An Exploration of Programming by Demonstration in the Domain of Novice Programming

by

David Gilligan

A Thesis

Submitted to Victoria University of Wellington
in fulfilment of the requirements for the degree of
Master of Science in Computer Science

August 14, 1998

Abstract

Learning to program is hard. Novice programmers have difficulty translating a task's specification into behaviour for the computer that will achieve the task, and translating way the computer should behave into instructions that will make the computer achieve that behaviour. Novices are hindered in both these steps because they lack a good model of the way the computer carries out the instructions in a program.

This thesis presents an environment designed to provide novices with a good model of the way a computer carries out a program, and to support them as they learn to program in an imperative language. The environment makes the underlying state and operation of the computer and program explicit. The environment allows the novice to program by demonstrating how the computer should behave. The environment then shows them the instructions that would make the computer behave that way.

The thesis presents two models as the basis for the environment: a model of computation, and a model of programming. The model of computation uses a coherent set of analogies from the real world to explain the way a computer carries out a program. The model of programming extends the model of computation to include programming using Programming by Demonstration. The thesis also presents two prototypes of the environment, and an exploration of the models and environment using the prototypes. The thesis presents many issues for a Programming by Demonstration environment for novices that this exploration shed light on.

Acknowledgements

None of this would have been possible without the following people and institutions:

Pondy, who was a better supervisor than I could ever have hoped for. I want to thank him particularly for his guidance and support, for making me explain the things that made no sense to anyone but myself, for reading drafts that no one should have to read, for his unique and indecipherable writing, for being interrupt driven, for bearing with my wildly optimistic deadlines estimations, and for an office with more interesting clutter (including the Weetbix) than than anyone else I know.

My family, who provided food, shelter, and a car (when they weren't using it), and for having a lot of patience with the hours that I kept.

The staff and students of the Computer Science at Victoria, who created a great environment to work in, and for the employment that enabled me to keep in salt & vinegar chips and chocolate. Special thanks to those who discussed this work with me, offered advice, and let me scavenge through their collections of papers, particularly: Robert, Ewan, Mimi, and the SERG and AI groups.

DrinksCorp, who provided the steady supply of salt & vinegar chips and chocolate.

Special thanks go to those others who kept me sane through all this:

The Botch-It Boyz and their den of grease and iniquity in the Land of Darkness and Shadows$^{(TM)}$, for offering some small escape. To the other dwellers therein: Trouble the aptly named chair stealing cat, the pizza box monster, the coke bottle hive, 3Dfx cards, and bell south specials with all the knowledge of physics, chemistry, carpentry and engineering that they entail.

Fenella, for the much needed advice during the twilight times, and (along with Tim) for daring to read some of my early gibberish.

Kit, for letters from afar that told of life on the outside, and for the occasional game or two.

VUWSF, all those who've put on plays by Mr. W. Shakespeare, and the Incredibly Strange and International Film Festivals, for much needed doses of Kulture$^{(TM)}$[1]

[1] Finally, thanks to whoever invented the footnote.

Contents

Chapter 1

Introduction

This thesis presents an exploration of Programming by Demonstration in the domain of novice programming. The goal was to create an environment that supports novice programmers as they learn to program and helps them understand the way a computer behaves when it is carrying out a program.

1.1 The Problem

Understanding how a computer works is hard. Getting a computer to perform a task can be described from three different perspectives: the specification of the task, the behaviour of the computer as it achieves the task, and the program to achieve the task. The specification describes what the program has to get the computer to accomplish. The behaviour describes how the computer will accomplish the specification. The program determines the behaviour of the computer as it carries out the program's instructions. For example, suppose the task is to get the computer to display the numbers from 1 to 1000 (the specification). A program that would achieve this specification is (there are many): "put the value '1' in a place, while the value in that place is less than or equal to 1000, print the value in the place and increment the value in the place". The behaviour is something like: set a counter to '1', print the counter, increment the counter, print the counter, increment the counter, etc. Relating these three different perspectives correctly is part of the complexity of understanding how a computer works.

Programming is hard. Learning to program is harder. Programming involves two inter-twined steps that connect the perspectives: (1) designing behaviour that will achieve the specifications, and (2) translating that behaviour into instructions that will generate that behaviour. Experienced programmers often carry out both steps at once; however, this is extremely difficult for novices. The difference between what the programmer wants the computer to do, what the programmer specifies for the computer to do, and what the computer actually does causes even more trouble for novices than it does for experienced programmers (Lieberman & Fry 1995).

Novices have trouble translating the specification of the task and how they might solve it into a program that they can get the computer to execute. Designing instructions for the computer to achieve a task is not trivial because the computer does not operate and reason in the same way that a human does. A novice may know the desired behaviour, and may even be able to show someone what they want the computer to do, but they can still have great difficulty coming up with the loop, the counter, the print statement, and the correct relationship between those components that will produce the desired behaviour.

One reason why this is hard is that novices do not have a good model of the way a computer works, how it carries out instructions, or what the instructions mean to the computer. Without an accurate model of the way a computer behaves, they cannot predict what the computer will do with an instruction. The behaviour of a computer and the program that specifies that behaviour are quite different, and the mapping between them is not straight-forward. Thus when they come to design their own programs, novices cannot easily come up with instructions that will produce the desired behaviour.

Novices are often taught with models that give incomplete, contradictory, or incorrect analogies for the computer's behaviour. These analogies can confuse novices and unintentionally give them the wrong idea about how the computer behaves (Erickson 1995). Common programming concepts affected by this kind of problem include variables, parameters, and recursion. For example, one common (and misleading) analogy is that a variable is like a box with a piece of paper in it, and the piece of paper has the variable's value written on it. Thus, thinks the novice, if you put another value into the variable, the box will contain *two* pieces of paper, and so the computer can access both values. The novice then proceeds to write instructions that they think will use the variable's first value when appropriate. These instructions are likely to be incorrect because in reality once a second value is put into a computer variable, the first value is lost.

Novices are expected to form a conceptual model of the computer's behaviour. However, they have difficultly making the link between the dynamic behaviour and the static instructions because traditional programming environments do not let the user see the insides of the computer as it carries out a program. Thus, they cannot see what the computer is doing with their instructions.

1.2 The Solution

The solution presented in this thesis is a programming environment that shows the program's data and the internal state of the computer as the program is carried out and allows novices to construct their programs by showing (demonstrating) the computer what they want it to do. The state of the computer and the program's data are made explicit so that the computer's behaviour is clearly visible. The environment allows the novice to program by directly manipulating the state of the computer to demonstrate how it should behave. The computer will watch what the novice does and record the instructions that would cause the computer to carry out the novice's actions. The environment allows the novice to learn to program much more gently — doing just the first step of programming: designing the computer's behaviour to achieve the specifications. The novice should not have to worry about the syntax of the instructions — the environment should enable the novice to concentrate on understanding the mapping between the computer's behaviour and the instructions. The environment allows the novice to make mistakes and encourages exploration by allowing the novice to undo any action they make.

Constructing an environment like this requires two underlying models: a model of computation, and a model of programming. The model of computation is a view of the computer that describes how it carries out the instructions. The model of programming describes how the novice will use demonstration to program in this environment.

The model of computation should provide analogies that accurately describe the components of the computer and their behaviour at the right level of detail for understanding programming. Its aim is to help novices scaffold from things they already know about to an understanding of the way a computer behaves when it is carrying out a program. A good model will help the novice's understanding of the computer and allow them to accomplish the second step (translating behaviour to instructions) more easily. Coming up with analogies based on things that people already know about, that fit together in a reasonable way, and still accurately describe the computer and programming language is not a simple task. The analogies have to be clear enough so that a novice can understand the computer through them, and accurate enough to avoid misleading the novice.

The model of programming should combine the model of computation with programming by demonstration to provide a description of how to program in this environment. In programming by demonstration, the programmer shows the computer an example of what they want it to do, the computer watches the programmer's actions, makes some inferences about them to generalise them, and records the results as a program. The aim here is to use programming by demonstration to help novices learn by having them do what the computer would do, and to remove the burden of translating the desired behaviour into program instructions while they are learning. The model needs to describe how the novice can manipulate the components of the computer to demonstrate actions, and how the computer translates those actions into instructions. Coming up with actions that are appropriate for novices to use for demonstrating behaviour, that map to instructions in a computer program, and that enable novices to understand what a computer does when its carries out instructions is not a simple task.

1.3 The Results

Two prototypes of such an environment were implemented to explore this solution. The two models were developed alongside the prototypes: the model of computation (the abstract machine model), and the model of programming (the demonstration model). The first prototype implemented a core subset of an imperative programming language, and served as a starting point for the models. The second prototype elaborated on the first, and experimented with different analogies and techniques for the models. The prototypes were used as testbeds for exploring the models and various demonstration techniques. They served to uncover a number of conflicts and issues between the models and the environment. The issues and trade-offs that arose during this work are discussed in the thesis. Some were resolved, and the others had several possible solutions but no clear "best" solution.

The abstract machine model consists of a set of real world analogies for the components of a computer and program. It describes how a computer carries out a program at the level of the programming language. The model uses a general purpose imperative programming language for the basis of its view of the computer (a tidied up version of `Pascal`). The model is influenced by our desire to focus on the kinds of instructions in the programming language (its ontology) and what the instructions mean (their semantics), rather than on the syntax of the language. The model is also influenced by the goal of ensuring that it handles some common novice misconceptions, and does not introduce new ones. Our personal view of programming style and what a novice programmer needs to know also affected the model. The model itself serves as the basis for the environment. It determines the objects in the environment, their appearance, and the way they relate to each other.

The demonstration model determines the way that the novice (programmer) interacts with the objects in the environment to create a program. The model helps the novice to understand how the computer operates by forcing the novice to demonstrate actions in the same manner as the computer would carry those actions out. The model consists of a way of programming and a theoretical machine that supports this way of programming — the demonstrational machine. The demonstration model uses the abstract machine model as the basis for a visualisation of the computer, and makes use of techniques for directly manipulating the computer's components to demonstrate code. The demonstrational machine serves as the basis for the programming environment. The demonstrational machine extends the abstract machine by adding the notions of the demonstrator (a novice programmer), and the recorder, who records the demonstrator's actions in the form of a program. In this it differs from other programming by demonstration methods that use the notion of the novice *teaching* the computer to perform a task, which the computer remembers and carries out later. In our model, the recorder is a friendly expert that translates the novice's actions into a program that the (non-expert) computer will carry out later. The demonstration model also differs from other programming by demonstration systems because the recorder does not do any inference to turn the demonstrator's actions into more general instructions.

1.4 Outline of the Thesis

This thesis presents an exploration of programming by demonstration in the domain of novice programming. The rest of this chapter describes some of the related work in areas relevant to this work. Chapter 2 details the abstract machine model that provides real world analogies for the computer's behaviour when it carries out a program. Chapter 3 introduces the demonstration model, the demonstrational machine, and the two prototypes of the environment. Using the two prototypes, it presents three detailed examples of the way of programming supported by the demonstration model. Chapter 4 describes the demonstration model and prototypes in more detail, and discusses issues that arise from the differences between them. Chapter 5 discusses larger issues for the models and environment. Chapter 6 presents a preliminary evaluation of programming with each prototype. Chapter 7 describes the implementation of the prototypes, and points out some of the pitfalls in implementing this kind of environment. Finally, Chapter 8 presents the conclusions and contributions of the thesis, along with some possibilities for future work.

1.5 Related Work

This thesis touches on several different fields of computer science. Rather than give a survey of all the relevant work in all the fields, this discussion is limited to the most useful and significant work. There are pointers to other work that was not necessarily directly relevant, but was nonetheless consulted, in the the bibliography at the end of the thesis. The related work is loosely divided into four areas: computer environments for teaching novices, end-user programming, programming by demonstration systems, and calculators and their models.

1.5.1 Computer environments for teaching novices

There are many approaches to teaching programming. Some of them seek to use computer environments to help novices learn. One group of learning environments are the so called 'Intelligent Tutoring Systems'. The aim of these kinds of systems is to construct a computer tutor that guides a student through exercises, detecting the student's mistakes, and offering advice and help. We have taken a different approach: our environment is not a system that guides and offers advice to the novice, rather it is an environment that lets novices explore freely.

Eisenstadt, Price & Domingue (1993) describe four fallacies that typically befall intelligent tutoring systems and their designers:

1. Trying to generate a complete bug catalogue so that the system can provide a solution to any problem the novice might create.
2. Poor user interfaces that create a barrier to learning.
3. Working only on toy examples and not scaling well to larger examples.

4. The “we know what the best way to learn is” fallacy.

Though these fallacies primarily apply to intelligent tutoring systems we believe that they should also be avoided by other novice support environments. Our approach avoids the first problem because our environment is not an intelligent tutoring system. The prototypes described in this thesis do suffer from the second problem because we chose not to focus on their user interfaces. Although we did not address them specifically, we believe that our system will satisfactorily cope with the third and fourth problems, but we cannot be sure without further evaluation.

Eisenstadt et al. (1993) object to typical intelligent tutoring systems. They believe that any distinction between learning to program and the ‘design/edit/run/debug’ cycle is artificial and not a good idea. Through their work on various intelligent tutoring systems they came to the view that all novices really needed was to see what was going on inside the computer, in an environment that allows exploratory learning. They present software visualisation as the key to avoiding the fallacies and creating a useful learning environment. We agree with their point of view, and our general approach matches theirs even though we are building a programming by demonstration environment and not just a programming environment that uses software visualisation.

Pane & Myers (1996) present a broad coverage of usability issues that the designers of novice programming environments should take into account. The article contains plenty of examples of novice confusions and the causes for them. They cover many issues but only briefly discuss each one. The issues that were particularly important for this work are:

- Supporting incremental running and testing with immediate feedback.
- Choose an appropriate metaphor.
- Consistency with the metaphor.
- Consistency with external knowledge.
- Supporting direct manipulation and definition by example.
- Avoid requiring premature commitment to code.

Note that although this paper was read late in the course of this thesis, our work is consistent with the way they recommend addressing these problems and other common novice confusions.

Programming environments are a critical part of learning to program for novices. The programming environment that a novice learns on will colour their first experience of what it means to program. The main problem with programming environments in Lieberman & Fry’s (1995) view is that they do not allow the user to easily view the past and present states of the program. Making the underlying state visible and unwindable are not new ideas, but most programming environments do not support them. Lieberman & Fry (1995) present a program debugging environment called ZStep94 — a visual debugger that shows the behaviour, code, and keeps a complete execution history (ZStep95 — a later iteration of the system — is described in Ungar, Lieberman & Fry (1997)). The user can display values in the state by selecting a piece of code and the system overlays the code with a window that shows the value. However, ZStep94 does not show all the values in the current state at once. This has the advantage of allowing the user to pick out the parts that they deem important and prevents information overload. The user can click on a point in the program that they want to see and the code will step back or forward to that position. The user can also look at the previous values of a particular expression to see the expression's evaluation history.

ZStep94 is a debugging environment for programmers, who presumably know how the computer operates. If programmers are aided this by kind of environment (as Ungar et al. (1997) argue that they are) then it will certainly help novices. Our environment is different — instead of just showing certain values in the state, it shows all the whole state as the computer carries out the program. The environment uses our model of the computer's operation to make all data and operations explicit so that the novice can see what happens inside the computer as each instruction is carried out.

Programming is a complex process that involves two often intertwined steps: working out how the computer should behave to achieve the task, and constructing instructions that tell the computer how to behave. Doing both steps at once is hard for novices.

Soloway (1986) presents an approach to teaching novices to program by teaching them to use and combine useful components of algorithms and chunks of code (partial plans). Soloway says that novices do not know the useful plans to use and are not able to merge the ones they do know about correctly to get the computer to achieve the task. Expert programmers often carry out simulations (on the computer or in their heads) of their programs and plans to find and fix weak points in their designs. The plan-based approach is to teach novices these partial plans (a counting loop, a sentinel loop etc.) and provide an environment that lets them combine the plans and see the results (Soloway 1986, Spohrer & Soloway 1986).

This approach gives the novice the pieces and lets them work out the order. In effect, it attempts to simplify both of the steps involved in programming by giving the novice a higher level language to work with, the hope being that they will learn the way the computer works and the problem solving skills necessary to program along the way. Teaching problem solving skills using partial plans is important for programming, however this thesis takes a different approach.

Our work takes a lower level view: it takes care of the instructions (the second step) and lets the novice concentrate on how the computer should behave to achieve the task. What the novice needs is a good model of the way the computer operates. They need to see the underlying state and how the computer carries out the instructions in order to understand the computer's operation. We want to teach them this model by making it explicit and allowing them to demonstrate what they want the computer to do. They should see the results of their actions, and the code needed to make the computer carry out those actions, the hope being that they will make the connection between their actions and the instructions generated by the computer. Making the state explicit and easily manipulated backwards and forwards is critical to this. The novice should be able to see what is going on to notice and explore solutions for their mistakes incrementally. They will be able to see where and how their ideas about what the computer needs for the program are going wrong.

1.5.2 End User programming

A related area to novice programming systems is end user programming. Instead of making an environment that supports people as they program or learn to program, the goal of end user programming is to give programming power to non-programmers through applications that allow the user to customise the applications for their own ends without having to teach them programming (Nardi 1993). Spreadsheets and CAD systems are examples of applications that often have this kind of power built into them.

One environment that aims to bring the power of programming to people using computers is Boxer (diSessa & Ableson 1986). Boxer is based on doing away with the distinction between the operating system and the application, between the programmable and the non-programmable. The environment is built on `SmallTalk` hierarchy of objects in a visual medium that aims to make programming more accessible to the user. It does this by making every part of the computer environment a box-like object that can be opened, viewed, and altered to suit the user needs.

Cypher (1993b) contains a score of systems that use programming by demonstration techniques for end user programming in various domains. Despite the fact that end user programming is a common domain for programming by demonstration, the work in this thesis is not about end user programming. Our focus is solely on turning novices into programmers[1]. It would be a stretch to call this work end user programming, even if one extended the usual definitions of an "end user" to be a novice programmer, and "customising their environment" to defining new subroutines for their programs.

1.5.3 Programming by Demonstration systems

In "Watch What I Do" (Cypher 1993b), Myers (1993a) presents a taxonomy of the different kinds of programming by demonstration systems. The paper gives a brief overview of the area of direct manipulation, programmable, and demonstrable systems. He classifies programming by demonstration systems ("programming by example" in his terminology) across three axes: demonstrational or not, programmable or not, and intelligent or not (does it do inferencing). Although the paper seems to present each category as either one thing or the other, *e.g.* either a system is programmable or it isn't, he does recognise

[1]Real Programmers$^{(TM)}$ in the more traditional Computer Science sense.

that the axes are really a continuum. According to his terminology, the prototypes of our demonstrational machine environment are roughly demonstrable, programmable, but not intelligent. The demonstration model and prototypes make more use of what he calls "programming with example" than "programming by example", the difference being the amount of inference the system uses — our environment does not do any but the most trivial inferencing.

There are quite a variety of programming by demonstration systems. One variety focuses on making common repetitive tasks easier. For example, Eager is an intelligent agent system that watches as users perform tasks and tries to work out what they are doing (Cypher 1993a). Eager never asks users for any information, it just anticipates what they will do next and gives them visual feedback of its guess. When users are satisfied that Eager is anticipating correctly, they can get it to complete the task for them, and produce a shell script to do it in future. Eager operates independently of the application that the user is working in. Other programming by demonstration systems of this variety operate in specific applications such as text editing (Witten & Mo 1993) and graphical editing or drawing (Kurlander 1993, Lieberman 1993a).

The most well known variety of programming by demonstration systems are the graphical user interface builders that help the designers build interfaces without having to program. Notable systems include: Peridot (Myers 1993c), Garnet (Myers 1993b), Marquise (Myers, McDaniel & Kosbie 1993), and Pavlov (Wolber 1996).

Peridot is a system for laying out graphical user interface components (Myers 1993c). It produces code from one example, generalises by inferencing using action rules and abduction with domain specific knowledge to make it manageable. Its influence can be seen in its successor Garnet — a collection of tools for building various parts of graphical user interfaces that make use of various demonstrational techniques (Myers 1993b).

Marquise is another system that focuses on demonstrating graphical editors and the interactions that their users can perform (Myers et al. 1993). It uses icons to give a representation of the actions that the mouse performed during a sequence of events. It tries to get the inference right on the first demonstration, but the user can change the data if the system guesses incorrectly.

Wolber (1996) and Wolber & Fisher (1991) present programming by demonstration systems (Pavlov and its predecessor DEMO) that use a stimulus-response model for building interactive (animated) graphical user interfaces. The systems make use of techniques for demonstrating continuous updates in response to time related events, for example, performing linear equation generalisations to determine the dynamic motion of a dial rotating, when its movement is demonstrated as a response to moving a slider.

The systems described so far (and the majority of the other systems in Cypher (1993b)) are not closely related to our system because they use programming by demonstration for variety of end user programming goals: constructing user interfaces by demonstration, text editing by demonstration, drawing and graphical editing by demonstration, and automating common interface tasks involving the mouse. Our system differs — it aims to support novices who are learning general purpose imperative programming.

1.5.3.1 Closely related Programming by Demonstration systems

The programming by demonstration systems that are close to our demonstrational machine environment include: Pygmalion (Smith 1993), Tinker (Lieberman 1993b), Rehearsal World (Finzer & Gould 1993) and Leogo (Cockburn & Bryant 1996) a hybrid programming system that incorporates a programming by demonstration component.

Pygmalion is claimed to be the first programming by demonstration system and the first system to use icons (Smith 1990a, Smith 1993). Pygmalion is a kind of sketching board for experienced programmers. It allows the programmer to create executable pictures (icons) that represent parts of their program by drawing them and combining them with icons for built-in operations. The program is demonstrated by the programmer editing their graphical sketches. The system automatically carries out completely specified icons (operations), and pauses at incomplete points where what to do hasn't been specified and waits for the programmer to show it what should be done. Pygmalion requires the programmer to think ahead and demonstrate actions in the right order, it does not have unwind or undo. Pygmalion does not use any particular model of the computer, rather it is a workspace for ideas and modelling on which the programmer can create any graphical model they like. It was from Pygmalion that we saw the need to handle partial instructions and programs. Our target users (novices) and interactions are different to those of Pygmalion.

Another programming by demonstration system that is related to this work is Tinker, an environment for novices learning general purpose programming in `Lisp` (Lieberman 1993b). Tinker allows the programmer to produce `Lisp` programs by providing examples of input, and then directing what the system does with the input. The programmer can select from a series of subroutines or type code directly. Tinker works out the results of each operation and displays them. Tinker doesn't use inference or do much generalisation, instead the programmer selects from a list of objects. However, it can detect when the program requires a conditional statement by using multiple examples. Tinker's method for demonstrating conditional statements is discussed further in section 4.3.5.1.

Our environment is like Tinker because it does general purpose programming, but Tinker does not present a model of computation, or let the programmer view and manipulate the internals of the computer. Our system does not recognise any programming constructs from multiple examples, but it does require multiple examples to demonstrate each branch of a conditional statement. Tinker also seems to require that its user know a bit more `Lisp` than a novice might reasonably be expected to know because Tinker's user is allowed (and sometimes required) to enter arbitrary `Lisp` expressions.

Rehearsal World (Finzer & Gould 1993) is a system that facilitates the building of educational (and other) programs by (non-programmer) designers and educators by giving them programming power. It is strongly based on a metaphor of the program as a play and every object in the program as an actor or a part of the stage on which the play is performed. To use the system, the designer leads the objects (actors) through the events and operations that should happen in the program (they rehearse the play). The user cannot interact with the code: everything is done by watching and recording. Rehearsal world focuses on the interface to the program and presents the components of the system in a purely object oriented fashion. It is close to a macro recorder because it does not use inferencing, it only allows single actions (method invocations) to be demonstrated, and no program constructs can be demonstrated. It just records the `SmallTalk` code for the demonstrated actions (invocations). However, it does show some of the internals of a program, though its model of computation is higher level than ours because it is designed to shield designers from the machine within. Its model is also quite different to ours because it is based on the `SmallTalk` object hierarchy.

The Leogo programming environment (Cockburn & Bryant 1996), based on `Logo`, is an attempt to combine three programming interaction paradigms in one environment: text editing, direct manipulation using programming by demonstration, and what they called "iconic programming" that is based on using buttons, sliders, and other widgets. The environment attempted to keep all three paradigms consistent so that the user can choose the one that most suits them for a given task. They encountered a number of difficulties in the process and had to make some compromises. Particularly interesting is that Leogo's programming by demonstration component was quite limited, little more than a macro recorder: no conditionals, parameters for procedures, no ability to call procedures, and it did not generalise or use inference. `Logo` is simpler than a general purpose programming language, and Leogo's target was school children. Our system goes further with programming by demonstration because it allows conditionals, parameters, and procedure calls. However, the interfaces on our prototypes uses a similar style of interface to their "iconic programming" component, though our prototypes are less restrictive than their interface.

None of the systems described in "Watch What I Do" (Cypher 1993b) make any distinction between the computer as the semi-intelligent recorder/observer of the user's actions, and as the unintelligent/literal agent that carries out the programs. Some systems use a teacher and student metaphor to describe the relationship between the user and the system (Maulsby & Witten 1993, Lieberman 1993b) — the user is *teaching* the computer to perform some task. Others simply assume that the computer is watching the user with the intent to produce a program for them (Cypher 1993a). The demonstration model presented in this thesis differs from all these systems because it makes a distinction between the two roles that the computer plays in programming by demonstration. The distinction is made in order that the model of how the computer works is clear to the novice. This avoids one of the observations that Pane & Myers (1996) make about the way novices can be confused by the role of the computer as program manager and program executor in programming environments.

1.5.4 Calculators: Users' Models and Inconsistencies

Programming involves constructing formula and mathematical expressions for the computer to evaluate. Part of our work was coming up with good ways of constructing expressions, and analogies for how the computer works out expressions. A seemingly obvious and good model for expression construction and evaluation is the pocket calculator. For example, MiraCalc (Goldson 1994) uses a calculator-like interface for teaching a functional language (`Miranda`) and logic programming, though it is far more like an evaluation and programming shell than a normal calculator.

However, using a calculator-like analogy is not as straight-forward one might think. Using four-function calculators, Mayer & Bayman (1981) looked at the differences between users' models of how a calculator works and the real model that the calculator operated on. They found that people came up with many different (and bizarre) models, and not all the users understood the calculators well. To make the user's view match the way the device operates, they recommend (among other things) making the underlying state and model visible, and matching the user intuitions and assumptions.

Young (1981) describes different kinds of models for explaining the inner workings on three kinds of calculators: four function, reverse polish notation or stack based, and algebraic calculators. Each kind of calculator operates slightly differently and requires a different model to explain its inner workings. Additionally, each kind of calculator suffers from its own internal inconsistencies and problems. Four function calculators mix up the order values should be entered in when calculating reciprocals. Stack based calculators have two unrelated uses for their 'Enter' keys: signalling the end of the input, and shifting the (hidden) stack down to duplicate the bottom value. Algebraic calculators do not properly calculate expressions involving unary operators when they are entered in the order that they are written down. The fact that most calculators hide their inner workings compounds these inconsistencies and makes their model of operation difficult to understand.

These papers show that using a pocket calculator to evaluate or construct an expression is not straight-forward in reality. For our work, this suggested that using a normal pocket calculator as a demonstration tool for constructing an expression would probably not be easy. This proved to be correct. Each of our two prototype environments explored a different method for constructing expressions: the first was based more closely on a calculator interface, and the second was based more closely on the computer's evaluation method for expressions. Neither was particularly satisfactory for demonstrating expressions.

Chapter 2

The Abstract Machine Model

The first underlying model for our novice programming environment is a model of computation. The environment will use this model to visualise the state and operation of a computer carrying out a program. This chapter describes a model of the computer's operation that uses analogies to link the parts of the computer to objects in the real world. This model is known as the abstract machine model, or just the machine model, to distinguish it from the demonstration model (the second model) described in chapter 4. The machine model provides the basis for the visualisations and interactions that are used in the prototype environments.

The aim of the machine model is to make it easier for the novice to understand how the computer carries out instructions. The model provides analogies that the novice can use to scaffold from what they know about to help them understand the way a computer behaves when it carries out a program.

Section 2.1 outlines biases in the model's construction. Section 2.2 presents a general overview of the model and analogies for the components of the computer. Section 2.3 presents the model's analogies for variable, parameter, and constant declarations. Section 2.4 presents the model's analogies for instruction statements. Section 2.5 presents analogies for several programming concepts that were not fully developed for this model (*e.g.*, parallel programming). Section 2.6 describes the programming language used in the model. Section 2.7 discusses how well the model addresses some confusions common to novices.

2.1 Biases in the model

The machine model is biased by three factors that we deem important: emphasising ontology over semantics and syntax, common confusions that novices have about programming, and a personal view of what novice programmers do and do not need to know. Each bias is described below.

2.1.1 The importance of the programming language's ontology

Our view is that the syntax of a programming language is not as important as the semantics, and that the semantics are not as important as the ontology. The ontology of a programming language lists the kinds of things, the concepts and objects, that the language makes distinctions between, *e.g.* variables v.s. parameters. The most important thing is to make the ontology explicit in the model, and then to make the semantics follow naturally. The syntax is not important.

2.1.2 Common novice confusions about computer programming

Another source of bias in the model is our attempt to address the most common confusions of novices. People learning to program often misunderstand certain programming concepts. Pane & Myers (1996) describes many examples of the misunderstandings made by novice programmers. Problems result from these misunderstandings so frequently that special care was taken to create good analogies to avoid the misunderstandings. The problems singled out for special attention were:

Confusion over variables

This confusion stems from the common metaphor that a variable is a box with a piece of paper in it. A consequence of this metaphor is that you should be able to pick up the pieces of paper and move them from box to box. This causes problems because variables in a computer's memory are not quite like this.

One misconception arises from the fact that the types of the values are ignored by this metaphor. The values are all on pieces of paper, so why can't any box hold the value from any other box?

A second misconception is that swapping the values in two variables can be done with just the two variables. You simply swap the pieces of paper in the two boxes with two assignments. When their program to do this fails to work properly, novices are often confused as to why.

Another typical misconception arises over empty boxes. Is a newly created box empty, or does it have a blank piece of paper in it? Does a box lose its value and become empty when its value is assigned to another box? In either case, it seems that a box can sometimes contain no piece of paper and hence no value. However, a variable always has a value. A new variable's value may be arbitrary and meaningless, but it is still a value.

Confusion over passing parameters

There are two confusions surrounding passing arguments to subroutine calls (procedures and functions). One confuses passing an argument by value and passing an argument by reference. The other confuses a local variable in the calling scope (actual parameter) and a formal parameter with the same name.

Problem of recursive calls to the same code.

Another problem related to scope is understanding recursion. Recursive calls to subroutines confuse novices about where the program is up to, which variables are being used, and how the computer carries out a program with a recursive call.

The computer can only consider a few values at a time.

Computers cannot easily do the kind of fast visual processing that humans can. This applies especially to arrays. The computer cannot look at an entire array, or a group of variables, all at once and pick the value it wants for the current task. It has to consider the values one at a time. For example, to find the smallest number in a list of numbers the computer must consider each in turn.

The computer has no common sense.

People can equate the speed and complexity of what a computer can do with the computer being able to reason like a human and "just do things". In reality, a computer can only do certain things, and it carries out instructions without thinking about them. It does not operate the same way that we seem to operate. A computer cannot do anything without specific instructions on how to do it.

The model attempts to provide analogies that minimise the confusions, and provide good explanations for the way the computer operates.

2.1.3 Personal bias towards programming (Comp 102 style)

Another source of bias in the model is our view on how programming should be learnt, what things people need to know to program, and how people should program. This view is taught in the first year computer science courses at Victoria University of Wellington, Comp 102 and Comp 103. It includes such things as: you do not need any global variables, all constants should be global, abstract data types should always be used, and program design and style are as important as a program working correctly. These ideas affect the programming concepts and machine details that are included in the model.

2.2 The model of computation

The model contains a set of analogies that match programming concepts to things in the real world. The purpose of these analogies is to make it easier for someone learning to program to understand the abstract concepts that are part of computers and programming. The model consists of an abstract machine, a `Pascal` based imperative language for the instructions the machine carries out, and a set of analogies to link the abstract machine and the real world. There are analogies for the components of the machine, for the way the machine carries out the programs, and for the instructions in the programming language.

The model is designed to explain how a computer carries out a program, *not* how a compiler or interpreter handles program problems to do with syntax, scoping, or style. The model provides little support for these things; it assumes that the computer has a correct and complete program to work with. Only teaching novices good design habits and skills will give them this kind of support.

The machine model was the basis for two prototypes of a new programming environment, which will be described in chapters 3 and 4. The model was improved after each prototype environment was constructed. The model presented here is the machine model as it stands after the completion of the second prototype.

2.2.1 General overview of the analogical model

The model personifies the flow of control in the computer as a clerk. The other parts of the computer become machines, calculators, devices, shelves, and a clipboard that the clerk uses to carry out the program. The clerk and the other parts of the computer are in a room. The clipboard holds worksheets (activation records), which like tax forms, contain places to put values and instructions to be followed. The clerk follows these instructions in order and to the letter. The clerk knows how to carry out the instructions on the worksheets and operate the devices in the room. However, the clerk has no common sense, initiative, or short-term memory.

The clerk uses the devices in the room to carry out the instructions on the worksheets. The clerk knows how to use each of the devices in a limited and highly structured fashion. There are several calculators, I/O devices, and a machine called the worksheet machine. The clerk uses the calculators to work out the values of formulae and to do calculations. The clerk uses the I/O devices to communicate with the computer's user, who is outside the room. The clerk uses the worksheet machine to print worksheets.

The clerk carries out the instructions on a worksheet in order. In doing so, the clerk records values on the worksheets, and uses the other objects to carry out some of the instructions. When the clerk gets to an instruction that it does not know how to carry out, the clerk goes to the machine that prints worksheets. The clerk gets the worksheet machine to print out a worksheet with detailed instructions on how to carry out the puzzling instruction.

The clerk puts the new worksheet on top of the other sheets on the clipboard, and begins to carry out the instructions on the new sheet. When the clerk has finished the instructions on the top worksheet, the clerk takes it off the clipboard, throws it away, and carries on working from the point it had reached on the previous worksheet. When the clipboard has no more worksheets on it, the clerk has nothing more to do, and the computer has finished the program.

2.2.2 The analogies for the components of the computer

Each component of the abstract machine has its own analogy with something in the real world. In order to keep the overall model as consistent as we can, and to reduce the amount of handwaving and magic[1], some of the analogies do not match the behaviour of their components as closely as they could.

2.2.2.1 The Clerk

The clerk represents the program counter, and most of the CPU. The clerk does not represent the computer's arithmetic and logic unit, or memory. The computer carries out a program when the clerk uses the clipboard, and carries out instructions on worksheets. The clerk cannot remember any values, and must write everything down. Exact instructions must be provided for the clerk to carry out a task.

2.2.2.2 Calculators

The calculators are the computer's maths processor, and logic unit. There are several calculators in the room that the clerk uses to work out the value of an expression. Each calculator specialises in a particular kind of calculation, one does numbers, another does logic, and so on.

2.2.2.3 I/O devices

The computer's case is the walls of the room. Outside the room are the computer's keyboard, screen, mouse, and the computer's user. These are connected to the three I/O devices inside the room. The I/O devices have some buttons, and either a slot, an L.C.D. screen, or both. Each device is explained below:

Input device

> The input device is connected to the computer's keyboard and reads keyboard input from the user. The device represents the computer's typing and input buffers. It has a small L.C.D. screen, some buttons, and a slot. Whatever the user types on the keyboard is displayed on the device's screen. To obtain a value the clerk pushes a button and the device dispenses a value on a piece of paper through the slot. The clerk uses the value and then throws the piece of paper in the garbage. Only values on the device's screen can be dispensed. Once a value has been printed, it is no longer displayed on the device's screen. The buttons on the input device correspond to built-in functions for reading different kinds of values.

---

[1] This use of 'magic' is the technical Computer Science term which means "something that you do not need to know about, but should accept that it works like this". In this sense, magic is indistinguishable from technology, if you have enough of either of them (to paraphrase Clarke's Law).

Output device

The output device is connected to one part of the computer's screen. The device represents the computer's standard text output stream. It has a slot and several buttons. The clerk uses the device to display text on the computer's screen. The screen is not visible from inside the room so the clerk cannot see what the screen displays.

Graphics device

The graphics device is connected to the other part of the computer's screen. The graphics device can draw simple graphics on the screen, and can obtain mouse input from the user. As well as a slot and some buttons, the graphics device has two L.C.D. screens. Each screen displays a number, and together the screens display the coordinates of the mouse pointer.

The buttons on the output and graphics devices correspond to built-in procedures in the language. The output device handles the text output procedures; the graphics device handles graphics procedures, and mouse procedures. The graphics device also provides two constants[2] that are continually updated[3] to reflect the coordinates of the computer's mouse.

To perform an operation, the clerk presses the button on the device that corresponds to the operation. The device produces a piece of paper from its slot. The piece of paper contains the name of the built-in operation, and places for the operation's parameters. The clerk fills in the details, and feeds the piece of paper back into the device's slot. Each device only accepts the piece of paper that it produced. The device carries out the operation, and then the clerk carries on.

For example, if the clerk pressed the `WriteNUMBER` button on the output device, the device would produce a piece of paper with a place for a `NUMBER`. The clerk would take the piece of paper and put it on the clipboard. Then the clerk would write a number in the place, say `13`, and feed the paper back into the output device's slot. The device would take the paper, reads the details, and display `13` on the computer's screen.

2.2.2.4 The clipboard

The computer's activation stack is the clipboard. The worksheets on it represent activation records. Putting a new worksheet on the clipboard is analogous to calling a new subroutine. Removing a worksheet and throwing it in the trash is returning from the subroutine call. The pieces of paper produced by the I./O. devices are partial worksheets. The clerk uses partial worksheets in same way as worksheets.

[2] Constants in the sense that the clerk cannot alter them.
[3] They are variable constants.

2.2.2.5 Worksheets and templates

The computer's memory consists of a stack of blank paper for printing worksheets.[4]. The worksheets are printed from master copies that are known as templates. The I/O devices and the worksheet machine contain templates. The templates for worksheets printed by I/O devices are built into the devices themselves. However, the templates for worksheets printed by the worksheet machine are provided by the programmer.

2.2.2.6 The worksheet machine and templates

The worksheet machine prints worksheets for programmer-defined subroutines. The templates for these worksheets, and templates for the types of places to print on them, are put into the back of machine by the programmer. Together, all the templates in the worksheet machine make up the program.

2.2.2.7 Buttons and Worksheets

Subroutines represent actions that the computer can perform. They are activated through buttons on the various devices and machines in the room. Some of the buttons are attached to the worksheet machine; these actions are defined by the programmer. The rest of the buttons are attached to the calculators and I/O devices; these actions are built into the computer and the language.

To perform an action, the clerk presses the appropriate action button. The effect of pressing a button depends on where the button is located, and is described in the following sections:

A button on the worksheet machine

Pressing a button on the worksheet machine causes the machine to print a worksheet for the subroutine associated with the button. Producing a folded worksheet is analogous to allocating space for the subroutine on the activation stack. A worksheet consists of two or three parts: a part containing places for the parameters and the subroutine's variables, and a part for the subroutine's instructions. If the subroutine is a function, then its worksheet has a third part at the bottom of the sheet containing a place for the return value (see figure 2.1). When the machine produces a worksheet, the worksheet is folded up so that only the parameter places are visible.

The clerk places the folded worksheet on the clipboard, works out the values of the call's arguments, then copies the values into the parameter places on the folded

[4] To properly model a finite amount of memory, the analogy should include a waste paper bin linked to a paper recycling plant. The recycling plant could recycle paper without loss. However, because there is only a finite amount of paper available for the computer's use, the computer can run out of paper. When that happens and the clerk asks for another worksheet, the computer will crash and it is time to buy more paper, ah . . . memory.

