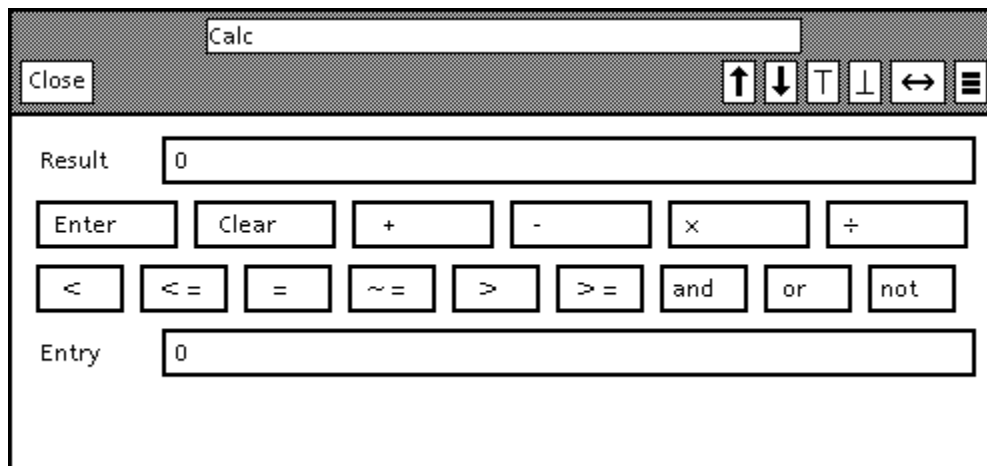


Figure 5-11. The calculator window.

Originally, the calculator was used only to create predicates. The if statement only tested values for being equal to 'Yes', and the value tested was almost always the Result field in the calculator. However, the resulting programs were quite hard to read, as can be seen in the figure. So I implemented the more flexible way of specifying predicates, described at the beginning of this section, which is akin to a syntax-directed editor.

Further additions could be made to conditionals. The calculator could be used to specify the predicate expression, but the resulting computation expression could be displayed algebraically, and not as a sequence of calculator commands. This would improve readability greatly. Or, the conditional property sheet could be enhanced to allow more complicated expressions.

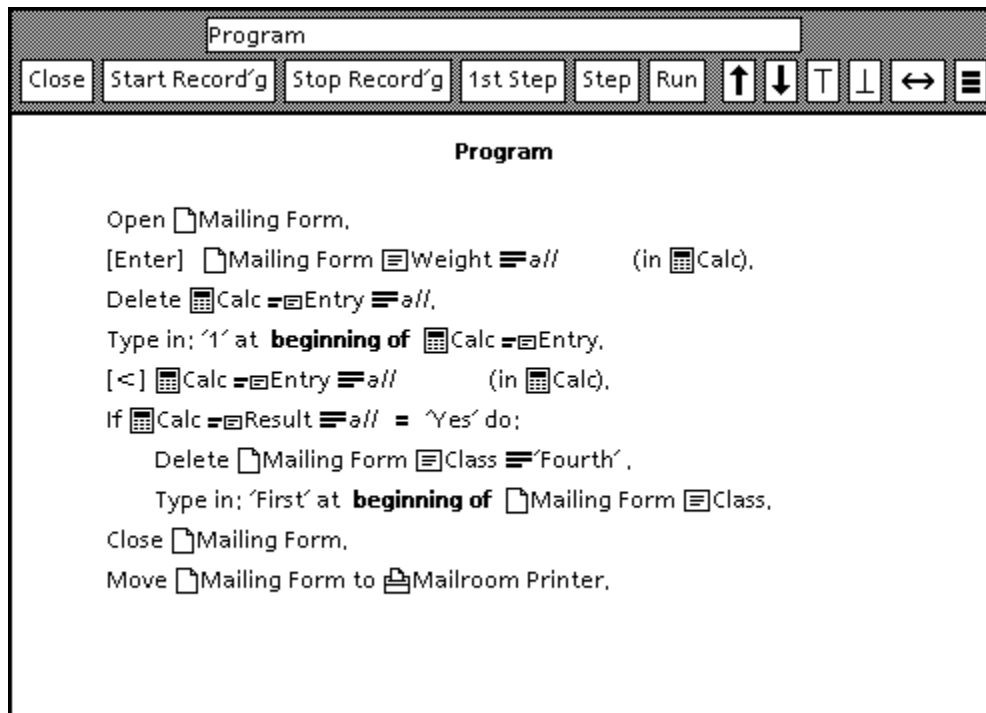


Figure 5-12 Mailing program written using the calculator.

The problem of when to record the statements to be executed conditionally, as discussed in section 5.5, is present, but is not much of an annoyance. In the example given, the conditionally executed statements were recorded when the main-line of the program was recorded, but they could have been added later to the program. Since the user is free to add statements at will, he can set up the state needed to record some conditionally executed path after he has recorded other parts of the program.

## 5.7 Other control structure

While-loops are not provided in SmallStar, but could be added easily. The body of the loop would be wrapped in a **while** statement just as with set iteration and conditionals. A predicate could be specified as is done for conditionals.

Procedure calls are another kind of control structure, not yet mentioned. When a procedure is called, arguments are passed to it, it is invoked, and then results are returned from it. SmallStar allows existing program icons or buttons to be invoked while a program is being recorded, but no simple way of passing arguments explicitly has been implemented. The data to be passed can be left in known places, so that for instance, a called program that computes shipping charges based on some weight would know to look in a field in a particular document for its input, and would, by convention, leave the computed charges as output in another field in the same document.

One can imagine several different methods for argument passing. In a called program, operands could be parameters. The descriptions for those operands would specify whether the operand was an input or output parameter, and what position the parameter was in (first, second, third parameter, etc.). When a program was called during program recording, the user would select the objects to be passed as

parameters, one by one, and the recording mechanism would build an argument list consisting of the descriptions for the selected object.

Another possibility would be to provide input and output parameter slots associated with a program. A program icon or button would have parameter fields and frames attached to it. Before running the called program, the user would move the input arguments to these input slots. After running the program, the results would appear in the output slots and could be moved somewhere else.

## Chapter 6

### User trials

#### 6.1 Purpose of trials

The ultimate goal of the research described in this dissertation is to show that it is possible to build a practical, usable programming-by-example system. One can state that SmallStar is a demonstration of a practical system, but the real test is to see whether it can be used by its intended audience. This chapter describes a set of trials of SmallStar by experienced Star users who are potential users of a programming-by-example mechanism in Star.

Ideally, programming by example should be compared to other kinds of programming to see whether it is easier to learn and use. However, it would be very difficult to design an experiment to test this which would yield meaningful quantitative results. One plausible experiment would compare programming in the Star programming language Cusp (see section 3.3) with the programming-by-example mechanism in SmallStar. Pools of subjects would be trained to use both systems. Their speed of learning, error rates, ability to program specified tasks, and so forth would be compared.

However, there are a number of variables in such an experiment which would be difficult to control. The instruction used to teach the two different systems must be similar in quality and approach. The functionality provided by the two programming systems must be similar. The tasks the users are asked to do must be realistic, and must not accidentally favor one programming method over the other.

Performing such an experiment is beyond the scope of my research. The number of subjects needed to produce statistically useful results is unknown and potentially large. Cusp and SmallStar are currently sufficiently different in functionality that a particular task is much more suited to one than the other. Bringing them closer together would have required much more programming. It was more interesting and fruitful to spend the time evolving the SmallStar programming-by-example mechanism than to bring it closer to Star and Cusp.

Instead of attempting such an experiment, I elected to evaluate SmallStar programming by example in a more quantitative, subjective way. Discovering whether users *perceived* programming by example to be easy to use was just as interesting as finding out whether it actually was better than conventional programming. So a number of Star users were taught how to use programming by example in SmallStar, taken through a few sample tasks, and then asked to produce similar tasks on their own. Along the way they were questioned about their understanding of programming by example, how easy they found it to use, and what problems they encountered. I called these tests *user trials* rather than experiments, to emphasize their non-rigorous nature.

Other kinds of experiments could be done, which would answer questions about the success of the individual design decisions made in SmallStar. The programming-by-example functionality in SmallStar could be implemented in different ways, and those ways compared. For instance, the user could be allowed to edit data descriptions only while the program was being recorded. Or, the user could be required to specify control structure during program recording, rather than inserting it by editing afterwards. These experiments would test the hypothesis, now assumed to be true, that such operations are better done after the program has been recorded. Methods for presenting data descriptions in formats other than description sheets could also be tried, to see whether one method is more or less understandable than another.

These sorts of experiments would be useful for fine-tuning the user interface to a programming-by-example mechanism, but of course would not show, by themselves, whether or not programming by example is useful.

## 6.2 The trial users

User trials were done on five employees of the Xerox Office Systems Division. The users will be named A, B, C, D, and E. All are experienced Star users who use Star in their daily work. In no particular order, one is a product planner, one a technical writer, one a secretary, one a project manager, and one a Star consultant who helps others learn and use Star. All will be referred to as “she”, though there was actually a mix of men and women.

Surprisingly, no user tested was without programming experience. User A had not taken any programming courses, but had done some exercises in a book that taught simple programming in terms of commanding a robot. B had just started a beginning Pascal course. C had taken courses in PL/I, COBOL, and BASIC, but had not done any programming outside of school. D had taken a programming courses two decades ago, and has since written a few BASIC programs. E had taken a Pascal course and a course in algorithm design and data structures.

Some, but not all of the users had attended a talk I gave on programming by example and SmallStar. However, during the trials, none of those who saw the talk referred to something I was teaching them as a detail they remembered from the talk.

## 6.3 Procedure

Each user trial took place in a quiet cubicle, with a minimum of interruption. Because of logistical difficulties, no audio or video tape records of the trials were kept; I took notes instead. I spent between an hour and two hours with each user. The user always manipulated the keyboard and mouse; I told the user what to do, and did not take over the input devices when demonstrating something.