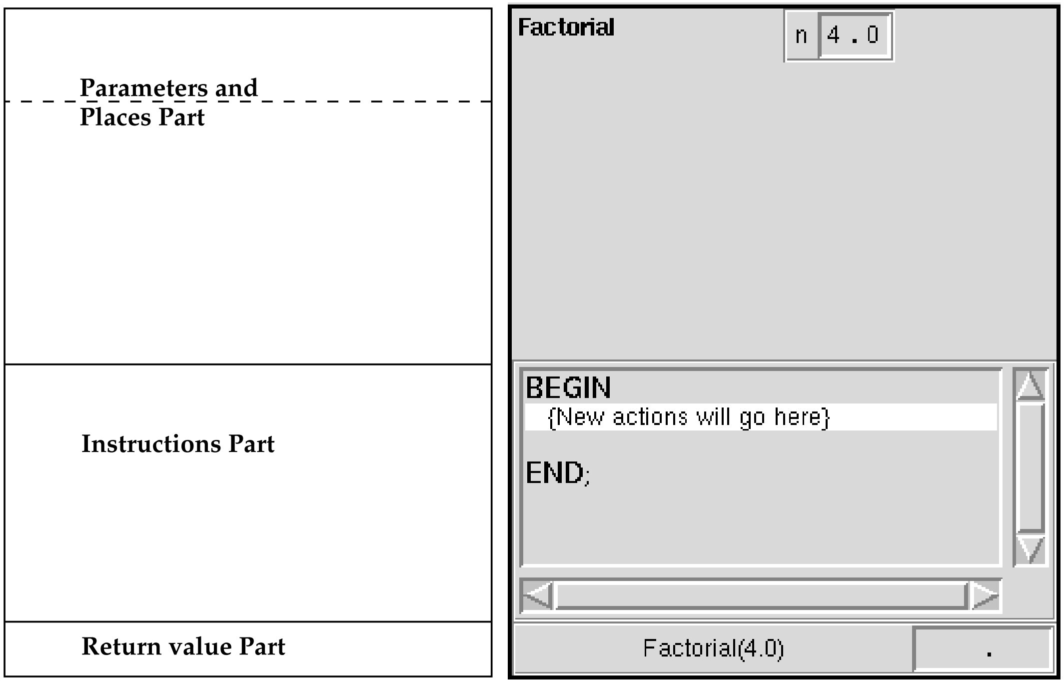


Figure 2.1: The layout of a worksheet with an example from the second prototype.

worksheet. Once all the parameter places are filled in, the clerk unfolds the worksheet, and places it on the clipboard. Unfolding the worksheet is analogous to calling the subroutine.

The clerk carries out the instructions on the unfolded worksheet until all the instructions have been completed. Then, if the worksheet is for a procedure, the clerk removes the worksheet from the clipboard and discards it. If the worksheet is for a function, the clerk folds it up so that only a return value place is visible (at the bottom of the sheet). This allows the clerk to copy the return value to another place before discarding the worksheet. Folding up and discarding the worksheet is analogous to returning from a subroutine call.

Subroutine worksheets are folded, unfolded, and re-folded in three stages to distinguish the three steps in the execution of a subroutine call: the invocation, the execution, and the return.

When a subroutine is first invoked, its worksheet is folded to hide its variables and instructions. The subroutine's parameter places are left visible so that the clerk can fill them. The clerk must be able to use values from places on the unfolded worksheet to fill in the invoked subroutine's parameters; a folded worksheet does not obscure the unfolded worksheet beneath it because the invocation is made in the scope of the unfolded worksheet.

When a subroutine is executed, its worksheet is unfolded to display the names of its parameter places, its instructions, and variables. The unfolded worksheet completely covers the worksheets beneath it because its subroutine is now the current scope.

When a function returns, its worksheet is folded up to display its return value in the scope of the worksheet that called it. When a procedure returns, its worksheet is simply discarded because it has no return value.

A button on the calculator

Pressing a button (an action button) on a calculator causes the associated operation to be carried out on the current value in the calculator. The action buttons on the calculators represent built in functions that the computer can calculate. The calculators have other buttons on them too, *e.g.* digits. The clerk uses the calculators in the same way a person would, they copy values from the screen into places on the worksheet, and visa versa.

A button on the input device

Pressing a button on the input device causes the device to carry out a `Read` function of some kind, and return the function's value by printing a partial worksheet. The partial worksheet contains only the third part of a full worksheet: the part with a place for the function's return value. The clerk places the partial worksheet on the clipboard, copies the return value to where ever the value is required, and then discards the partial worksheet. The input device prints partial worksheets because all its subroutines are built-in functions that do not require any arguments, and do not require the clerk to carry out any instructions.

A button on an output device

Pressing a button on an output device causes the device to print another kind of partial worksheet. A partial worksheet for an output device contains only the first part of a full worksheet: the part for the subroutine's parameter places. The clerk places the partial worksheet on the clipboard, works out the values of the arguments to the subroutine, and then puts the values into the partial worksheet's parameter places. Once all the parameter places are filled in, the clerk takes the partial worksheet and feeds it back into the device that printed it.

There are two output devices: the text output device and the graphics output device (see section 2.2.2.3). Each device only accepts partial worksheets that it produced. Once a device accepts a partial worksheet, that device carries out the subroutine using the details on the sheet. The output devices print this kind of partial worksheet because none of the subroutines on the devices are functions. The output subroutines are procedures, and most of them require one or more parameters (only `WriteLine`, which outputs a newline character, does not require any).

2.2.2.8 The first worksheet in a program

One of the templates provided by the programmer is the special template for the program's "main" procedure. The template for the "main" worksheet contains the definitions of all the programmer-defined constants in the program. The constants are printed along the worksheet's right edge. The "main" worksheet is larger than the other worksheets so that the constants will be visible under a stack of worksheets.

The worksheet for the program's "main" procedure is special because it is the first worksheet printed when the program is begun. To begin a program, the clerk empties the clipboard of all old worksheets, and presses the button on the worksheet machine for the program's "main" procedure. The worksheet machine prints the "main" worksheet. The clerk puts the worksheet on the empty clipboard, and begins to carry out the instructions.

2.3 Analogies for the places on worksheets

Places are printed rectangles that the clerk can write values in. Each rectangle is marked to determine the type of values that the place can contain. There are four kinds of places: parameters, variables, return places, and constants. The first part of a worksheet contains a subroutine's parameter and variable places. Return places are contained on the return value part of a function's worksheet (see figure 2.1). Constant places are printed only on the special "main" program worksheet. The analogies for each kind of place, and for types are described in the following sections.

2.3.1 Variables, Parameters, Constants and Return Places

Variables, constants, and parameters are all places used by the computer to hold values. The size of the place, the shape of the place, and what the place can contain all depend on the type of value the place is designed for. Types will be described in section 2.3.3. Whether the clerk can modify the value in a place depends on the kind of place.

2.3.1.1 Variables and Parameters

Variables are named places. They occur on worksheets, where they appear as outlined rectangles that mark out a space on the worksheet. The value of the variable is contained in that space. The value can be used and replaced by the clerk. Each variable is named so that the clerk can find the correct place when one is referred to in the instructions.

Parameters are variables positioned at the top of a worksheet so that they are visible when the worksheet is first printed. However, the parameter's name is not visible when the worksheet is folded up; its name appears only when the worksheet is unfolded.

2.3.1.2 Constants

Constants are unchangeable named places. Like variables, constants are named, but the clerk cannot alter their values. Constants can only hold a value of one of the three simple types, which will be described in section 2.3.3. The values of most constants are determined by the special "main" program template that the programmer puts in the computer. Most constants occur on the special "main" program worksheet, where they can always be seen be the clerk. However, there are two constants that aren't printed on the "main" worksheet. These constants are represented by two named L.C.D. screens on the graphics device (see section 2.2.2.3).

2.3.1.3 Return places

Return places occur on the return value part of a function's worksheet. They are like variables but they are not named; instead, they are labelled with the same instruction that caused the clerk to call the subroutine in the first place (see figure 2.1).

2.3.2 Alternative model for places

A place on a worksheet can only hold values of the correct type. The model does not represent this constraint clearly because anything can be written in a rectangle on a piece of paper. We also considered a stricter representation of variables and types, in order to explain types as clearly as possible and make explicit the idea that values of different types are incompatible (see section 2.1).

This representation used the idea that each variable is a box that is constructed so that it can only hold particular kinds of objects. The kind of object determines the type of values that the variable can contain. For example: a `NUMBER` place is a box with a row of tumblers in it. Each tumbler represents a number column *e.g.* ones, tens etc., and has the digits from 0 to 9 printed on its rim. By aligning the tumblers appropriately, the box can display any number. A `NUMBER` place cannot hold a letter because there aren't any letters printed on the tumblers.

Similar analogies for the other types result in places that are more than just flat rectangles on printed paper. However, this clashes with the analogy for worksheets — a machine that prints worksheets will have trouble printing a box that contains a series of working tumblers. If we say that the worksheet machine can construct boxes because it is designed to, or gloss over this by saying that places are somehow magical, then the model becomes less plausible and less consistent. It is far more plausible to have a machine that prints rectangles on paper, than it is to have a machine that prints magical boxes on paper.

We choose to use a less strict analogy for places. We traded strictly consistent behaviour for places for the consistency of the overall model. The relaxed analogies for types are described in the next section; the stricter analogies are described too, for the sake of comparison.

2.3.3 Types

The types are the templates that places are made from. The worksheet machine uses them to print places on worksheets. The other devices use them to display different kinds of values, and print parameter places. The type of a place is represented by the size, and shape of the place. A place can only hold values of the right type.

There are two kinds of type: simple types, and compound types. There are three simple types built into the machine: number, strings, and booleans. Simple types are the only types that can be used to make constants. Compound types can only be created by the programmer. There are two kinds of compound types: array types, and record types.

2.3.3.1 Simple Types

`NUMBER`

A place of type `NUMBER` has a permanent decimal point. It only makes sense to put numbers in the place because of the decimal point. Its stricter analogy is a row of tumblers example given above (see section 2.3.2). The tumblers are stricter because they prevent arbitrary characters from being put into the place. The decimal point does not prevent arbitrary characters, but it does make them look out of place.

`BOOLEAN`

A place of type `BOOLEAN` has the word "`TRUE`" and the word "`FALSE`" separated by a vertical line. To put a value in the place, the clerk crosses out the word that doesn't apply. The value in a `BOOLEAN` place is the word that is still visible.

A stricter analogy would make a `BOOLEAN` place a box that holds a switch, a red light (`FALSE`), and a green light (`TRUE`). One of the lights is always on; which one depends on the position of the switch. There are plenty of variations on this theme. The type could equally well be just a switch, or a switch and one light. The red and green light idea has the advantage, and disadvantage, of tapping into the traffic light symbolism of "red is for stop" and "green is for go".

`STRING`

A place of type STRING has a pair of permanent single quotes, one at each end of the rectangle, *i.e.* $\boxed{\text{‘}\qquad\quad\text{’}}$. The clerk can write any characters in the place. The quotes indicate that its contents are always a `STRING`.

A stricter analogy would make a `STRING` place a box that holds a stand for scrabble pieces. Each scrabble piece is marked with a single character, and slots precisely onto the stand.

These analogies provide the novice with a view that separates the kinds of values a computer can deal with. The view is not as strong as one that the stricter analogies would provide. However, it should be enough to distinguish between different types of values, *e.g.* 5.2 and '5.2'.

The analogies for simple types raise a question about how much space a place takes up on a worksheet. This model assumes that a fixed size area is marked off by a place's rectangle. For `NUMBER` places the area is large enough for the clerk to write approximately 20 digits. For a `STRING` place the area is large enough for 80 characters. For a `BOOLEAN` place the area is just large enough to hold the words `TRUE` and `FALSE`. These sizes are chosen arbitrarily.

The problem with using fixed sizes is that they don't work in certain situations, say when the clerk needs to write 81 characters into a `STRING`. As an alternative solution to the problem of place size, the model could allow places of arbitrary size, which grow and shrink as needed. However, this weakens the worksheet analogy by taking it further away from real worksheets. A more reasonable solution is to allow the clerk to extend a place's rectangle when it needs to fit a larger value into a place.

2.3.3.2 Compound Types

The programmer uses compound type constructors to produce new types. New types are customised templates that the worksheet machine uses to print new kinds of places. A new type is specified using one of the two compound type constructors, an `ARRAY` or a `RECORD`. A place of a new type will always be based on one of these two constructors.

`ARRAY` types

A place of an `ARRAY` type is made up of a row of component places. Each component place in the row holds the same type of value as all the other places in the row. How many places there are in the row, and their type, are specified when the programmer defines the new type. For example, an `ARRAY` of 10 `BOOLEAN` places is a row of 10 rectangles. Each rectangle contains the words "`TRUE`" and "`FALSE`" separated by a vertical line. The individual places inside the `ARRAY` do not have names like other places do. Instead, they are numbered from 1 to the size of the `ARRAY` row, and the instructions refer to an array's components by number.

A stricter analogy would make a place of an `ARRAY` type be a carousel of component boxes. The inside of each box is custom-built to hold the appropriate value. An `ARRAY` of 10 `BOOLEAN` places, would be a carousel of boxes, with each box containing two lights and a switch.

`RECORD` types

A place of a `RECORD` type is made up of a collection of component places surrounded by a rectangle. Unlike an `ARRAY` based place, the component places in a `RECORD` are not necessarily all made from the same type. The size and shape of each place inside the `RECORD` is determined by that place's type. Each component place has a name that is unique within the `RECORD`. A "personal details" section in a tax form is an example of a `RECORD` type. It has a place for the person's name, a place for their address, and a place for a contact phone number.

A stricter analogy would make place of a `RECORD` type be a box that wraps up a number of other boxes for easy shipping, *e.g.* a custom-built trunk, case, or medicine chest with a variety of different compartments.

The value of a place based on a compound type is a combination of all the values in its component places.

New types made from `ARRAY` and `RECORD` can be used to specify other new types. An `ARRAY` place that is a row of `RECORD` places is common. Places that are printed from a compound type can potentially take up a lot of space. A complex compound place could be too big to print on a worksheet of a reasonable size. The model doesn't directly address this issue, it assumes that the worksheets are big enough[5].

2.4 Analogies for the instructions on worksheets

The second part of each full worksheet contains instructions that the clerk will carry out. There are three kinds of instructions: assignments of values to places, calls to subroutines, and control structures. The analogies for each kind of instruction are described in the following sections.

2.4.1 Assignment

The clerk does assignment by replacing the value in a place with another value. To replace the value in a *simple* place, the clerk erases the old value and writes in the new value. To replace the value in an *entire* compound place, the clerk changes the values in each of the place's components. To replace the value in a *component* of a compound place, the clerk does whichever of the previous operations is appropriate for the component place. The clerk has no short-term memory, and cannot recall the old value once a variable is changed.

An assignment instruction requires two things: a specification of the place to put a value, and a value to put in the place. The description of the place is usually the place's name. If the place is a component of a compound place, then the description is the compound place's name combined with either the component's internal name (if it is part of a `RECORD`), or the component's index (if it is part of an `ARRAY`). Parameter places have no names on folded worksheets and cannot be assigned to. Variable places can be assigned to, but constant places cannot.

[5] If a place becomes too large, the worksheet machine can always use a smaller font, and the clerk can put its reading glasses on.

A value must be of the same type as the place it is going to be put. There are three groups of values used in assignments (and other instructions): literal values, values in places, or expressions to evaluate.

2.4.1.1 Literal values

A literal value is a value that appears in the instructions on a worksheet, *e.g.* `42`, or '`hello`'. To assign a literal value to a place requires nothing more than replacing the value in the place with the literal value.

2.4.1.2 Values in places

A value in a place is referred to in the instructions by a description of the place (see above). To assign a place's value to another place involves finding the first place, looking at its value, and then changing the value in the second place to match the value in the first. The values in parameter places on folded worksheets cannot be used in assignment; the values in constant and variable places can be.

2.4.1.3 Expressions to evaluate

An expression instructs the clerk on how to calculate a value. Typically, an expression is a mathematical formula, but it need not be. It could be a programmer-defined function call, or a series of nested programmer-defined function calls such as '`factorial(factorial(n))`'. If the expression contains a mathematical formula, the clerk uses calculators to work out the value of the expression. If the expression contains a programmer-defined function, then the clerk carries out the function and uses the value that the function returns (see section 2.2.2.7). In either case, to assign the value of an expression to a place, the clerk works out the value of the expression, and replaces the contents of the place with that value.

2.4.2 Calls to subroutines

Subroutine calls instruct the clerk to press the button that corresponds to that subroutine. Once the clerk has pressed the correct button, the clerk does what is necessary to carry out the subroutine according to which machine the button is on (see section 2.2.2.7).

2.4.3 Control Structures

There are two kinds of control structure that the abstract machine can use: `IF` statements, and `LOOP` statements. `IF` and `LOOP` statements are representative of selection statements and iteration statements respectively.

2.4.3.1 IF statements

The IF statement instructs the clerk to choose one of two sets of subsequent instructions to carry out. Each IF statement has a BOOLEAN condition that the clerk evaluates to work which set of instructions to carry out.

2.4.3.2 LOOP statements and LEAVE statements

LOOP statements instruct the clerk to repeat a group of instructions. The clerk will carry out each instruction in the loop and then go back to the first instruction in the loop and carry on from there. A loop may contain a LEAVE statement. A LEAVE statement instructs the clerk to skip the rest of the instructions in the loop and carry on with the instructions after the end of the loop.

2.5 Extending the machine model

The following programming concepts were not fully developed for the machine model: external files, units (modules), exceptions, parallel programming, abstract data types, objects, and pointers. These concepts were not developed in the demonstration model, or implemented in the prototype programming environments, both of which will be discussed in chapters 3 and 4. However, the abstract machine model could be extended cleanly to handle these concepts. The analogies for these concepts are presented in the following sections.

2.5.1 Files and file operations

There are two kinds of files in Pascal: TEXT for text files, and FILE for binary files. The following analogies can apply to both kinds of file. They will both be referred to as FILE because I do not wish to distinguish between them.

Files require two extensions to the model: another I/O device in the room, and an analogy for another built-in type. The new device is the computer's interface to files. The FILE device has a number of buttons and a slot. It operates like the other I/O devices (see section 2.2.2.3) with the addition that all its operations involve a FILE parameter.

The FILE type itself becomes the fourth built-in type. A place made from a FILE type is a rectangle permanently marked with the words "The" at one end and "File" at the other. The space in between the two words is used to hold the name of the current FILE associated with the place.

The FILE device is connected to a filing cabinet, which holds folders that contain paper files like a company's files. The device can perform four kinds of built-in operations: OPEN a file for use, CLOSE a file, a variety of READ operations, and a variety of WRITE operations. The device uses the FILE parameter to find the correct file for the operation. The OPEN operation finds the FILE, and moves it to into the FILE device so that it can be accessed.

`CLOSE` puts the `FILE` away in the cabinet. The `READ` operations get different types of values from an open `FILE`. The `WRITE` operations add different types of values to an open `FILE`.

2.5.2 Units

A `UNIT`, or a module, is an extra worksheet machine. For every `UNIT` that a program uses, the room contains a machine that prints worksheets for that unit. Each unit machine contains all the type and subroutine templates (the declarations) for that `UNIT`.

Note that all the templates from all the units could be put into a single worksheet machine. However, this would not support the notion of abstract data types, or encapsulation.

2.5.3 Exceptions, events, and handlers

Exceptions, events, and their handlers are not something novices need to worry about. The analogy for exceptions and events depends on the kind of exception and event handling that you want to teach. This section presents one possible analogy for exception handling. A similar analogy can be used for events.

Exceptions could be incorporated into the machine model with a special worksheet printing machine with a panel of warning lights — the exception machine. The warning lights are connected to all the machines in the room, and each light has a button under it.

After finishing an instruction, the clerk checks the panel to see if any of the lights are on. If a light is on, the clerk presses the button under the light and the exception machine prints a worksheet for a subroutine. The subroutine is the exception handler for the particular exception. The clerk immediately puts that worksheet on the clipboard and begins to carry out the instructions on it. The worksheet is not folded up when it is printed because the exception machine has already printed the subroutine's arguments, if any, in the sheet's parameter places.

2.5.4 Parallel programming, and concurrency

Again, parallel programming, concurrency, and related issues are not something novices need to worry about. However, they can be incorporated into the machine model in a straightforward manner.

In the current model, there is only one clerk in the room. The clerk has a clipboard with a stack of worksheets, and carries out the instructions on the worksheets. In a parallel programming version of the model, there can be more than one clerk in the room. Each clerk has their own clipboard, worksheets, and carries out their own instructions. Making a new process is like getting another clerk to carry out the instructions at the same time. The room needs to have a supply of clerks (processes), who sit around waiting to be given a task.

2.5.5 Abstract Data Types, Objects, and Pointers

Abstract data types, objects, and pointers are used for building complex data structures. A.D.T.s are the most easily incorporated into the machine model. Pointers are the next easiest, but they require several extensions to the model. Objects are the most difficult to incorporate; they require more changes to the model. The required changes to the model for each concept are outlined in the following sections.

2.5.5.1 Abstract Data Types (A.D.T.s)

An abstract data type is a type that has all its operations defined in a single `UNIT`. The template used by worksheet machines for making places of the A.D.T. is also in that `UNIT`. To extend the current machine model to include A.D.T.s merely requires extending it to include `UNIT`s (see section 2.5.2).

2.5.5.2 Pointers

The analogy for the computer's memory has to be expanded to handle pointers. In addition to the stack of blank paper for worksheets (see section 2.2.2.5), the computer's memory must also include a box of unused filing cards and a filing cabinet for cards being used. The box of cards is attached to the worksheet machine. Each card has an index number, and space for a value of some type.

A `POINTER` is a new type constructor that the programmer can use to create new types. A place of a `POINTER` type can hold the index of a card. The `POINTER` based type determines the type of place that must be printed on that card.

For pointers, the worksheet machine must be extended to allow it to print places on cards. The machine has extra buttons on it for printing places on cards. Each pointer based type definition adds a button to the machine, which allows the machine to print the type of place that the new pointer type can refer to.

The clerk understands two new instructions relating to pointer places: `NEW` and `DISPOSE`. The `NEW` instructions tells the clerk to get the worksheet machine to print the correct type of place on the card, copy the card's index into the pointer place, and put the card in the filing cabinet. The `DISPOSE` operation tells the clerk to find the card referred to by the pointer, take the card out of the cabinet, put it back in the card box, and erase the card's index from the pointer place.

An instruction to dereference a pointer causes the clerk to looks in a pointer place to get the index number, and find the card in the filing cabinet with the same index. Once the clerk has the card, the clerk can change or use the values in the place on the card.

The clerk keeps the card for only as long as it is needed, then puts the cards back in the cabinet.

2.5.5.3 Objects

The analogy for objects depends on the kind of objects that you want to teach. One analogy for objects assumes that all objects are referred to by reference only. This analogy requires extending the current machine model to handle pointers. Objects are like `RECORD` places on cards, which are created, kept in a filing cabinet, used, and destroyed just like the places on cards described for pointers (see above).

Objects also require a couple of extensions to the pointers analogy: a filing cabinet per type of object, and a worksheet machine per filing cabinet. Objects of the same type are all kept in the same cabinet. Each cabinet's worksheet machine holds the templates for all that type of object's methods.

To carry out a method associated with an object, the clerk finds the object's card in the correct cabinet, slots the card into the associated worksheet machine, and presses the method's button on that machine. The machine takes the object and prints a worksheet for the method. The worksheet has a special pointer constant called `this_object`, printed on it. The value of `this_object` is the index of the object that is slotted into the machine. The clerk takes the worksheet and the card from the machine, puts the card back in its filing cabinet, and carries out the instructions on the worksheet.

2.6 The language

The machine model is based on an abstract machine that carries out programs in a `Pascal` like language. The language is a `Pascal` that has been altered to tidy it up, and to fit it in with our view of good program design and style (see section 2.1).

`Pascal` was changed to remove some of its nastier constructs, and to simplify the things it makes needlessly complex or confusing. Some concepts we disallowed completely; others we modified to fit with our view of programming. Making our modifications removed most of the things that confuse people, added nothing to what they need to know to understand programming, and still enabled them to do general purpose programming.

The result is a language, which - one could say - is representative of the class of general purpose, imperative, non object-oriented, non concurrent, programming languages (this includes `C`). The language has `Pascal` like syntax, semantics, and ontology - including `Pascal`'s distinction between functions and procedures. The following sections describes the parts of `Pascal` we disallowed, changed, ignored, and added to the language.

2.6.1 Parts disallowed

We completely disallowed the following constructs from the language:

- `PACKED` types or variables.
- variant records.
- nested scope functions or procedures.
- global variables.
- `GOTO` statements.
- the built-in `SET` type.
- `VAR` parameters.
- anonymous types.
- enumerated types.

Each of these concepts was thought to be too confusing, too complicated, bad style, or unnecessary for novice programmers.

2.6.2 Parts changed

The following changes were made to make parts of the language clearer, easier to model, and easier to explain. Some of the changes bring `Pascal` into line with THINK `Pascal`, which is the version of `Pascal` used in first year programming courses at Victoria University of Wellington.

2.6.2.1 Added `RETURN` statements.

The "`function_name := result_value;`" convention for returning a value from a function was replaced with a "`RETURN result_value;`" statement. Using a `RETURN` statement distinguishes the function's name from a magical temporary place used to hold a value. The `RETURN` statement evaluates the `result_value` expression, stops the function, and returns the value immediately.

2.6.2.2 `LOOP` and `LEAVE` statements replace all iteration constructs.

The iteration structures `WHILE`, `REPEAT`, and `FOR` loops were removed and replaced with a general `LOOP` statement. The reason for this is to reduce the variety of looping structures, and make the concept of iteration more important than trying to remember the syntax for a particular iteration construct. The `LOOP` statement itself has no exit condition, but a `LEAVE` statement was added to allow a loop to exit.

`WHILE`, `REPEAT`, and `FOR` statements can be mimicked with a `LOOP` statement. Which statement you mimic depends on how you set up the loop; the point in the `LOOP` where you position a `LEAVE` statement determines the iteration structure of the `LOOP`.

2.6.2.3 Matching END statements for all control statements.

All the control statements, selection and iteration, now have their own explicit `END` statement. `IF` statements end in `ENDIF`, and `LOOP` statements end in `ENDLOOP` (`CASE` statements, if we had them, would end in `ENDCASE`). The reason for introducing `END` statements is to reduce the confusion associated with the problem of trying to match up blocks of code with the control statements that surround them. None of these statements requires a `BEGIN` keyword anymore because the `BEGIN` is an implicit in the statement.

2.6.2.4 Simplified the built-in types.

The number of built-in types was reduced to three: `INTEGER` and `REAL` were replaced by `NUMBER`, `BOOLEAN` remains unchanged, and `STRING` and `CHAR` were combined into `STRING`. The `STRING` type is not just an `ARRAY` of `CHAR`s, it is a proper built-in type. Functions for manipulating `STRING`s were added to the language to support the `STRING` type. A sampling of these will be described in section 2.6.4.

2.6.2.5 Restricted use of type declarations.

The structures for creating programmer-defined types, `ARRAY` and `RECORD`, can only be used to define new named types. No anonymous types are allowed. All variables must be created from a named type.

2.6.2.6 Compound types can be returned from functions.

The function value return mechanism was extended to allow compound types to be returned from functions.

2.6.2.7 Restrictions on using constants.

Constants are restricted to literals of the three built-in types: `NUMBER`, `BOOLEAN`, and `STRING`. Constants can only be declared globally.

2.6.2.8 Changes to input and output operations.

The input and output subroutines have been changed to give simpler I/O for the language.

Input using `READ` (including `READLN`) is split into three separate functions: `ReadNUMBER`, `ReadSTRING`, `ReadBOOLEAN`. Each is a function that requires no parameters but returns a value of the appropriate type. The functions read text input from the user until they read a value of the appropriate type. `ReadSTRING` simply reads until the end of the line.

Output using `WRITE` (including `WRITELN`) is split into four separate procedures, one for each of the three simple types (`WriteNUMBER`, `WriteBOOLEAN`, `WriteSTRING`), and one for ending lines (`WriteLine`). Each procedure, except for `WriteLine`, requires one parameter of the appropriate type, and does not allow any format specifications. `WriteLine` has no parameters.

2.6.3 Parts ignored

Several parts of `Pascal` were not considered for the model. They were not disallowed in the model, nor did we alter them to fit our view. The parts were left out not because of some complexity or confusion in the way `Pascal` does them (see section 2.6.1), but because we decided to leave them out and concentrate on the remainder of the language. The following parts of `Pascal` were left out:

- The `FILE` type.
- The `TEXT` type.
- `UNIT`s, and abstract data types.
- Pointers.
- `CASE` statements.

Even though they were ignored, suggestions on how to incorporate most of them into the model, by using suitable analogies, can be found in section 2.5.

2.6.4 Parts added

The language has the following extra built-in procedures added to it for doing simple line based graphics and manipulating the mouse:

- `WaitForMouseClick()`, which waits until the mouse is clicked inside the graphics window and then finishes.
- `DrawLine(firstX, firstY, secondX, secondY)`, which draws a line between two points.
- `DrawRectangle(topX, topY, bottomX, bottonY)`, which draws a rectangle using the given coordinates as the top left and bottom right corners.
- `DrawCircle(x, y, radius)`, which draws a circle of the given radius at the specified point.
- `DrawSTRING(x, y, string)`, which draws a `STRING` value centered over the specified point.
- `DrawNUMBER(x, y, number)`, which draws a `NUMBER` value centered over the specified point.
- `DrawBOOLEAN(x, y, boolean)`, which draws a `BOOLEAN` value centered over the specified point.

All these operations are attached to the graphics device.

Several operations to support the `STRING` type were also added to the language. The following are a sample of the operations (there should be more operations for full support of `STRING`s):

- `StringLength(string):NUMBER`, which returns the number of characters in the `string`.
- `GetSubString(string, start, end):STRING`, which returns the `STRING` that begins at the `start`-th character and finishes at the `end`-th character in the given `string`.
- `FindSubStringsStart(string, substring):NUMBER`, which returns `-1` if `substring` does not occur in `string`, or otherwise it returns the number of characters into `string` that `substring` starts at.
- `FindSubStringsEnd(string, substring):NUMBER`, which returns `-1` if `substring` does not occur in `string`, or otherwise it returns the number of characters into `string` that `substring` ends at.

These `STRING` operations are attached to a `STRING` calculator, which is similar to the numerical calculator. Neither the `STRING` calculator nor any of these operations have been implemented so far in either of our prototype programming environments.

2.7 How the model addresses novices' confusions

Some programming concepts cause people learning to program (novices) a lot of difficulty. Providing support for understanding the concepts that cause problems was an important goal for the machine model. Section 2.1.2 described several of the problems and their related concepts; how well the model handles these problems is addressed in this section.

With a good model, the correct behaviour is "obvious" and the confusion won't arise. The machine model is a good model for most of the problematic concepts, it avoids one of the problems, but it does not provide much help for a couple of the problems.

The model has been used in the lectures of two offerings of Comp 102, an introductory programming course at Victoria University of Wellington. Two kinds of preliminary evidence suggests that the model helps novices: lab interaction between students and tutors, and an improvement in internal exam marks for questions on recursion.

2.7.1 The confusion over the values of variables

Three typical misconceptions arise from the common metaphor that "a variable is a box with a piece of paper in it". The first is that a variable should be able to hold any type of value. The second is that the old value in a variable is still available after a new value has been assigned to the variable; this shows up particularly when novices try to swap the values in two variables. The third misconception is that a variable can sometimes have nothing in it, *e.g.* when first created or during assignment.

The model handles the first problem by explaining variables of different types as different kinds of places (see section 2.3.3). For example, a `NUMBER` place has a decimal point that makes it clear that it must hold a number. A `STRING` place has quotes at each end, which make it clear that it contains a piece of text. It makes little sense to put letters into a decimal `NUMBER` place.

The second and third problems are caused by the box and paper analogy for variables. The machine model uses a different analogy — a variable is a rectangle on a worksheet that the clerk writes values into. The model's analogy does not cause the problems that the box and paper analogy does.

The model avoids the second problem by providing a good explanation of the way the clerk does assignment, and what happens to old values. The clerk does assignment by erasing the place's old value and copying a new value into the place (see section 2.4.1). The clerk doesn't remember a value after it has been copied; the old value is gone.

The model doesn't address the third problem completely, since nothing is stated about a place's initial value. However, the problem that boxes can sometimes have no piece of paper in them, and hence no value, does not apply to printed rectangles; the only time the value in a rectangle place is erased is when another value is about to be written there.

Whether a place initially contains no value[6], a zero value, or a junk value is not terribly important for the model. This is an implementation issue for any environment that supports the machine model. Provided the initial values are generated consistently, novices should have little problem.

2.7.2 The confusion over passing parameters

Passing parameters to subroutines give rise to two problems: the difference between passing arguments by value and passing them by reference, and the difference between a local variable in the calling scope (actual parameter) and a formal parameter with the same name. The first problem is not directly addressed in the model. The model merely avoids it by eliminating the passing of arguments by reference (`VAR` parameters) from the language.

The model handles the second problem with folded and unfolded worksheets, the way worksheets obscure one another on the clipboard, and the visibility of parameter names. The problem involves two worksheets: an unfolded worksheet (the calling worksheet) that contains an instruction to call a subroutine, and a worksheet for the subroutine that is printed folded up and later unfolded. The folded up subroutine's worksheet is placed on top of the calling worksheet while the clerk fills in the subroutine's parameters. Once all the parameters have been filled, the subroutine's worksheet is unfolded to obscure the calling worksheet.

[6] From the model's notion of worksheets and places as printed rectangles, one might expect that places are initially empty.

Recall that a parameter's name is not visible on a folded up worksheet, but it is visible on an unfolded worksheet (see section 2.3.1.1). A variable's name is visible as long as the variable is visible; variables are only visible on unfolded and unobscured worksheets (see section 2.2.2.7).

The variable places on the calling worksheet are quite separate from the parameter places on the subroutine's worksheet. A parameter's name is not visible until the subroutine's worksheet is unfolded. At which point, the subroutine's worksheet completely hides the variable on the calling worksheet. If a variable on the calling worksheet and a parameter on the subroutine's worksheet both have the same name, then the clerk is always sure which place the instructions refer to because at any given time only one of them is named.

2.7.3 Issues surrounding recursive calls to the same subroutine.

The main problems with recursive calls are that novices get confused about the multiple invocations of the same subroutine, and multiple copies of the subroutine's local variables. Giving a good explanation for subroutine invocation and local variables removes the source of the confusion. The model gives this explanation with the clipboard, worksheets, and worksheet machine (see section 2.2.2.7).

To carry out a recursive call, the clerk gets the worksheet machine to print out another copy of the subroutine's worksheet. The clerk takes the copy, and carries it out as if carrying out a normal subroutine call (which the recursive call is from the clerk's point of view). At this point, two copies of the same worksheet are on the clipboard, one on top of the other. Each copy has a distinct set of the same instructions, and a distinct set of the same variable places, but the clerk is up to a different instruction on each sheet. The clerk treats them as completely different subroutines and carries on as normal.

2.7.4 The computer only can consider a few values at a time.

This problem primarily concerns arrays. For example, the computer cannot look at all the values in an array at the same time and pick out the minimum value. The model's analogy for arrays does not give a novice the correct idea about the way a computer handles arrays. The stricter analogy of arrays does address this problem by preventing all the elements of an array from being visible at once (see section 2.3.3.2).

2.7.5 Conveying the idea that the computer has no common sense.

The problem is that novices expect a computer to find the solutions to problems in the same way as a human can. However, a computer does **not** operate the same way the a human does. In particular, a computer has no common sense, or human reasoning capabilities, and requires detailed instructions to carry out a task. The model does not provide much help with this idea. The explicit instructions, and the clerk following instructions blindly does help a little, but not a lot.

2.8 Summary

This chapter described the abstract machine model in detail. The aim of the model is to help novice programmers understand the way a computer carries out a program, enabling them to scaffold from what they already know to a better understanding of programming. The model is made up of an abstract machine, a `Pascal` based language, and a coherent collection of analogies. The analogies link parts of the abstract machine to objects from the real world.

The design of the model was biased by our belief that the ontology and semantics of the programming language are more important than its syntax: to aid the novice learning to program in the language, the ontology and semantics should be made explicit. Another influence on the model was the goal of providing analogies that would avoid several common misconceptions that novices have about particular programming concepts (*e.g.*, recursion). Our personal view of what is good programming style also affected the model.

The abstract machine model describes how the computer carries out a program from the point of view of a programmer who uses the programming language. The flow of control in the computer is personified by a clerk who carries out the program by following the instructions on worksheets, which are stacked on a clipboard. The clerk has various devices, *e.g.* calculators and a worksheet printing machine, to help them carry out the instructions. Each worksheet contains places to write values and instructions to be carried out. The clerk working through the instructions and worksheets, and using the other devices, represents the computer carrying out the program.

The stack of worksheets on the clipboard represents the activation stack, with each worksheet representing a particular subroutine. The clerk uses the worksheet printing machine to print new worksheets when subroutines are called. New worksheets are printed from templates, which constitute the program. New worksheets are printed folded up so that only their parameters are visible. The clerk fills in a worksheet's parameters before unfolding it and carrying out the instructions on it. The first worksheet to be printed when the program is begun is the program's "main" procedure.

The model detailed the analogies for the places (variables, parameters, constants, and return values) and instructions on the worksheets. Two analogies for places were described: printed rectangles that are marked (with decimal points and quotes etc.) to determine the type of values they can hold, and boxes whose contents strictly determine the type of values they can hold, *e.g.* a set of tumblers with digits on them. The rectangle based analogy does not provide as good a mapping as the box based analogy for the behaviour of places, types and values. However, the model used the printed rectangle analogy because it is more compatible with the notion of printed worksheets. In doing so, the consistency of the analogy for places was traded off against the consistency of the overall model.

The analogies for the instructions on worksheets described how the clerk carried out the different kinds of instructions. Assignment is done by erasing the value in a place and writing a new value there. Expressions are evaluated using the calculators. Control structures are used to determine what to do next. Subroutine calls are carried out by pressing a button that prints the appropriate worksheet.

While the model described most of the concepts in the `Pascal` based programming language, some typical programming concepts were not fully developed for the abstract machine model. These concepts were: files, units, exception handling, parallel programming, abstract data types, pointers, and objects. Nevertheless, possible analogies for incorporating them into the model were presented after the concepts that were fully incorporated into the model.

`Pascal` was cleaned-up and modified to produce the `Pascal` based language that the model used. The changes made to `Pascal` were all based around simplifying and clarifying the language for novice programming. These changes were made based on our personal view of programming style and what novices need to know in order to program.

Finally, the model was informally evaluated with respect to how the model addressed the common confusions of novices that were part of its original bias. The abstract machine model provides a good model for most, but not all, of the common novice confusions that were discussed.

The abstract machine model provides the basis for the demonstration model, the demonstrational machine, and the way of programming they support.

Chapter 3

Examples of the Demonstration Model

The second underlying model for our environment for novice programmers is the model of how to program in this environment. The demonstration model is the environment's model of programming. It extends the abstract machine model (the first model) by adding programming by demonstration, and describes the way the programmer interacts with the computer to produce a program.

This chapter introduces the demonstration model and provides three detailed examples of programming with it.[1] A more detailed look at the model and the issues raised during our exploration of the prototypes will be presented in chapter 4. The demonstration model consists of two things: a way of programming, and a special environment for programming — the demonstrational machine. The way of programming is based on techniques from programming by demonstration, program visualisation, and direct manipulation.

The demonstrational machine extends the abstract machine to allow programming. Two new roles are added to interact with the components of the machine: the demonstrator who can carry out actions that the clerk would do, and the recorder who writes the demonstrator's actions down as the program. Two prototype programming environments based on the demonstrational machine were implemented. These environments were used to explore the analogies in the abstract machine model, and the programming techniques of the demonstration model.

Section 3.1 describes the techniques that the demonstration model is based on. Section 3.2 outlines what the demonstration model is *not*. Section 3.3 presents the demonstrational machine that is required for this way of programming. Sections 3.4 and 3.5 describe the first prototype of the demonstrational machine, and give a detailed example of programming with it. Section 3.6 describes the second prototype of the demonstrational machine, and outlines its differences from the first prototype. Section 3.7 and

[1] What better way to explain this kind of programming by demonstration than with a demonstration.

Section 3.8 give examples of programming with the second prototype. Section 3.9 provides a summary of the chapter. A more detailed description of the way of programming supported by this model will be given in chapter 4.

3.1 Techniques underlying the demonstration model

Programming in the demonstration model and prototype environments is based on a combination of three techniques: programming by demonstration, program visualisation, and direct manipulation (see figure 3.1). Programming by demonstration provides a way for a novice to program without knowing programming details. Visualising the computer from the abstract machine model provides a basis for the demonstrational machine — the environment that this way of programming requires. Direct manipulation provides the means for a novice to interact with the components of the demonstrational machine to create programs. Each technique is described in the following subsections.