The trial started with an introduction to SmallStar, its functionality, and how it is similar to and different from Star. The keys on the keyboard are in different places, so that was pointed out, and the difference between Star and SmallStar documents was shown. Property sheets for text and icons were demonstrated.

I walked the user through the recording of a simple program which moved a document with a particular name out of a folder, similar to the program shown in figure 3-5. The user saw how the program transcript was built up as her actions were being recorded.

I then asked the user to record a program that moved the first document out of a folder. This task is exactly the one shown in figures 4-2, 4-3, and 4-4. The user was shown how operands in the program could be selected, how several operands could correspond to a single data description, and how descriptions could be modified using description sheets. I asked the user to write a simple program using the knowledge she had already learned, such as deleting the last document in a folder.

Set iteration and conditionals were covered next. I showed the user how to add a set iteration loop to a program that moved a document out of a folder. The steps for this program are shown in figures 5-1 through 5-5. I then took the user through the construction of a program with a conditional, using the example shown in figures 5-6 through 5-10.

At this point, rather than supplying another fixed task, I asked the user to think of a task she would like to try to program. If there was time, I also took the user through the order entry example shown in figures 4-6 through 4-9.

Finally, I asked each user what her general impression of the system was, and whether she would find it useful if it were present in the real Star system.

## 6.4 Observations

The user trials were quite encouraging. Most users understood the concepts presented during the trial without much trouble. Conceptual blocks did not persist for long. A number of users had comments and criticisms about details of the user interface, but few provided comments about the basic ideas of programming by example and the techniques used for data description and control structure.

I had given a number of demonstrations of SmallStar previously, and expected difficulty with certain features. In particular, the **Make Description Different** and **Create Description** commands, described in sections 4.4. and 4.5, respectively, had been stumbling blocks in the past. But surprisingly, no users during the trials had any difficulty understanding what the commands did. Long paths, as mentioned in section 4.11, were a problem during the demonstration and to some extent, during the trials. Also, Star users do not know the pictographs in figure 3-6 that are not pictures of icons, and so some SmallStar users found them hard to comprehend.

Following are detailed comments on each user's trial:

A: User A had no trouble grasping the concepts presented. A did find description paths difficult to read, because of their left-to-right nature: the object of ultimate interest was far away from the action. A found the use of the word **any**, as used in descriptions for set iteration, confusing; she would have preferred **every** or **all**. She thought of and started to program a form letter application, but was unfortunately stymied by a bug in SmallStar (subsequently fixed).

A was somewhat familiar with the Cusp programming language used in real Star, but disliked it because she found it too easy to type something that was incorrect. By contrast, she liked programming by example because it did not require her to remember programming language syntax. She found herself motivated to create programs because of the speed and ease of use of the mechanism. She said that debugging was “fun”, and also remarked that “even if the program has bugs, part of it always works”.

B: User B had some difficulty understanding set iteration. B started to write a program for a task she had in mind before formulating the task clearly; as a result she became stuck in the middle, and had to be coached to state clearly what it was she wanted to accomplish. Some of the tasks she wanted to program were unprogrammable because the data descriptions did not provide enough choices (for instance, she wanted to be able to choose the two newest documents in a folder).

B was inclined to edit an existing program into a new one, by deleting and adding steps. She had some difficulty doing so because of the limited program editing SmallStar provides. She later commented that she would in some cases prefer just to type in the commands rather than record them, and found the programming pop-up menus and recording occasionally tedious to use.

C: User C used the single-stepping commands in the programming window a great deal in order to try out small parts of programs. She could read and understand description paths, but would have preferred a more pictorial approach, especially for the descriptions of parts of tables: she suggested larger pictographs with the descriptions contained within them. C thought of a simple icon

manipulation task and programmed it easily. We also went through the order entry example, which she understood with no trouble. C found herself mentally translating descriptions to the corresponding real objects; she would have liked a command that did this for her.

- D: User D had little trouble with most concepts. She found the separate single-stepping commands **1st Step** and **Step** somewhat confusing. Since steps in a program could be deleted, she expected other kinds of programming editing to be possible. She wanted to see the list of possible descriptions, perhaps displayed in a pop-up menu, that could be used to fill in the *-fill this in-* blanks found in control structure and description sheets (for instance, see figure 5-4). Like user C, D found paths hard to read, and would prefer larger pictographs with their descriptions inside.

D constructed a program to merge the text from two documents into a single document. The program had two errors in the middle, and D was able to correct them by deleting a step and inserting two new ones, after a small amount of coaching.

D considers programming by example to be an “essential” capability for Star. Though she finds the programming-by-example mechanism complex in places, she feels the complexity is necessary for the power it provides.

- E: User E understood every concept with no misunderstandings. She did not object to the current form of paths, and found them easy to read, but would prefer a slightly more graphical approach for paths to parts of tables. She said she found conditionals easy to understand because of her programming experience. E could not think of a program to write, so we went through the order entry application, which she understood perfectly.

## 6.5 Results

The user trials can be considered to be successful. None of the users had any difficulty understanding the basic concept of programming by example. Given surprisingly little instruction, most were able to comprehend and use all the concepts presented. Users had no trouble understanding data descriptions or the idea that one data description could be used in several places in the program.

The choices available on certain data description sheets could be improved. Some users were stymied when they could not describe a certain object or set of objects.

Some users had difficulty with set iteration, but this seemed to be a result of the awkward use of the data description choice *any*. In retrospect, it would be better if data descriptions indicated more clearly when they were being used to generate a single value, and when they were intended to choose a set.

Most users wanted an undo facility, so they could back up one or more steps in the program if they had made an error.

All the users tested were enthusiastic and said they would definitely use programming by example if it were available in Star. They did not consider it too difficult to learn or use. The trials were too brief to be able to ascertain what limitations they might encounter while trying to program other tasks by example. Nevertheless, their uniform enthusiasm and acceptance of the idea of the system is the most telling result. It means that programming-by-example mechanism functionality like that in SmallStar, if expanded and added to the real Star system, would be accepted and used by a large number of Star users. The ordinary Star user would then be able to program Star.

## Chapter 7

### Discussion and conclusions

#### 7.1 Summary of work; principal contributions

The original motivation for the research described in this dissertation was to explore programming for the non-programmer. Programming by example seemed to be the most promising direction to take; I wanted to show that it could be a practical technique for letting the user of a system write his own programs easily.

I started my research into programming by example by adding a simple programming-by-example mechanism to SmallStar, a simulation of icon user interface of the Xerox Star office information system. Because the resulting programs operated only on constant data, and did exactly the same thing each time they were run, I added the notion of *generalization* to the programming-by-example mechanism. The programmer could generalize a data object that had been a constant; when the program was run again, the system would choose some (possibly different) object that matched certain characteristics of the original. A simple set iteration mechanism was also added.

I then spent some time adding many more Star features to SmallStar, so as to be able to write more realistic and varied programs by example. The somewhat vague notion of generalization evolved into the concept of *data description*. The system created a data description for every object used while recording a program. A data description contained specifications for choosing an object when the program was run; the system supplied a reasonable default specification in each description it created, but if the specification was wrong, it could be modified by the user. Control structure could be added to a program by editing, after the program had been recorded; I implemented set iteration loops and conditional statements.

Finally, I tested SmallStar by teaching it to a number of experienced Star users, to see whether they found the system usable and useful.

The research described in this dissertation produced three notable contributions to the study of programming by example:

- The notion of *data description* is useful: data descriptions search for and select the data to be used each time a program is run. They let the user say why he chose the data he did when he created a program by example, by providing a mechanism that indicates which characteristics of the data are important.
- Certain programming operations, such as adding control structure, are better done after, rather than during program recording. The operations can be easily accomplished by program editing, if the programming mechanism provides the user with a readable static representation of the recorded program
- A practical system for programming by example can be incorporated into a richly functional, commercial software system.

The first two contributions are techniques that can be used in programming-by-example systems; the last contribution is a goal, not a technique, that was met with the help of the first two ideas. Much of the rest of this chapter discusses various aspects of programming by example and how these contributions relate to them.



## 7.2 Syntactic and semantic programming by example

A programming-by-example mechanism can be included in many different kinds of systems. However, it seems easier to add programming by example to some systems than to others. Text editor macros, such as EMACS keyboard macros (section 2.1), and the typescript programs provided by Exemplary Programming (section 2.7) both have straightforward programming-by-example mechanisms, because they simply keep a transcript of the user's keystrokes. Compare this with the mechanisms in Pygmalion (section 2.3) and SmallStar, which must go to some trouble to record the user's actions; they must interpret the user's keystrokes and mouse-button clicks in terms of the objects in the Pygmalion or SmallStar worlds.

These two categories of systems illustrate the difference between syntactic and semantic programming by example. A syntactic programming-by-example system records the user's actions in the form in which they were given; a semantic programming-by-example system is more interested in the meaning of what the user does.

Typescript systems like those mentioned above lend themselves to syntactic programming by example, while direct manipulation systems such as Star need semantic programming by example. There are several reasons for this difference.

In typescript systems it is possible to record the raw user input, play it back, and have the system perform the same task. In direct manipulation systems, the raw user input consists of pointing as well as keystrokes. Some commands may be invoked from the keyboard, but others will be invoked by pressing "buttons" on the screen, pointing at items in menus, or directly manipulating (for instance, moving) objects on the screen. Simply adding information such as cursor position or mouse movement to the program transcript will work only if objects and commands always appear in the same place on the screen, which is unlikely. Instead, the system must look a level deeper than the raw user input, and record what commands were invoked, not just what keys were pressed.

Second, and more importantly, typescript systems refer to objects explicitly, by name or by description. A denotation for an object is included in the typed command. For instance, filenames and filename patterns are common operands in operating system executive languages. When a programming-by-example mechanism records commands in such systems, it need not understand what the meaning of the syntax is; it merely has to record the syntax.