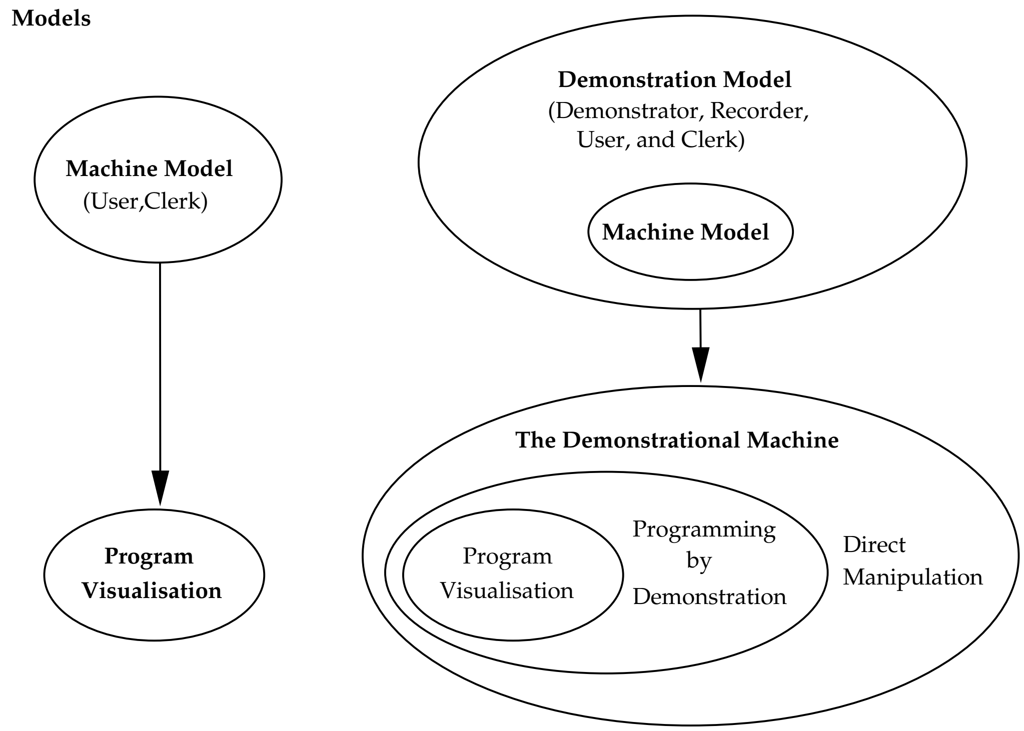


Figure 3.1: The relationship between the models and techniques.

3.1.1 Programming by Demonstration

Programming by demonstration (also known as programming by example) is a technique for making the programming of complex tasks easier for people to specify. The idea is to show the computer what you want to do, and let it work out how to carry out your task. The programmer shows the computer an example, or a series of examples, or demonstrates to the computer how to do something. While the programmer demonstrates, the computer records the programmer's actions, and makes some inferences about those actions. When the programmer has finished demonstrating, the computer tries to construct a program that is a generalised version of the task (Cypher 1993b).

The idea of showing the computer what to do, rather than trying to work out how the computer would do it, seems to be a useful thing for novices. It allows the novice to show the computer what they want it to do, without having to worry about the instructions that would make the computer do it because the computer figures out the details for them.

One component of many programming by demonstration systems is generalising from the programmer's actions. The hard part about generalising from a programmer's actions is trying to infer the programmer's intent. If the computer cannot work out why the programmer did something, then the computer is unlikely to produce a useful general version of the programmer's task (Cypher 1993b). For example, correctly inferring loops and conditions in the context of general programming is a difficult task.

The programming by demonstration used in this model is concerned primarily with generating instructions, declarations, and control flow; this is tricky enough especially for general purpose programming for novices. The most inference that is used in the model is working out where a value came from, to establish the proper description of the value for an instruction. The computer will not be inferring loops from the programmer's actions.

3.1.2 Direct manipulation

The demonstrational machine uses direct manipulation as the interaction method through which the programmer will demonstrate what the computer should do. Direct manipulation is more usually used as a user interface technique that enables the user to feel as though they are interacting directly with the important objects in a system, rather than instructing the computer to do things to those objects. Properties typically expounded as features of direct manipulation include (Shneiderman 1990): immediate feedback when an action is performed, easily reversible actions, clear visibility of objects, incremental actions, the user directly alters the object rather than using a command language, a sense that the user is really altering the object, and a sense that the system is transparent.

In the demonstrational machine and its prototypes, the programmer uses direct manipulation to interact with the parts of the computer in the abstract machine model. The programmer will directly manipulate the state and components of the computer using actions similar to the actions that the clerk performs. In the prototypes, not all of the actions that the programmer can demonstrate will be done through direct manipulation. For convenience and simplicity some actions are demonstrated less directly, *e.g.* unfolding a worksheet.

3.1.3 Program visualisation/animation

The interface to the computer that the programmer directly manipulates is provided by visualising the abstract machine and its program in action. A program in action can be visualised at many different levels of abstraction. Program visualisation visualises the program at the level of the values in variables and subroutines. Program visualisation is one facet of software visualisation. Software visualisation includes visualising programs and algorithms at a higher levels of abstraction (algorithm animation), and often includes more abstract visual representations (Price, Baecker & Small 1993).

The way of programming supported by the demonstration model is based on visualising the abstract machine (see chapter 2) at the program visualisation level. The clipboard, worksheets, devices, and calculators should be visible, right down to the values in variable places on the worksheets. The aim is to make the internal state of the machine visible to the programmer at each step in the program. This should make it easier to understand the program and the way the computer carries it out.

3.2 Contrasting this approach with other application areas

There are a number of common application areas for programming by demonstration systems. The way of programming supported by our demonstration model is not interface building by demonstration, nor graphical editing by demonstration, nor is it programming by demonstration that uses multiple examples or input and output pair to infer a generalised program, nor is programming with a visual language.

Direct manipulation and programming by demonstration are usually used together in graphical user interface builders (Myers 1993b, Myers 1993c, Myers et al. 1993, Wolber 1996). In those kinds of systems, graphical components of a user interface are set out and their positions specified with direct manipulation. Programming by demonstration is used to produce code that determines the user interface's operations and reactions to events, and enforce relationship constraints between interface elements from the specific positions and relations demonstrated by the user. In contrast, the demonstration model uses direct manipulation to obtain instructions for the program.

Another common use of direct manipulation and programming by demonstration is in graphical editors (Kurlander 1993, Lieberman 1993a). Programming by demonstration typically is used in these kinds of systems to produce a procedure that carries out a particular graphical operation, *e.g.* creating a particular kind of graphical shape such as an arch. The user performs the operations that create an instance of the arch and the program produces a procedure that draws an arch. This kind of programming by demonstration is the closest to the *way* a programmer constructs a program in our model.

In contrast to programming by demonstration that infers a subroutine from multiple examples of input and output pairs, the demonstration model obtains subroutines from traces of the programmer carrying out the instructions. The demonstration model has more to work with than just the initial data (the input) and the final result (the output).

The phrases 'visual programming', 'visual programming language', 'iconic programming' are applied to a wide variety of different systems (Chang 1990, Myers 1990, Hyrskykari 1995). Whether these phrases apply to a particular system depends upon your definition of the phrases. One group of systems referred to by these phrases consists of programming environments in which the programmer lays out various graphical symbols, and "wires them up" to each other. The objects that the programmer manipulates are the components of the program. In contrast, the way of programming supported by our demonstration model has the programmer manipulate data objects and components of the computer, not program elements. Other kinds of programming described by the phrases 'visual programming' and 'visual programming language' may well apply to the demonstration model.

3.3 The Demonstrational Machine

The way of programming used in the demonstration model requires a particular programming environment, which we call the demonstrational machine. The machine allows the programmer to demonstrate actions that will be part of the program. The demonstrational machine consists of three things: the abstract machine presented in chapter 2, a new person called the demonstrator (the programmer), and a new person called the recorder. The abstract machine provides the computer to program. The demonstrator and recorder extend the abstract machine to allow it to create programs. Both the demonstrator and the recorder are present in the room with the clerk. The programmer takes the roles of the demonstrator and the user, while the computer takes the roles of the clerk and the recorder (see figure 3.2).

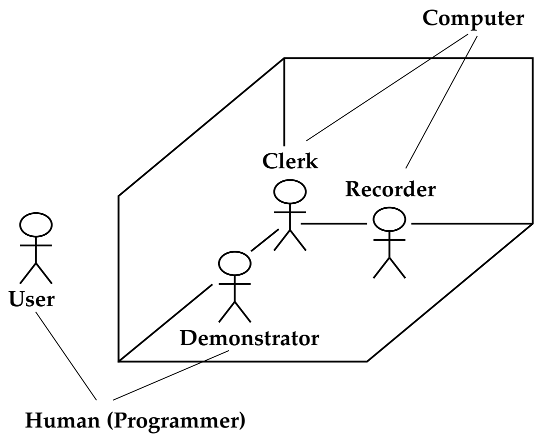


Figure 3.2: Roles in the demonstrational machine.

The first two parts of this section describe the roles of the demonstrator and the recorder in the demonstrational machine. The third part of this section outlines interactions that are critical for programming with the demonstrational machine. The fourth part outlines the way that a program is built up in the demonstrational machine. Finally, the two prototypes of the demonstrational machine are introduced.

3.3.1 The Demonstrator

In the demonstrational machine, the programmer takes the role of the demonstrator. The demonstrator controls the computer and, to a lesser extent, the recorder. The demonstrator can get the computer to do things by taking the clerk's place, or by asking the clerk to carry out the next instruction. The demonstrator can ask the recorder to unwind the computer's state to a previous one, or reset the computer. The demonstrator can also ask the clerk to start the program from the beginning (both of these will be discussed in section 3.3.3).

If the demonstrator takes the clerk's place, then the demonstrator takes the clipboard and begins to do things that the clerk could do: the demonstrator uses the calculator, writes values into places on worksheets, asks for new copies of worksheets, and so on. The clerk waits and does nothing while the demonstrator acts as the clerk. The demonstrator *does not* follow the instructions on the worksheets. However, the demonstrator *does* have to carry out actions in the same manner as the clerk would.

3.3.2 The Recorder

The demonstrator alone cannot write any programs; to do that, requires the help of the recorder. The recorder represents a smart agent who job it is to help the demonstrator create a program. The recorder watches the demonstrator when the demonstrator takes control of the machine, and records what the demonstrator does in the form of a program. The recorder is also responsible for recording the current state of the machine after any action by the demonstrator (this will be discussed in section 3.3.3). In traditional programming by demonstration, the recorder would be the computer that is trying to work out what the programmer wants to do.

The recorder is necessary to clearly distinguish between the computer's two roles in this way of programming: figuring out the program, and carrying out the program. The recorder figures out the program, and the clerk carries the program out. The clerk is fundamentally robotic and naive; it follows instructions blindly because that is the way that computers work. The distinction between the clerk and the recorder is there to ensure that novices get the right model of the computer — it follows the instructions it is given to the letter.

One part of the program is the sequence of instructions that the clerk must follow; they are recorded from actions that the demonstrator does in the clerk's place. For each action the demonstrator carries out, the recorder writes down the instruction that would have caused the clerk to carry out the same action. The recorder writes these instructions directly onto the template for the top unfolded worksheet on the clipboard because that is where the computer is up to. The recorder also writes the instruction onto all other worksheets on the clipboard that were printed from that template. The recorder does this in order to keep all the printed worksheets consistent with their templates. The instructions are added to the template right after the last instruction that the demonstrator or clerk carried out. The recorder makes all these modifications before the demonstrator demonstrates the next action.[2]

The instructions on worksheet templates are one part of a program; the other part of a program is the declarations. While acting in the clerk's place, the demonstrator can demonstrate instructions — assignment, subroutine calls, expressions, returns, conditionals, and loops — for the recorder because the instructions are actions that the clerk carries out in the course of a program. However, the demonstrator cannot demonstrate declarations — new places, types, and subroutines — while acting *strictly* like the clerk because these are things that the clerk uses, but does not carry out or produce. For instance, the clerk can't define a new type in the course of a program, though the worksheet machine can use a type to print a place.

However, the demonstrator isn't the clerk and is not required to act strictly like the clerk. For declarations, the demonstrator can do things that the worksheet machine would normally do. To produce a new declaration, the demonstrator produces the object created by the declaration, and begins to uses it. The recorder tries to figure out what declaration is required to produce that object, and records the declaration's details on the appropriate templates.

- For a new variable, the demonstrator draws the variable place on the worksheet by using a type template; the recorder adds the declaration to that worksheet's template.
- For a new constants, the demonstrator draws the constant place on the special "main" worksheet; the recorder adds the declaration to the "main" worksheet's template.
- For new type declarations, the demonstrator declares a new type by drawing the type on a blank type template using a type constructor; the demonstrator fills in the type's details using the other types that exist. The recorder asks the demonstrator to name the new type and adds it to the other types in the worksheet machine.
- For new subroutine declarations, the demonstrator gets a blank worksheet from the worksheet machine and fills it out with the details of a new subroutine; the recorder writes out a new worksheet template and connects it to one of the worksheet machine's unused buttons.

Note that this is not the only way that declarations could be produced. An alternative

[2]There is an alternative way of updating the worksheets — the recorder could wait until the top unfolded worksheet changes and then update the new top worksheet to match its template. The effect is the same, but the recorder is less rushed.

method (a less demonstrational one) would be for the demonstrator to step back from taking the clerk's place and ask the recorder for a new declaration.

- For a new variable, the recorder draws a variable place on the current worksheet's template and then on the worksheets themselves.
- For a new constants, the recorder draws a constant place on the special "main" program worksheet, and its template.
- For new type and subroutine definitions, the recorder creates new templates, and puts them into the worksheet machine.

The recorder asks the demonstrator for any details that are needed to fill out the new declaration, *e.g.* the name and type of a new variable. The demonstrator cannot fill these details in because the clerk cannot.

Both views on how to produce declarations have advantages and drawbacks. The first method has the advantage of providing a consistently demonstration based approach. However, it means that the clerk's role in the computer could become blurred to novices because the demonstrator takes the clerk's place but then does things that the clerk couldn't normally do. The second method has the advantage of clearly defining the clerk's role by limiting what the demonstrator can do when they take the clerk's place. However, it has the disadvantage of not consistently using demonstrational methods to produce all the code. It is worth nothing that the interfaces of the two prototypes of the demonstrational machine do not favour one of these views over the other; they are (more or less) compatible with both views.

3.3.3 Vital interactions between the agents in the machine

Three agents interact during the running of the demonstrational machine: the clerk, the demonstrator, and the recorder. The vital interactions between them are described in this section.

3.3.3.1 Demonstrator and Clerk

There are three interactions that occur between the demonstrator and the clerk:

1. The demonstrator is able to take over from the clerk to perform actions that the recorder can record as the program.
2. The demonstrator is able to ask the clerk to carry out the instructions on the current worksheets in the current state.
3. The clerk is able to ask the demonstrator to take over when the clerk reaches an incomplete part of the program.

The demonstrator is able to take over from the clerk and "drive" the computer, while the recorder watches and writes down code. The implication is that most of the time the clerk is driving the computer. However, in the prototypes of the demonstrational machine this is not the case. The prototypes have the demonstrator in control, driving the machine all the time, and occasionally the demonstrator relinquishes control to the clerk.

The clerk is the person who carries out the instructions when the computer is running a program normally. It is useful for the demonstrator to test their programs by getting the clerk to carry them out. Also the demonstrator may want to add an instruction at a much later point in the program. However, before the demonstrator can add the new instruction, the computer must reach that point in the program. The demonstrator could follow the instructions on the worksheet until the program reaches that point. However, it is quicker, more convenient, and less tiring to have the clerk do it.

The third interaction occurs because the program is demonstrated incrementally. The clerk can reach points in the program that haven't been fully demonstrated yet. When this happens, the clerk stops and asks the demonstrator to take over because the clerk doesn't know what to do. Recall that the abstract machine model assumed that the computer's program was complete (see section 2.2), which is not the case in the demonstrational machine. Examples of partial programs will be shown throughout the examples of programming in this chapter, particularly in section 3.7. How the clerk knows when to ask the demonstrator for help will be discussed in section 4.1.3.

3.3.3.2 Demonstrator and Recorder

There are three important interactions that occur between the demonstrator and the recorder:

1. The demonstrator is able to tell the recorder when to record and when not to record.
2. The demonstrator can ask the record to unwind the computer's state.
3. The demonstrator can ask the recorder to undo the demonstrator's last action(s) entirely.

These interactions give the programmer who uses the demonstrational machine a great deal of latitude for exploring programs. The second and third interactions relate to the recorder's role of recording the state of the machine after every action the demonstrator performs.

Sometimes the demonstrator does not want the recorder to watch and record their actions. For instance, the demonstrator may want to alter the value of a variable in order to demonstrate a particular branch of an `IF` statement. An example of this can be found later in this chapter in section 3.7.13.1. The demonstrator is able to tell the recorder when to watch, and when not to watch the demonstrator's actions. Telling the recorder not to watch ensures that the recorder does not record any code for actions that the demonstrator does. However, the recorder still records the changes to the state to allow unwind and undo.

When programming by demonstration it is useful to take the program back one or more instructions to see them carried out again, to help find a problem in the program, or to demonstrate an alternate case (usually in conjunction with telling the recorder not to watch). Unwinding the computer's state causes the computer to return to the state it was in prior to the last instruction it carried out. Unwinding the state has no effect on the program's instructions or declarations. An example of unwinding can be found later in this chapter in section 3.7.13.1.

The demonstrator will make mistakes while using the demonstrational machine. The demonstrator may demonstrate the wrong action to the recorder, which will put the computer in the wrong state and cause the wrong instruction to be recorded. Asking the recorder to undo the effects of actions provides the demonstrator with a quick and easy way of correcting obvious mistakes. Undoing an action usually unwinds the computer's state *and* erases whatever the code recorder recorded about the action.

3.3.4 The Program

The demonstrational machine starts with completely blank templates, and the program is built by the recorder writing down what the demonstrator does and produces. The program is a little bit like a history of the demonstrators actions. It is certainly *not* a linear history because it can be replayed, stepped through, and altered at any point. Declarations are not necessarily used immediately, and the order that the clerk carries out the instructions is not necessarily the same order that they were demonstrated in.

3.3.5 The Prototypes of the Demonstrational Machine

Two prototypes of the demonstrational machine were built as testbeds for the abstract machine model and demonstration model. The prototypes differed from each other in their complexity, the way they handled some programming concepts, and in the versions of the models that they used. The first prototype is based on a subset of the machine and demonstration models. Evaluating the first prototype resulted in changes to both models. The second prototype was based upon the revised models, and implemented almost all of each model.

The prototypes showed the state of the computer and program using analogies from the machine model (see chapter 2). However, neither prototype was built with any concern about the usability of their user interface. The primary concern was exploring the model, especially the way of programming. Ideally, a properly implemented demonstrational machine would have beautiful 3-D graphics and use lots of animation. However, the prototypes made do with simple 2-D graphics and no animation. Pretty graphics, icons, and user friendliness are things that can be addressed in later versions.

The first prototype allowed programs to be built using:

- Variables and constants of built-in types.
- Assignment.
- One form of selection statement, `IF`, and one form of iteration statement, `LOOP`.
- Complex expressions constructed with a single (large) calculator.
- A variety of built-in subroutines for simple input, output, graphics, and mouse operations. The subroutines are demonstrated using a precursor to worksheets.

All programs built with the first prototype are linear programs. The first prototype does *not* allow any programmer-defined procedures, functions or types. The prototype allows the demonstrator to change the state of the machine, without producing code, by asking the recorder not to watch what the demonstrator is doing. However, the prototype had no capability to undo actions, or to unwind the program's execution. The only way to go back to a previous statement is to carry out the program from the beginning.

The second prototype allows programs to be built using everything that the first prototype allowed. It extended the kinds of program the demonstrator could demonstrate, by adding the following:

- Programmer-defined procedures and functions.
- Programmer-defined types with `ARRAY`s.
- The capability to undo an unlimited number of actions, and to unwind the program's execution back to an arbitrary point in the past.

With these additions, the only part of the model that the second prototype did not support was `RECORD`s. The second prototype differed from the first in the way it supported the construction of complex expressions, and the invocation and execution of subroutines. The second prototype used worksheets extensively, and had several small calculators rather than one large calculator.

The rest of this chapter is divided into two parts: the first part gives an example of using the first prototype to program, the second gives two examples of programming with the second prototype environment. The examples of the second prototype focus on constructs that are not part of the first prototype, or places where the second prototype differs from the first prototype. All the examples show what happens when the programmers makes no mistakes in the course of demonstrating their programs. Understanding how and why the environment works the way it does is not important at this stage. That will be discussed in chapter 4.

3.4 The First Prototype

The first prototype's interface is shown in figure 3.3.

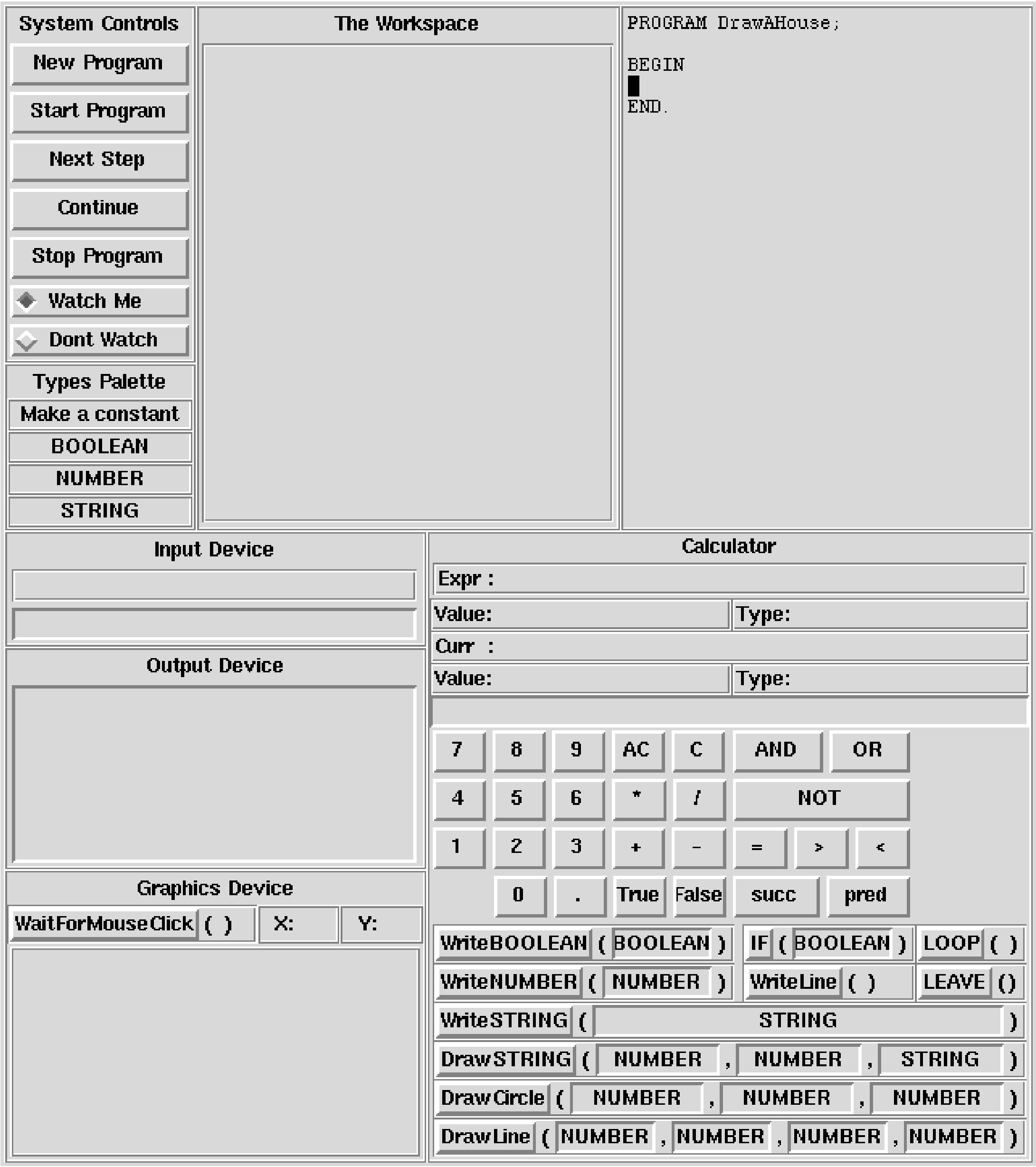


Figure 3.3: The interface of the first prototype.

The interface consists of these components:

- A workspace that represents the memory of the computer (at the top in the center of the interface).
- A region that shows the program, and the instruction that the computer is up to in the program (at the top right of the interface).
- Three regions for the input, output, and graphics devices (at the bottom left). Each region is a combination of one of the devices in the room and the hardware on the

outside the room that the device is connected to. The graphics region contains a pseudo mouse for demonstrating mouse input.

- A palette of types, which displays all the types available for creating places (at the center left).
- A large calculator, which incorporates all the built-in operators, functions, and procedures available to the clerk (at the right beneath the program display). The area of the calculator that contains procedures and functions is called the action(s) palette (at the bottom right). The first prototype implemented a representative subset of the subroutines required in a full environment.
- A number of controls for carrying out the program (at the top left of the interface).

3.5 Example 1 - The Button Program

This example shows the way a programmer uses the first prototype to produce a complete program. The example program is based on a program used in Comp 102 at Victoria University of Wellington. This example shows the following features of the demonstration model and environment:

- Programming by carrying out an action that demonstrates an instruction to the recorder.
- Programming by manipulating the data structures and state of the program and machine.
- Simple text input, text output, and graphics routines — including mouse input.
- Subroutines being carried out via the precursors to partial worksheets.
- Drag and drop manipulation of the arguments to subroutine calls.
- Drag and drop manipulation of values to demonstrate assignment.
- Drag and drop manipulation of types for declaration of simple variables and constants.
- Complex expressions being constructed, and evaluated using a calculator and direct manipulation.
- Control structures being specified though procedures, which have a single expression parameter.

The task is to build a program that asks users to enter a message, draws a simple representation of a button on the screen, puts the message in the button, and then asks them to click the mouse on the screen. If the user clicks outside the button, then the program asks the user to click again. If their click is inside the button, then the program congratulates them and stops. The program does not stop until the user clicks inside the button.

This program includes examples of calling built-in subroutines, arguments, variables, constants, assignment, expressions, iteration, selection, and user input. The full program is (the line numbers are for annotation only):

```
PROGRAM ClickInTheButton;

CONST
  LeftX : 50;
  RightX : 160;
  TopY : 100;
  BottomY : 75;

VAR
  x : NUMBER;
  y : NUMBER;
  message: STRING;

BEGIN
  WriteSTRING('Enter the words on the button.');
  message := ReadSTRING();

  DrawLine(LeftX, TopY, RightX, TopY);
  DrawLine(RightX TopY, RightX, BottomY);
  DrawLine(RightX, BottomY, LeftX, BottomY);
  DrawLine(LeftX, BottomY, LeftX, TopY);

  DrawSTRING(((LeftX + RightX) / 2), ((TopY + BottomY) / 2), message);

  WriteSTRING('Please click the mouse in the button.');
  WriteLine();

  LOOP
    WaitForMouseClick();
    x := MouseX;
    y := MouseY;
    IF (((x>LeftX) AND (x<RightX)) AND ((y>BottomY) AND (y<TopY))) THEN
      LEAVE;
    ELSE
      WriteSTRING('You clicked outside the button. Please try again');
      WriteLine();
    ENDIF;
  ENDLOOP;

  WriteSTRING('Well done! You clicked in the button.');
  WriteLine();
END.
```

To write this program, the programmer becomes the demonstrator and begins with a new program, which is named `ClickInTheButton`.

3.5.1 Demonstrating a subroutine call

To demonstrate the first step of this program, the demonstrator moves to the `WriteSTRING` button on the actions palette and types "`Enter the words on the button.`" into the `STRING` place. The `WriteSTRING` button highlights indicating the button is ready to carry out its action (see figure 3.4). Note that the figures used to illustrate the examples only show the relevant sections of the interface. The demonstrator clicks on the button, and the text they typed (the argument) appears in the output buffer (see figure 3.4). The demonstrator has carried out a built-in procedure call.

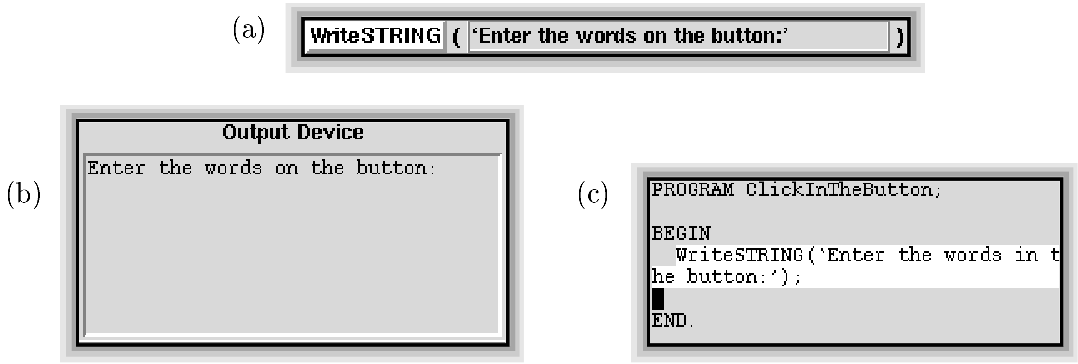


Figure 3.4: (a) `WriteSTRING`'s parameter is filled in and its button is highlighted. (b) After the button is pressed, the `WriteSTRING` call is carried out, (c) which results in the newly demonstrated code.

3.5.2 Making a variable

Before the computer can read a message to display on the button, the computer needs a place to read the message into. The demonstrator constructs a variable place by dragging from a type on the types palette to the workspace. Then the recorder asks the demonstrator to name the place. In this case, the demonstrator drags a `STRING` place to the workspace and calls it `message` (see figure 3.5).

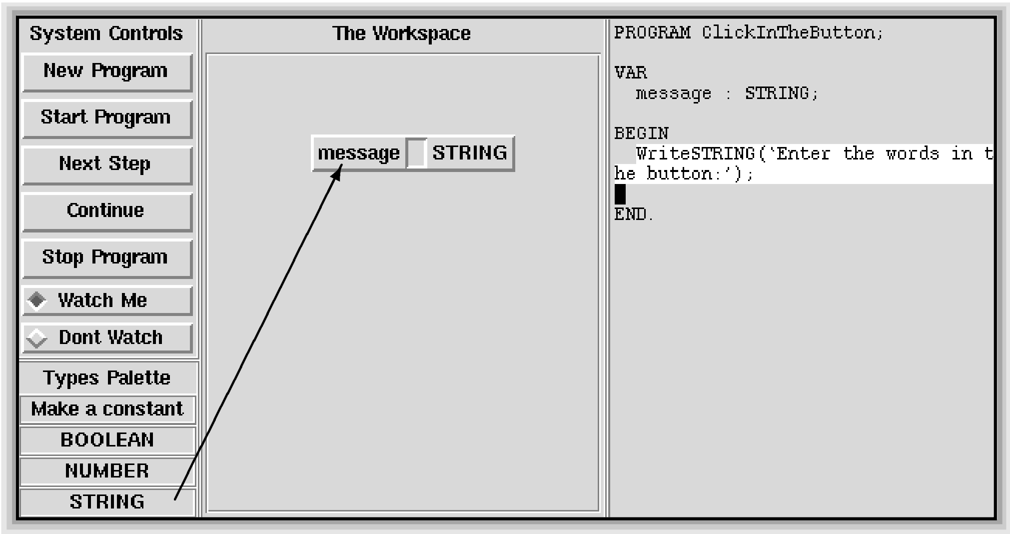


Figure 3.5: Demonstrating a new `STRING` variable, `message`.

3.5.3 Getting text input and assigning it to a place

The next action to show the recorder is reading in a message from the program's user. To do this, the programmer has to momentarily switch roles from the demonstrator to the user because some input must be entered before the demonstrator can show the action of reading input. The user moves the mouse to the input window, types "`Click Here!`", and presses the <`Return`> key. Each character is echoed to the input buffer, and the typing buffer is reset (see figure 3.6).

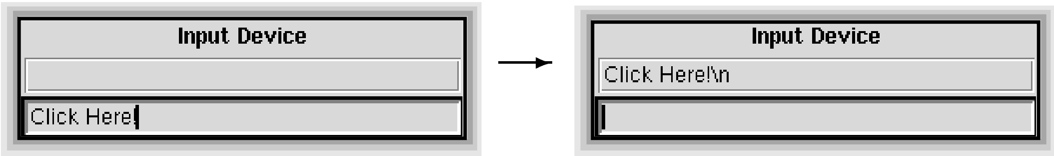


Figure 3.6: User's key presses are echoed to the input buffer from the typing buffer, which is then reset.

The input device is set up and the user returns to the role of demonstrator. The demonstrator selects all the text in the input buffer, and drags the selected text to the `message` place. By doing this, the demonstrator has demonstrated a `ReadSTRING` call, and assigned the function's return value to a place (see figure 3.7).

Figure 3.7: Dragging text input to `message`.

3.5.4 Creating constants for drawing the button.

To define the constants to hold the coordinates of the button's corners, the demonstrator clicks and drags from the constant area on the types palette to the workspace. This creates a constant in the same way that dragging from the `STRING` type to the workspace creates a variable. The demonstrator names the place `LeftX`, and gives it an initial value of `50` (see figure 3.8). The demonstrator constructs the remaining constants in the same way.

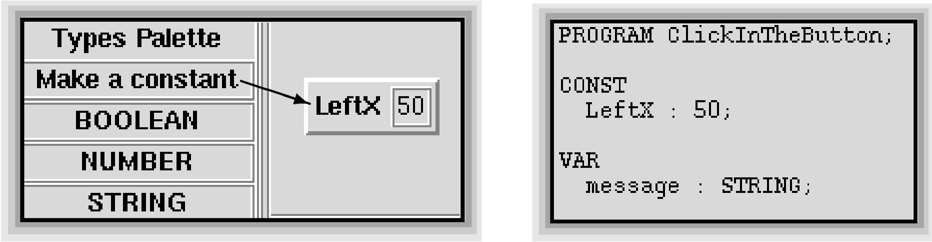


Figure 3.8: Constructing the constant `LeftX`.

3.5.5 Drawing the button.

Drawing the button requires the use of the `DrawLine` procedure. The demonstrator fills in the parameters for `DrawLine` by dragging the values from constants to the appropriate `NUMBER` parameter places next the `DrawLine` button (see figure 3.9). Once all the parameters are filled in, the `DrawLine` procedure is called by clicking on it (See figure 3.9). The other calls to `DrawLine` are demonstrated in the same way.

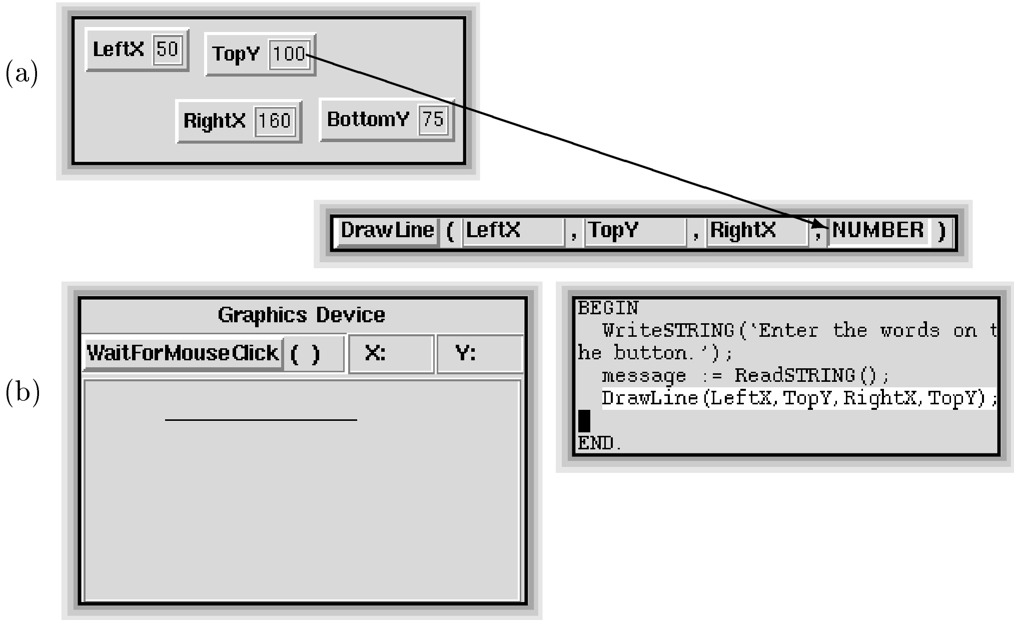


Figure 3.9: (a) Filling in `DrawLine`'s parameters with constants. (b) The results of the first call to `DrawLine`.

The demonstrator intends to use the `DrawSTRING` procedure to put the words inside the button. The demonstrator has to calculate the coordinates of the center of the button. This could be done by creating some more constants, but for this example the demonstrator constructs expressions to calculate the coordinates.

3.5.6 Constructing an expression

The demonstrator constructs expressions using the calculator. The calculator shows several things: the expression constructed so far, the value of the expression (if it can be calculated), the current sub-expression being worked on, and the value of the current sub-expression (if it can be calculated). The current expression is also highlighted in the whole expression. The demonstrator uses the calculator in a similar manner to normal calculator, and copies values between places on worksheets and the calculator.

To calculate the 'x' coordinate of the center of the button, the demonstrator has to construct the expression '$((LeftX + RightX)/2)$'. The demonstrator first drags the value of the constant `LeftX` to the calculator, which results in something similar to pressing a key for a constant ($e.g.$, π) on a standard calculator (see figure 3.10). Then the demonstrator presses the '`+`' button on the calculator and the expression becomes '$(LeftX + \square)$' (see figure 3.10). The place for the second argument to '`+`' is now the current expression.

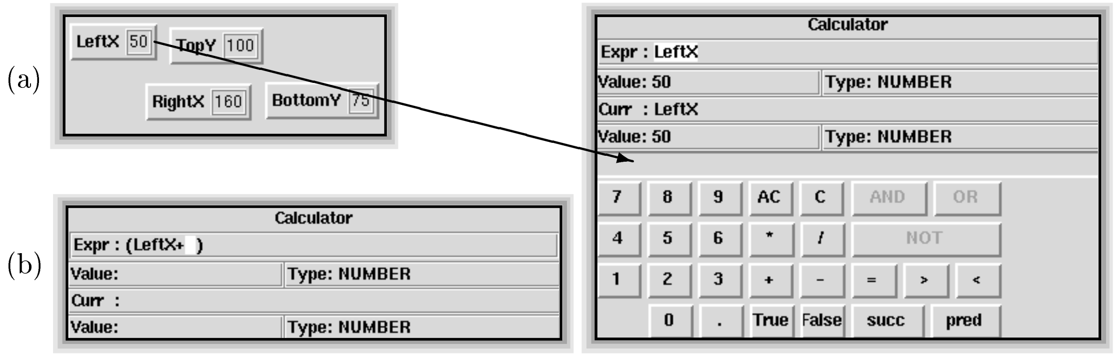


Figure 3.10: (a) Dragging the constant `LeftX` to the calculator. (b) The calculator display after pressing the '`+`' button.

Next the demonstrator drags the value in `RightX` to the calculator, and the expression becomes '$(LeftX + RightX)$'. The expression is complete, the whole expression becomes the current expression, and the value of the complete expression is shown (see figure 3.11). The demonstrator presses the '`/`' key on calculator, which causes the expression to become '$((LeftX + RightX)/ \ \square)$' (see figure 3.11).

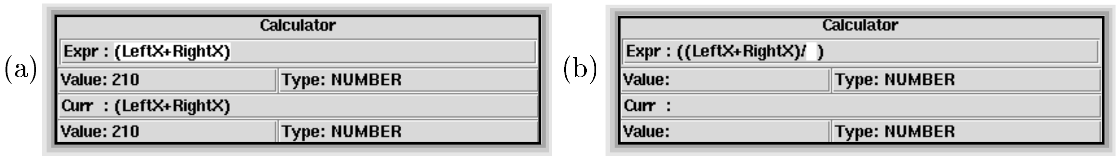


Figure 3.11: (a) The first complete expression. (b) The expression after pressing '`/`'.

To complete the expression the demonstrator enters the number '`2`' via the calculator's keypad. The demonstrator drags the completed expression from the calculator to the first parameter place of `DrawSTRING` (see figure 3.12).

At this point, the demonstrator goes back to the calculator, clears it, and constructs the expression '$((TopY + BottomY)/2)$'. This expression is put in the second `DrawSTRING` parameter. The demonstrator drags the value in the `message` variable to the last parameter of `DrawSTRING`, and clicks on the procedure. The demonstrator has completed drawing the button (see figure 3.13).

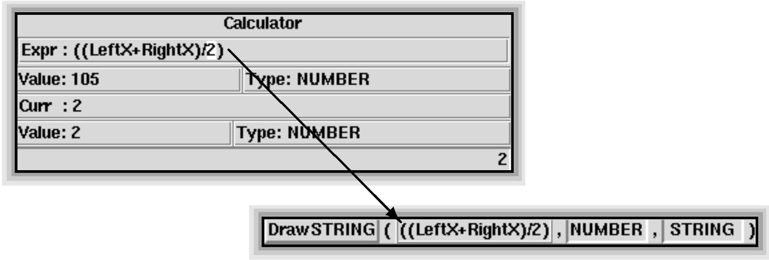


Figure 3.12: A `DrawSTRING` parameter is filled with the completed expression.

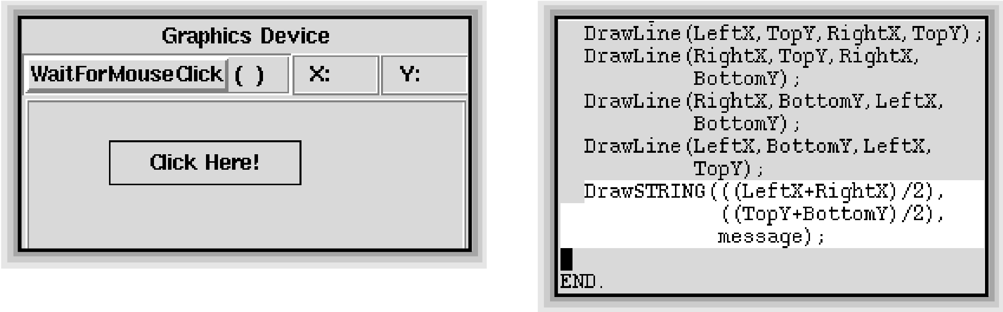


Figure 3.13: The results of the `DrawSTRING` call.

3.5.7 Prompting the user for mouse input

The demonstrator uses another call to `WriteSTRING` to display "`Please click the mouse in the button`" on the computer's screen. Then the demonstrator demonstrates a call to `WriteLine`, which outputs a newline to the text part of the computer's screen. `WriteLine` requires no parameters (see figure 3.14), and is carried out as soon as its button is clicked on. `WriteLine` and the other subroutines that require no parameters are not continuously highlighted because it is distracting.

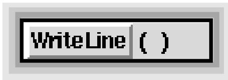


Figure 3.14: `WriteLine`, a procedure with no parameters.

3.5.8 Demonstrating the `LOOP`

A `LOOP` is demonstrated the same way as a procedure with no parameters. The demonstrator clicks on the `LOOP` button, which puts the structure of a `LOOP` in the program at the point the demonstrator is up to (see figure 3.15). A `LOOP` instruction consists of two statements, `LOOP` and `ENDLOOP`, which enclose the body of the loop. Once a `LOOP` instruction is in place, the demonstrator can demonstrate instructions in the `LOOP` body.

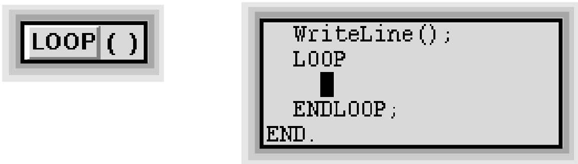


Figure 3.15: The `LOOP` button and the structure it creates in the program.

3.5.9 Getting mouse input

Inside the `LOOP`, the demonstrator calls the `WaitForMouseClick` procedure. This procedure has no parameters; It waits until the computer's user clicks the mouse somewhere inside the graphics window, and then finishes. Once `WaitForMouseClick` is called, the computer cannot do anything except wait until the procedure has finished.

At this point, the programmer takes on the role of the computer's user once more. The user clicks the mouse in the graphics window. This causes the graphics device to update the mouse coordinates it displays (see figure 3.16). It so happens that the user has clicked inside the button drawn in the window.

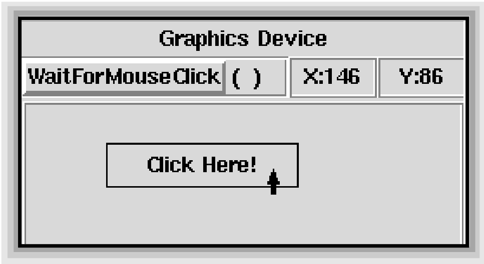


Figure 3.16: After the user has clicked the mouse to complete `WaitForMouseClick`.

The user goes back to the role of demonstrator and creates two new places with the `NUMBER` type. The demonstrator names the places `x` and `y`. The demonstrator drags the values in the mouse coordinate places to `x` and `y` respectively. Each dragging action creates an assignment (see figure 3.17).

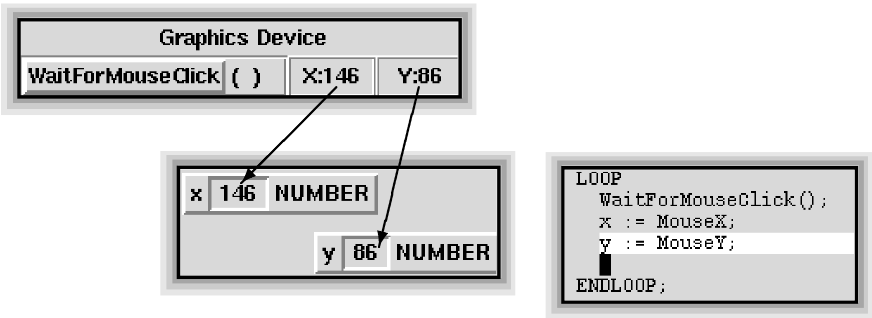


Figure 3.17: Assigning the mouse coordinates to variables.

3.5.10 Demonstrating an `IF` statement

The demonstrator shows the computer how to test whether the point is inside the button by using an `IF` statement. The demonstrator demonstrates the `IF` statement in a similar manner to the `LOOP`. However, the `IF` statement requires a single parameter — a conditional expression. The demonstrator has to construct a condition and drag it to the parameter box for the `IF` statement. Then the demonstrator can generate the `IF` statement by clicking on the `IF` button.

3.5.11 Constructing a complex expression

In this example, the condition for the `IF` statement is quite complex. The expression '$(((x > LeftX) AND (x < RightX)) AND ((y > BottomY) AND (y < TopY)))$' is `TRUE` if the point (x, y) is inside the button. The demonstrator constructs this expression using a slightly different approach to the one used to make the earlier expressions (see section 3.5.6). Both approaches use the calculator and expressions can be constructed with either approach. The second approach is presented here to show that there is more than one way to construct expressions.

The demonstrator is going to construct the expression by laying out the structure of the expression, and then filling in the expression's values. The demonstrator presses the '`AND`' operator on the calculator, which produces the expression '$(\square\ AND\ \square)$' (see figure 3.18). The current expression is the first parameter of the '`AND`' operator because that is the first part of the expression that is not fully specified.

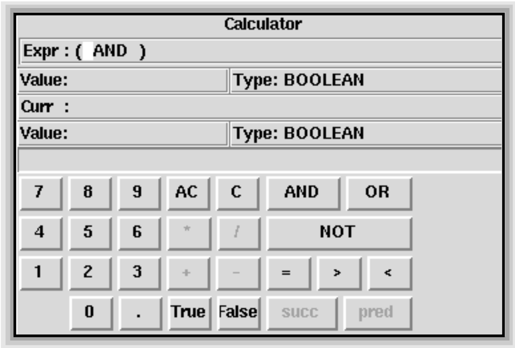


Figure 3.18: An expression containing only an '`AND`' operator.

Next the demonstrator presses the '`AND`' operation again. This creates the expression '$((\square\ AND\ \square) AND\ \square)$'. The current expression is the first parameter of the *inner* most '`AND`' function. However, the demonstrator wants to put a third '`AND`' operator in as the second parameter of the *outer* '`AND`'. So the demonstrator uses the arrow keys to traverse the expression and select the second parameter of the first '`AND`' as the current expression. Then the demonstrator presses the '`AND`' key again, which creates the expression '$((\square\ AND\ \square) AND (\square\ AND\ \square))$' (see figure 3.19). Each unspecified place in the expression will hold a comparison between one of the mouse pointer's coordinates and one side of the button.

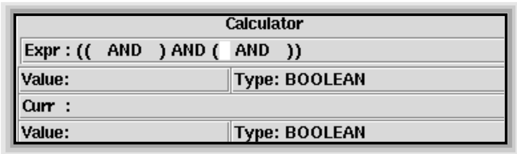


Figure 3.19: The expression after completing the structure of the `AND` operations.

The demonstrator moves the current expression marker back to the first unspecified place in the expression. There the demonstrator presses the '>' key, which produces the expression '$(((\square > \square) AND \square) AND (\square AND \square))$' (see figure 3.20). The demonstrator drags the value in the `x` place onto the calculator. This fills in the first parameter to the '>' operation (see figure 3.20). The demonstrator completes the '>' comparison by dragging the value in the constant `LeftX` to the calculator. The second parameter for the leftmost '`AND`' becomes current expression.

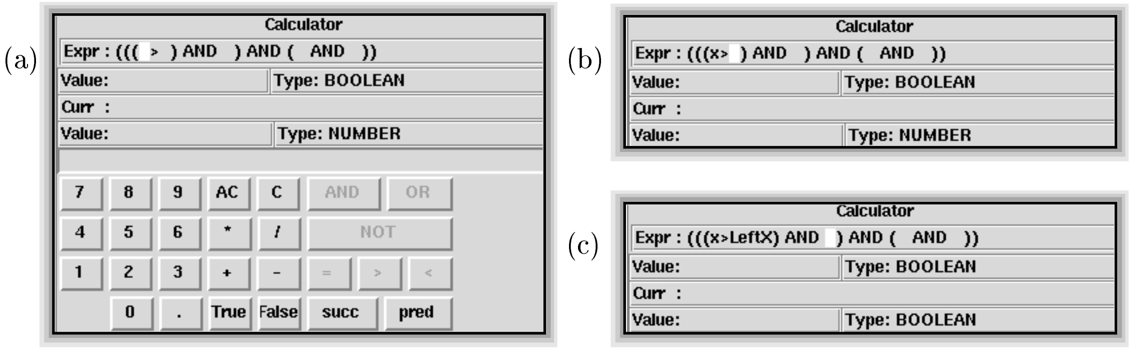


Figure 3.20: (a) The expression after adding a '>' operator, (b) after dragging the value in `x`, (c) and after dragging the constant `LeftX` to the calculator.

3.5.12 Carrying out the `IF` statement

From here, the demonstrator completes the remainder of the expression using the approach just described, or the approach described in section 3.5.6, or a combination of the two. Then the demonstrator uses the complete expression to create an `IF` statement (see section 3.5.10).

The `IF` statement is immediately carried out once it is created. In this case, the expression evaluates to `TRUE` because the user clicked the mouse inside the button. So the program moves to the first branch of the `IF` statement (see figure 3.21).

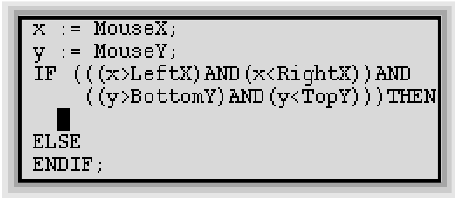


Figure 3.21: In the `TRUE` branch of the `IF`.

3.5.13 Demonstrating a `LEAVE` statement and the end of the program

The mouse was clicked inside the button so the program must stop. The first thing the demonstrator does is make the computer exit the `LOOP` by demonstrating a `LEAVE` statement. `LEAVE` is another control statement, which is specified with a zero parameter procedure like `LOOP` itself. Demonstrating a `LEAVE` statement causes the program to exit the `LOOP` (see figure 3.22). Once outside the `LOOP`, the demonstrator invokes `WriteSTRING` and `WriteLine` to display "`Well done!  You clicked in the button`".

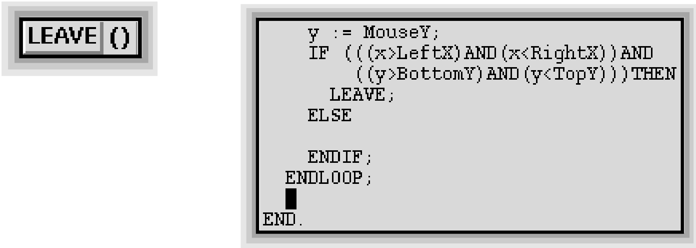


Figure 3.22: The program after leaving the `LOOP`.

At this point, the program is almost complete, except that no actions have been demonstrated as part of the `FALSE` branch of the `IF` statement.

3.5.14 Demonstrating what to do when the `IF` is `FALSE`

In order to demonstrate what to do in the `FALSE` branch, the demonstrator gets the clerk to start the program from the beginning. The demonstrator steps through the program watching the clerk carry out each instruction until the `FALSE` branch of the `IF` statement is entered. This is the only way alternative branches can be demonstrated in the first prototype.

To ensure that the computer enters the `FALSE` branch, the user must click outside the button. This will make the `IF` statement's condition evaluate to `FALSE`. Once inside the `FALSE` branch, the demonstrator takes over from the clerk, and demonstrates writing out "`You clicked outside the button.  Please try again`". Now the program is complete.

3.6 The second prototype

The abstract machine model and demonstration model were changed to reflect our experience with the first prototype environment. The second prototype is based on the newer models, which are described in chapters 2 and 4 respectively. The second prototype allows all the constructs that the first prototype does, and adds several new constructs. It also expands on the amount of control that the demonstrator has over the computer, allowing the demonstrator to unwind the computer's state to a previous point in time.

The examples show the following features of the altered models and environment:

- Programming with the worksheet analogy for subroutine invocation, execution, and return values.
- Demonstration of programmer-defined subroutines.
- Construction, declaration, and use of new data types based on the `ARRAY`.

- Construction of expressions using specialised calculators, and by invoking functions in the correct order of execution.
- Unlimited unwinding of the computer's state.
- Unlimited undo of the demonstrators actions.

The second prototype's interface is shown below in figure 3.23.

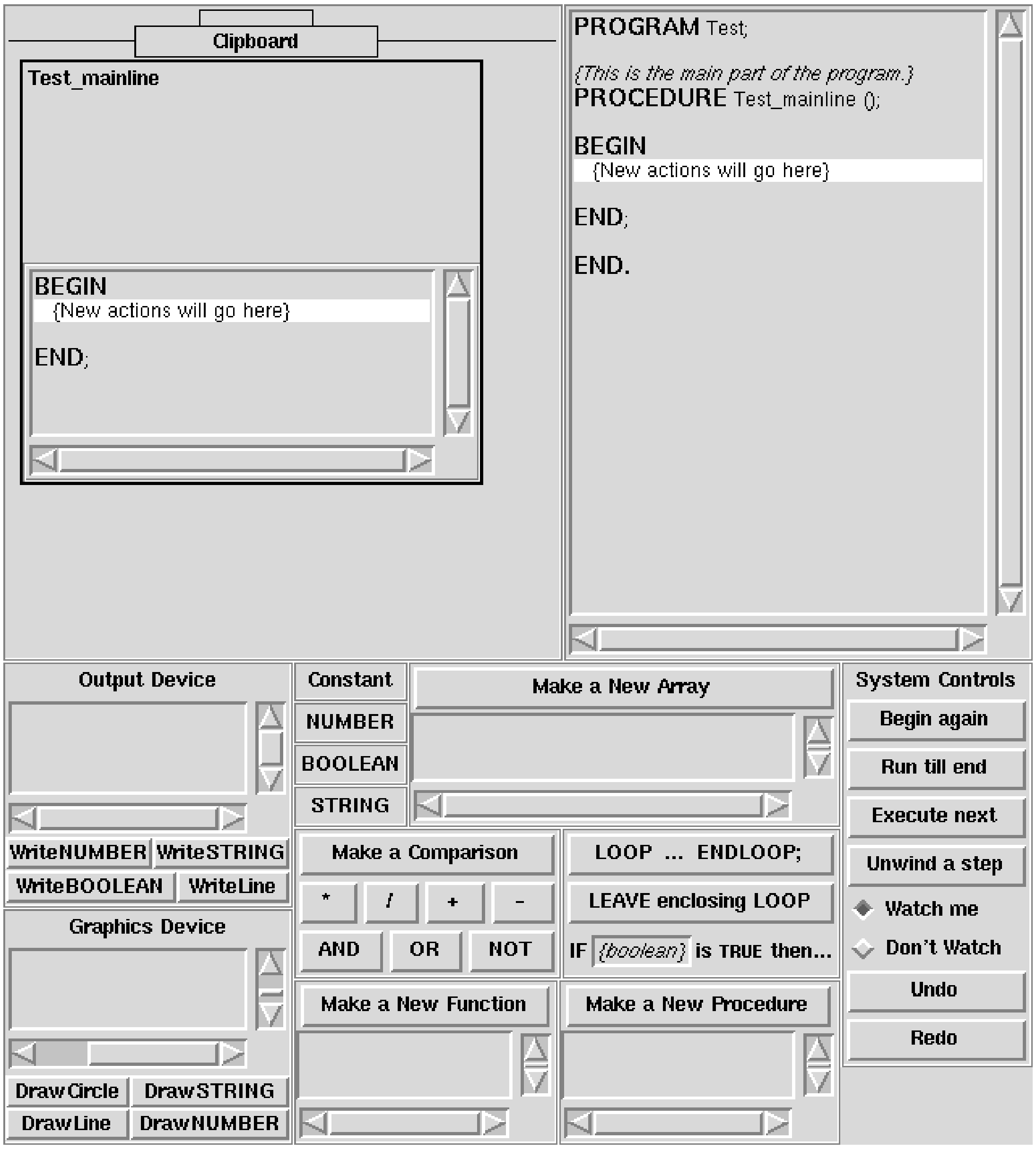


Figure 3.23: The interface of the second prototype.

The second prototype's interface makes the following changes to the first prototype's interface[3]:

- The clipboard replaces the workspace.
- The types palette is extended to include programmer-defined types.
- The subroutines are separated from the calculator and given their own palette.
- The calculator is replaced by several, more specific, calculators that are displayed as needed.

Reading text input from the user was not implemented in the second prototype. As with the first prototype, only a representative set of built-in subroutines were implemented. Only one of the specific calculators was implemented because it was enough for us to explore the method of programming. The other expression operations that were implemented were placed on the subroutines palette. The second prototype used the stricter analogy of array places that was omitted from the abstract machine model. As a result, arrays appear as a carousel of places. The reasons for using this analogy will be discussed in section 4.2.4.

The following two examples show programs that use the extensions of the second prototype. The first example shows a program with a programmer-defined subroutines. The second example shows a program using `ARRAY`s. The examples also show different ways that the second prototype provided for the demonstrator to demonstrate concepts already shown in the first prototype's example (see section 3.5).

3.7 Example 2 - The Factorial Program

This example shows the way a user produces a program using second prototype. The example shows the following features of the demonstration model and environment:

- Defining a new subroutine.
- Demonstrating a parameter for a new subroutine.
- Demonstrating the return type of a new subroutine.
- The invocation, the execution, and the return from programmer-defined and built-in subroutines.
- Constructing expressions using a specialised calculator, and by invoking functions in the correct order (of execution).

The task is to build a program that computes the factorial of a number, using a recursive programmer-defined function.

The program includes examples of programmer-defined functions, recursion, return types, return values, `RETURN` statements, arguments, and parameters.

---

[3]In addition to moving some of the components around.

The full program is (the line numbers are for annotation only):

```
{1 }  PROGRAM RecursiveFactorial;
{2 }
{3 }  FUNCTION Factorial(n : NUMBER) : NUMBER;
{4 }    BEGIN
{5 }      IF (n = 1) THEN
{6 }        RETURN 1;
{7 }      ELSE
{8 }        RETURN (n * Factorial(n - 1));
{9 }      ENDIF;
{10}    END;
{11}
{12}  VAR
{13}    num : NUMBER;
{14}    fact : NUMBER;
{15}
{16}  BEGIN
{17}    num := 5;         { the number is set here for simplicity }
{18}    WriteNUMBER(num);
{19}    WriteSTRING(‘ factorial is ’);
{20}
{21}    fact := Factorial(num);
{22}
{23}    WriteNUMBER(fact);
{24}    WriteLine();
{25}  END.
```

3.7.1 Beginning the Program

The user becomes the demonstrator (programmer) and begins with a new program, which is named `RecursiveFactorial`. The instructions from line `12` through to the `WriteSTRING` call on line `19` are demonstrated the same way as in the first prototype (see section 3.5). At that point, the demonstrator needs to demonstrate a recursive function to compute the factorial of the value in `num`, then return the factorial value to the `fact` place, and output the value in `fact`.

3.7.2 Demonstrating a new function

To do this, the demonstrator clicks on the “`Make a New function`” button in the subroutines palette. The recorder asks the demonstrator to name the function, and the demonstrator names it `Factorial` (see figure 3.24).

(a) (b)

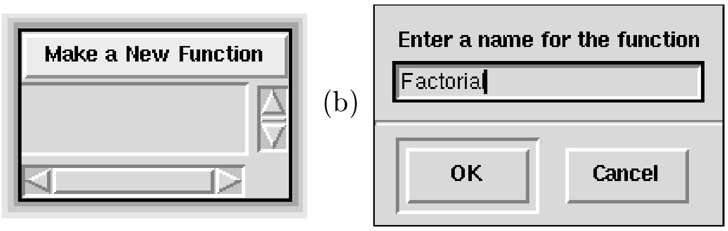


Figure 3.24: (a) The new functions region. (b) Naming the new function.

Making a new function causes the function to be invoked. Invoking a function does three things (see figure 3.25): (a) puts a partially complete function definition in the program, (b) puts a partially complete call to the function in the program, and (c) puts a folded up worksheet on the clipboard. The worksheet is folded up because the arguments for the function's parameters have yet to be specified. The call in the code is partial because neither the arguments for the call, nor the destination for the return value have been specified yet (indicated by the dots, see figure 3.25).

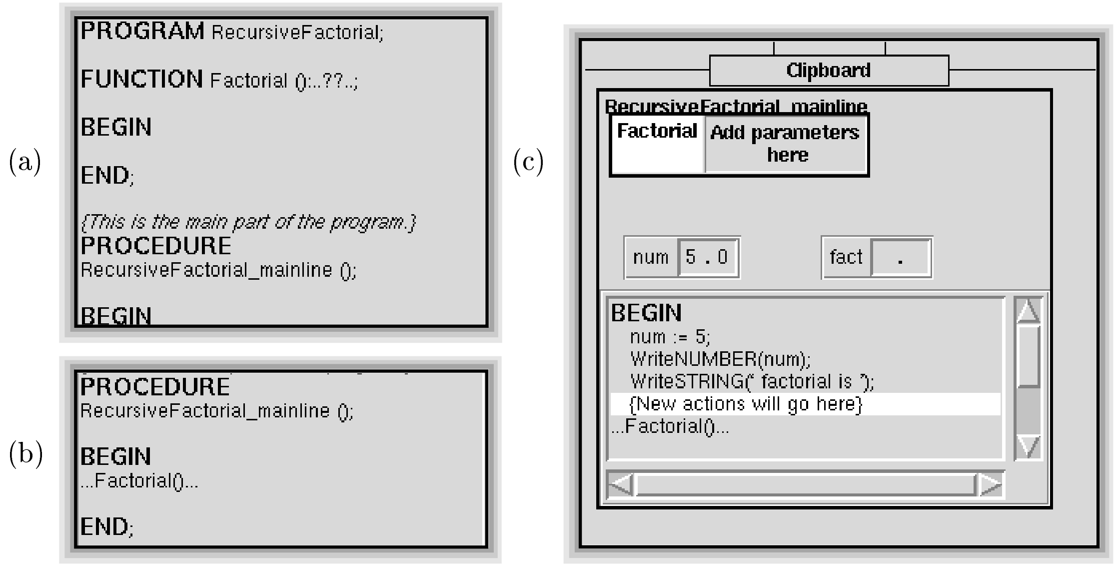
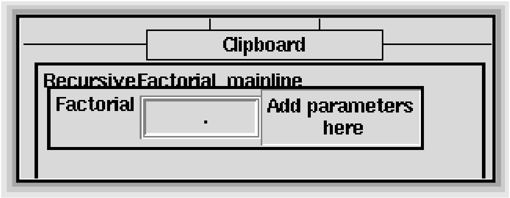


Figure 3.25: (a) The partial function definition, (b) the partial call, and (c) the folded worksheet for the `Factorial` call.

3.7.3 Demonstrating a parameter for the new function

The name on the folded worksheet is highlighted, indicating that all the parameters — there are none as yet — have been filled in and the function can be called (see figure 3.25). However, the `Factorial` function needs to have a parameter. So the demonstrator drags a `NUMBER` type from the types palette to the new parameters area on the folded up `Factorial` worksheet. This makes a new `NUMBER` parameter place. The recorder asks for the name of the place, and the demonstrator replies "`n`".

Now the function's name becomes unhighlighted because it has an unfilled parameter, and thus can't be called. Meanwhile in the program window, the instruction that defines the function's parameter are displayed (see figure 3.26). Note that the name of the parameter place is shown only in the function definition and not on the folded worksheet (see section 2.2.2.7).

(a) 

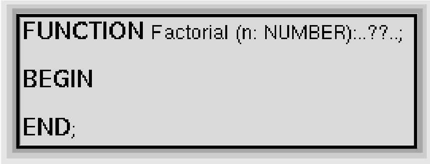
(b) 

Figure 3.26: (a) A new parameter on the worksheet. (b) The new parameter's definition.

3.7.4 Providing an argument to a function

To demonstrate an argument, the demonstrator drags the value in the `num` place to the newly defined `NUMBER` parameter on the folded `Factorial` worksheet. The value appears in `Factorial`'s parameter, and the name of the `Factorial` function becomes highlighted once more indicating that it can be called (see figure 3.27).

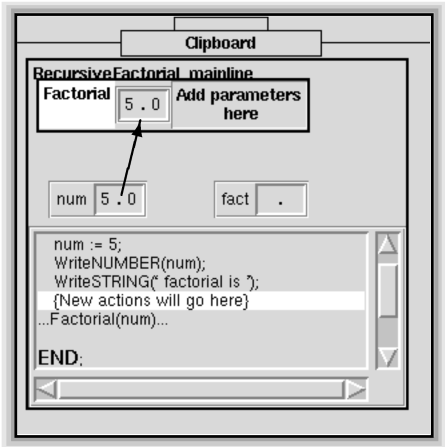


Figure 3.27: Once its parameter is filled in, `Factorial` is ready to be called.

3.7.5 Calling a new function for the first time

The demonstrator calls the new `Factorial` function by clicking on its name on the folded worksheet. At this point, the recorder asks the demonstrator whether the demonstrator wants to: demonstrate the function now, or fake the function's execution and supply a pretend return value. The latter option allows the demonstrator to create functions, but defer demonstrating them in detail until later. Why the recorder even offers the demonstrator this choice will be explained later in section 4.2.3.1.

The demonstrator chooses to demonstrate the function now and the `Factorial` worksheet is unfolded and positioned over the worksheet of the main program (see figure 3.28). The worksheet for the new function now reveals the name of its parameter place. The worksheet has empty areas for local variable places and for instructions. At this point, a button for invoking the `Factorial` function appears in the demonstrator-defined functions palette.

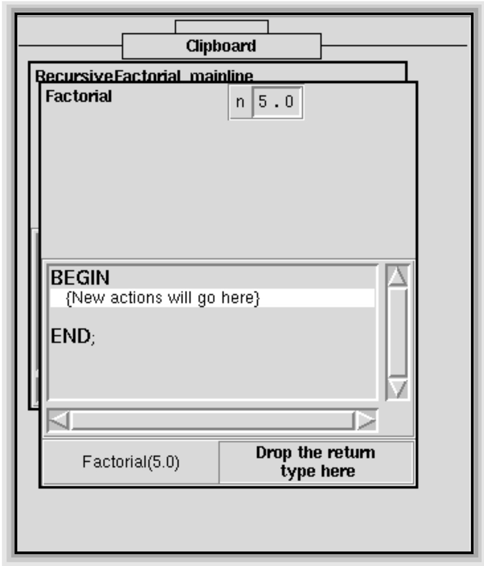


Figure 3.28: The unfolded `Factorial` worksheet.

3.7.6 Demonstrating a return type

To define the type of the return value, the demonstrator drags a `NUMBER` from the types palette to the return place on `Factorial`'s worksheet (see figure 3.29).

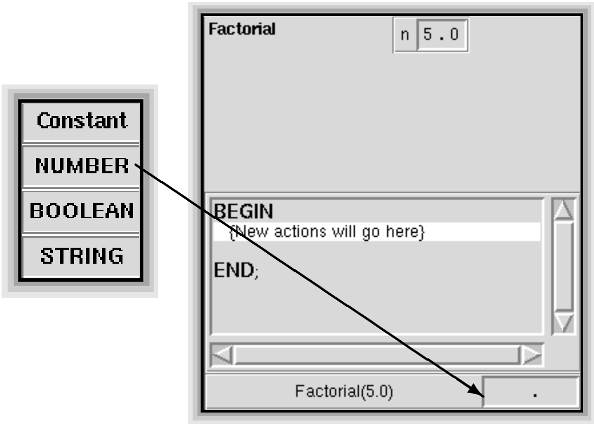


Figure 3.29: Defining `Factorial`'s return type on the unfolded worksheet.

Now the demonstrator demonstrates instructions for the `Factorial` function. The `IF` statement (line `5`) is demonstrated in a similar way to the first prototype: the demonstrator constructs a conditional expression and uses it to create the `IF` structure (see section 3.5.10). However, constructing an expression is a little different with the second prototype.

3.7.7 Demonstrating an expression with the comparison calculator

To create the expression '$n = 1$' the demonstrator clicks on the "`Make a comparison`" button. This brings up the comparison calculator, or comparator, which is one of the specialised calculators available to the computer (see figure 3.30). The comparator only has operations that compare two values of the same simple type.

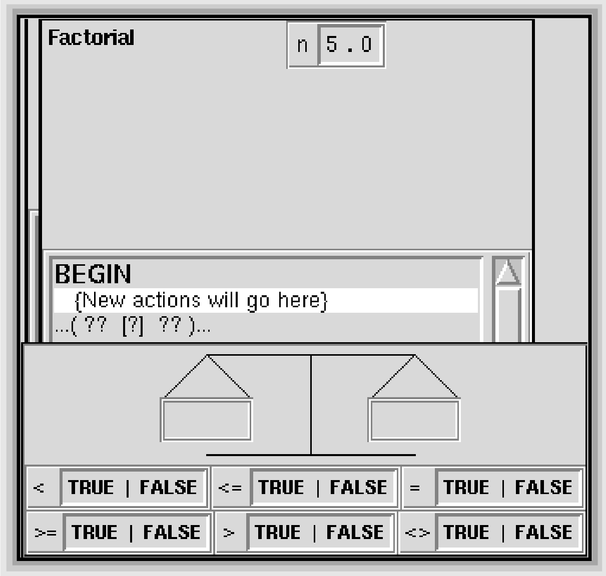


Figure 3.30: The comparison calculator (the comparator).

To make the comparison, the demonstrator drags the value in `n` to the first parameter place on the comparison calculator, the left hand side of the scales. This immediately causes both parameter places to become `NUMBER` places — the same type as the type of value that the demonstrator dropped (see figure 3.31) — which ensures that the same type of values will be compared. Then the demonstrator types "`1`" into the second parameter place, the right hand side of the scales. Immediately, the six comparison operations are carried out and their results displayed in the places below the operators (see figure 3.31). The comparison calculator is like a special function that carries out several comparison operations at once. The places below the operators show their return values.

(a)

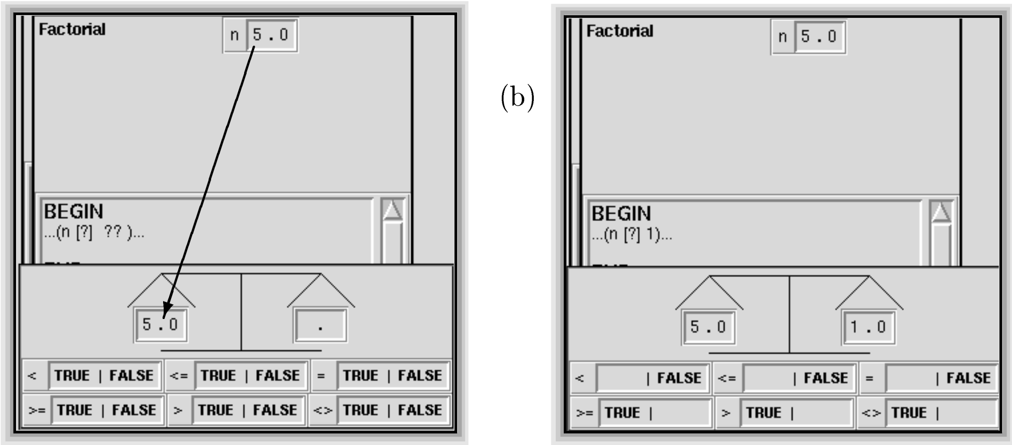


Figure 3.31: (a) One value in the comparator. (b) Both parameters filled in.

To create the expression, the demonstrator selects the return value from the '`=`' comparison and drags it to the parameter of the `IF` statement. This creates an `IF` statement with the condition '$n = 1$', which is carried out immediately. As the value in `n` does not equal `1` (`n`'s value is `5`) the program enters the `ELSE` branch of the `IF` statement (line `8`).

3.7.8 Demonstrating an expression by invoking functions in order

To demonstrate the `RETURN` statement, the demonstrator must construct the expression and drag it to the return place. The comparison calculator is one new way that the second prototype has of constructing expressions; the other new way is to call functions and nest their return values in the correct execution order. The demonstrator constructs the expression '$(n * Factorial(n-1))$' for the `RETURN` statement by nesting function calls.

To do this, the demonstrator invokes a call to the built-in '`*`' function by clicking on the '`*`' button in the subroutines palette. This causes a folded up multiplication worksheet to appear on top of the `Factorial` worksheet, and a disconnected '`*`' function to appear in the code (see figure 3.32).

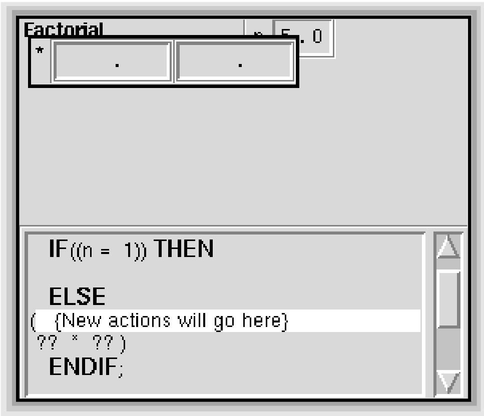


Figure 3.32: The invocation of '`*`'.

The demonstrator fills the first parameter to the multiplication call by dragging and dropping the value from `n` on the `Factorial` worksheet. To specify the second parameter, `Factorial(n-1)`, the demonstrator has to demonstrate a recursive call to `Factorial`.

3.7.9 Demonstrating a recursive call

The demonstrator invokes the `Factorial` function by clicking on its button in the functions palette. This produces a folded up `Factorial` worksheet, which is distinct from the one already open. The new folded worksheet is placed partially over the folded worksheet for '`*`', see figure 3.33.

(a)

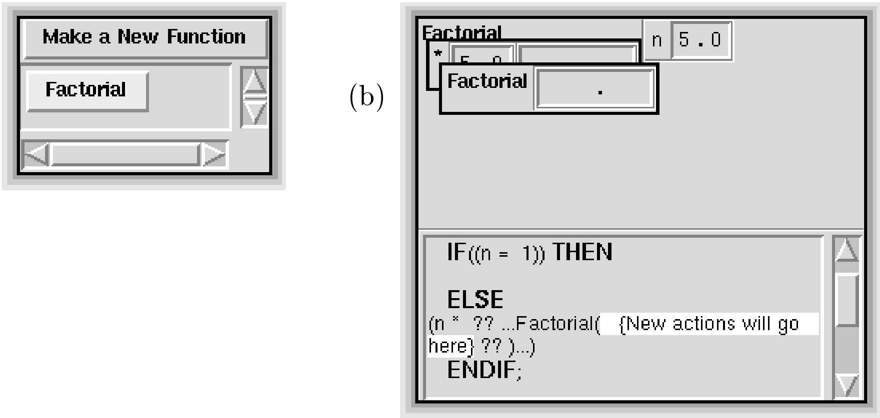


Figure 3.33: (a) The `Factorial` button in the functions palette. (b) The invocation of a recursive call to `Factorial`.

The user can't fill in the parameter for the recursive `Factorial` call yet because they need the return value from '$n - 1$' (see line 8). So the demonstrator invokes a call to '`-`', and a folded up worksheet for '`-`' is placed on top of the folded `Factorial` sheet (see figure 3.34). The demonstrator drags the value in `n` to, and enters "`1`" into, the two parameter places on the folded '`-`' worksheet. The subtraction function can be called now, and the demonstrator clicks on the '`-`' to call it. The subtraction is carried out, and its worksheet turns into a folded up worksheet which has the bottom of the sheet visible (see figure 3.34). The unfolded worksheet stage was skipped because '`-`' is a built-in operation. The folded up partial worksheet with its return part visible shows the return value of the '`-`' operation.

(a)

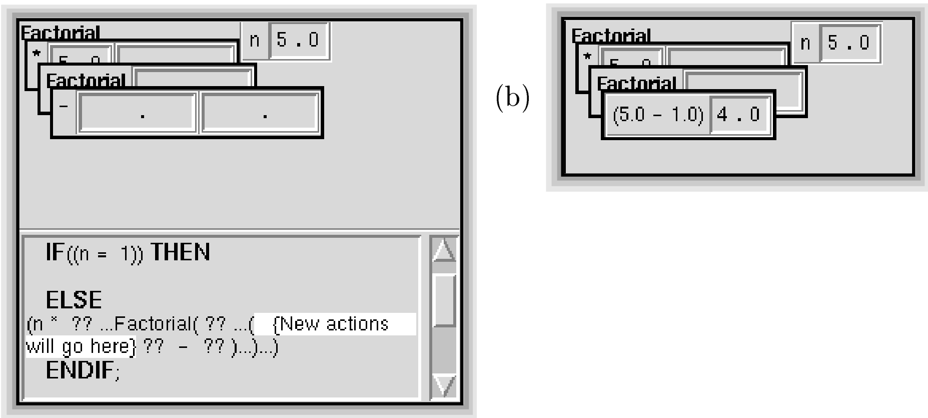


Figure 3.34: (a) The invocation of '`-`'. (b) The return of '`-`'.

3.7.10 Using the return value of a function

From here the demonstrator drags '`-`'s return value to the parameter place of the folded `Factorial` worksheet (see figure 3.35). The '`-`' call becomes connected to the `Factorial` call in the instructions. Previously, it was not properly connected to any other instruction.

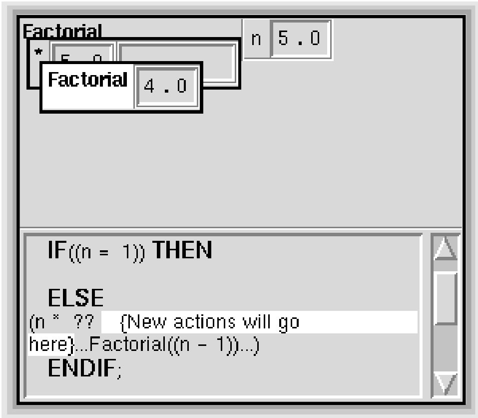


Figure 3.35: After the return value of '-' is used as an argument to the `Factorial` call.

All the parameters of the recursive invocation of `Factorial` are filled in, so the demonstrator calls the function. At this point, the recorder again asks the demonstrator whether the demonstrator wants to carry out the function, or just *pretend* to carry it out.

3.7.11 Two approaches to completing the demonstration of `Factorial`.

There are a two primary approaches that the demonstrator can take to demonstrate the rest of the instructions in the `Factorial` function. The demonstrator could follow the execution of the recursive call as it calculates the factorial of `4`. Or, the demonstrator could fake the recursive call, and then unwind the computer's state to demonstrate instructions in the `TRUE` branch of the `IF` statement.

The following sections show both approaches to demonstrating the rest of the program — the recursive approach is shown first, and then the unwinding approach is shown. Once the `Factorial` function is completely defined, demonstrating the final few lines of the program is the same for either approach.

3.7.12 Approach 1: Following the recursion down

If the demonstrator carries out the recursive `Factorial` call. `Factorial(4)`'s worksheet unfolds to full size and covers the other worksheets on the clipboard (see figure 3.36). The demonstrator asks the clerk to carry out the next statement, by clicking on the "`Carry out next instruction`" button in the control panel. Each of the following steps happens as the demonstrator asks the clerk to carry out the instructions. The demonstrator has nothing to demonstrate until the clerk reaches the `TRUE` branch of the `IF` statement.