In contrast, one usually refers to objects in direct manipulation systems directly, by pointing to them. There is no need, and frequently no opportunity, to give a description for an object. Since objects and many of their characteristics are visible, it is easier for the user to search for an object visually than to give a description that denotes the object. The user does not need to communicate the mental description he is using to the system; the mental description substitutes for the explicit description the user would have given in a typescript system.

If the programming-by-example mechanism in a direct manipulation system records the user's actions only in a simple syntactic way, the transcript will lack some crucial semantic information. The mechanism must therefore also record the user's intentions even though he has not stated them explicitly.

This recording of the user's intentions can be done in several ways, as was discussed in section 4.2. The system can require the user to state his intentions explicitly, it can use inductive inference, or it can make informed guesses as to what the user means, and let the user correct the guesses if they are wrong. SmallStar implements the last choice, using data descriptions to record the user's semantic intent. Data descriptions substitute for the descriptions that would be given in a typescript system; they are denotations for objects.

These contrasts between typescript and direct manipulation systems only illustrate the differences between syntactic and semantic programming by example; they do not define them. Other systems fall in between these two extremes, and combine aspects of both kinds of programming by example.

The Tinker system (section 2.5) combines aspects of both syntactic and semantic programming by example. Tinker understands the structure of Lisp functions. It knows they consist of nested function applications which return values. Tinker lets the user build Lisp functions in terms of these components; it does not just blindly record what the user types in. On the other hand, Tinker does not try to understand the meaning of each function call. Thus, Tinker does semantic programming by example to help the user build a function out of function calls, but does a sort of syntactic programming by example when recording the individual components of the function.

Pygmalion (section 2.3) is a direct manipulation system, but the semantic aspects of its programming-by-example mechanism are not as complex as those of SmallStar. The Pygmalion system has graphical cells which can contain numbers. When the user uses or changes the contents of a cell, the system always records a constant reference to the cell itself, not to its contents. (Section 4.3 discusses the differences between referring to data directly, and referring to it in terms of its container.) For instance, if the user moves the number 5 from one cell to another, the operation recorded will always refer to the two cells, not to the number 5. Because of this uniform way of referring to data, Pygmalion does not need a mechanism like SmallStar data descriptions. On the other hand, choosing a cell at run-time is correspondingly more difficult; the user must record an explicit computation that makes the choice, instead of using a data description.

The differences between syntactic and semantic programming by example provide a convenient way of classifying programming-by-example systems. Many systems will mix the recording of syntax and semantics. Separating the two when examining a system, or when considering the design of a new system will help clarify what functionality is really being provided by the programming-by-example mechanism.

### **7.3 Creating the program: recording vs. editing**

The fundamental idea of programming by example is simple: the user creates a program by giving an example of what he wants the program to do. The system records his actions; the resulting transcript is the program.

Clearly, SmallStar does not use pure programming by example. Certain parts of the program are specified, not during recording, but afterwards, by editing an already recorded program. SmallStar data descriptions are constructed by the system while the program is being recorded, but the default descriptions the system chooses may be wrong, and it is up to the user to edit and correct the data descriptions after the program has been recorded. In addition, all the control structure in a SmallStar program is added after the main-line steps in the program have been recorded.

It is a mistake to try to do everything by example. Programming by example is best at recording serial user actions. When it strays from this best use, the serial nature of the mechanism may intrude and make certain tasks awkward. If the user must give many commands related only to program recording, or if he must change or augment the manual steps of a task in order to record it, then one of the virtues of programming by example will be lost: recording the example will no longer be similar to doing the same task by hand.

It is therefore better to let the user record the program in as straightforward a way as possible. Complications such as adding control structure or specifying exactly how the program should choose its data should be done after the example is given, so that the user can concentrate on recording the algorithmic steps of his task. In earlier versions of SmallStar, data descriptions and control structure were

done in a more by-example way. The user specified data generalizations (section 4.7) and indicated the boundaries of set iteration loops (section 5.3) as he was recording the program. But doing these parts of program creation during recording was awkward, and now these tasks are done by editing, rather than during program recording.

A few programming-by-example systems are able to combine recording and program editing, because they are willing to re-execute parts of the program every time a change is made. Juno (see section 2.8) and the Build system for dynamic program building (section 2.9) are examples of such systems.

A program created by example in the Juno system describes a static geometric figure on a display screen. Each time the user modifies the program, by adding, modifying, or deleting an operation or a constraint, the program is run again, and the figure is redrawn.

In the Build system, the user types in statements in a BASIC-like language, which are executed immediately. If the user starts to delete or insert statements in an existing part of the program, the program is re-executed from the beginning, and then is stopped at the point just before where the editing is to take place. Rerunning a program, can of course, take quite a while.

Both Juno and Build must be able to restore the objects their programs can change to their original states before rerunning a program. For Juno, this is a simple matter of clearing the screen. But Build must keep snapshots of its world or undo what the program has done. Saving and restoring state is discussed further in section 7.4; it is impractical for many real-life systems.

Combining program editing with recording is attractive, but it will be feasible only with certain systems. If rerunning a program every time a change is made is time-consuming or impractical, the system will have to rely on more conventional program editing.

If some part of program creation is done by editing, there must be something to edit. The program must be made visible to the user. One way is to display dynamically what the program does by letting the user single-step the program; the user can stop and edit when necessary. But for other than very simple programs or systems this will be awkward. The user needs to be able to view the whole program, to see what actions are being enclosed in what control structures, and to see quickly what data is being used where.

Program editing therefore implies the system must provide some human-readable static program representation. Designing a concise and comprehensible program representation is a major part of the design of a programming-by-example system.

## **7.4 Real and contrived examples**

When a user decides to write a program by example, he may wish to supply his example in a tentative way. If his example does not work for some reason, he may not want his actions to have damaged any real data, or to have performed any undesirable or irreversible actions. Such caution may not be necessary if the program being recorded is straightforward. Indeed, one of the goals of programming by example is to make it more likely for a program to work the first time. Nevertheless, the user may want to be careful when executing his example. For this reason, the user may prefer to give a contrived, “made-up” example while recording the program, rather than incidentally performing the task he wants to do on real data as he records it.

There are several ways to attack this problem. The user can avoid using real data, the system can suppress undesirable side effects while a program is being recorded, or the system can provide a way of saving and restoring the state of its world.

To avoid mangling real data, the user can manufacture some sample data, either by creating some data that resembles the real data accurately enough to be used in a recorded example, or by making copies of already existing real data, and manipulating the copies to avoid damaging the originals. (In fact, creating some sample data may be necessary if the user is writing a program to be used in the future, but for which he has no immediate application and no data to manipulate.) The distinction between “real” and “made-up” data is entirely in the user's mind; the system cannot tell the difference.

The undesirable side effects mentioned here are generally actions that produce effects outside the user's world; they are actions he cannot retract or undo easily, or which may interfere with other users. In the Star system, such actions can merely be annoying, such as printing a document the user does not really want printed, confusing, such as automatically sending mail to other users which may be inappropriate, or really damaging, such as changing files that are shared with others. Such unwanted actions can be prevented by turning them off while a program is being recorded or tested. For instance, the user would move a document to a printer icon, but nothing would actually be printed. Changes in a shared file would be visible only to the user who made them, and so on. But the system would have to indicate clearly to the user when he was in such a “safe” mode, to avoid confusion.

A third way to prevent the user from mangling his world is to provide a way to restore its state to some previous time. Before the user started to record a program, the system would *checkpoint*, or take a *snapshot* of the state of its world. The user could restore his world to the saved state if he made an error or simply wanted to undo the effects of the example he gave. An alternate way of accomplishing the same task is to remember each command the user executes, and to provide a way to undo these actions, step-by-step. Such a facility would of course be generally useful, both during normal operation of the system, during program recording when the user wished to back up a step, and during the recording of an example the user later wished to retract.

However, in many systems, checkpointing is impractical, because large amounts of state must be saved. A complete undo facility may be similarly difficult to provide, since some operations might require saving too much state. In the real Star system, both checkpointing and undo are unimplemented. Checkpointing the contents of a user's desktop is possible, but one cannot checkpoint the entire state of the external things to which the user has access, such as objects stored on shared file servers. There is an **UNDO** key on the keyboard, but it presently does nothing. Implementing an undo mechanism in SmallStar would have been possible, but it would have taken quite a while, and so I did not choose to take the time. So I have not done any empirical tests on these ideas, and leave them for future research.

## **7.5 Adding programming by example and SmallStar techniques to other systems**

Programming by example is a concept which can clearly be applied to many different systems. The number of systems described in chapter 2 confirms this. However, if a system designer would like to incorporate programming by example into a new or existing system, there are a number of design decisions to make.

The first thing to consider is whether simple syntactic programming by example will suffice. For systems such as an operating system executive or a small data base system, a simple typescript mechanism such as EMACS keyboard macros (section 2.1), or a parameterized typescript mechanism such as the one used for Exemplary Programming (section 2.7) may be all that is necessary. Even for richly functional direct manipulation systems, syntactic programming by example may be quite suitable for use in creating animated tutorials and automating very simple tasks. Syntactic programming by example is especially good for reducing long, frequently used sequences of keystrokes to single commands. If some non-programmable action needs to be done in the program, such as choosing among a set of alternatives, the program can instead pause and let the user do it. As long as the user understands

that the mechanism is just substituting for the input devices of the system, syntactic programming by example is an effective

However, if the system designer would like to give the user a more sophisticated programming capability or if syntactic programming by example is not suitable, the recorded program must include some semantic information as well. The techniques used in SmallStar can be applied in many cases.