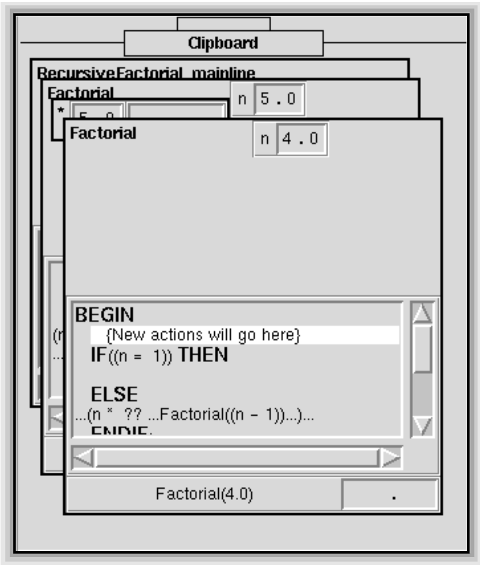
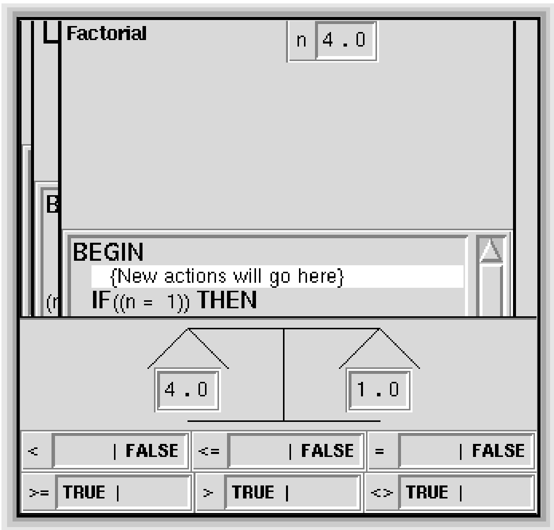


Figure 3.36: The unfolded `Factorial` worksheet when '$n = 4$'.

The demonstrator asks the clerk to carry out the next statement, the `IF` statement on line `5`. The clerk carries out the `IF` statement by evaluating the conditional expression. The clerk uses the comparator with the values of `4` (the value in `n`) and `1`. The condition '$4 = 1$' is evaluated and the clerk goes to the `FALSE` branch of the `IF` (see figure 3.37).

(a)



(b)

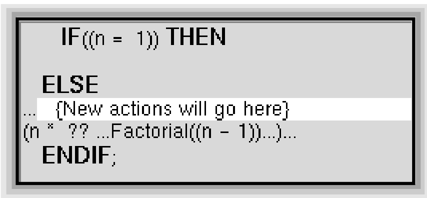


Figure 3.37: (a) Comparing `4` and `1`. (b) Inside the `FALSE` branch.

The clerk begins to carry out the next statement. However, the statement is not a complete statement. Rather, it is an expression that hasn't been connected yet. So the clerk goes ahead and evaluates the expression '$\ldots(n*\ldots(Factorial(n-1))\ldots)\ldots$'. First the clerk invokes '`*`' (see figure 3.38).

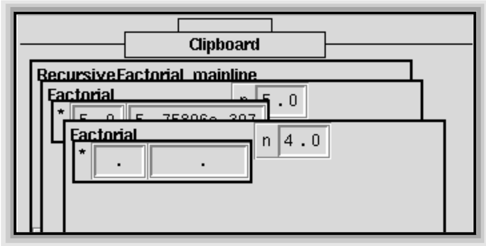


Figure 3.38: After the clerk has invoked `*`.

Then the clerk tries to evaluate the arguments for the '`*`' call. The clerk fills in the first parameter with `4` (the value in `n`), and moves to the next parameter of '`*`', which is unspecified. The clerk checks to see if there is an expression that hasn't been fully connected and finds a call to `Factorial`. So the clerk invokes the `Factorial` function (see figure 3.39). If there hadn't been a partially connected function there, the clerk would have stopped and asked the demonstrator take over.

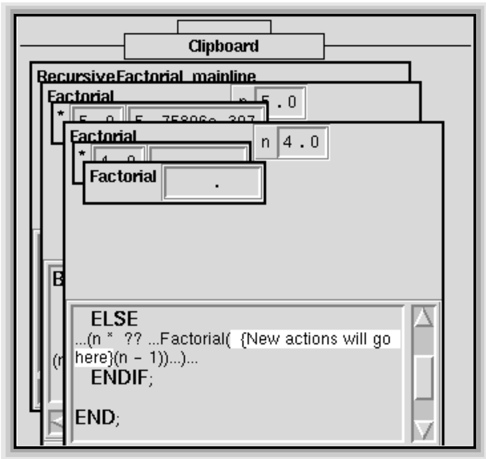


Figure 3.39: After the clerk has invoked `Factorial`.

The clerk works out the argument for the `Factorial` call's parameter by invoking '`-`', filling in the two parameter with `4` and `1`, and calling '`-`'. The '`-`' function returns `3` which the clerk uses as the argument for the `Factorial` call. Then the clerk calls the `Factorial` function (see figure 3.40). The clerk goes through the same process but with '$n = 3$', and then with '$n = 2$' (see figure 3.40). However, '$n = 1$' is different because it is the base case.

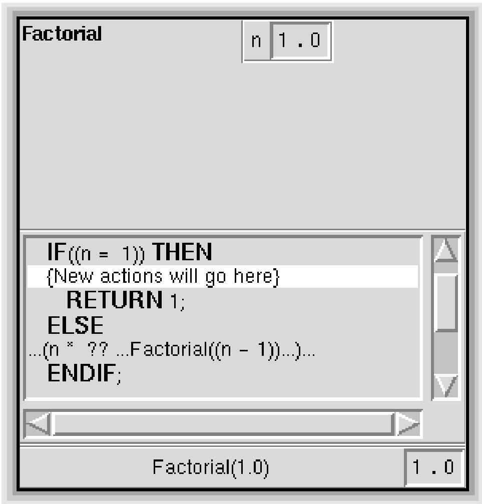
(a) (b) (c)

Figure 3.40: (a) Carrying out `Factorial` when '$n = 3$'. (b) Carrying out `Factorial` when '$n = 2$'. (c) The unfolded `Factorial` worksheet when '$n = 1$'.

3.7.12.1 Demonstrating the base case at the end of the recursion

The clerk begins the instructions for `Factorial(1)` by carrying out the `IF` statement (see figure 3.41). The clerk evaluates the condition and goes to the `TRUE` branch because the value in `n` is equal to `1`. At this point, the demonstrator takes control and types "`1`" into the return place on `Factorial`'s worksheet to demonstrate a `RETURN` statement (see figure 3.41). The `Factorial` worksheet is folded up leaving only the bottom section with the return place visible (see figure 3.41).

(a)



(b)

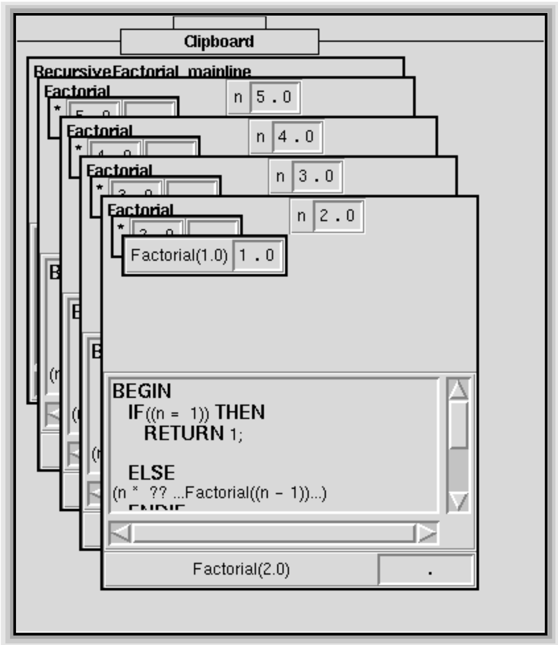


Figure 3.41: (a) `Factorial`'s return place holding the value `1`. (b) The re-folded `Factorial(1)` worksheet on top of the stack of worksheets.

Then the demonstrator drags `Factorial(1)`'s return value to the second parameter for the '*' call, which is inside the call to `Factorial(2)`. This makes the call to `Factorial(n-1)` become '*'s second parameter (see figure 3.42). Up to this point, these two function calls were not properly connected.

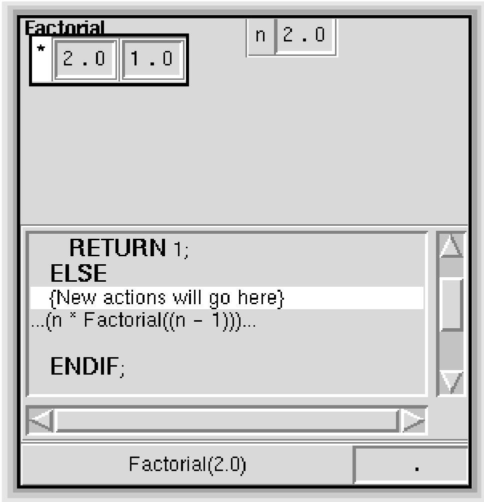


Figure 3.42: '*' with all parameters filled in.

3.7.12.2 Following the recursion back up

Now that '*' has both its parameters filled in, the demonstrator calls it, and drags its return value (2) to the return place for the `Factorial(2)` call. This creates a `RETURN` statement with the expression `(n * Factorial(n - 1))`. The statement causes the `Factorial(2)` worksheet to fold up and return 2 (see figure 3.43). Prior to this, the '*' function was not connected to the rest of the instructions.

(a) (b)

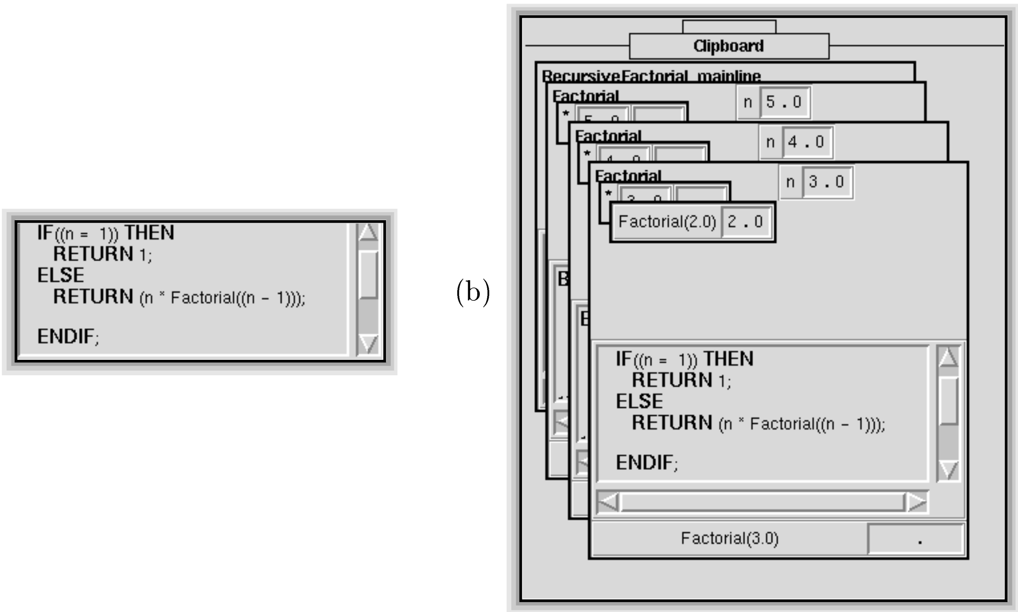


Figure 3.43: (a) The code that resulted from using the '*' call in a `RETURN` statement. (b) The return from the `Factorial(2)` call.

Now the demonstrator gets the clerk to carry out the instructions until all the `Factorial` calls have returned (see figure 3.44). The demonstrator does not have to demonstrate anything for the earlier `Factorial` calls because the recorder has propagated the changes made to the `Factorial` function to all the copies of its worksheet that are on the clipboard. So the expressions in all the `Factorial` calls are fully connected to the instructions. Demonstrating the final part of the program is described in section 3.7.14.

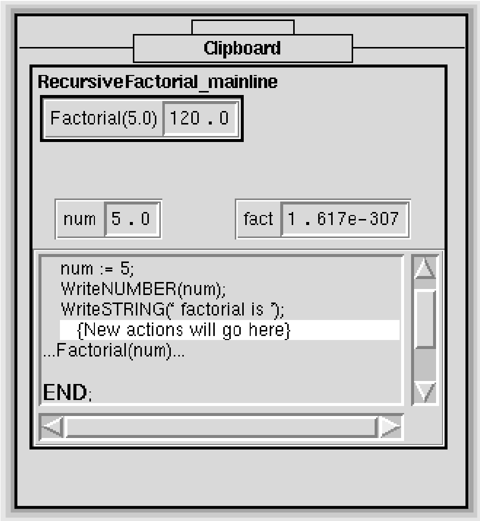


Figure 3.44: The return from first `Factorial` call with a value of `120`.

3.7.13 Approach 2: Pretending to carry out the recursion and unwinding instead

In this approach, the demonstrator decides to *pretend* to carry out the `Factorial(4)` call. The recorder asks the demonstrator for a `NUMBER` value to use as the value returned from the call. The demonstrator enters "`13`", and `Factorial(4)`'s worksheet is folded up to display a return place just as if it had been carried out and returned (see figure 3.45).

Then the demonstrator drags the value "returned" from the "call" to `Factorial(4)` to the second parameter of the '`*`' call. The demonstrator carries out the multiplication, and drags the result to the return place on the `Factorial(5)` worksheet. This creates a `RETURN` statement that uses the completed expression, '$(n * Factorial(n-1))$' (see figure 3.46). The `RETURN` statement is carried out and the `Factorial` worksheet is folded up (see figure 3.46).

At this point, the demonstrator could go on to demonstrate the end of the program, except that the `TRUE` branch of the `IF` statement inside `Factorial` is not complete. Until the computer knows what to do when '$n = 1$', the `Factorial` function will not work properly.

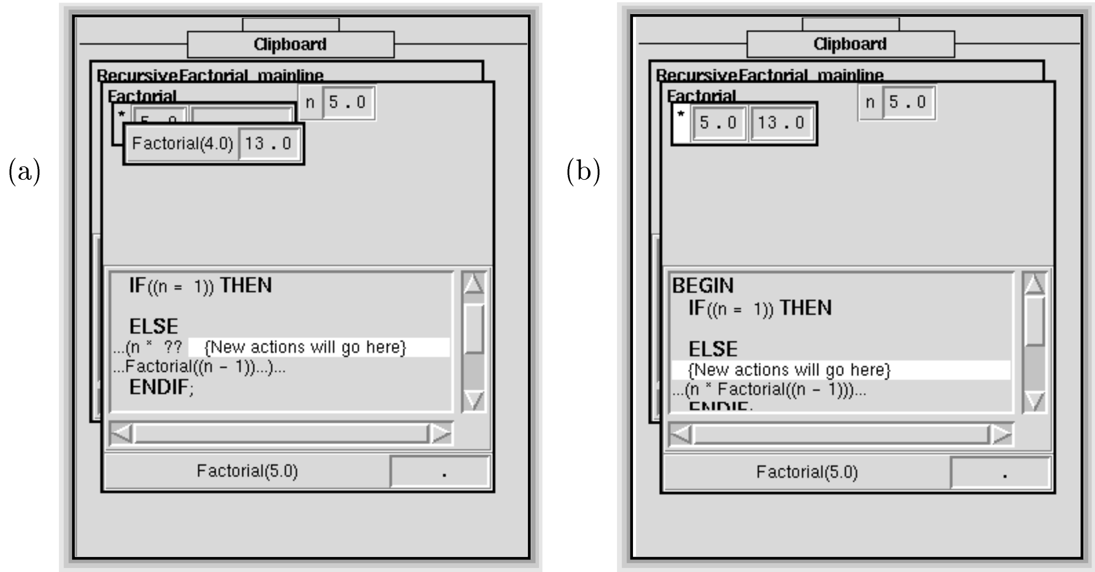


Figure 3.45: (a) The return value from the pretend `Factorial` call. (b) The pretend return value is used as an argument to '`*`'.

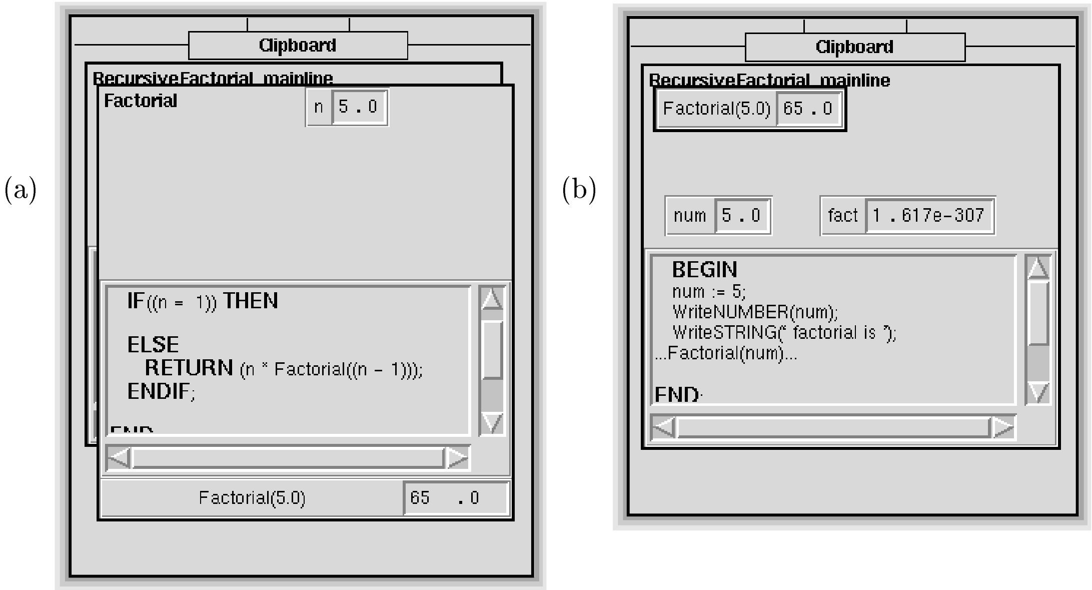


Figure 3.46: (a) The `RETURN` statement created for the `Factorial` function. (b) `Factorial(5)` after returning with the value of `65`.

3.7.13.1 Demonstrating the base case without using recursion

To demonstrate what to do in the `TRUE` branch, the demonstrator unwinds the state of the computer to just before the `IF` statement. Unwinding is done one step at a time. Each step reverts the state of the computer to a previous state, but does not change the instructions that are part of the program. The demonstrator stops unwinding when the computer's state reaches the point to just prior to the `IF` statement (see figure 3.47).

(a) (b)

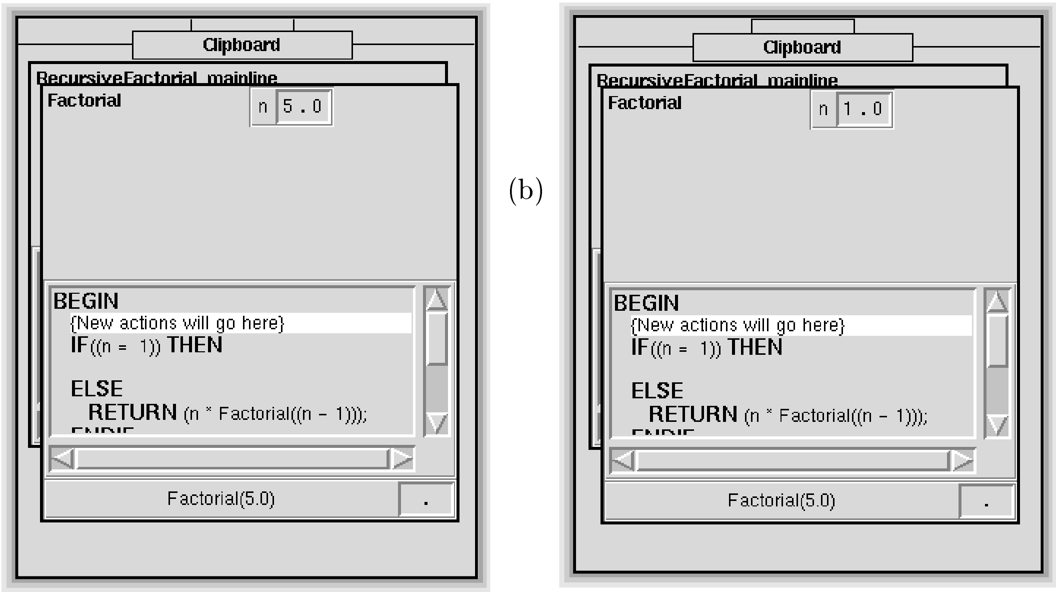


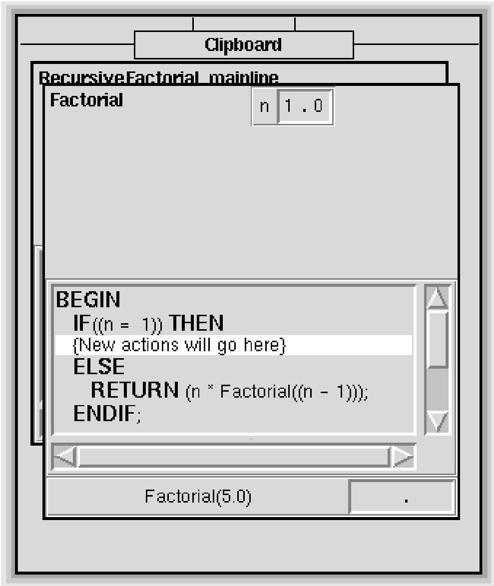
Figure 3.47: (a) Unwound to a point just to prior to the `IF` statement. (b) The new value in the `n` place.

The demonstrator selects the "`Don't Watch`" option on the control panel. Now the recorder is not watching what the demonstrator does, and will not make any changes to the instructions. The demonstrator can still do things though. The demonstrator enters "`1`" into the `n` place. This would normally result in the recorder writing down an assignment instruction ('`n:=1;`'). However, because the recorder isn't watching, all that happens is that the value in `n` changes (see figure 3.47).

Then the demonstrator selects the "`Watch me`" option, which makes the recorder pay attention to what the demonstrator does. The demonstrator asks the clerk to carry out the next instruction, the `IF` statement. The clerk carries out the `IF` statement, evaluating '$n = 1$' to `TRUE` in the current state, and goes to the `TRUE` branch of the `IF` (see figure 3.48).

At this point, the demonstrator takes over and demonstrates another `RETURN` instruction by typing "`1`" into the return place on `Factorial`'s worksheet. The `RETURN` instruction folds up the worksheet leaving only its return value visible (see figure 3.48).

(a)



(b)

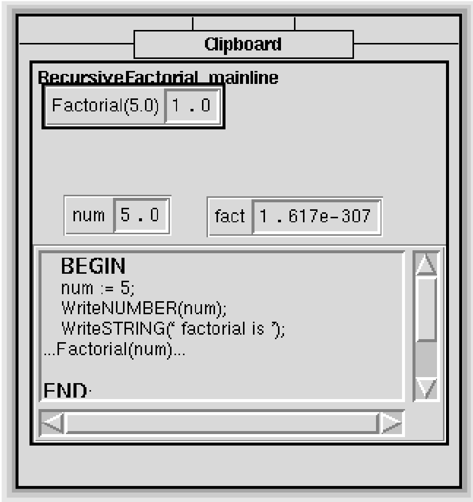


Figure 3.48: (a) In the `TRUE` branch of the `IF` with '$n = 1$'. (b) Returning the value `1` from `Factorial(5)`.

3.7.14 The end of the program

To complete the program the demonstrator drags the return value of the `Factorial` function to the `fact` place, and demonstrates the last few `Write` procedures that output the value in the `fact` place. The value returned by the `Factorial` function depends on the path the demonstrator took to demonstrate it. Following the recursion down to the base case produces a value of `120`, which is the correct value of `Factorial(5)`. Whereas, pretending and unwinding produces a value of `1`, which is not the correct value. However, in both cases the instructions for calculating the factorial of a number with the `Factorial` function are quite correct. If the program was run again from the beginning it would produce the correct result for `Factorial(5)`.

3.8 Example 3 - Selection Sort with an Array

This example focuses on demonstrator-defined types. The demonstrator can create new types with two constructs: `ARRAY`, and `RECORD`. This example deals only with `ARRAY` because `RECORD` is not implemented in the second prototype. The example shows the following new features of the demonstration model and environment:

- Demonstrating a new programmer-defined type.
- Demonstrating a variable of a new type.
- Demonstrating parameters and return values with a new type.
- Using values of a new type.
- Using the elements in a new type.

The task is to build a program that creates a list of numbers, sorts the list using selection sort, and outputs the sorted list. The list is to be stored as a new type, sorted in a new function, and written out with a new procedure. The full program is (the line numbers are for annotation only):

```
{1 }  PROGRAM SortAList;
{2 }
{3 }  CONST
{4 }    MaxList : 10;
{5 }
{6 }  TYPE
{7 }    NUMBERLIST : ARRAY[1..MaxList] of NUMBER;
{8 }
{9 }  PROCEDURE WriteNUMBERLIST(l : NUMBERLIST);
{10}    VAR
{11}      index : NUMBER;
{12}
{13}    BEGIN
{14}      index := 1;
{15}      LOOP
{16}        IF (index > MaxList) THEN
{17}          LEAVE;
{18}        ELSE
{19}          WriteNUMBER(l[index]);
{20}          index := index + 1;
{21}        ENDIF;
{22}      ENDLOOP;
{23}      WriteLine();
{24}    END;
{25}
{26}  FUNCTION SortNUMBERLIST(unsorted_list : NUMBERLIST) : NUMBERLIST;
{27}    VAR
{28}      index : NUMBER;
{29}      sortedto : NUMBER;
{30}      min : NUMBER;
{31}      temp : NUMBER;
{32}
{33}    BEGIN
{34}      sortedto := 1;
{35}      LOOP
{36}        IF (sortedto > MaxList) THEN
{37}          LEAVE;
{38}        ELSE
{39}          LOOP
{40}            min := sortedto;
{41}            index := sortedto;
{42}            IF (index > MaxList) THEN
{43}              LEAVE;
{44}            ELSE
{45}              IF (unsorted_list[min] > unsorted_list[index]) THEN
{46}                min := index;
{47}              ELSE
```

```
{48}                { Nothing demonstrated for this branch yet }
{49}              ENDIF;
{50}              index := index + 1;
{51}            ENDIF;
{52}          ENDLOOP;
{53}
{54}          temp := unsorted_list[sortedto];
{55}          unsorted_list[sortedto] := unsorted_list[min];
{56}          unsorted_list[min] := temp;
{57}          sortedto := sortedto + 1;
{58}        ENDIF;
{59}      ENDLOOP;
{60}      RETURN unsorted_list;
{61}    END;
{62}
{63}
{64}  VAR
{65}    list : NUMBERLIST;
{66}
{67}  BEGIN
{68}    list[1] := 42;
{69}    list[2] := 34;
{70}    list[3] := 1;
{71}    list[4] := 10;
{72}    list[5] := 50;
{73}    list[6] := 4;
{74}    list[7] := 56;
{75}    list[8] := 703;
{76}    list[9] := 12;
{77}    list[10] := 1500;
{78}
{79}    WriteNUMBERLIST(list);
{80}    list := SortNUMBERLIST(list);
{81}    WriteNUMBERLIST(list);
{82}  END.
```

This example does not describe the details of demonstrating most of this program. It only describes the parts that concern new types. The parts of the program not covered below are demonstrated using the methods described in the previous examples.

3.8.1 Demonstrating a new type

Let's assume that the demonstrator has made the constant `MaxList` (line `4`). The demonstrator clicks on the "`Make a new type with an ARRAY`" button on the types palette. The recorder asks the demonstrator to provide a name for the new type. The demonstrator calls it `NUMBERLIST`, and a new type button labelled `NUMBERLIST` appears in the types palette (see figure 3.49).

The button has two boxes associated with it: one is the place for the element type of the ARRAY, the other is a place for the size of the array. The demonstrator must fill in the size and element type places before the NUMBERLIST type can be used. The demonstrator drags the value in the MaxList to the size place, which sets the size of the NUMBERLIST. Then the demonstrator drags from the NUMBER type on the types palette to the element place, which makes the elements of the NUMBERLIST be NUMBER places. Now the new type is ready to use (see figure 3.49).

(a)

(b)

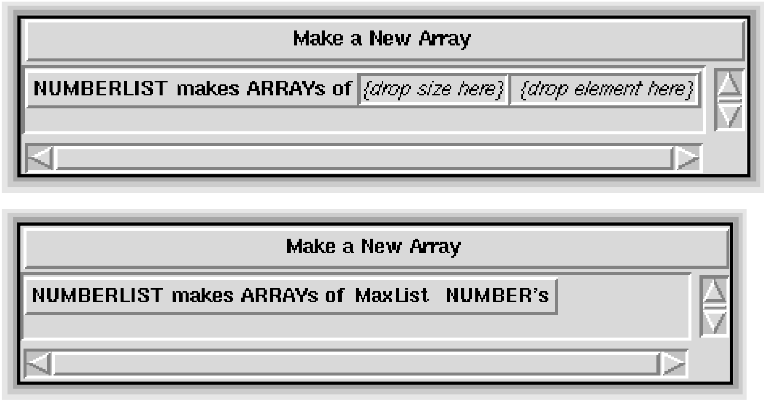


Figure 3.49: (a) The initial NUMBERLIST type. (b) The complete type.

3.8.2 Demonstrating a variable of a new type

The demonstrator demonstrates a variable of type NUMBERLIST by dragging from the NUMBERLIST type on the types palette to the local variables area of the main worksheet. The demonstrator names the place list (see figure 3.50). This method is exactly the same as the one used to declare variables of any of the built-in types (see examples 3.5 and 3.7).

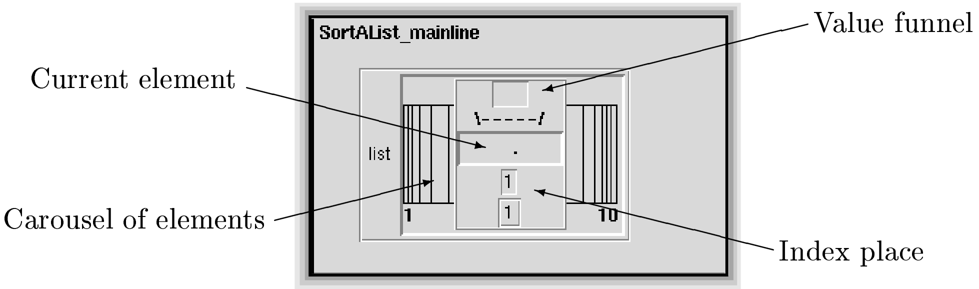


Figure 3.50: list — a variable of type NUMBERLIST.

An ARRAY place appears as a carousel, which the clerk turns to find a particular element. Note that this is not the analogy that the machine model uses (see section 2.3.3.2). The second prototype uses the stricter analogy that was presented as an alternative (see section 2.3.3.2). An array consists of the name of the variable, the current element place, a compressed view of the elements that aren't visible, an index place, and a funnel-like place (see figure 3.50).

The index place shows the index of the element that is displayed in the current element place. The demonstrator turns the carousel to show different elements by putting values into the index place. The first and last indices of the array are shown to either side of the index place. The funnel place is used to hold values to be put into an array element.

3.8.3 Assignment with `ARRAY` elements

Assignment to an array element is much more like carrying out a procedure than assigning a value to a simple variable. The value funnel and index place act like parameters that must be filled in before the assignment procedure can be called.

To assign the value `42` to the first element of `list` (line `68`), the demonstrator the enters "`42`" into the funnel of the `list` place (see figure 3.51). Then the demonstrator enters "`1`" into `list`'s index place. Immediately, `list` displays the element at index number `1` (see figure 3.51). At this point, the assignment button (trapdoor) under the funnel is highlighted. The demonstrator clicks on `list`'s assignment button. This action assigns the value `42` to the first element in `list` (see figure 3.51). The demonstrator initialises the remaining elements of `list` in this way.

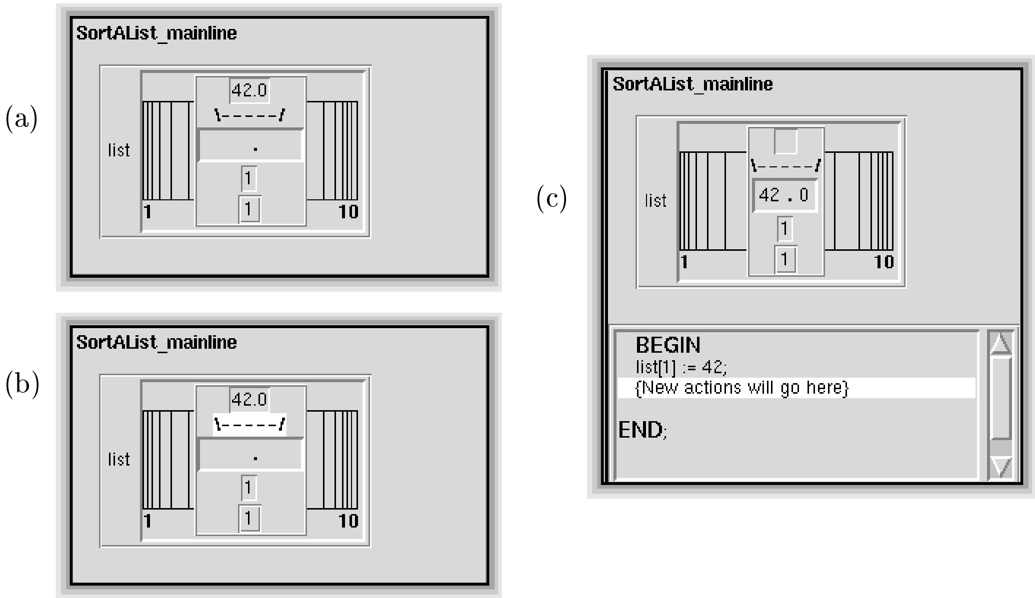


Figure 3.51: (a) A value is put into `list`'s value funnel. (b) A value is put into `list`'s index area. (c) After the assignment has been carried out.

3.8.4 Using a new type in subroutines

The demonstrator demonstrates the `WriteNUMBERLIST` procedure by asking for a new procedure and demonstrating it in the same way as a demonstrator-defined function (see section 3.7.2). The one exception is that a procedure does not return a value, so there is no need to demonstrate return types or values. A procedure finishes when the last `END` statement in that procedure is carried out. At that point, the procedure's worksheet folds up and disappears.

The `NUMBERLIST` type can be used like a simple type to define parameters for subroutines, and return types for functions. The demonstrator drags the `NUMBERLIST` type to the new parameters area of `WriteNUMBERLIST`'s worksheet to add a new parameter (see figure 3.52).

(a)

(b)

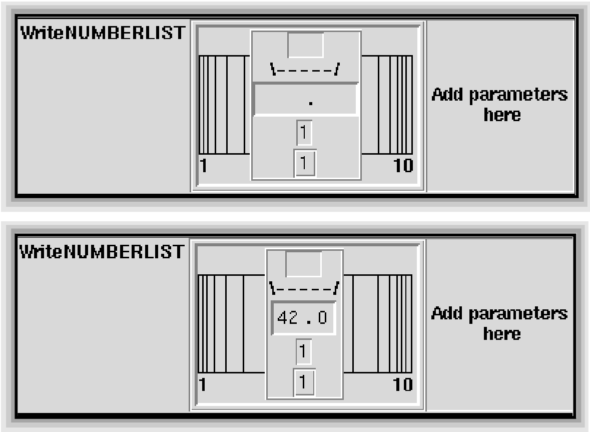


Figure 3.52: (a) `WriteNUMBERLIST`'s new `NUMBERLIST` parameter. (b) `list` is used to fill in the parameter.

The demonstrator drags `list` to the new parameter place to fill the parameter in (see figure 3.52). Each element in `list` is copied over to the corresponding element in the parameter place.

To sort the list of numbers, the demonstrator makes a new function, `SortNUMBERLIST`, which takes a list and returns a sorted copy of that list. The details of how the demonstrator demonstrates this function are not given here except for two actions: demonstrating the comparison of two `NUMBERLIST` elements, and assigning the value in one element of a `NUMBERLIST` place to another element of the same place. Examples of both actions can be found in the `SortNUMBERLIST` function (line `45` and line `55` respectively).

3.8.5 Comparing two elements of an `ARRAY` type

To demonstrate the comparison on line `45`, the demonstrator uses the `ARRAY` place's index to locate the first element to compare, and then drags that element to the comparator (see figure 3.53). Then the demonstrator alters the value in the `ARRAY` place's index, and drags the newly displayed element to the other place on the comparator (see figure 3.53).

3.8.6 Assigning one element to another element of the same array

To demonstrate the assignment on line `55`, the demonstrator finds the `min`-th element, and drags its value into `unsorted_list`'s funnel (see figure 3.54). Then the demonstrator alters `unsorted_list`'s index so that the `sortedto`-th element is displayed, and carries out the assignment by clicking on `unsorted_list`'s assignment button (trapdoor) (see figure 3.54).

(a) (b)

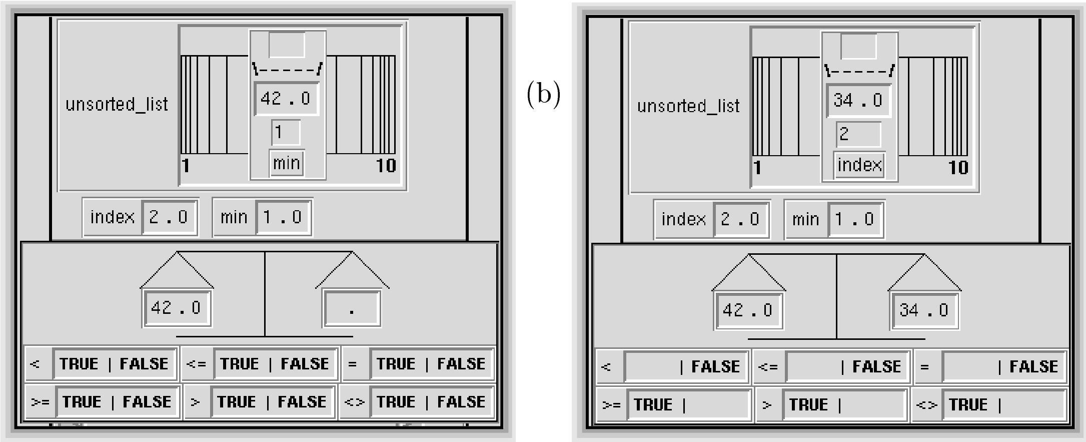
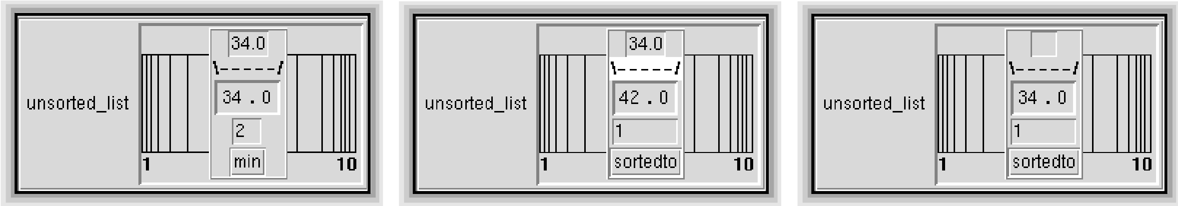


Figure 3.53: (a) One element is used in the comparison. (b) Another element is used in the comparison.



(a) (b) (c)

Figure 3.54: (a) The value of one element is put into the funnel. (b) The destination element is displayed by changing the index. (c) After the assignment.

3.8.7 Returning an array value from a function

To return a `NUMBERLIST` from the `SortNUMBERLIST` function, the demonstrator drags from the `NUMBERLIST` type on the types palette to the function's return place, and then drags `unsorted_list` to the return place. This is exactly the same method used to for a function that returns a `NUMBER` (see section 3.7.6).

When `SortNUMBERLIST`'s sheet folds up, the demonstrator drags the value returned to `list`. This action copies all the elements in the returned `NUMBERLIST` value to their counterparts in `list`.

3.9 Summary

The demonstration model extends the abstract machine model to allow programming by demonstration. The way of programming supported by the demonstration model is based on programming by demonstration, direct manipulation, and program visualisation. The result is a demonstrational machine that extends the abstract machine by adding the roles of the demonstrator and the recorder to support programming the machine.

The demonstrator (the human programmer) can take over the running of the computer described in the abstract machine model, and demonstrate what the program should do by carrying out the actions that will make up the program. The recorder watches the demonstrator and writes down the code that describes those actions. The recorder also records the previous states of the machine so that they can be restored, and all the demonstrator's actions so that they can be undone. In other programming by demonstration systems, the role of the recorder is not usually distinguished from that of the clerk who carries out the program. The distinction is made here so that the two roles that the computer plays are clear to the novice: the fundamentally robotic and naive clerk, which is the computer when running a program, and the smarter and more sensible recorder, which is the programming by demonstration application that works out what the demonstrator wants the program to do.

As well as running the machine, the demonstrator can get the clerk to carry out parts of the program. The clerk will stop and ask the demonstrator to take over when an incomplete section of the program is reached. The demonstrator can also ask the recorder to unwind the state of the machine to a previous state, or to completely undo the effects of one of the demonstrators actions.

Two prototypes of the demonstrational machine were implemented with the aim of exploring the abstract machine and demonstration models in the light of an environment for supporting novice programmers. The first prototype implemented a subset of the program constructs covered by the models. It had no unwind or undo capability, and did not support programmer-defined types or subroutines, which limited it to linear programs. The prototype was shown in action with an example of building a program that draws a button, asks the user to click the mouse in the button, waits for the user to click the mouse button on the screen, and tells the user whether or not their click was inside the button.

The second prototype extended the first to include programmer-defined types and subroutines, and added undo and unwind capabilities. The second prototype was based on the modified models that resulted from our exploration with the first prototype. The second prototype used different techniques from the first prototype for demonstrating expressions and subroutines. Two examples of the second prototype in action were given: building a program to recursively compute the factorial of a number, and building parts of a program to sort a list of numbers.

The demonstration model, the way of programming with the demonstrational machine, and the issues that arose from exploration with the prototypes are described in more detail in the next few chapters.

Chapter 4

The Demonstration Model

Theory often differs from reality. There are a number of discrepancies between the theoretical demonstrational machine and the prototypes, between the model and the two exploratory environments. This chapter presents a detailed description of the demonstration model and issues that were uncovered during our exploration of the prototypes.

Section 4.1 describes issues to do with creating programs by demonstration and the consequences for the demonstrational machine. The two sections that follow (sections 4.2 and 4.3) describe how individual pieces of code are created using the way of programming supported by the demonstration model. Section 4.2 describes how declarations are programmed, and section 4.3 describes how actions (instructions) are programmed. Each section describes creating code from two perspectives: how a demonstrator interacts with the demonstrational machine in the model, and how a demonstrator interacts with the prototype demonstrational machines. Issues relating to demonstrating particular pieces of code statements are discussed after each pair of code creation perspectives. Section 4.4 presents expressions in the model and examines the ways of programming them that were explored in the prototypes. Section 4.5 provides a summary of the chapter.

4.1 Programming issues

4.1.1 Compiling, interpreting, and demonstrating

The demonstration model's way of programming is quite different to the traditional way of programming. Traditionally, the programmer types in program code, compiles it, and the computer carries it out. If something unexpected happens in the course of the program, the programmer debugs the program, alters the code, and begins the cycle again.

The way of programming supported in the demonstration model is different because the program is running all the time. As it runs, the program is coded incrementally by demonstrating what it should do using the state of the demonstrational machine. The demonstrational machine is usually paused, waiting for the demonstrator to demonstrate (or clerk to carry out) the next instruction. When an action is carried out by the demonstrator, the state changes to reflect the effects of the action and then the code (an instruction) is recorded.

This is quite different to taking a compiled program, running it, pausing it, and then stepping through it. It is more like an interpreter in that it allows new code to be carried out as the program is running. However, it is also different from an interpreter, because the demonstrator does not specify the code and get the machine to execute it; the demonstrator executes the code and the machine records it. In terms of the traditional method, the process is more like carrying out the program, and then typing it in.

It is possible that this so different from the traditional method that novices who learn to program this way may find it difficult to transfer to more traditional programming methods. This possibility will be discussed further in chapter 5.

4.1.2 Program construction, structure editing, and demonstrating

The way of programming supported by the demonstration model necessitates two things: it constrains the programmer (demonstrator) to produce code (demonstrate it) in meaningful chunks, and it requires the demonstrational machine to be able to cope with partially complete instructions and programs.

A "meaningful" chunk of code is a piece of code that is not less than an atomic action on the demonstrational machine; a chunk quite often covers more than one atomic action. An atomic action on the demonstrational machine is one that the clerk either carries out completely or does not do at all, the clerk cannot do half the action. A meaningful chunk of code is (usually) smaller than a whole instruction; carrying out a whole instruction (usually) involves several steps, each of which corresponds to a meaningful chunk. The demonstrator is constrained to demonstrate meaningful chunks for two reasons: the demonstrator can only use what is available in the current state, and only the actions that have an effect on the state of the machine (including the templates) will produce code.

The recorder will only record code for actions that have an effect on the state of the machine. The demonstrator is constrained to actions that use what is available in the current state because the demonstrator cannot use what isn't there. For example, the demonstrator cannot demonstrate an assignment to a variable that doesn't exist. The demonstrator would have to create the variable and then assign to it. This is similar to an interpreter and more constrained than traditional programming by text or structure editing.

The demonstrational machine has to be able to cope with partially complete instructions and programs because, at any point in the program's execution, the clerk could run across a part of the program that the demonstrator has not yet demonstrated. The fact that the demonstrator is constrained to producing meaningful chunks of code, helps the demonstrational machine deal with partial instructions.

Each chunk of code that is produced, partial or not, is an executable unit; each is a well formed part of a program. Programs built with the demonstrational machine are going to be well formed at all times. This is like a structure editor, but the program structure is stronger because each chunk of code is more fully specified when it is created. There are less pieces of fill in later. For example, a structure editor may do assignment as <DEST>:=<VALUE>, but in the demonstrational machine the parts of the assignment must be specified in order to created the code, such as '$num := 42$'. As a result, there are fewer places that partial instructions can occur. In fact, there are four places, and each place gives rise to a particular kind of partial instruction.

4.1.3 Partial instructions

The recorder indicates the unspecified components of a partially specified instruction or program with special "Not Specified" instructions. There are four different kinds of "Not Specified" instructions for different situations that involve partial code: "Not Specified Expressions" for empty arguments, "Not Specified Connections" for expressions with unspecified destinations for their results, "Not Specified Statements" in empty control structure and subroutine bodies, and "Not Specified Returns" in incomplete functions. The first two "Not Specified" instructions occur in partial instructions, and the last two in partial programs.

"Not Specified" instructions are meta-instructions that tell the clerk to stop and ask the demonstrator to take over. They are not the same as "there is no code here", but rather "I don't know if there is any code here yet". The recorder erases each "Not Specified" instruction as and when the demonstrator demonstrates a replacement for it.

"Not Specified" instructions could be displayed for the demonstrator in a variety of ways, or they could be kept hidden. In the second prototype, "Not Specified Expressions" appeared as ' ?? ', and "Not Specified Connections" as '. . . '. The other two "Not specified" instructions were displayed as empty space. Each kind of "Not Specified" instruction is described in one of the following sections.

4.1.3.1 "Not Specified Expressions" for empty arguments

Every time any subroutine is invoked, the recorder places a "Not Specified Expression" in each of the invocation's parameters. In the first prototype, "Not Specified Expressions" are implemented by not allowing a subroutine to be called, or an expression to be used, until all the arguments are specified. In the second prototype, "Not Specified Expressions" are implemented by not allowing the program to be carried out past an unspecified argument. The difference between them is that the second prototype allowed partial subroutine calls in the code, *e.g.* '*Factorial*(??)' (see section 3.7.3), whereas the first prototype did not allow this — a subroutine call was either complete or it was not recorded.

4.1.3.2 "Not Specified Connections" for return values with unspecified destinations

Until the demonstrator uses the value returned from a function, the function's return destination is unknown. The recorder denotes this with a "Not Specified Connection". A "Not Specified Connection" prevents the clerk from carrying out the next instruction until the function's return value is taken to a destination. "Not Specified Connections" only appeared in the second prototype, where they were displayed as '. . . ' (see section 3.7.2). The first prototype didn't need "Not Specified Connections" because of its way of constructing expressions (which will be described in section 4.4).

4.1.3.3 "Not Specified Statements" in empty control structure and subroutine bodies

The recorder writes a "Not Specified Statement" in the body of each new control structure, `IF` (which has two bodies) and `LOOP`, and in the body of every new subroutine that the demonstrator decides to define. "Not Specified Statements" represent places in the program that have not been carried out before by either the clerk or the demonstrator. "Not Specified Statements" are most useful for drawing the demonstrator's attention to a part of the program that was sectioned off sometime ago but not demonstrated in detail. Neither prototype implemented "Not Specified Statements". Instead, they relied on the demonstrator to remember what hadn't been demonstrated in detail, or to pick up on missing pieces as the code was carried out.

4.1.3.4 "Not Specified Returns" in incomplete functions

The recorder writes a "Not Specified Return" at the end of each new demonstrator-defined function to prevent the clerk from folding up the function's worksheet until a return value is specified. "Not Specified Returns" aren't needed for procedures because procedures do not return any usable values. The first prototype did not allow new functions to be defined, and so did not implement "Not Specified Returns". The second prototype implemented them to ensure that the demonstrator would demonstrate a `RETURN` instruction in every new function.

4.1.4 Model and implementation trade-offs

The demonstration model describes a way of programming in a special environment, the demonstrational machine. Programming in an implementation of the environment that was completely faithful to the model would be painful, *e.g.* lining a worksheet up to feed it into a slot. There are three general ways that the prototype demonstrational machines differ from the model's demonstrational machine:

- The prototypes have "short-cuts" that do not use the same way of manipulating objects that the model describes, *e.g.* outputting values.

- Some of the manipulations in the graphical user interfaces do not support the metaphor completely, *e.g.* worksheets are not unfolded, positioned, or slotted by the demonstrator.
- Some of the demonstration methods the prototypes use are less demonstrational than the model describes *e.g.* demonstrating control structures.

Behind all these differences is the trade-off between ease of use and keeping faithful to the model. This is equivalent to a trade-off between experts and novices, or one between ease of use and ease of understanding. The differences between the prototypes and the demonstrational machine will described in more detail in the sections on programming in the model, sections 4.2, 4.3 and 4.4.

4.2 Programming declarations

Declarations are one part of a program (instructions are the other). Declarations determine part of what is printed on the worksheets of the abstract machine. There are three kinds of declaration: places, types, and subroutines. Declarations cannot be demonstrated by mimicking the clerk because the clerk does not create any declarations during the course of a program. This section describes the way declaration code is demonstrated in the demonstration model. Each kind of declaration is described in this section using the same general format:

- A description, in terms of the demonstrational machine, of how the demonstrator demonstrates (programs) the declaration.
- A description of the way that each of the two prototypes implemented demonstrating the declaration.
- A discussion of the issues involved with demonstrating the declaration.

The last part of this section discusses particular issues that apply to arrays.

4.2.1 Place declarations

In the model, the demonstrator declares a new place by drawing a place rectangle of the appropriate type on the appropriate part of a worksheet. The recorder asks the demonstrator what the place's name is, and records a new declaration on the worksheet's template. For a constant, the recorder also records the initial value that the demonstrator gives the place.

There are four kinds of places: variables, parameters, return places, and constants. Types are used like stencils or stamps to draw the places. In both prototypes, the declaration of a new place is demonstrated by dragging the type of the place from the types palette to a certain part of a worksheet. The first prototype only allows constants and variables; the second prototype allows parameters and return places as well as constants and variables.

- Variables are declared by dragging a type to the parameters and places area (see figure 2.1) of an unfolded worksheet. This area is only visible during a subroutine's execution. Variables cannot be declared at any other time. The recorder ensures that when a variable is declared, the variable place is created on all copies of the subroutine's worksheet.
- Parameters are declared by dragging a type to the "new parameters" area on the folded worksheet of a new subroutine. This area is only present the first time a new subroutine is invoked. New subroutines will be discussed in section 4.2.3.
- Return places are declared by dragging a type to the return value part of the worksheet. This area is only visible on function's unfolded worksheet (see figure 2.1). In the second prototype's implementation, the function's worksheet had to be unfolded. This made the stages in a subroutine's execution clear — invocation, execution, returning from. It would be possible to make the return value part visible on the folded worksheet when the function is first invoked (*e.g.* hanging beneath the parameters). However, having the return place visible when a function is first invoked seems to put the return stage out of order.
- Constants are declared by dragging a type to the "constants area" on the special "main" worksheet. Constants can be declared at any time because the "constants area" is alway visible (the "main" worksheet is slightly larger than the other sheets see section 2.2.2.8).

Places can be renamed by clicking on their names, and entering a new name. Renaming a place updates all references to that place. The values in places can be changed in a similar manner. The effect of changing a place's value depends on the kind of place (assuming that the recorder is recording):

- Changing a variable's value creates an assignment statement.
- On a folded subroutine's worksheet, changing a parameter's value changes the corresponding argument in the subroutine call. If the subroutine's worksheet is unfolded then parameters are identical to variables, and thus an assignment statement is added to the subroutine's body.
- Changing a return place's value creates a `RETURN` statement.
- Changing a constant's value causes the constant's definition to be altered, it does not create an assignment.

Assignments and `RETURN` statements will be discussed in section 4.3.3. Arguments to subroutine calls will be discussed in section 4.3.2.

4.2.1.1 Issues for place declarations

There are two issues related to place declarations: initial values, and type inference.

It is not clear whether place should initially have no value, a nil value, or a random garbage value (see section 2.7.1) Neither prototype handled this issue satisfactorily. The first prototype gave places no values at all initially. The second prototype was not consistent because it used two initialisation methods: it gave `NUMBER` places an initial value of random garbage, whereas `BOOLEAN` and `STRING` places were initialised to nil values (`FALSE` and `''` respectively).

Each initialisation method has its pros and cons. An argument for no values is that the places are supposed to be on newly printed worksheets so they shouldn't have any values in them. An argument for random garbage values is that new declared places in computer programs always have some data in them. The data is usually nothing to do with what the place will be used for, but the programmer often has to account for it; thus, the novice should be reminded of it. An argument for nil, or zeroed out, values is that places always have data in them but the language and computer should zero out places for the programmer's benefit. I do not think it matters which method is used provided it is used consistently.

There are other ways a place could be declared. The demonstrator could just mark off a place on a worksheet, and then use it. The recorder would then infer what type of place the demonstrator has drawn from the kind of values the demonstrator puts in the place. The difference between this and the prototype's method (apart from the inferencing) is the order in which the place is declared and used. The prototype's method requires the place to be declared first, and then used. The type inference method requires the place to be used in order for it to be fully declared. The prototype's method is the less demonstrational method; however, it is not clear which method is more appropriate for novices or experts (understanding or programming).

4.2.2 Type declarations

In the model, the demonstrator declares a new type by drawing the type on a blank type template with a type constructor; the demonstrator specifies the type's details using the other types that exist. The type's details are the component fields and elements that make up the new type. The recorder asks the demonstrator to name the new type and adds it to the other types in the worksheet machine.

The first prototype did not allow new types to be constructed. The second prototype only implemented one of the two type constructors: the `ARRAY`. In the second prototype, the demonstrator declares a new type by clicking on the appropriate "Make a new type" button to create a blank type (see section 3.8.1). The demonstrator provides a name for the new type, and fills in the type's details. The details of a new type do not have to be filled in immediately, but the type cannot be used to declare any places until it is complete.

For arrays, the demonstrator has to specify the size of the array with a literal or constant value, and the type of the array's elements by dragging another type to the array. Arrays will be discussed in more detail in section 4.2.4 because there are several issues specific to them alone. For records, had they been implemented, the demonstrator would have to drag another type to specify each field of the record, and name each field.

4.2.2.1 Issues for type declarations

There are two issues for type declarations. The first is whether the type is specified in the general case, or specified in terms of a particular value or place. The second issue is when the details of a type must be specified: when the type is first declared, before it is used to create a place, after a place is created but before the place can be assigned to, or later. The first issue provides the two main approaches to declaring new types: defining the type as a type template, and demonstrating the type with a particular instance. The second issue provides a spectrum of variants within each of those approaches.

Defining the type as a template requires the demonstrator to specify a general case of the type, and then use the type to create places. There are three main variants of this approach:

- **Template method 1**

  The details of the type must be completely defined when the type is *first* declared, before the type is used to create a place. The components of the type must be specified using other types (and, for an array's size, constant or literal values), for example, by dragging the `NUMBER` type to one of the fields.

- **Template method 2**

  The details of the type have to be completely defined *before it can be used* to create a place, but the details do *not* have to be defined when the type is first declared — they could be added at a later point. This is the method used by the model and prototypes. Like the previous method, it requires using other types (and constants or literal values) to specify the new type's details.

  This method and the previous one distinguish clearly between declaring a type and using a type to declare a place, but they are not particularly demonstrational. A minor variation of this method is to allow the details of the type to be specified using example values from which the types of the components are inferred. This variation differs only in that it uses a small amount of inference. However, the variation is quite strange and may cause confusion among novices because it allows values to be put into a type declaration, which is quite different to putting values into a place. An issue related to this will be discussed in section 4.3.1.

- **Template approach 3**

  The details of the type do *not* have to be completely defined *before it is used* to create a place, but all the type's details are still specified on the type itself. This method differs from the previous ones in that it allows partially specified places to be created. This means that when the type's details are further specified or extended, all the places of the type must be altered to match the new type; this includes places in previous states of the machine.

Demonstrating the type from an example place requires the demonstrator to create a particular place, and the specifications of a new type are inferred from that place. There are four main variants of this approach:

- **Demonstration method 1**

  To define a new type, the demonstrator would draw a blank place on a worksheet using a type constructor (`ARRAY` or `RECORD`). The demonstrator would specify the number of components in the place, and the types of those components *before* putting values into the place. The recorder would ask the demonstrator to name the new type of which this place is the first example, and would record a new type template for the worksheet machine based on the place. Once all the type template's details were recorded, the demonstrator could assign values to the place.

  This method is the demonstration equivalent of template method 2. It uses the least inference of all the demonstration methods.

- **Demonstration method 2**

  The demonstrator would draw a blank place on a worksheet using a type constructor (`ARRAY` or `RECORD`). The demonstrator would specify the number of components in the place, but *not the types* of those components. The recorder would ask the demonstrator to name the new type, and would record a new type template based on the place for the worksheet machine. Then, as the demonstrator assigns values to the place's components, the recorder would fill in the type template's details based on the types of those values.[1]

  This method is more demonstrational than the previous methods because the recorder infers the details of the new type from an example (see section 4.2.1.1). However, it blurs the distinction between the declaration of a new type, the declaration of a new place with that type, and the use of that place in demonstrating instructions.

- **Demonstration method 3**

  The demonstrator would draw a blank place using a type constructor, but would *not specify the number of components* that the place has. The demonstrator draws in each component of the place as it is needed, and puts a value in it. The details of each component are inferred based on the types of those values. The demonstrator can keep on adding extra area to the place's rectangle whenever they need to. This

[1] This type inference is quite simple in our language because it has no type hierarchy. However, in a language with a type hierarchy the inference would be far more complicated.

method offers more flexibility than the previous method, but suffers from the same problem of blurring distinctions. In an implementation of the demonstrational machine, adding a new component to a place could be done in the same way as parameters are added to a subroutine in the second prototype (see section 3.7.3). This will be discussed further in section 4.2.3.

- **Demonstration method 4**

  The demonstrator would draw a blank place, but would *not even specify the general structure* of the place. The general structure and details of the type would be inferred by the recorder based on how the demonstrator sections off and uses the place.

  This method takes type inference to an extreme (see section 4.2.1.1), and would be highly problematic to implement without losing the distinction between arrays and records (and simple types). The recorder will never really know whether a place with several components of the same type is an `ARRAY` or a `RECORD`. For example, if the demonstrator sections a place into two components and puts a number in each of them, the recorder cannot tell if the place is an `ARRAY` of size 2, or a `RECORD` with two fields.

  There is also the question of the difference between a `NUMBER` and a new kind of place that only contains a `NUMBER` component.

  It would also be problematic if there were two places of different types, and the whole value of one of them was dragged to the other. It is not clear how the recorder should combine the fields and values. Possibilities include unifying the two types into a single type, changing the type of one of the places to be the type of other place, or extending the type of the destination place by adding fields of the other place's type.

  One solution to these problems is to have the recorder ask the demonstrator what do to when these problems arise. However, this turns the method into one that uses less inference than the one described here.

It is not clear which of these methods is the most appropriate for learning to program, or which is the most appropriate for programming. It is likely that the method that is the best suited for one, may not be suitable for the other. However, some of the methods are unsuitable for novices. Template methods 1 and 2 require the demonstrator to commit to the details of the type before the demonstrator has seen and used a place that the type would create. Demonstration method 4 is so unconstrained and blurs the distinctions between types, places, and values so much that it would be too confusing for novices.

4.2.3 Subroutine declarations

In the model, the demonstrator demonstrates a new subroutine declaration by getting a *blank* folded worksheet from the worksheet machine and using it to carry out a new subroutine call. The recorder knows that the demonstrator is demonstrating a new subroutine because the worksheet is blank. The recorder asks the demonstrator for the new subroutine's name, writes out a new template for the subroutine, and records a call to the subroutine in the current scope. The demonstrator specifies the subroutine's parameter places, and fills them in. As the demonstrator specifies each parameter, the recorder asks the demonstrator for the parameter's name, which is recorded along with the type of parameter on the subroutine's template. The recorder records the arguments as part of the call instruction. When all the parameters are filled in with arguments, the demonstrator unfolds the worksheet (calls the subroutine).

For functions, the demonstrator demonstrates the return type for the function, which the recorder records, once the worksheet is unfolded. The instructions that make up the new subroutine can be demonstrated during any call to the subroutine. The way that instructions are demonstrated will be discussed in section 4.3.

The first prototype does not support programmer-defined subroutines, but the second prototype does. To simulate the demonstrational machine, the second prototype had to provide a way for parameters and return types to be demonstrated for a new subroutine. It also had to ensure that a subroutine could not be called until all its parameters were filled in. An example of the declaration of a new function using the second prototype was covered in the example `RecursiveFactorial` program (see section 3.7).

New parameters are added to a new subroutine by dragging types to the "new parameters area" on a folded worksheet (see section 4.2.1). The "new parameters area" only appears on the folded worksheet for the first invocation of a demonstrator-defined subroutine because that invocation is the one that defines the subroutine's interface by demonstration. The action of dragging a type to the "new parameter's area" to declare a place is the prototype's equivalent of the demonstrator drawing a new parameter on the new worksheet in the demonstrational machine. Possibly a better interface would be to allow values to be dragged to the "new parameter's area" to declare a new parameter.[2] The new parameter would be of the same type as the value, and the value would be used as the argument to the new parameter for this call.

A subroutine (new or built-in) cannot be called until all its parameters are filled in. On a new subroutine's first call, it is perfectly valid for the demonstrator to declare a parameter, fill it in, and then declare another parameter before calling the subroutine. The prototype unhighlights, highlights, and unhighlights the subroutine's name on its worksheet to indicate that the subroutine cannot or can be called (see section 3.7.3). More details on demonstrating subroutine calls can be found in section 4.3.2.

[2]This involves type inference.

4.2.3.1 The wishful programming check

Wishful programming is a programming technique that is often employed when a programmer needs to carry out a task and there isn't a subroutine that will do the task. To solve this problem, the programmer uses wishful programming and makes up a subroutine to do the task. The programmer works out what parameters the subroutine requires, and what task the subroutine should perform, but does not bother to define the subroutine fully. The programmer leaves the subroutine as a stub, with plans to define it later.

The recorder gives the demonstrator the option to use "wishful" programming. When the recorder comes across a call to a demonstrator-defined subroutine, the recorder asks the demonstrator to choose whether they wish to demonstrate the subroutine now, or pretend to so they can come back to it later. The demonstrator also has the option of telling the recorder not to ask about this particular call again. If the demonstrator chooses to pretend, the subroutine is not called at all and the program is jumped forward to the next instruction (see section 3.7.13).

The recorder inserts a wishful programming check at each call to a demonstrator-defined subroutine, after the parameters are filled in but before the subroutine is called. A wishful programming check is a meta-instruction that is only visible to the recorder. The check is the recorder's way of reminding itself to ask the demonstrator before the subroutine is called. Wishful programming checks were only implemented in the second prototype (see section 3.7.5 for an example).

Wishful programming checks on functions are more complicated, because even functions that weren't really carried out still have to return a value that can be used in the calling scope. Thus, the recorder asks the demonstrator to enter a return value for a pretend function call. The value provided may well be an incorrect result for the function, but this doesn't matter provided the demonstrator is aware that it is incorrect (see section 3.7.13 for an example).

4.2.3.2 Issues for subroutine declarations

There is one issue for subroutine declarations: with respect to carrying out the subroutine, how much flexibility should the demonstrator have over *when* the parts of a subroutine can be demonstrated? A subroutine declaration consists of two parts: the parameters, and the body of the subroutine. The amount of flexibility over how much of a subroutine has to be demonstrated *before* it is used (called), and how much has to be demonstrated *as* the subroutine is used, provide a spectrum of various methods. There are less flexible methods and more flexible methods. Four possible methods are described below.

- **Subroutine method 1**

  The subroutine's parameters and body must be demonstrated completely *before* the first call. Here 'call' refers to invoking, filling in the parameters, carrying out, and returning from the subroutine. The subroutine has to be defined as a template before the demonstrator can call it. This method is the least flexible because the subroutine must be completely defined before it is used. It is also less demonstrational than the prototype's method. However, it does distinguish more clearly between the declaring the subroutine and calling the subroutine — this will be discussed further in section 4.2.3.3.

- **Subroutine method 2**

  The subroutine's parameters and body must be demonstrated completely *during* the first call; the subroutine's definition cannot be altered after that. This method has a stronger demonstrational flavour because it allows the demonstrator define the subroutine by calling. By restricting the demonstration to the first call, it is still less flexible than the prototype's method. However, by preventing modifications to the body during later calls, this method avoids some of the issues to do with copies of instructions on worksheets. These issues will be discussed further in section 5.3.

- **Subroutine method 3**

  The subroutine's parameters must be demonstrated during the first call, but the body of the subroutine can be demonstrated during any call. This is the method that the second prototype uses. Parameter declarations were restricted to the first call because of the complexities involved in allowing parameters to be declared during any call (these are discussed in the next method). This method is flexible because it allows the demonstrator to alter the instructions in a subroutine during any call to handle unforeseen situations.

- **Subroutine method 4**

  The subroutine's parameters and body can be demonstrated during any call. This is the most flexible method presented here, and also the most problematic to implement and use. If new parameters can be added any time a subroutine is called, then the demonstrational machine has to have some way to update all the previously demonstrated calls to the subroutine that occur in the program.

  One way is to have the recorder update every other call to the subroutine on all of the templates and worksheets to include the new parameter. In the other calls, the new parameter would be filled in by a "Not specified expression". However, this means that the program would be incomplete until the demonstrator had gone back over all the calls to the subroutine and filled in the extra parameter. It also puts the demonstrational machine in an inconsistent state because any calls earlier than the one that was changed should not have been invoked the way they were. The demonstrator must take the machine back to a state prior to the first call to properly demonstrate arguments to the new parameters. Declaring new parameters like this this makes the demonstrational machine more complicated and difficult to use.

An alternative is to distinguish subroutines by their parameters as well as their names, and allow multiple subroutines with the same name and different numbers of parameters. New parameters declared during later calls to a subroutine would cause a new subroutine to be created. However, this isn't really a solution — it is just another version of the prototype's "parameters on the first subroutine call" declaration method (subroutine method 3), with a more flexible naming scheme.

As with type declarations, it is not clear which of these methods is best suited for programming and understanding programming. One thing that is clear is that there is some common ground between methods for declaring new types and those for declaring new subroutines.

4.2.3.3 Approaches to demonstrating declarations

The methods for producing type declarations fall into two categories: those that define the type directly as a template, and those that define the type from a particular example place (see section 4.2.2). The difference between these two categories is whether the declaration is created as a general case from which instances can be made (the specification-based approach), or whether the declaration is created by going from an instance to the general case (the example-based approach). This difference can also be seen in the methods for declaring subroutines, but it is less apparent because the subroutine methods tend to use a mixture of the two approaches.

To better explain what I mean by the example-based approach and the specification-based approach, Table 4.1 contrasts the two approaches.

Example-based Approach	Specification-based Approach
Going from the particular instance out to the general	Going from the general in to the particular instance
Demonstrator works from inside the program	Demonstrator has to step outside the program
From inside the calling scope	From outside the calling scope
On a worksheet	In the worksheet machine
As the clerk	As the recorder
As, or after, it is used	Before it is used
With an instance	With a general case

Table 4.1: Contrasts between the two approaches to declarations.

Subroutine method 1 uses the specification-based approach completely because the subroutine's parameters and instructions have to be defined before the subroutine can be called. The remaining subroutine methods either use the example-based approach completely or use a combination of the two approaches, depending on which approach is used to declare the parameters. Both the subroutine's initial declaration and body are declared using the example-based approach: the subroutine is initially declared by calling it, and its instructions are defined by carrying them out during the call. However, the parameters can be declared with either approach. If the parameters are declared by using a type to define the parameter (as in the prototype's implementation), then the method uses the specification-based approach — the type is the general case that is used to set the parameter place up for specific kinds of values. If the parameters are declared by using specific values, from which the type of the parameters can be generalised[3], then the subroutine declaration method uses the example-based approach completely — the value is a particular instance of the type of value that the parameter should hold.

It is interesting that the prototype used template method 2 for declaring new types (specification-based) and subroutine method 3 for declaring new subroutines (mostly example-based). The inconsistency was not noticed at the time because we did not understand the spectrum of alternatives well enough. A future implementation of the demonstrational machine should use a more consistent approach for demonstrating new types and new subroutines because they are both declarations.

4.2.4 Issues with `ARRAY` types

One of the common novice confusions, which the abstract machine model was trying to solve, was the idea that the computer can only consider a few values at a time (see section 2.1.2). For arrays, this means that the clerk only focuses on one of the elements at a time. The abstract machine model did not solve this problem (see section 2.7.4); the demonstration model does not even address this problem. However, the second prototype's implementation of arrays *does* address the problem. It gets across the right idea of the computer's behaviour by forcing the demonstrator to deal with an array as the clerk does — one element at a time — because only one element of the array is visible for use at at time (see section 3.8.2). The demonstrator and clerk select the particular element to display by altering the array's index (see section 3.8.3). Note that the demonstrator can alter an array's index to view different elements before using one to demonstrate an instruction, but the clerk will only alter an index in order to use the corresponding element.

To assign to an array element, the second prototype treats the assignment like a procedure, but compresses the invocation and filling in of the first parameter into one step by making the invocation implicit (see section 3.8.3). The value that the demonstrator wants to assign to an element is dropped into the array's funnel place. This signals the beginning of the assignment procedure (the invocation), and thus the array's current index is discarded. The demonstrator then specifies the index of the element to assign to, and calls the assignment procedure by opening the "trapdoor" beneath the funnel, which carries out the assignment (see section 3.8.3).

Implementing arrays in the second prototype raised four issues specific to `ARRAY` types

[3]This is mentioned near the beginning of section 4.2.3 as a better interface to the "new parameter's area", but it was not implemented.

and places: the degree to which this implementation of `ARRAY` places enforce the right model of the computer's behaviour on the demonstrator, the type of an `ARRAY`'s index place, the problems with `ARRAY` indices and stale values, and the amount of screen space an `ARRAY` place takes up. The rest of this section discuss these issues.

4.2.4.1 The degree to which this implementation of `ARRAY` places enforces the right model

The second prototype's implementation of arrays prevents the demonstrator from looking at the whole array, and instead forces the demonstrator to view the elements one by one. In this respect, the prototype does a good job of enforcing the right model; a better job than either the abstract machine model, or the demonstration model.

However, in order to enforce the right behaviour the prototype depicts arrays differently from the way the abstract machine model does (see section 2.3.3.2). The prototype uses the carousel or rola-deck analogy that was not used in the machine model because it requires magic to integrate it with the worksheet analogy (see sections 2.3.3.2 and 2.3.2). The prototype gets away with using it because the prototype is a computer simulation of the machine and can perform the necessary magic. Even so, the fact that the prototype differs from the models in its exposition of arrays may be problematic for novices. It is not clear which, if either, should be brought in line with the other.

4.2.4.2 `ARRAY` index and types

The language has no integer type, only real numbers (`NUMBER`). This lack of distinction causes a problem because `ARRAY` indices can only be positive integers. The `INTEGER` type was removed from the language initially because there seemed to be no need for it. After all, reals and integers are both just numbers; combining them removed the need for converting between them and explaining them to novices, or so we thought.

The second prototype worked around the problem by allowing `NUMBERS` to be used as indices and secretly truncating their values. The hope was that the discrete nature of the elements in an array would be clear enough. However, there are at least three better ways to fix this problem: add `INTEGER` to the types in the language, provide devices for explicitly converting `NUMBER` values to integers so that they can be used in indices, or make the automatic conversion of `NUMBER` values to integers explicit by providing a brief animation of the process when they are put in an array's index.

It is not clear which approach would provide the best solution. Adding the `INTEGER` type solves the index problem, but introduces new issues, such as "Why are there two sorts of numbers?". Conversion devices would be quite annoying, if they were only needed for array indices and every number had to be converted before being put into an index. Providing an animation of the conversion process makes the truncation work around more obvious, but it may not be the best way to solve the problem.

4.2.4.3 The `ARRAY` index and stale values

A stale value is a value in a place that would not appear if the program were to be re-executed. Stale values are caused by side effects of instructions and will be described in more detail in section 4.4.1. In general, the second prototype prevented stale expression values from occurring by restricting how expressions could be constructed and used. This will be discussed in section 4.4.3. However, stale values were not prevented in array indices and caused problems from there.

Stale values were not prevented in array index places because the index places are used to perform two different tasks: as a way for the demonstrator to view elements in the array, and as a way to determine which element is involved in demonstrating an instruction. The demonstrator could specify an index, *but* then do nothing immediate with the element. Instead, the demonstrator could demonstrate several unrelated instructions *before* using use the element. These unrelated instructions may make the array index stale. A stale index will mean that the element used in the demonstration may not be the element used when the program is re-executed.

Array places display only one element at a time to constrain the demonstrator to use arrays in the way that the clerk uses them. However, this is a problem when it comes to finding the correct element to use in a demonstration. The demonstrator has to be able to view elements in the array without using them in instructions because the demonstrator has to be able to see example values before using them.

There are various ways of handling the problem of stale values and array indices:

- **Do nothing**

  Let the expressions go stale and let the demonstrator beware. This is *not* an optimal solution. It is not *even* a solution.

- **Discard the index**

  Discard the index value at the end of each instruction so that an array doesn't display any specific element. This means that a new index value has to be entered if the demonstrator wants to use an array element. It also means that once an index is entered, the array element must be used immediately if it is to be used at all in this instruction.

- **Re-specify the index**

  Require the demonstrator to re-specify the index before an element is used. This ensures that the index value is current. The demonstrator could be provided with the option to re-calculate the index using the previous expression, for example, in the instruction '$A[2*i] := A[2*i] + 1$'. The old index value and expression would only be discarded if a new index expression was specified.

  This method makes assignment to an array element act like a procedure that takes two parameters: the value to assign, and the index of the element to assign to. The prototype used this method for coping with stale values when assigning to an array element. Array element assignment as a procedure raises another issue that will be discussed in section 4.3.1.1.

- **Track expressions**

  Track expressions that might be affected by side-effects and re-evaluate them if they have gone stale. Expressions that contained variables and/or demonstrator-defined functions would have to be tracked. This method ensures that the state of the machine is kept correct. A potential problem with this method is that if the re-evaluation is done implicitly, then the demonstrator may be unsure of what is happening and get confused. Animations could be used to alleviate this problem.

- **Separate controls**

  Provide a separate set of controls for viewing elements in the array. Restrict the array index places to specifying which element to use in an instruction. The viewing controls should allow the demonstrator to see the elements before using them, but would not let the demonstrator access any of the elements in the array. The demonstrator has to use the index place to access a particular element.

The second prototype did not provide a complete solution to this problem. It coped with assignment to an array element, but did nothing about the indices of array elements that were used in other places, *e.g.* in expressions.

4.2.4.4 The amount of screen space an `ARRAY` place takes up

An array with many elements will be too large to display completely, because there is not enough screen space to display the whole array. This is the same problem that was first brought up in the abstract machine model with compound types and the size of worksheets (see section 2.3.3.2). There are various solutions to this problem: limiting the size of arrays, displaying only a few elements, or using scrolling windows to display the array etc.. The second prototype took the approach of displaying only one element of the array at a time.

However, even if only one element of an array is displayed at a time, an array place with complex elements could still be too large for the screen space. To alleviate this problem, we intended to display a reduced encapsulated representation of the element whenever the array had elements of a compound type. Elements of simple types — `NUMBER`, `BOOLEAN`, and `STRING` — would still be displayed in full, and their values could be used normally. Compound elements wouldn't be displayed in full, their internal elements could not be accessed while in the array, but they could be used as a whole unit. This would mean that the demonstrator could not get at the internals of a compound element until it was taken out of the array. We planned to do this to enforce a strict view of abstract data types, as well as to help manage the space and complexity of compound types. However, this solution was not implemented in the prototypes.

4.3 Programming actions

Instructions are one part of the program (declarations are the other part). Instructions determine what actions the clerk does to carry out the program. The program's instructions are printed on the worksheets. To demonstrate an instruction the demonstrator carries out the action that the clerk would do upon reaching that instruction. This section describes the way instructions/actions are demonstrated in the demonstration model. There are four kinds of instructions: assignments, subroutine calls (built-in and demonstrator defined), return statements, and control structures. Each kind of instruction is described using the following general format:

- A description in terms of the demonstrational machine of how the demonstrator demonstrates (programs) the action.
- A description of the way that each of the two prototypes implemented demonstrating the action.
- A discussion of the issues involved with demonstrating the action.

Particular issues relating to built-in I/O subroutines, and inconsistent states are discussed in subsections of their own.

4.3.1 Assignment

In the model, the demonstrator demonstrates an assignment instruction by copying a value into a variable place. The value could be something entered by the demonstrator (a literal), or from another variable or constant place (a reference), or from a calculator (an expression), or a function's return place (an expression). The demonstrator must erase the old value before writing down the new value. The recorder records the assignment using a reference to the variable and the source of the value.

Assignment is done in both prototypes by dragging a value to the variable place, or by typing directly into the place. Only literal values can be typed directly into variables. Values from other variables, calculators, and function calls have to be dragged and dropped on the destination variable. Literals can also be entered into a calculator as an expression which consists only of the literal value, and then dragged to a variable. The new value replaces the place's old value, and the assignment is recorded.

4.3.1.1 Issues for assignment

There are two issues for assignment: the confusion caused by using drag and drop for both assigning a value and declaring a place, and an inconsistency in assignment to array elements.

In both prototypes, assignment and variable declaration (among other things) are performed by dragging and dropping. Assignment is performed by dragging a value to a place. Variables are declared by dragging a type to a worksheet. Assignment and variable declaration both involve variables but are quite different: one is an instruction, the other is a declaration. By using the same interaction technique for both, the prototypes blur the difference between *declaring* a place and *using* a place. This may adversely affect what novices learn. However, the drag and drop method does give the prototypes a consistent interface. It is not clear which is more important. It may be better to use a less consistent interface, in terms of demonstrating, to provide a clearer distinction between the two sorts of instructions.

The other issue for assignment concerns the way that array assignment was done in the second prototype. Assignment to an element of an array is carried out in the same way as a procedure. This is done to avoid stale values in array indices (see section 4.2.4.3). However, using this approach means that different methods are used for assigning to array elements and for assigning to whole variables. If you have a value, assigning it to an array element requires several steps (like a procedure), whereas assigning it to other places requires just one step, *i.e.* a drag and drop. This is another inconsistency that may cause confusion for novices.

One way to resolve the inconsistency would be to do all assignments with procedures. To be consistent with the strict typing used by the rest of the system, you would need a procedure for each type of destination. Another way to resolve this would be to fix the array index and stale value problem so that array elements could be assigned to directly (see section 4.2.4.3 for ways of doing that). Both ways make assignment consistent across all places, but it is not clear which way is better. The first way treats assignment more formally in a structured manner. The second way is less structured, but has a more demonstrational direct manipulation feel to it.

4.3.2 Subroutine calls

In the model, the demonstrator demonstrates a subroutine call by pressing the button for the subroutine, taking the subroutine's worksheet, filling in the subroutine's parameters, and (for programmer-defined subroutines) unfolding the worksheet or (for built-in subroutines) slotting the worksheet back into the device that printed it. The recorder records the call using the subroutine's name and the arguments that the demonstrator filled the parameters in with.