Consider, for example, a symbolic algebra system with a direct manipulation user interface. In such a system, the user can select parts of expressions, and ask that they be transformed in some way. For instance, the user could select the denominator in a rational function, ask for it to be factored, then select the whole function and ask that it be simplified by canceling like terms in the numerator and denominator. Data descriptions would be well suited for describing the expressions the user chose. Including sophisticated pattern matching in the data description mechanism would also be useful. Suppose the user were trying to move all terms containing  $y^2$  to one side of an equation; a data description could be used to describe the set of such terms.

Sophisticated data descriptions can also be used in more mundane applications, such as text editors. SmallStar provides relatively simple data description choices for text strings (see figure 4-16, which shows a text string description sheet). But a system whose major purpose was to manipulate text would provide many more ways of choosing text strings. The data descriptions in such a system would allow choices as varied as all the text between two patterns, the third word in the fourth paragraph, all text strings beginning with "Samuel", and ending with, but not including a carriage return, and so forth.

Data descriptions could also be used to augment syntactic programming by example systems, in order to increase their power. An operating system command language usually includes a pattern matching mechanism for file names. In a programming-by-example system built on such a language, the pattern mechanism would be used instead of data descriptions to select files. But if the mechanism were inadequate, data descriptions could supply more flexibility without changing the command language. However, the existence of two ways of searching for files, one in the command language, and one in the programming-by-example mechanism, would probably confuse the user, and so, if possible, it is probably better to augment the command language.

Data descriptions are a powerful notion by themselves, and do not always need to be used in conjunction with programming by example; they could be used in other programming systems that do not provide programming by example at all. For instance, one can imagine a system like SmallStar in which the user creates the program, not by example, but by typing it in or using a syntax-directed editor. Special data description objects would be inserted in the program text, and could be edited just as they are now in SmallStar. Such a system would not be as convenient to use as SmallStar, and would not provide the advantages of programming by example, but it would be as powerful.

A system with data descriptions but without programming by example might also be easier to implement. If programming by example is not designed into a system from the start, it may be difficult to implement. Intercepting the user's actions in order to record them, at a level other than the raw user input, may not be easy, and may require a considerable amount of recoding.

The *ex post facto* methods provided in SmallStar for adding control structure are also applicable to many systems. They could be used for both syntactic and semantic programming by example. The only requirement is that the system provide a readable static program representation, so the user can indicate where the control structure is to be inserted. Small pieces of programs could be recorded separately, and later combined with control structure. This would probably be more convenient than the current SmallStar method of requiring that everything be recorded in-line in a single program.

## 7.6 Programming by example and conventional programming

So far, programming by example has been discussed as an end unto itself, as a way of letting an ordinary user program the system he uses. But programming by example can also be used as a tool to teach conventional programming in a relatively painless way. If the system records the user's actions and shows the resulting transcript to the user in a static form (that is, in a programming language), then the user will be able to see that certain actions map into certain programming language constructs. Eventually, the user should be able to remember these mappings, and write programs himself without needing to use the program recording mechanism. If good user interfaces for conventional programming languages were built, programming by example might be an excellent way to teach beginners how to program.

Interactive programming systems such as Lisp, Smalltalk, and APL are close to this goal. They allow programmers to try out small pieces of programs quickly and, see what they do.

Programming by example could also be used as an adjunct to conventional programming, even by professional programmers. Many parts of a program could be written by example much more quickly than by hand. The programmer's time could be spent more constructively on the difficult parts of the program. Producing code by example would also reduce errors, since creating the code would include testing it on an example.

If a programming-by-example system were built for creating programs in some conventional programming language, data descriptions would probably not be included. Data descriptions substitute for code that would normally be included explicitly in a program written in a conventional language (see section 4.8, which discusses this point with respect to generalizations). However, the *ex post facto* method of adding control structure would be useful in such a system. Bodies of conditional statements and loops could be recorded separately and later merged into the program when control structure was added.

## 7.7 Conclusions

The research described in this dissertation was motivated by a long-term goal of mine: making computers easier to use. Making programming easier is a part of this goal; I want to enlarge the community of people who can program computers. In this research I have concentrated on programming by example, a technique that lets a user program a system in its normal user interface, by giving an example of what the program should do.

To explore the promise of programming by example, I incorporated it into a simulation of a richly functional system with a modern user interface. Doing so meant evolving new techniques, most notably data descriptions and the addition of control structure by program editing. Finally, I performed trials on potential users to see whether the resulting system could really be used and learned.

This research has demonstrated that programming by example is a viable method of programming by the user that can be applied to real systems. The techniques used in SmallStar are applicable to other systems as well. Programming by example can make the computers and programming more accessible to more people; that is its ultimate worth.

## References

- [Anderson]  
Bruce Anderson. Programming in the home of the future. *International Journal of Man-Machine Studies* **12**: 341-365 (May 1980).
- [Attardi & Simi]  
Giuseppe Attardi and Maria Simi. Extending the power of programming by examples. *ACM Conference on Office Information Systems*. Philadelphia, June 1982.
- [Bauer]  
Michael A. Bauer. Programming by examples. *Artificial Intelligence* **12**: 1-21 (May 1979).
- [Biermann & Krishnaswamy]  
Alan W. Biermann and Ramachandran Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering* **SE-2**: 141-153 (September 1976).
- [Brown]  
Peter Brown. Dynamic program building. Stanford University, Department of Computer Science. Report No. STAN-CS-81-838, February 1981.
- [Cuff]  
Rodney N. Cuff. On casual users. *International Journal of Man-Machine Studies* **12**: 163-187 (February 1980).
- [Curry]  
Gael A. Curry. *Programming by Abstract Demonstration*. Ph.D. dissertation, University of Washington, Seattle. Technical Report No. 78-03-02. March 1978.
- [Digital]  
Digital Equipment Corporation. *DECsystem-10 User's Handbook*. Maynard, Massachusetts, 1972.
- [Faught 1980a]  
W. S. Faught, D. A. Waterman, P. Klahr, S. J. Rosenschein, D. M. Gorlin, S. J. Tepper. *EP-2: A Prototype Exemplary Programming System*. Rand Corporation. Report R-2411-ARPA. February, 1980.
- [Faught 1980b]  
William S. Faught. Applications of Exemplary Programming. AFIPS Conference Proceedings. 1980 National Computer Conference **49**: 459-464. AFIPS Press, 1980.
- [Goldberg & Robson]  
Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [Gould & Finzer]  
Laura Gould and William Finzer. Programming by rehearsal. Xerox Corporation, Palo Alto Research Center, Palo Alto, California, Report No. SCL-84-1, May 1984.
- [Halbert 1980a]  
Daniel C. Halbert. Programming for non-programmers. Internal memorandum, Xerox Corporation, Office Products Division, Palo Alto, California, July 1980.

[Halbert 1980b]

———. Programming by the users: command recording in a prototype of Star. Internal memorandum, Xerox Corporation, Office Products Division, Palo Alto, California, September 1980.

[Halbert 1981]

———. An example of programming by example. Masters' report, University of California, Berkeley, June 1981. Also an internal report of the Xerox Corporation, Office Products Division, Palo Alto, California, 1981.

[Joy]

William N. Joy. An introduction to the C Shell. University of California, Berkeley, 1979.

[Kahan]

W. Kahan. Personal communication, 1980.

[Lieberman]

Henry Lieberman. Seeing what your programs are doing. *International Journal of Man-Machine Studies* (to appear).

[Lieberman & Hewitt]

Henry Lieberman and Carl Hewitt. A session with Tinker: interleaving program testing with program writing. *Conference Record of the 1980 LISP Conference*. Stanford University, August 1980.

[Mathlab]

Mathlab Group. *MACSYMA Reference Manual: Version 10*. Massachusetts Institute of Technology, Laboratory for Computer Science, January 1983.

[Nelson]

Greg Nelson. How to use Juno. Xerox Palo Alto Research Center, Computer Science Laboratory. Manuscript CGN-11, January 1984.

[Shneiderman]

Ben Shneiderman. Direct manipulation: a step beyond programming languages. *Computer* 16(8): 57-69 (August 1983).

[Smith]

David Canfield Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Ph.D. dissertation, Stanford University, 1975. Also published by Birkhauser Verlag, Basel and Stuttgart, 1977.

[Smith et al 1981]

David Canfield Smith, Charles Irby, Ralph Kimball, and Eric Harslem. The Star User Interface: An Overview. *Proceedings of the National Computer Conference*. Houston, 7-10 June 1982.

[Smith et al 1982]

David Canfield Smith, Charles Irby, Ralph Kimball, and Eric Harslem. Designing the Star User Interface. *Byte* 7(4): 242-282 (April 1982).

[Stallman]

Richard M. Stallman. EMACS: manual for ITS users. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, AI Memo No. 554, June 1980.



[Tanner]

William R. Tanner, editor. *Industrial Robots - Volume 1: Fundamentals*. Society of Manufacturing Engineers, Dearborn, Michigan, 1979.

[Vonnegut]

Kurt Vonnegut, Jr. *Player Piano*. Holt, Rinehart & Winston, 1952.

[Wang]

Wang Laboratories, Inc. *Word Processor Glossary Decision Processing User Manual*. Lowell, Massachusetts, 1978.

[Waterman]

Don A. Waterman. Exemplary programming in RITA. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern Directed Inference Systems*. Academic Press, 1978.

[Witten]

Ian H. Witten. Programming by example for the casual user: a case study. *Seventh Conference of the Canadian Man-Computer Communication Society*. Waterloo, Ontario, June 1981.

[Xerox]

*INTERLISP Reference Manual*. Xerox Corporation, October, 1983.

[Zloof & de Jong]

Moshe M. Zloof and S. Peter de Jong. The system for business automation (SBA): programming language. *Communications of the ACM* **20**: 385-396 (June 1977).

[Zloof 1977]

Moshe M. Zloof. Query-by-example: a data base language. *IBM Systems Journal* **16**: 324-343 (Fall 1977).

[Zloof 1981]

Moshe M. Zloof. QBE/OBE: a language for office and business automation. *Computer* **14**, No. 5: 13-22 (May 1981).