The first prototype only allows built-in subroutines, and does not have worksheets, instead it has a precursor to them. The demonstrator does not have to invoke a subroutine to get its worksheet and see its parameters — all subroutines' parameters were always visible. The demonstrator fills in a subroutine's parameter by dragging a value to the parameter place. The subroutine's parameters can be specified, and re-specified, in any order. Once all the parameters are full the subroutine's name becomes highlighted, and the subroutine is called by clicking on it. None of the code for the call is recorded until the demonstrator clicked on the highlighted button.

The second prototype used worksheets and supported both built-in and programmer-defined subroutines. The demonstrator first has to invoke a subroutine to get its worksheet, and then fills in the subroutine's parameters. Values can be dragged to parameters, or typed directly into them (for literal values only). The parameters have to be specified in order from left to right, as opposed to any order. Why this is so will be discussed in section 4.4.3. Once all the parameters are filled in, the demonstrator can click on the subroutine's worksheet to call the subroutine. If the subroutine is programmer-defined, then its worksheet will unfold; if it is built-in, the subroutine will just be carried out.

When a subroutine is first invoked in the second prototype, a partial call is inserted into the code (see section 3.7.2). As the subroutine's parameters are filled in, the code for each argument is added to the partial call. By the time the subroutine is called, the code for the call is already complete, unless the subroutine is a function. The code for a function call is not complete until the demonstrator has specified what to do with its return value ('...'s are displayed around the call to indicate this, see section 4.1.3). Returning from a function will be described in section 4.3.3.

4.3.2.1 Issues for subroutine calls

There are two issues for subroutine calls (primarily in the second prototype): why each step in the process of calling a subroutine is demonstrated individually, and how this method of programming deals with multiple nested subroutine invocations (usually functions). The first issue will be discussed below. However, the second issue will be discussed in section 4.4.3 because it is closely related to the construction of expressions.

In the second prototype, each step in the process of calling a subroutine — the invocation, filling in each parameter, the call — is demonstrated separately to enforce the idea that the demonstrator is "driving" the machine. The reasoning was that when the clerk drives the machine, subroutines are carried out in these separate steps, so the demonstrator should do them that way too. The question is whether some of the steps should be handled more implicitly. Having to carry out each step for each subroutine call can be quite tedious, especially for zero parameter subroutines (*e.g.*, `WriteLine`). Subroutines could be invoked by dropping the argument for their first parameter on the button, they could be called as soon as the last parameter was specified.

Which way is better? Both ways of doing it have drawbacks. I suspect it comes down to the novice versus expert trade-off. At one stage, the first prototype automatically called subroutines as soon as the last parameter was filled in. This gave the impression that the computer was doing something magical and beyond the demonstrator's control. It also caused problems if the demonstrator made a mistake with one of the parameters (especially the last one), because the subroutine was carried out before they had a chance to check the parameters' values. Exactly the same kind of problem was noticed with Pygmalion (Smith 1993), which also carried out a subroutine as soon as all its parameters were filled in.

4.3.3 Returning from a function

In the model, once a function has been completely carried out and has returned a value, the demonstrator has to demonstrate what to do with the value. The demonstrator can copy the value to a variable place, or copy it to a parameter place of another subroutine or calculator, or use it to make a decision about which set of instructions to carry out next. The recorder records the function call as part of the appropriate kind of instruction: an assignment, another subroutine call, or an `IF` statement (respectively).

The first prototype has two ways to demonstrate where a function's return value should go; one way applies to the calculator functions, and the other way applies to the input functions. In both ways the code for a function call is not created until the function's return destination is specified. Using the calculator, the demonstrator drags the function's return value to a variable, or parameter. In this case, the function is usually the main operator of an expression. The calculator will be discussed in more detail in section 4.4.2. The other way is limited to functions on the input device, and will be discussed in the next section 4.3.4.

In the second prototype, the demonstrator has to drag the function's return value to a place. This specifies a destination for the value, and allows the recorder to properly record the function call in the instructions. Issues relating to partial function calls and their return destinations will be discussed in section 4.4.3 because functions are a major part of constructing expressions.

4.3.4 Demonstrating built-in subroutines with the I/O devices

In the model, the clerk uses the I/O devices to carry out the built-in subroutines for communicating with the computer's user. The demonstrator uses the I/O devices to demonstrate those calls to subroutines. There are three I/O devices in the demonstrational machine: the input device, the output device, and the graphics device. Each device was presented in section 2.2.2.3. The following parts of this section describe how the demonstrator makes use of the I/O devices in the demonstrational model, and how the devices were implemented in the prototypes. Examples of programming with the devices can be found throughout section 3.5. After the devices has been described, the last part of this section discusses some of the issues that arose to do with the devices.

4.3.4.1 Using the Input device

All the subroutines associated with the input device are `Read` functions with no parameters: `ReadNUMBER`, `ReadBOOLEAN`, or `ReadSTRING`. The demonstrator demonstrates reading text input from the user by pressing one of the buttons on the input device, taking the sheet it prints with the value, reading the value, and copying the value to a place. The recorder records which button the demonstrator pressed and what the demonstrator did with the value. However, the prototypes provided the demonstrator with a different way of interacting with the input device's functions.

The input device was only implemented in the first prototype, and was made up of two components: the typing buffer, and the input buffer (see figure 3.6). The typing buffer is where the user types and edits their input. The input buffer is what the demonstrator uses to demonstrate reading the user's input. When the user enters text, the characters that are entered appear in the typing buffer. When the user presses <`Return`> the contents of the typing buffer are echoed to the input buffer. When an input function is carried out, the device scans the input buffer for the first set of characters that match a value of the type that the function reads, and returns those characters. All the characters up to and including that set are discarded from the input buffer.

The input buffer provides the demonstrator the ability to see the characters that the user has entered before the computer reads them. To demonstrate an input function, the demonstrator highlights a region of the input buffer, and drags the selected characters to a place. Note that the demonstrator does not explicitly invoke the function at any point. Which of the three input functions is being invoked is *constrained* by the value selected in the buffer, and is *determined* by the place the value is dragged to. For example, in the first example program in chapter 3 a `STRING` was read from the user (see section 3.5.3). A `ReadSTRING` function was clearly being demonstrated because the value, "`Click here!`", could only be a `STRING` value, and its destination, the `message` place, was a `STRING` place. However, if "`42`" had been selected in the input buffer it could have been dragged either to a `NUMBER` place, or to a `STRING` place.

This way of demonstrating input functions is quite different from the way that the machine model describes the clerk carrying out an input function (see section 2.2.2.7). It is a strange way of specifying a function's destination because the return value is dragged to place *without* the function being explicitly invoked. For example, dragging a `STRING` from the input device to a variable creates an assignment statement without explicitly invoking `ReadSTRING` (see section 3.5.3). In effect, the demonstrator uses the likely return value to demonstrate that they want to call the function.

This is inconsistent with the way the clerk uses the input device, and inconsistent with the way other functions in the prototype are demonstrated. This suggests that the prototype should not allow the demonstrator to do it this way. Rather, the demonstrator should use the input device in the way that the model describes — by pressing the button for the appropriate input function on the device, taking the sheet with the value it prints, reading the value, and copying the value to a place.

4.3.4.2 Using the Output device

All the subroutines associated with the output device are procedures. The demonstrator demonstrates writing out a value by pressing a button on the device, taking the partial worksheet that the device prints, filling in the procedure's parameters, and slotting the worksheet back into the device. The recorder records which button the demonstrator pressed, and what values the demonstrator wrote on the worksheet. However as well as allowing the demonstrator to use this method, the prototypes provided the demonstrator with a more direct way of interacting with the output device.

The output device was implemented in both prototypes, and did not change significantly between implementations. The device consists of two components: a list of buttons, and the text window. The buttons correspond to the various output procedures: (`WriteNUMBER`, `WriteSTRING`, `WriteBOOLEAN`, and `WriteLine`). The text window is a representation of the text output part of the computer's screen that the user sees.

The `Write` procedures can be demonstrated by pressing the buttons and filling in their parameters in the manner described above. Alternatively, the demonstrator can just drag a value onto the text window and drop it. This is equivalent to invoking, filling in the parameter, and calling the output procedure that is appropriate for the value.

4.3.4.3 Using the Graphics device

All the subroutines associated with the graphics device are also procedures. The demonstrator demonstrates a graphics procedure by pressing a button on the device, taking the partial worksheet that the device prints, filling in the procedure's parameters, and slotting the worksheet back into the device. The recorder records which button the demonstrator pressed, and what values the demonstrator wrote on the worksheet. The device also has two constant places, `MouseX` and `MouseY`, that the demonstrator can copy values from.

The graphics device was implemented in both prototypes, and did not change significantly between implementations. The device consists of three components: a list of buttons, a pair of constant places, and a graphics window. The buttons correspond to the graphics procedures, the two constants hold the coordinates of the mouse pointer, and the graphics window is the graphical part of the computer's screen that the user sees. The graphics procedures are demonstrated by pressing the buttons and filling in their parameters as described above.

The graphics window contains a mouse pointer that is known as the pseudo-mouse in order to distinguish it from the real mouse pointer. The user uses the pseudo-mouse to interact with the demonstrational machine's mouse. When the user (who is often the demonstrator pretending to be the user) uses the mouse on the graphics window, the position of the pseudo mouse pointer and the coordinate places are updated.

4.3.4.4 Issues with using the I/O devices

There are two issues relating to the I/O devices: the discrepancy between how they are used in the prototypes and in the model, and the blurred distinction between the demonstrator and the user in the prototypes.

There are two different ways in which the demonstrator could interact with the I/O devices. One method, the machine model method, is the same as the way the clerk interacts with the devices — pressing buttons, filling in worksheets, etc. — and is used in the demonstration model. The other method, which I call the "direct demonstration" method, is used by some of the I/O devices in the prototypes, *e.g.* dragging a value directly out of the input buffer to a variable.

In the first prototype, the input device only allowed the demonstrator to read a value using direct demonstration. The output device allowed both methods of interaction; dragging values to the output window was the direct demonstration option. The graphics device did not allow any direct demonstration. However, the `DrawSTRING` procedure is a prime candidate for using direct demonstration.

The direct demonstration of built-in subroutines speeds up the demonstration process by reducing the number of steps involved in demonstrating the whole instruction. It also provides a more demonstrational feeling of doing what the clerk would do. However, directly demonstrating built-in subroutines is quite different from the demonstration model's description of how the demonstrational machine is used. This difference may cause confusion among novices.

On one hand, the demonstration model is based on the machine model, which describes the way the clerk uses the devices. The demonstration model provides a clear guide to how the computer works, the demonstrator's role, and what the demonstrator has to do to produce a program. On the other hand, the prototypes of the demonstrational machine allow (require in the case of the input device) a different method for using the devices. Thus, these demonstrational machines that are supposed to provide support for the model's way of programming, support an alternate method.

Which method is closer to the way the computer works? The answer to this appears to be that the way the clerk uses the machine is the closer way. This only *appears* to be the answer, because directly demonstrating can be seen as merely a user interface short-cut for demonstrating built-in subroutines. For example, dropping a `NUMBER` value on the text window is a short-cut for invoking the `WriteNUMBER` procedure, filling in its parameter, and calling it. Thus, direct demonstration only conflicts with the demonstration model at first glance[4]. Under this view, direct demonstration is a feature for experts. As such, it does not clash with the demonstration model. However, the direct demonstration short-cuts are still not consistent with the way the model describes the demonstrator interacting with the I/O devices. This inconsistency will not help novices. Using the machine model method on all the devices would be better for helping the programmer understand the model of the computer.

Which method is the best for learning to program? Which is the best for programming by demonstration? The answer to these two questions requires some experimental analysis, and feedback from novice and expert programmers. However, I would guess

[4]Of course, this may be enough to confuse novices.

that the machine model method is better for novices, and that direct demonstration is better for experts. Thus ideally, the next generation of demonstrational machines should support both interaction methods.

However, if the current direct demonstration methods in the prototypes are meant to be more than just user interface short-cuts, then they would be at odds with the current demonstration model. For example, if the demonstrator drew a line directly onto the graphics window and the recorder was supposed to infer the variables that the demonstrator was thinking of for the `DrawLine` call, then this would not be consistent with the demonstration model. If this were the case, then either direct demonstration should be integrated into the model, which would probably mean substantial changes to the model, or it should be removed completely.

The second issue concerns the distinction between the two roles that the human programmer plays while using the demonstrational machine: the demonstrator, and the user (see section 3.3). In the prototypes, the representation of the I/O devices combines the parts of the computer's screen that the user sees with the devices that the clerk uses. This blurs the distinction between when the programmer is acting as the demonstrator and when the programmer is acting as the user. This may cause a problem for novices.

A clearer way to represent the devices would be to split them into two clearly defined parts: one for the parts of the computer that the demonstrator interacts with, and one for the parts of the computer the user interacts with. The representations of the I/O devices could be split across a graphical boundary to clearly distinguish the areas of the computer that each role can interact with (see figure 4.1).

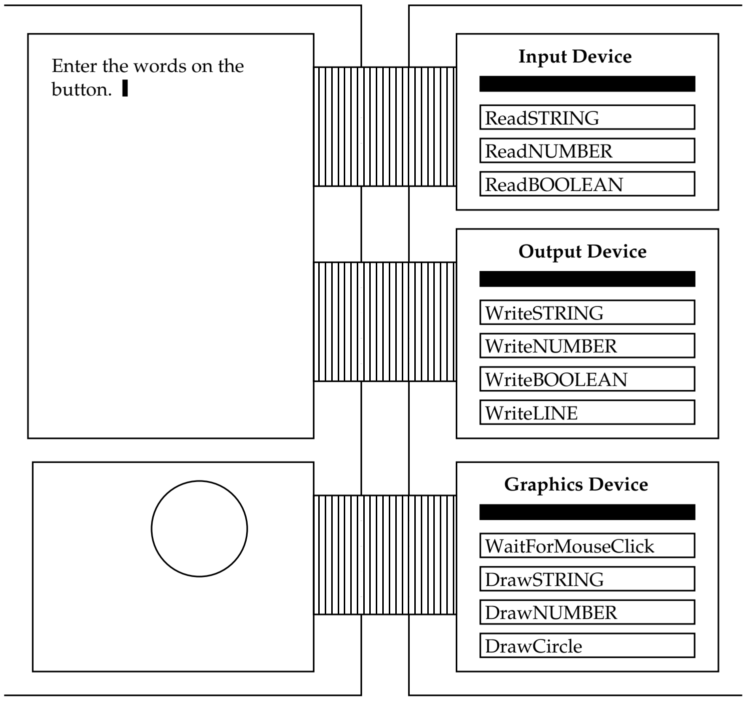


Figure 4.1: Separating the parts of the I/O devices cross a graphical boundary.

A consequence of this is that the user's view of the input and output devices could be combined into a single text window. The prototypes separate text *input* from text *output* because they are separate from the clerk's inside view of the computer. However, from the user's point of view (outside the computer) any sort of text usually appears on the same window, which would make the prototype more consistent with the way a computer usually appears to a user. Perhaps the graphics window should be part of this window too. On the other hand drawing graphics may be sufficiently different that it should have its own window.

4.3.5 Demonstrating control structures

There are two kinds of control structures: `IF` and `LOOP` (with `LEAVE` statements).

In the model, the demonstrator demonstrates an `IF` statement by constructing an expression, evaluating it, using its value to determine which course of action to take, and indicating to the recorder that the value is being used in a such determination. The recorder records this as an `IF` statement using the expression. How the demonstrator demonstrates an expression will be discussed in section 4.4.

The demonstrator has to explicitly signal to the recorder what they are using the expression's value for. The use of this value cannot be demonstrated explicitly because the demonstrator is acting as the clerk and the use occurs *inside* the clerk's (and therefore the demonstrator's) head.[5]

The demonstrator can demonstrate an `IF` statement at two points:

1. **Before demonstrating the instructions in either branch.**

   Demonstrating an `IF` statement before either branch is described above, and one of the branches is demonstrated the first time through the `IF` statement. This leaves one branch unspecified until the statement carried out again on a later execution of the program.

2. **After demonstrating one of the branches.**

   An `IF` statement is demonstrated after one of its branches by adding it in the middle of a set of instructions. At the end of the first branch of the statement, the demonstrator can jump the program forward over some of the instructions that follow the `IF` statement. The recorder moves the skipped instructions into the other branch of the `IF` statement.

The demonstrator demonstrates a `LOOP` statement by jumping the program counter to an earlier instruction on the same worksheet, and getting the clerk to carry it out again. The recorder records this action as a `LOOP` because the demonstrator has acted as the clerk would have if the clerk had encountered an `ENDLOOP` instruction. The recorder puts

[5]Unless the recorder is given the ability to read the demonstrator's mind (*i.e.*, enable correct inferences), which would constitute a major modification of the whole model.

a `LOOP` at the point the demonstrator jumped *to*, and an `ENDLOOP` at the point the demonstrator jumped *from*. The recorder prevents the demonstrator from interleaving loops. Between a `LOOP` and an `ENDLOOP`, the demonstrator can demonstrate a `LEAVE` statement by jumping the program to just after the `ENDLOOP` statement in the instructions. The recorder records a `LEAVE` statement at the point *inside* the `LOOP` that the demonstrator jumped the program from.

However, neither prototype follows the model completely because they do not allow the demonstrator to jump the program counter around. Rather, the way control structures are demonstrated in the prototypes is similar to the way built-in procedures are demonstrated — filling in parameter places with values, and then calling them. The "call" to a special control structure "procedure" causes the recorder to record an outline of the control structure at the current point in the instructions. Even though `IF` statements are actually consistent with the demonstration model, their interfaces look a lot like a procedure that takes one parameter. The second prototype distinguishes the control structure "procedures" more clearly than the first prototype because the control structure "procedures" do not have worksheets.

There are three special control structure "procedures": `IF`, which requires a single `BOOLEAN` parameter, `LOOP`, and `LEAVE`, neither of which have parameters. The structure of an `IF` statement is:

```
IF ( expression ) THEN
  true body
ELSE
  false body
ENDIF;
```

Where the `expression` is the parameter of the IF "procedure". The structure of a `LOOP` statement is:

```
LOOP
  body
ENDLOOP;
```

A `LEAVE` statement has no structure, it consists of just the keyword '`LEAVE`'. A `LOOP` has one body, and an `IF` has two bodies, one for each branch. Each body contains a "Not Specified Statement" (see section 4.1.3.3). Demonstrating a control structure causes the program to enter the body of the structure, and the demonstrator can demonstrate from there. For `IF` statements, the branch that the program enters is determined by the current value of the expression. The other branch can only be demonstrated when the program enters it, typically on a later execution of this part of the program.

Thus, `IF` statements can only be demonstrated before its branches, and not as an alternative to previous sets of instructions (branch). `LOOP` statements are demonstrated quite differently — as a special "procedure" that adds instructions to the program. The prototype's method of demonstrating both control structures is more like asking the recorder to put the appropriate structure in at a point, than demonstrating it, *e.g.* saying "I need a `LOOP` here" instead of saying "I want to go back to this point and do these instructions again". For `IF` statements, the prototypes allow the demonstrator to say "I only do this sometimes", but they do not allow the demonstrator to say "Wait, I want to do something different this time around".

4.3.5.1 Issues for control structures

There are two issues for control structures: why the prototypes use a different method to the model for demonstrating `LOOP` statements, and how control structures should be demonstrated.

The prototypes use a different method for demonstrating `LOOP` statements because we wanted a quick and easy way of implementing them. The prototypes did not seriously address `LOOP` statements. There are several ways to demonstrate control structures (the prototypes implemented `LOOP` statements using the first method described below):

- **By specifying the structure before any instructions**

  This method is easy to implement. It has two problems: it is top down (like structure editing) because it requires the demonstrator to realise in advance that they need the control structure, and (therefore) it is not particularly flexible.

- **By selecting a set of instructions to place the structure around**

  The demonstrator highlights a set of instructions, and asks the recorder to create the control structure around those instructions. An `IF` statement could be created by putting the highlighted instructions in one of the branches and leaving the other branch empty. The demonstrator would have to specify which branch to fill, and an expression for the `IF` statement. Instructions could be added to the empty clause on future executions. An advantage of this method is that it enables control structures to be added on subsequent executions of the instructions, *e.g.* when the demonstrator sees that the program needs to do something different in this case. A drawback of this method is that is requires the demonstrator to interact directly with the instructions, and guess how many of the instructions should be part of the structure *before* the structure has been carried out.

- **By marking the start and end of the structure as you go**

  The demonstrator demonstrates a control structure by specifying the start of a control structure, which the recorder records as the first statement of the structure (*e.g.*, `LOOP`). Then the demonstrator demonstrates instructions normally, until at some later point the demonstrator specifies the end of the control structure. The recorder records that as the end of the control structure (*e.g.*, `ENDLOOP`). For `IF` statements, the demonstrator has to provide an expression for the first part of the structure, and has the option of specifying an `ELSE` or a `ENDIF` at the "end" point.

If the demonstrator specifies an `ELSE`, the other branch of the structure could then be demonstrated.

This method is like specifying a region of a video or audio tape to play repeatedly. An advantage of this method is that it requires no direct interaction with the code, rather the demonstrator chooses the right state or point in time to end the structure. As a result, it is more demonstrational than the previous method. A disadvantage is that it allows programming with the equivalent of constrained `GOTO` statements.

- **By using the demonstration model's method**

  The demonstrator demonstrates control structures by using the methods described at the front of this section (see above). For `IF` statements, this means constructing an expression, and using the expression's value to decide what to do. At the end of first branch, the demonstrator may also move the program counter forward to skip over previous instructions that should be in the other branch. For `LOOP` statements this means that the demonstrator would decide that a previous instruction is the next one to carry out, and move the program counter back to that statement — the first statement in a `LOOP`.[6]

  This method is more demonstrational than the previous methods, and it is more consistent with the way other instructions are demonstrated because the demonstrator acts in the same way the clerk would.

- **By inferring the control structure, but not the condition**

  One method along these lines is the method used by Tinker (Lieberman 1993b). Tinker's method is for the demonstrator to demonstrate one case of a function, then demonstrate another case of the same function. When Tinker comes to carry out the function it finds that there are two versions, and asks the demonstrator to enter a distinguishing condition that will cleanly separate the two versions of the function. Tinker uses the condition to combine the two function into a single function with an `IF` statement. Tinker's method applies to languages which allow many functions with the same name or multiple clauses. Tinker doesn't work out the condition, it just works out where the condition needs to be. It allows further versions of the function to be written and combined on subsequent iterations.

  An equivalent method for `IF` statements in the demonstrational machine would be to demonstrate a set of instructions, then interrupt the clerk carrying out those same instructions at a later time, and demonstrate another instruction. The recorder should ask the demonstrator whether the instruction is an alternative to or follows on from the current instruction. If the instruction was meant to be an

[6] For `LOOP` statements, the demonstrator takes the program counter back to the future. For `IF` statements, the demonstrator takes the program counter forward to the future.

alternative, the recorder would record an `IF` statement and ask the demonstrator to demonstrate a conditional expression for the `IF` statement.

Neither of these methods for demonstrating `IF` statements attempts to infer the conditional expression. Doing that is hard, but it would be extremely useful.

It should be possible to infer `LOOP` statements from repetition. For example, the demonstrator carries out the same sequence of instructions to the first few elements on an array, and from that the recorder infers a `LOOP`. The inference might be simple enough for iterations along each element in an arrays, but it is likely to get quite complex for other kinds of loops. Eager (Cypher 1993a) and Metamouse (Maulsby & Witten 1993) are examples of systems that infer loops from repetition.

- **By inferring the control structure and the condition**

  Methods in this category attempt the difficult task of inferring the condition as well as the structure. For an `IF` statement, this means working out the conditional expression to use; for the `LOOP` statement, it means working out the stopping condition.

It is not clear which of these methods is the most useful for programming. The inference-based methods may help novices and experts by relieving them of some of the burden of having to figure out in advance how many cases there are, or what the condition should be. However, these methods might confuse a novice, if the novice cannot follow what the recorder is doing, or if the recorder makes a mistake. The non-inferencing methods that allow the structure to be demonstrated after the instructions are more likely to suit an expert, who will be used to planning for different cases, whereas, the program counter manipulation that these methods require may cause novices some trouble.

4.3.6 Inconsistent states

The demonstrational machine is in an inconsistent state if it is not in the state it should be in as a result of carrying out all the instructions up to this point in the program. This means that given the instructions that should have been carried out, the state of the I/O devices and/or values in data structures are not correct. An inconsistent state is dangerous because some value or device state is incorrect, and the demonstrator may depend on that value or device to demonstrate the next instruction. As a result, the demonstrator may well demonstrate the wrong instruction, even though the instruction will appear to be the correct one in the current state.

The way that programs are "written" with the demonstrational machine makes it extremely easy to reach an inconsistent state. If any new instruction is added to a section of code after the first time that code has been carried out, then the demonstrational machine will be put into an inconsistent state. The combination of adding instructions by demonstrating an action and allowing programs with repetition makes it difficult to avoid inconsistent states. Repetition can arise from iterative constructs or from calling the same procedure from two different points in the program (which includes recursion).

Inconsistent states can be caused in at least three other ways: wishful programming (see section 4.2.3.1), certain subroutine declaration methods (see section 4.2.3.2), and stale values. Stale values are a special kind of inconsistent state that involves inconsistent expression values. Stale values will be discussed further in section 4.4.1.

The demonstration model assumes that the state of the machine is always kept consistent, but does not specify how this should be done. The recorder could do one of three things to cope with the state becoming inconsistent: automatically take the machine back to an earlier consistent state, remind the demonstrator that the state is inconsistent and leave it to them to deal with, or automatically unwind to a consistent state and then re-execute forward to the current point (this could be done visibly or invisibly). The prototypes do not use any of these methods, and as a result it is easy to get them into an inconsistent state. However, when using the prototypes the fact that the state was inconsistent did not seem to matter because I took the inconsistencies for granted, and often re-ran the program from the beginning. It remains to be seen whether this is sufficient for novices. Chapter 6 discusses my experiences when programming with the prototypes in more detail.

4.4 Programming expressions

In the model, the demonstrator demonstrates an expression by using a calculator to construct and evaluate the expression. The recorder records the values and operations that the demonstrator entered into the calculator as the expression.

In the prototypes, several techniques for constructing expressions were implemented. The first prototype had a single calculator, which allowed functions and operators to be used in any order to construct complex expressions. The second prototype used a different approach with multiple calculators and functions on worksheets, which required expressions to be constructed in the same order as the clerk would evaluate them.

The following subsections describe the techniques used in the prototypes, and discuss the issues involved with those techniques. This section concludes with a description of some of the other methods that could have been used, and a description of the way the next version of the demonstrational machine should allow expressions to be demonstrated.

4.4.1 Stale values

A stale value is caused by evaluating an expression in one state but not using its value until a later state. The expression hangs around and is still associated with the value. The value is stale if the expression would evaluate to a different value in the later state. If a stale value is used to demonstrate an action, then the recorder will use the expression in the instruction, but the stale value will be used to update the state. This is inconsistent

— subsequent executions of that instruction may have a different effect from what the demonstrator had intended. Stale values arise from side effects that occur during the interval between evaluating an expression and recording the expression in the code.

The first prototype avoided stale values by always re-evaluating expressions as they were used. The second prototype prevented stale values in almost all cases by preventing the demonstration of other instructions while an expression was under construction, and by requiring the expressions to be used in an instruction immediately after being evaluated. These strictures will be discussed further in section 4.4.3. The exception to them was expressions put into an array's index place, which caused problems (see section 4.2.4.3).

4.4.2 Single calculator

A single large calculator was used in the first prototype. It consisted of four main components: a display area for the entire expression, a display area for the current sub-expression that was being entered, a keypad, and a list of subroutines (see figure 4.2). Each expression display area shows the expression, its value (if it can be calculated), and the type of value that the expression will return.

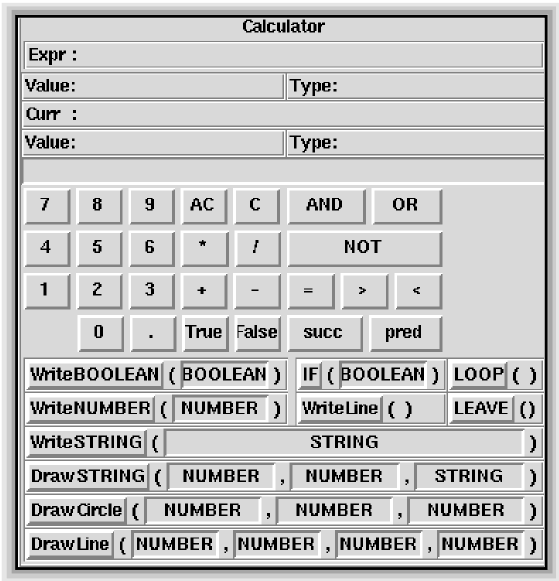


Figure 4.2: The first prototype's single calculator

The calculator is used to construct expressions and evaluate them. It works in a similar way to a normal calculator, but with some enhancements.

- The expression is displayed as it is being constructed, and is bracketed automatically.
- The demonstrator can drag values from variables and constants to the calculator in order to add them to the expression.

- All the built-in functions (and operators[7]), except input functions, can be used to construct expressions. Input functions are not allowed because they have a side effect on the input buffer, which makes it problematic to re-execute them.
- Each function extends the expression by adding its name and places for it arguments at the point where it is entered. For example, entering '+' at a "Not Specified Expression" adds the structure '($\square$ + $\square$)'.
- The calculator is type sensitive so only functions that return the appropriate type of value can be entered.
- The expression can be specified left to right, right to left, from the outside in, or from the inside out, but the expression must have a well structured expression tree at all times. The expression tree can be incomplete (*i.e.*, by having "Not Specified Expressions"), but it cannot consist of two or more disjoint sub-expressions.
- The expression can be traversed to view the values of its sub-expressions.
- Sub-expressions can be replaced by selecting them and entering new expressions.

As the demonstrator uses the calculator, the entire expression is displayed. The expression will contain a "Not Specified Expression" for each part of the expression that hasn't been specified yet (see section 4.1.3.1). The next place to be filled in is highlighted in the entire expression's display area; this is known as the current expression. The details of the current expression are shown in its own display area (see figure 4.2).

If a value is entered into the calculator, either by typing it, or by dragging it from a variable or constant place, then that value replaces the old current expression. The current expression moves on automatically to the next "Not Specified Expression" if the value was entered by *dragging* a variable or constant to the calculator. If the value was *typed* in, then the current expression does *not* move on until the demonstrator presses the <`Enter`> key (unlike a standard algebraic calculator in which the next operator ends the value, and similar to an RPN calculator). The reason for this is that a variable value is entered completely in one action, whereas typing in a value can go on for an unknown number of key presses and the calculator has to be told when the value is complete.

If a function is entered into the calculator (by pressing a key), the current expression is used as the first parameter to that operation. An example of this is shown in section 3.5.6 when '$(LeftX + RightX)$' was used as the first parameter to a '`/`' operation (see figure 3.11). After adding the function's structure to the expression, the current expression moves automatically to the next "Not Specified Expression". This will generally be the first parameter of the new function.

Even though procedures were placed on the calculator in the first prototype, procedures cannot be used to construct expressions because they do not return values. However, as a short-cut for dragging a completed expression to a procedure parameter, the

[7] I do not distinguish between functions and operators. Addition, subtraction, division etc. are all functions.

demonstrator could click on a procedure with an empty parameter, and (if the types matched) the expression in the calculator would be put in the parameter place.

The calculator prevents the demonstrator from type mismatches. Functions and keys that produce a different type of value from the current expression are disabled. For example, if the expression is '(2 + $\square$)' (where '$\square$' is the current expression), then only numbers and functions that return numbers will be enabled on the keypad (see figure 3.10).

The position of the current expression can be changed by traversing back and forth through the whole expression with the aid of the arrow keys (see figure 4.3). The new current expression is highlighted and its details displayed. The traversal follows the expression tree (which the demonstrator cannot see) in steps of one parameter, with a sub-expression considered to be a parameter of its parent expression (see figure 4.3).

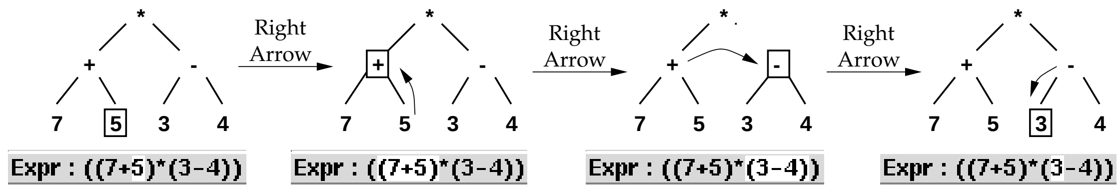


Figure 4.3: An example of traversing an expression.

Being able to select different parts of the expression enables complex expressions to be constructed and altered in any order (see section 3.5.11 for an example). This is important because it solves a problem that arises as a result of the calculator automatically bracketing expressions.

The problem with automatic bracketing (or structure building) is that if the demonstrator enters '7 + 5' and then presses '*', should the expression become '$\underline{(7+5)} * \square$' or '$7 + \underline{(5 * \square)}$' (where '$\square$' is the current expression). On a typical algebraic calculator the expression would become '$7 + (5 * \square)$' because an algebraic calculator uses a certain operator precedence (of course, the person using the calculator wouldn't usually be able to see the expression). However, the demonstrator may have wanted the other expression. Algebraic calculators get around this problem by having brackets, '(' and ')', and forcing the expression to be entered with the correct bracketing. By allowing the demonstrator to view and traverse the expression, our calculator enables the demonstrator to see and choose the appropriate place for the '*' function without the need to enter brackets explicitly or use a certain operator precedence. The user does not have to plan ahead to make sure that they enter the correct formula as they have to do on an algebraic calculator (Young 1981).

A complete expression is evaluated and its value is shown in the appropriate display area (entire or current). Once the *entire* expression is complete, it can be dragged to another place, such as a variable, parameter, or a return place, or used as part of an `IF` statement (see sections 4.3.1, 4.3.2, 4.3.3, and 4.3.5).

4.4.3 Demonstrating functions in the order of evaluation

The first prototype's single calculator let the demonstrator construct an expression in any order by editing the structure of the expression. The single calculator did not have to deal with built-in functions that had side effects, or programmer-defined functions (possibly with side effects), or the possibility of unwinding the state at any stage during an expression's construction. This meant that it was easy to construct expressions in (almost) any order. However, the second prototype had to deal with these things.

There were three differences related to expressions between the first and second prototypes: the order of construction, the degree of demonstration involved, and the number of calculators. The order in which expressions were constructed was the greatest difference. The second prototype's method for building expressions is less flexible than the first prototype's method, but it is also more demonstrational.

The second prototype planned to have three calculators in place of the first prototype's single calculator, however only one of the three calculators was implemented. That calculator will be presented below in section 4.4.4.1. The second prototype required the demonstrator to demonstrate expressions in the order the expression would be evaluated by the clerk (see section 3.7.8) because the demonstrator was taking the clerk's place to produce the program. Each function has its own worksheet that has to be filled in and carried out before the function's value can be used: built-in functions have calculators or partial worksheets, and demonstrator-defined ones have full worksheets. Built-in functions for expressions (such as the arithmetic operators) should have been implemented on a calculator, but as only one of the three calculators was implemented, most of the functions were left as worksheets. For example, to demonstrate the expression '$(4 + (6 * 2))$', the demonstrator must invoke '+', fill in its first parameter with '4', then invoke '*', fill in its parameters with '6' and '3', call '*' and drag its return value to the second parameter of the '+' invocation, and finally call '+'.

There are two reasons for this method: the second prototype had to handle side effects and state, and we wanted the demonstrator to use a more demonstrational method of specifying an expression. Restrictions on the use of calculators and worksheets were imposed to ensure that no stale values or strange side effects occurred in the process of demonstrating an expression. Expressions had to be demonstrated in the order of evaluation, *i.e.* from the outside in and from left to right. This meant that functions had to be demonstrated in the order they would be called in (see section 3.7.8). The following rules were enforced to ensure that stale values and inconsistent states did not arise:

1. **Invoking a new function over another function's folded worksheet forces the new function to be a parameter of the other function.**

   This prevents the demonstrator from invoking multiple functions without deciding which function is the parameter of the other.

2. **Parameters must be specified in order from left to right.**

   This constrains the demonstrator to always fill in the next unspecified parameter, which prevents arguments from being evaluated out of order.

3. **A return value must be dragged to a place on the worksheet directly beneath its own worksheet; it cannot be dragged over an intervening worksheet to third worksheet.**

   This prevents the demonstrator from re-ordering the expression, or demonstrating another instruction, *e.g.* an assignment, in the middle of constructing the expression.

4. **A completed function's return value must be dragged to a place before anything further is demonstrated.**

   This ensures that completed functions are used straight away, and are not left hanging.

These restrictions ensure that the demonstrator demonstrates the components of an expression in the order that the clerk would evaluate them, and therefore cannot generate stale values. Issues to do with demonstrating expression in this order and a comparison with the first prototype's method will be presented in section 4.4.5.

4.4.4 Multiple specialised calculators

The single calculator in the first prototype was large and quite complex. Most of the time, only about half of its keys could be used because the rest were disabled to prevent type mismatches (see section 4.4.2). Experience with the single calculator, especially for entering strings and constructing conditional expressions, led to the idea of splitting the calculator into several smaller type and operation specific calculators. We envisioned three specialised and more manageable calculators, each restricted to certain operations and types of values:

- A numerical calculator working only with numerical operations on `NUMBER` values.
- A logical calculator working only with logical operations on `BOOLEAN` values.
- A comparison calculator working only with comparison operations between two values of the same simple built-in type, *i.e.* two `NUMBER` values, two `BOOLEAN` values, or two `STRING` values.

Only the comparison calculator, or "comparator", was implemented in the second prototype. The numerical and logical calculators were going to be smaller versions of the first prototype's calculator, complete with expression traversal.

4.4.4.1 The Comparator

The comparator is used only for building comparison expressions, and is quite different to a standard calculator. It consists of two main parts: a pair of weighing scales with two places for values, and a set of comparison return places (see figure 4.4). The comparator has one return place for each of the six built-in comparison operations: $<$, $>$, $=$, $<>$, $>=$, and $<=$. The comparator is rather like a special function that takes two parameters, and carries out six operations in parallel.

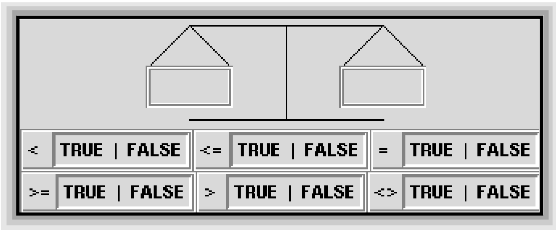


Figure 4.4: The Comparator

To demonstrate an expression with the comparator, the demonstrator first fills in both of the comparator's places with values, and then drags one of the return values to a variable, a parameter, or an `IF` statement (see section 3.7.7 for an example). Dragging a particular return value from the comparator demonstrates that demonstrator wants to use the corresponding comparison operator.

Until the first value is dragged to the comparator, neither of the comparator's value places has a defined type; they are both just empty rectangles (generic places). As soon as the first value is entered into the comparator, both places become places for that type of value. This ensures that only values of the same type will be compared. It also allows the comparator to be used with any of the simple built-in types, without requiring a special comparator for each type.

Once both arguments have been filled in, the comparator calculates all the return values for all of its comparison operations, and displays them in the appropriate return places. This allows the demonstrator to view all the results and choose the one that they want to use. The fact that the demonstrator can see all the options should help to avoid the off-by-one errors that can occur when specifying a comparison in traditional programming, *e.g.* when a programmer must choose between '$>$' and '$>=$' for a `IF` statement.

However, displaying all six results and allowing the demonstrator to choose one could be the cause of another problem. When a demonstrator (especially a novice) comes to select a return value, there is nothing to stop them grabbing the nearest one that looks like it has the result that they are expecting. If the demonstrator is completely unsure what the comparison should be, then showing all the return values may confuse more than it helps.

4.4.5 Evaluation of the methods

The first prototype's method for constructing expressions is based on specifying the structure of the expression. The demonstrator could edit the expression, and view the values of sub-expressions until the expression was completed to their satisfaction. Then the expression could be used in an instruction. This method had the advantages of being able to specify the expression in any order, being able to see the whole expression as it is built, and of being somewhat familiar in terms of being somewhere between a normal calculator and the way we would write an expression down. However, the method is not very demonstrational and it does not show the order of evaluation as well as the second prototype's method.

The second prototype's method was more demonstrational than the first prototype's method. The demonstrator had to specify the expression in the order that it would be evaluated: from the outside to the inside, and from left to right. This method is more restrictive because the expression had to be built in a particular order. An expression with more than two operators is quite cumbersome to build using this method, *e.g.* '$n * Factorial(n - 1)$' (see sections 3.7.8 to 3.7.10). Each operator is another function invocation, with parameters to be filled in, which has to be called, and with a return value has to be used somewhere. Despite the presence of unlimited undo and unwind it was tedious and cumbersome to construct expressions this way, especially if the demonstrator got the order wrong. However, with this method it was clear how the clerk would evaluate the expression.

Neither prototype used a satisfactory method, though each had its good points.

4.4.6 Related expression building techniques

Lots of other applications need to provide methods for constructing expressions, *e.g.* spreadsheets, document production systems, and calculators.

Spreadsheets typically allow straight text editing of expressions, which is extremely flexible and allows expressions to be constructed like written formula. However, it also allows invalid expressions. To alleviate this, spreadsheets often incorporate some form of structure editing expression builder. For example, Excel's function wizard that allows the user to choose from a list of functions that explains the function and its arguments. Once a function is chosen, it allows the user to fill in each of the function's parameters. It gives good feedback by displaying the value of the parameters, and the function as it is being constructed. Some spreadsheets (Excel among them) allow the equivalent of dragging variables to an expression because they allow particular cells to be selected and used in the expression's construction.

FrameMaker (Adobe 1996), a document production application, has a comprehensive equation builder. FrameMaker's equations palette is a sophisticated structure editor, and evaluator, which can perform a variety of rewrite and evaluation rules on complex equations. It allows the user to specify the structure of an expression and fill in the blanks. It allows the user to type the equation, choose from a palette of symbols and equations, and select regions of the equation to edit, delete, and evaluate. The unspecified parts of the equation (marked with '?'s) can be traversed using the arrow keys. If the equation is clicked on, a cursor is inserted and the whole equation can be edited as text.

There are a wide variety of calculators, both hand held and computer applications, which use slightly different methods for constructing and evaluating expressions. There are three main kinds of hand held calculators: four function, stack based reverse polish notation (R.P.N.), and algebraic. Each kind of calculator has a different model of operation that is often unclear to the calculator's user (Young 1981, Mayer & Bayman 1981). Computer applications that implement calculators come in many varieties including: those that mimic hand held algebraic calculators, those that mimic adding machines with tapes that provide a history of the calculations and can be reused, R.P.N. calculators that have a visible stack, calculators with a number of named registers or variables that can be used in expressions, and mathematical working and evaluation environments like Maple and MathCAD (*e.g.*, Goldson (1994)).

These applications and their methods contain a lot of good ideas, but it is not clear which method or even combination of methods is the best. It is clear that there is no single "right" method, nor is there a single conceptual model that should be used as guidance.

4.4.7 An expression building method for the next prototype

The next implementation of the demonstrational machine should use a combination of the methods used in the first two prototypes. It would have specialised calculators for numbers, logic, and comparisons. The comparison calculator would remain as it was in the second prototype. The numerical and logical calculators would be like smaller versions of the first prototype's calculator. Expressions on a numerical or logical calculator could be traversed and altered like the first prototype's calculator. However, such traversal would have to unwind and redo the evaluation of the expression to avoid creating stale values.

The overall method of demonstrating expressions would be similar to the second prototype, with a use of one of the calculators treated as a function invocation. It is hoped that with most of the built-in functions on calculators, a balance between the methods used in the prototypes could be achieved. Using a calculator would create partial code just as in the second prototype (see section 3.7.2). Demonstrator-defined functions would still have their own worksheets, and would still have to be invoked, have their parameters filled in, be called, and have their return values used. Calculators and worksheets could both be used in the same expression.

However, more freedom to demonstrate an expression in any order would be provided by handling multiple invocations differently to the way the second prototype handled them. Instead of being forced to demonstrate functions in the order of evaluation (from the outside in), the demonstrator would be allowed to invoke multiple functions at once. If a second function was invoked before the first has been called or has had its parameters filled, then the second is put alongside the first. Neither is constrained to be a parameter of the other, and therefore, initially, the functions would not be nested inside each other, *e.g.* '$\ldots Factorial(\ldots min(\square, \square)\ldots)\ldots$', but would all be at the same level, *e.g.* '$\ldots Factorial(\square)\ldots min(\square, \square)\ldots$'.

Once one of the functions has had all its parameters filled in and has been called, then the order can be specified. The demonstrator drags the return value of one function to a parameter of another function to specific how the functions are connected. This allows multiple functions to be connected in any order. For example continuing the above example, *Factorial* could have its parameter filled in, be carried out, and have its return value dragged to one of *min*'s parameters, or vice versa. Any number of functions could be invoked at the same time, and re-ordered to suit the expression that the demonstrator wants. This provides a less constrained method than the second prototype's method. However, it is not completely unconstrained because in order to keep the state consistent some problems have to be addressed:

- **Stale values**

  This method with have to deal with the stale values that will be created by re-ordering functions. We don't want to constrain the way expressions can be constructed because that would result in the problems of the second prototype's method. Instead of preventing stale values, as the second prototype does, it should deal with stale values by re-executing the appropriate parts of the code with a mixture of unwinding and re-executing. Whether this is done silently by the recorder so the demonstrator is unaware, visibly by the recorder so the demonstrator isn't surprised if the values change, or only when explicitly requested by the demonstrator, is something that must be determined. How to detect the possibility that stale values may arise in an expression is another issue that must be considered.

- **Function worksheets**

  If multiple functions and stale values are allowed, then function worksheets can be left lying around after the point at which they were invoked. Presumably, the demonstrator could fill them in and call them as needed, which is acceptable because stale values will be dealt with somehow. However, having lots of unconnected function worksheets hanging around will affect the speed of the system, especially if expressions are constantly checked for stale values. Unconnected worksheets also raise scoping issues. For example, it is not clear what should happen to an unconnected function if the subroutine that it was called from is completed (*i.e.*, the subroutine's worksheet is folded up). These issues will have to be addressed.

A simple but constraining solution is to prevent the program from being carried out past an instruction which involves multiple functions, until all the functions' return values have been used or removed.

- **Multiple copies of calculators**

  If this method is going to give the demonstrator freedom to construct an expression in any order, then it has to allow multiple copies of the same kind of calculator. For example, the expression '$min((index+1),(counter-1))$' could be constructed by constructing the arguments, '$(index+1)$' and '$(counter-1)$', before invoking 'min'. In order to have both arguments constructed and waiting, there have to be two copies of the numerical calculator because both arguments require the numerical calculator.

  Alternatively the calculators could be extended to hold several expressions at a time. However, the current plan is that each calculator can only hold one expression at a time.

These issues are by no means an exhaustive list of the things that will have to be dealt with to make this method work. Even if the issues are dealt with, there is no guarantee that this method will be a good way of building expressions in this context. It will no doubt be a question of trading-off between what is the best for learning, and what is the best for ease of programming. I suspect that in some situations, expert programmers will always want to specify an expression by directly typing it in.

4.5 Summary

This chapter described in detail the demonstration model, the prototypes' support of the demonstrational machine, and a number of issues related to aspects of the model. The chapter was separated into four parts: programming, declarations, instructions, and expressions.

The "programming" section described the difference between the demonstration model's way of programming and traditional programming. The demonstrational model's way of programming has the programmer carry out the actions that the computer would do. The program is created by recording the demonstrator's actions as meaningful chunks of code. This requires the demonstrational machine to deal with partial instructions and partial programs. The four different kinds of "Not Specified" instructions created to deal with partial instructions and programs were described.

The "declarations" and "instructions" sections described how declarations and instructions are constructed with this way of programming. The way to demonstrate each declaration or instruction was described in terms of the theoretical demonstrational machine, and the two prototype implementations of it. Issues related to the various declarations and instructions were also described. Several spectra of different methods for demonstrating declarations (and control structures) were discussed. These methods ranged from less demonstrational to more demonstrational.

The "expressions" section focused on the different expression construction methods used in the prototypes. The first prototype used a single special calculator that allowed expressions to be constructed in almost any order. The second prototype used multiple function-like calculators and forced expressions to be constructed in the order that they would be evaluated by the clerk. The problems caused by stale values were described, along with what each prototype did to avoid to them. The prototype's methods for constructing expressions were evaluated with neither being clearly better than the other. Finally, a method for constructing expressions that might be used in the next implementation of the demonstrational machine was outlined.

The demonstration model describes what the demonstrator does as part of the demonstrational machine to produce a program. The prototypes differed from the demonstrational machine in several places. Often it was not clear which had the better approach. Most of the unresolved issues discussed in this chapter relate to the trade-off between novice and expert programmers, which is equivalent to the trade-off between ease of understanding and ease of use.

Chapter 5

Issues

Exploring the prototypes revealed other issues that were related to more than just the demonstration model. This chapter presents the other issues that affect the models and environment. Resolving most of the issues here is beyond the scope of this thesis. The intent of this chapter is to shed some light on the issues.

Section 5.1 discusses the various operations that the environment needs to be more usable. Section 5.2 discusses alternative ways of displaying the code to help novices see the structure of the program. Section 5.3 discusses the accuracy of the models and prototypes with respect to worksheets and real activation records. Section 5.4 discusses a problem with the way that the prototypes use the "drag and drop" interaction technique. Section 5.5 discusses the problem of constructing an expression that will be used in a program. Finally, section 5.6 briefly discusses unresolved issues for teaching novices with this environment.

5.1 Capabilities for serious programming

This section is about operations that have not been fully considered in the demonstration model, but which the demonstrational machine nonetheless needs to have. There are two operations that fall into this category: the ability to edit code, and better execution control of the machine. The first operation, editing, has not been mentioned so far because it was not something that either prototype explored. The second, better execution control of the machine, is a notion that arose out of my experience with the second prototype. This section describes each of these operations, why they are needed, how they help, and some of the problems that will have to be addressed in order to add them to the demonstrational machine *and* retain an environment with a consistent and unwindable state. Note that editing here includes the deletion, addition, and modification of pieces of code. Addition is already incorporated into the prototypes — the demonstrator can demonstrate action(s) to add a piece of code. Deletion and modification are more difficult.

5.1.1 Deletion

An operation for deleting statements was not considered a necessary part of the prototypes, which were meant for exploring this way of programming. However, a full programming environment based on the demonstrational machine would definitely need deletion. Undo only provides a limited delete facility because often you have to throw away too much in order to delete a statement. Undo is important, but is not as powerful as deletion because undo removes statements in a fixed order. Therefore, in order to delete a statement using undo, before you can delete the statement itself, you have to first undo (delete) all the statements created after it. However, deleting statements is not straight-forward in the demonstrational machine because the state of the machine has to be kept consistent. There are two kinds of statements in a program: instructions, and declarations. Each kind of statement creates different problems for deletion.

Deleting an arbitrary instruction makes the state of the machine inconsistent if the instruction has already been carried out. There are several ways around this problem:

- **Ignore it and allow the deletion of any instruction**

  This may make the current state inconsistent with the state that the program would produce at this point.

- **Only allow the deletion of the instruction that will be carried out next**

  The demonstrator has to manipulate the state so that the machine is just before the instruction that they want to delete.

- **Allow the deletion of any instruction that has not been carried out yet**

  This method splits the program up into to parts: the past instructions, and the future instructions. It preserves the state, but greatly restricts when statements can be deleted.

Note that the second method is a special case of the third. The last two methods do not completely solve the problem because of repetition. In either a loop or subroutine call, the next statement may have already been executed (during a previous iteration or call). An instruction may have been carried out in one call, but in another call the same instruction may not have been reached yet. If the state is to be preserved, it is not clear whether the demonstrator should be allowed to delete this instruction.

Declarations are separate from the state of the machine. At first glance, this seems to make them easier to delete. However, deleting a declaration is trickier because something has to be done about the code that references the declaration throughout the program: instructions refer to variables, variable declarations refer to types, and subroutine calls refer to subroutines. These statements depend on the declaration to work. When a declaration is deleted, it is not clear what should happen to the pieces of code that depend on it. If a declaration is not referenced, then there is no problem. Here are four possibilities for deleting a declaration and ensuring that its references are taken care of:

1. **Disallow deletion until it is unreferenced.**

   The demonstrator has to delete every piece of code that uses the declaration before they can delete it. This could be very tedious in some programs.

2. **Delete the references along with it.**

   The demonstrator can delete any declaration, and at the same time, the recorder will delete all the references to that declaration. Deleting all the references may trigger the deletion of other references, and may even cause the deletion of the entire program. For example, deleting a type will delete all the places of that type, which will delete all the assignments to those places. Implementing this method requires a way of scanning the program for these references, and should definitely check with the demonstrator before carrying out the deletion operation.

3. **Delete it and replace the references with "Not Specified".**

   The demonstrator can delete any declaration, and all the references to that declaration are replaced by a "Not Specified Declaration". This introduces another way that the program can be partially complete. A "Not Specified Declaration" is new kind of "Not Specified" instruction that performs a similar task to other "Not Specified" instructions (see section 4.1.3). When the clerk encounters a "Not Specified Declaration", the demonstrator could create a new declaration to replace the "Not Specified" one, or just delete the whole piece of code that contains it.

4. **Delete it and ask the demonstrator.**

   The demonstrator can delete any declaration, and the recorder asks what should be done about each reference: should it be deleted too, or should it be replaced with a "Not Specified Declaration". This method is a combination of the previous two methods.

All of these methods require the recorder to keep a dependency graph of all the statements in the program to determine which statements need to be updated. In addition to these issues, care must be taken to ensure that complex statements can be completely recovered if the deletion operation is undone.

5.1.2 Modification

The effect of modifying a statement can be achieved by deleting it and adding a new statement. However, this kind of modification facility is only of limited usefulness because deletion throws away too much. The prototypes allow the names of places and types to be modified, but the demonstrator should be able to modify the instructions and declarations too. Undo allows the demonstrator to fix obvious mistakes, but editing the program would allow the demonstrator to fix other mistakes more easily, *e.g.* mistakes that show up on subsequent executions.

One way to edit the program would be to allow the demonstrator to edit the code directly. We want to avoid this because in the demonstrational machine only the recorder can write code — the demonstrator just demonstrates what it should do. Editing is not straight-forward in the demonstrational machine. Consider the two kinds of program code: instructions and declarations.

To alter an instruction by demonstration, it seems reasonable that the demonstrator should wait until the instruction is being carried out, and then demonstrate an alteration to the instruction. One way to do this is to enter a mode that deletes the current instruction and replaces it with a new instruction that the demonstrator demonstrates. The trouble is that this is not really altering the old instruction. It certainly changes the instruction into another one, but it does not make any use of the components of the instruction. For example, the demonstrator may want to alter the instruction '$x := (2*y)+2$' to be '$z := (2*y)+2$', but deleting and replacing '$x := (2*y)+2$' means that the expression '$(2*y)+2$' has to be demonstrated again.

Instructions are carried out in meaningful chunks (see section 4.1.2). Thus, another option is to allow the demonstrator to interrupt the clerk and demonstrate a new chunk of code in the middle of an instruction. The demonstrator should be able to step in as the clerk completes a chunk and do something different with it. For example, take the value of '$(2*y)+2$' that the clerk has worked out and drag it to 'z' instead of 'x'. Similarly, as the clerk uses a calculator to evaluate an expression the demonstrator could stop them, and alter an operator in the expression, *e.g.* changing a '$=$' to a '$>$'. With this method, one kind of instruction could be changed into another kind of instruction by using a chunk of it differently. For example, if the demonstrator dragged '$(2*y)+2)$' to the output device, then the assignment would change to a call to `WriteNUMBER`.

This way of altering instructions has not been fully considered. The structure of chunks, which helps with "Not Specified" instructions and partial programs, may limit this way of editing to exactly the same chunks, thus preventing complete editing freedom. There are points in the execution of certain instructions beyond which they cannot be altered completely freely. For example, a subroutine call cannot easily be changed into an assignment once its second argument has been evaluated. Another thing that has not been considered is how to allow the demonstrator to modify the scope of a control structure. The current method used in the prototypes for demonstrating control structures is not amenable to modifying them in this manner, although some of the alternative methods for demonstrating control structures might be (see section 4.3.5.1).

How the demonstrator alters a declaration by demonstration depends very much on the method used for creating declarations (see sections 4.2.2.1, 4.2.3.2, and 4.2.3.3). To alter a declaration by demonstration in the second prototype, the demonstrator would work on the declaration directly. Once the declaration is altered, all the references to the declaration would have to be updated because altering the declaration is likely to invalidate most of instructions that reference it. For example, if one of the fields of a record is altered from a `NUMBER` to a `STRING`, then all the instructions that assign `NUMBER` values to that field are invalid.

The same problems that apply to deleting statements also apply to modifying them (see section 5.1.1). Once again, a dependency graph of the statements in the program should be used to determine what to update. The graph can also be used to determine how far back the program has to be unwound and re-executed in order to keep the state consistent after a modification (see section 4.3.6).

5.1.3 Better execution controls for debugging

From using the second prototype, it was clear that unwinding and executing forward by one step at a time does not give the programmer enough control of the state of the machine. In particular, it is not sufficient for debugging programs with complex expressions, recursive subroutines, or deeply nested subroutines. The demonstrator needs more control over the machine as it carries out the program.

The demonstrator needs controls similar to those on a video recorder: fast forward, rewind, and scan the program backwards and forwards. Controls like these were provided by Lieberman & Fry (1995) as part of the ZStep94 environment, and Ungar et al. (1997) argue in favour of all programming environments having these controls. Fast forward and rewind should run the program as fast as possible in the appropriate direction. Carrying out the program as fast as possible (in either direction) is useful because sometimes you just want to reach a certain state, and you do not care what happens along the way. At those times, the full animation of the execution process provided by the worksheets can hinder more than it helps. This will be discussed further in section 6.3. Scanning the program should run the program continuously at a pace that makes carrying out each chunk clear. Scanning is useful for quickly looking at the behaviour of the program without having to step though it, and for homing in on a problem. Ideally, the demonstrator would have a variable speed and direction control for carrying out the program (a kind of jogging control or direction throttle).

The demonstrator should also be able to get the clerk to stop at certain marked instructions (breakpoints). There should be two kinds of breakpoints: one that stops the execution each time the instruction is reached, and one that stops the execution only when an instructions is reached in a specific state, *e.g.* on the fifth recursive call.

5.2 Displaying the structure of instructions

The demonstration and machine models do not specify what the instructions should look like on the worksheets. The prototypes display the instructions on the worksheets using text (`Pascal` code). That is fine because we want the demonstrator to see the code and make the connection between their actions and the instructions. However, a useful thing that the appearance of the instructions should do is clearly show the structure of the flow of control. Even with good formating, text does not represent the nesting of instructions in control structures well. The structure of the code, especially alternative blocks of code in `IF` statements, is something that the demonstrator (especially a novice) should be able to see easily. A better way to display the instructions would be to use some kind of graphical layout.

There are three common kinds of graphical layout: an iconic representation (visual programming style), flowcharts, and Nassi-Shneiderman (N-S) diagrams (Nassi & Shneiderman 1990). An iconic representation is the least appealing because we want to retain the instructions as code in order to make the demonstrator aware of the link between the actions they demonstrate and the code that is recorded. For this reason, N-S diagram (or perhaps constrained flowchart) style display is preferable.

Flowcharts and N-S diagrams show the structure of the code clearly, and still have the instructions as text. We hope that this would be more beneficial to novice demonstrators than plain text. For more experienced demonstrators, the feedback that these diagrams give would be useful for seeing complex, and potentially confusing places in the code.

Hopefully, the clearer the structure is, the easier it should be to think about different cases and plan ahead[1]. However, merely using a graphic layout does not mean that the structure will automatically be easier to understand. A lot of depends on how well graphical layout shows the structure, and how well the demonstrator understands what the layout means (Petre 1995).

5.3 Worksheets: The Models versus The Reality

There is a problem with worksheets as they are used in the model — worksheets contain copies of the instructions (code) and yet they represent activation records. In an implementation of a programming language, it is unlikely that the activation records contain copies of the code — they are more likely to contain the *location* of the code. Although in the abstract machine model the worksheets all have copies of the code, in reality, only the worksheet machine contains the code. This is a discrepancy between the abstract model of the computer and the real computer. It can be seen more clearly in the demonstration model.

As it is based on the machine model, the demonstration model creates for each subroutine call a worksheet with a copy of the subroutine's code. However, two sets of instructions on the two separate worksheets are not just copies, they are the same set of instructions. If you alter one set of instructions, say by demonstrating an action, the other set of instructions changes as well. In the demonstration model, demonstrating an action during a subroutine's execution alters its definition (its template in the worksheet machine).

[1] This is purely for the demonstrator's benefit; the clerk does not plan ahead.

The demonstration model uses the recorder to work around this problem. The recorder writes new instructions on all the worksheets in order to keep them up to date with the templates in the worksheet machine (see section 3.3.2). It is hoped that having the recorder update the worksheets will smooth over this problem and help the novice understand that the templates constitute the definition while the worksheets just have to be kept up to date. However, it is not clear that it will.

An analogy that gives a more accurate description is that the instructions part of a worksheet is like a video screen or window that shows the instructions. The screen displays the instructions that are on the subroutine's template in the worksheet machine, which has the only real set of the instructions. Whenever the recorder a updates the template, the changes can be seen from the appropriate worksheets. While this analogy is more accurate, it clashes with the abstract machine model because worksheets that contain video screens are not printable.[2] A computer implementation of this analogy would avoid this clash because it has a level of magic sufficient to allow the printing of fold-able video screens on pieces of paper.

From the point of view of a expert programmer, that the activation records contain duplicate copies of the code does not seem to matter. There are two exceptions to this: if the size of the executable program is important, and if the programmer is writing self-modifying code. Neither of these should apply to novices. However, giving novices an incorrect model may cause them unforeseen problems in the future, *e.g.* when they shift to a more traditional programming environment.

5.4 Indiscriminate use of "Drag and Drop"

Many of the actions that the programmer can perform on the prototypes involve dragging an object from one place and dropping it in another. The effect of dragging and dropping an object depends on what the object is and where it is dropped. For example:

- Dragging a value to a place → puts the value in the place (*e.g.*, assignment).
- Dragging a type to a worksheet → declares a new place of that type (*e.g.*, a variable).
- Dragging a type to compound type → defines a component of a new type (*e.g.*, the type of an array's element).
- Dragging a simple value to the output device → implicitly calls and carries out the procedure that writes out the value (*e.g.*, `WriteNUMBER(13)`). A similar "short-cut" exists for the input device (see section 4.3.4).

Using "Drag and Drop" for all these actions removes the distinctions between the actions. For example, defining the type of an array's element is not the same thing as calling the `WriteNUMBER` procedure with the value `13`. The (ontological) distinction between types

[2] At least, not yet. Not doubt someone is working on it.

and values is also blurred because the same technique that is used to declare a place is also used to alter the place's value (see section 4.3.1.1). These distinctions are important, but are not likely to be obvious to novices because 'dragging and dropping' has been used indiscriminately.

A solution to this should support the distinctions more clearly. Dragging and dropping types to make new types and dragging, and dropping values to put them in places are fine, because they are performed on distinct regions of the interface: inside the worksheet machine for types (the type palette), and outside the worksheet machine for values. Dragging and dropping a type to *declare* a place is a problem because it mixes the two regions. This should be replaced by another interaction method. For example, selecting a type and using it to stamp out new places by clicking on points, like a stamp in a drawing program, or a copy and paste function. The I/O device "short-cuts" should be removed entirely, or at least relegated to proper short-cuts (see section 4.3.4.4).

5.5 Expressions: Specification versus Execution

In typical high level programming languages, the programmer has the freedom to specify complex expressions in the same way that mathematical expressions are specified — by writing them out as formulae. The formulae are translated by a compiler into a form that the computer easily can evaluate. On the other hand, the demonstration model seems to require that expression construction closely follow the way the computer will evaluate the expression. For example, the second prototype requires the programmer to construct an expression from the outside in and from left to right (see section 4.4.3). Apart from avoiding stale values, the main reason for this is that the demonstrator is taking control of the computer and operating it in the same way as the clerk operates it.

Not allowing the programmer to specify expressions by writing down a mathematical formula may be a step backwards, because it does not make use of what the programmer may already know from mathematics. Indeed, the second prototype appears to have gone back to assembly language like specification of expressions. One could claim that in doing so, it has lost the expressive power and flexibility for specifying expressions of a higher level language, which does not seem to be a good thing.[3]

Writing out a formula is fine if the programmer knows what they are doing. However, when the programmer is not so sure, it is likely to be more helpful to see the values of the expression's components and to work the expression though in the order that the computer will evaluate it (*i.e.*, from the computer's point of view). I suspect that expert programmers do something like that as they write expressions out, though they probably do not use strictly the computer's order.

[3] A related claim is that, because of the closeness of the machine model and demonstration model to the way the underlying assembly level machine works, programmers who learn to use the demonstrational machine should have little trouble writing compilers for `Pascal` (This was suggested by Ewan Tempero in a personal communication).

There appear to be two distinct ways that people approach formulae:

1. **Mathematical.**

   The mathematical way is usually used for specifying a formula in order to manipulate it; evaluation is merely something that is sometimes useful. For the mathematical way, the second prototype's method of constructing expressions is extremely annoying because restricts the specification of an expression so tightly. People who are used to thinking about formulae the mathematical way would probably prefer to be able to specify arbitrary expressions by typing them in directly.

2. **Computational.**

   The computational way is primarily concerned with evaluating the formula. A calculator, computer, or pen and paper is employed to work out the value of some formula. It may be helpful for people who use the computational way to see the values as the formula is evaluated. The specific order that a formula is evaluated in is controlled by the person carrying out the calculation, but they would always start with at least one value and apply an operation to it (working from the inside to the outside).

For either approach, restricting expression construction to "from outside in" is not helpful, except to see exactly how the computer would carry out the evaluation of the expression.

For constructing expressions that a computer is going to evaluate, there is a trade off between producing the correct expression and ease of specification. Specification and evaluation are different things, and it is not clear how much they should be combined. One possibility is to allow more freedom to specify expressions in any order, while displaying the values at each step in the calculation (see section 4.4.7).

5.6 Teaching novices

The demonstrational machine provides support for learning about and understanding programming with a particular model of a computer. It does not provide any support for teaching good programming habits, or programming design. So far, nothing has been said about using this programming environment in a course to teach novices to program. Merely giving a novice the execution model of the computer is not enough to teach them good habits and design.[4]

One large unanswered question is "How much is this model and way of programming going to help a novice?". To properly answer this question we need to carry out some experiments and evaluations with real novices. I believe that the machine model will help novices understand the computer and the way it carries out programs (see chapter

[4]This was noted by Mimi Recker in a personal communication.

2). It is not clear yet whether the way of programming supported by the demonstration model (see chapter 4) will help them.

There is a related issue that has to do with the programming paradigm that the demonstrational machine uses: the programming by demonstration model that it employs is so different from traditional programming environments that a person who learns to program in this environment might have some problems adjusting to a traditional programming method (see section 4.1). We have no idea if that person will even be able to produce code outside of the environment. Learning in this environment and then expecting a person to transfer to a traditional environment may cause them even more problems than if they had just learnt to program in the traditional way.

It is also not clear how much of the environment novices should be presented with initially. It is likely that giving them the whole environment would not be as helpful as introducing it gradually. One idea along these lines is to hide most of the environment initially, then reveal more parts of it as the novices become more competent. Once the whole environment is available, the novices could be allowed to edit first pieces of the code (*e.g.*, expressions), and then eventually the whole program. In this way, novices would learn to program using the machine and demonstration models, then gradually write programs more and more traditionally, and finally use the environment just for its debugging (unwind) and program animation facilities. This approach would support the gradual progression from novice to expert that people seem to go through (Eisenstadt et al. 1993).

Chapter 6

An Evaluation of the Prototypes

Our exploration of the prototype environments uncovered many issues for programming by demonstration in this domain. However, so far nothing has been said about how the prototypes were explored, or what they are like to use. This chapter presents limited evaluations of programming with the two prototypes of the demonstrational machine.

6.1 The kind of evaluations that were carried out

The only evaluation that has been performed on the prototypes is my own personal evaluation. I used the prototypes to write several small programs. In the process, I wrote down my thoughts and observations[1], and tried to answer the following kinds of questions:

- What was it like to program with the prototype?
- What was good or bad about it?
- What helped or hindered my programming?
- How much sense did the analogies and interaction methods make?
- What did I need that was missing?
- What other things were apparent about what I was doing?

This evaluation is fairly subjective and comes from the point of view of an expert, not a novice. A proper evaluation of the demonstrational machine will have to wait for the next version of the machine because neither of the prototypes are user-friendly or stable enough to carry out the required experimental analysis with novices and other experts.

6.2 The first prototype

The first prototype was evaluated by using it to produce programs for 10 simple tasks. The tasks required all the facets of the prototype. The tasks were introductory exercises

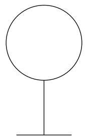
[1] I was watching what I was doing.

taken from various assignments for "Comp 102: Introduction to Programming" a CS 1 course at Victoria University of Wellington. The code for each program is contained in appendix A. A summary of my main observations is (my particular observations are contained in the remaining parts of this section):

- Being able to see the whole state of the computer as it carried out the program was immensely useful.
- The environment helps incremental development, making it easy to find mistakes on one iteration, and fix them on the next. It did not seem to matter to me that fixing a mistake made the state inconsistent (see section 4.3.6).
- This environment sorely needed unwind, deletion, and editing capabilities.
- I often positioned places relative to each other in a way that was meaningful to me in the context of the task. I noticed that I was using spatial reasoning to create various secondary (spatial) notations of my own.
- I often wanted to reuse parts of expressions that I'd already demonstrated, without having to re-demonstrate them. The problem was that I couldn't do this because the calculator was reset after each expression.
- Demonstrating programs that required nested `IF` statements was trickier than I had expected.

I also noticed that the first prototype had a tendency to slow down when the program got above a certain size (about 15 instructions). The prototype was implemented in `Tcl/Tk`, which is a string based language, and used a naive code representation for the program — the program was one long nested string. Extracting the next instruction to carry out from the program was not an efficient process. This did have the advantage that it was still easy to follow what the program was doing even when it was being run at full speed. The rest of this section presents my observations on what it was like to demonstrate each of the evaluation programs.

6.2.1 Draw a tree



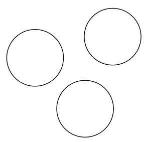
The task is to write a program that asks users for the size of a tree, and draws a simple representation of the tree on the screen.

I wrote a program that would draw a tree, scaled accordingly, but always fixed at a certain point on the screen. While writing this program, I found the pseudo mouse useful for finding coordinates on the graphics window — I would become the user and move the pseudo mouse to a point on the screen, then I would read the values in `MouseX` and `MouseY` and as the demonstrator use these values to define constants in the program, *e.g.* `GROUND`. This is not the intended use of the pseudo-mouse.

I noticed these problems while writing the program:

- The prototype needs some way of deleting the instructions. The only way to remove an incorrect instruction is to start again with a new program. This makes the prototype difficult to use, and almost impossible for large programs.
- I had to reconstruct several sub-expressions, which was quite annoying, because I didn't use variables to hold the expressions for the positions of parts of the tree. This suggests that either people will soon learn to use variables to hold values that they will need to re-use, or that the calculator should not erase an expression when you use it.
- I didn't feel that one current expression display area on the calculator was enough to get a feel for the values of each part of the expression. The current expression was great for constructing the expression in almost any order. However when checking the expression I wanted to see all the values of the important parts or components at once. For example, with the expression '$((HEIGHT * Size) + (LEAVES * Size))$' I wanted to view the value of '$(HEIGHT * Size)$' and '$(LEAVES * Size)$' at the same time.

6.2.2 Draw three circles



The task was to write a program that asks users to click the mouse on the screen three times and draws a circle at each point.

The program was straight-forward; writing it was much more pleasant than writing the "Draw a tree" program. Of course, I knew that the `LOOP` needed a counter, and what the `LOOP` condition was. One point of note was that I made a mistake during the first iteration of the `LOOP` — I forgot to carry out a `WriteLine` procedure after the first message. I was able to demonstrate an action to correct this after the message was printed for the second time. This seemed fine to me, even though it meant that the output buffer was in an inconsistent state (see section 4.3.6).

6.2.3 Output the largest of three numbers (version 1)

The task was to write a program that asks users to enter three numbers, determines which number was the largest using nest `IF` statements, and outputs that number.

Having to specify the `IF` statements in advance forced me to think hard about the outcomes of the conditions. I had to plan ahead to see where the program was going to end up when the user entered particular values. With a traditional programming environment, I'd have just coded it from the top down. In this environment, I had to go through the whole program (at least) *four* times to demonstrate all the `IF` branches correctly. I found it more difficult than I expected to set up the values each time. I needed the ability to unwind the program so that I could just alter the values while the recorder wasn't watching (see section 3.3.3.2), but the first prototype lacked unwind. I had to keep re-running the program, and entering new values. I had some trouble getting the values the right way around, which meant that a couple of times I had to step through branches that I had already demonstrated.

It didn't help that once the `IF` structures were in place, the program looked very cluttered. It was hard to see what I should be demonstrating next, *i.e.* which values I needed to enter to reach an unspecified branch. The purely text based layout did not make it easy to see the structure of the instructions.

6.2.4 Output the largest of three numbers (version 2)

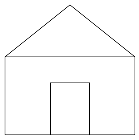
The task was to write a program that performs the same task as the previous one, but the program must use fewer `IF` statements and use `BOOLEAN` variables to store the comparison results. This program was not one of the programs that I had planned to write — I added it after my experience with the previous program. I wanted to try another approach to using `IF` statements and see if it made them easier to use.

Using three `BOOLEAN` variables to store the individual comparisons was helpful. Aside from only requiring two IF statements to write the code, the `BOOLEAN` variables allowed me to see more of what was going on in the program. Once I had declared them, I positioned them in the workspace near to the variables that held the three numbers — I put the boolean that represented the test '$num1 > num2$' between the $num1$ and $num2$ variables. I did the same for the other two booleans. I had to do a little rearranging at various points to get everything right.

I could then see at a glance which tests where related to which numbers (see figure 6.1). This made it easier to see where I was up to when demonstrating the branches of the `IF` statements. I knew which number to write out, because I could easily see the values in the booleans and their relation to the numbers. It was useful, especially as the values in the booleans were altered each time the program was run and I could see them. In the previous version, I could not see the values of the expressions after they had been evaluated; they were part of the `IF` statements, and their values are discarded after being used.

However, I still felt that writing this version of the program was as tricky as writing the previous version. This program had only one less branch because it had only one less `IF` statement. Once again, having to think which way around the values needed to be, and being unable to unwind made it difficult. Forcing a novice to do this sort of thinking may not be a bad thing though.

6.2.5 Draw a house



The task was to write a program that draws a simple house with a door and a roof.

Again while writing this program, I noticed that I wanted to reuse and alter previous expressions. It was annoying to construct the same sub-expression to use it in a different

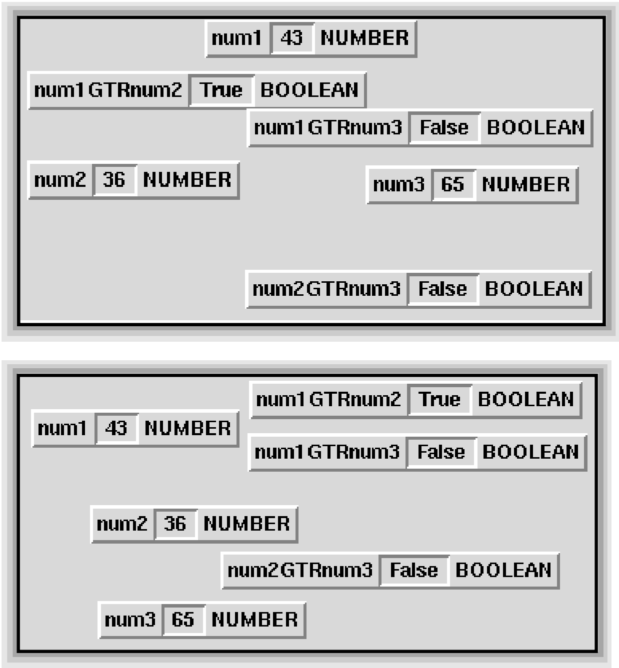


Figure 6.1: The spatial relationships between the numbers and the condition values at different points in the program.

place in another expression. I should have used variables, but I didn't because I was making it up as I went along.[2] Once an expression is used, there isn't an easy way to replace the expression with a variable that holds the expression's value.

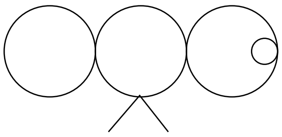
On a positive note, using lots of constants made it simple to resize the house to get the proportions looking right. The constants could be changed and the program re-run with the new values, *e.g.* `WIDTH` could be changed from `100` to `200` in order to lengthen the house.

6.2.6 Multiply two numbers

The task is to write a program that asks the user to enter two numbers, and then writes out the product of those numbers.

This program was straight-forward. I found that the prototype makes input easy because it shows the text that has been entered *but* not read.

6.2.7 Draw an insect



The task is to write a program that asks users for the size of the insect, asks them to click the mouse at the point where they would like the insect's head drawn, and then draws a simple insect.

The program that I constructed only drew an extremely simple insect: three body segments, two legs, and one eye. Even though I used variables to hold the values of some

[2]A dangerous observation in itself.

of the expressions, I still wanted a way to reuse expressions. The fact that the program was always running made incremental construction of the insect easy. However, I had to restart several times because I made a mistake and could not delete the incorrect instruction.

6.2.8 Calculate the properties of a number

The task is to write a program that asks the user to enter a positive number, and then displays some properties of that number: the square of the number, half the number, and whether it was even or odd.

This was straight-forward. However, often when I output a string to tell the user something, I would demonstrate a call to `WriteLine` directly after it. What I really wanted to do was to define my own subroutines, *e.g.* a procedure that took a `STRING`, a `NUMBER`, and output them both followed by a newline.

6.2.9 Click in the button

This is the same task as the example program in section 3.5. The task is to write a program that asks the user to enter a message, draws a simple representation of a button on the screen, puts the message in the button, and then asks them to click the mouse on the screen. If the user clicks outside the button, then the program should ask the user to click again. If their click is inside the button, then the program should congratulate them and stop.

Again, I used constants to make the proportions of an object I was drawing (the button) easy to change, and again I found that the environment makes prototyping or incremental development of small programs easy.

I found myself positioning places in a meaningful spatial way relative to each. I placed the constants in such a way that the coordinates of the button's corners were together (see figure 6.2). The first two constants were the x and y coordinates of the top left corner of the button, so I put them at the top left. The second two were the bottom right x and y, so they went at the bottom right.

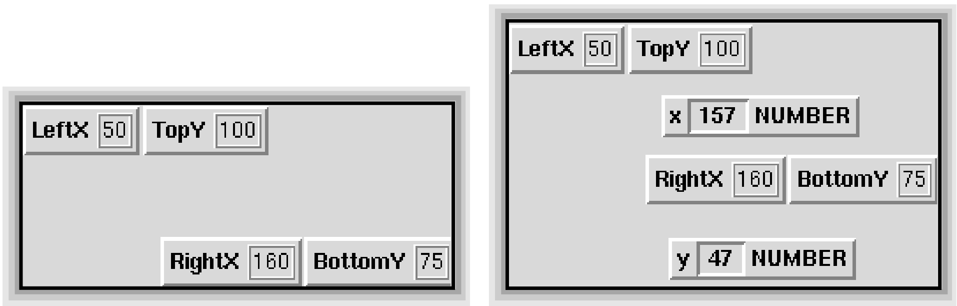


Figure 6.2: The constants for the button's corners, and the point coordinates positioned relative to each other.

I used the relative locations of the constants to keep track of the one that held the coordinate I needed for whichever expression that I was working on. When it came time to compare the corners with the mouse coordinates, I positioned the '`x`' place in the center of the "corners" to help myself think about the expression that I needed (see figure 6.2). This made constructing the complex comparison expression easier.

6.2.10 Read in a line and print it out, do this 10 times.

The task was to write a program that asks the user to enter a message, and then prints that message ten times.

This was straight-forward.

6.2.11 Evaluation summary of the first prototype

The majority of the ten evaluation tasks were straight-forward to construct programs with the prototype. The prototype made it easy to see mistakes because the state was always displayed and the program was built by carrying it out. The programs that used iteration were particularly helped by the environment because a mistake could be found on one iteration and fixed on the next.

The task that required nested `IF` statements was the most difficult to write (see section 6.2.3). Trying to demonstrate a program that works out the largest of three values did not feel nice (without unwind). Trying to code the same task using boolean variables (see section 6.2.4) led to the insight that I was using the spatial relationships between variable places to organise my thoughts about the programs. Further tasks uncovered further examples of spatial reasoning on my part (see sections 6.2.9). There seem to be two kinds of situations where I used the positions of the place to help me think about the problem: when comparing values, and when working with coordinates for drawing.

The lack of unwind, deletion, and editing capabilities hinders programming — if you make one mistake, you have to begin again. The second prototype went some way to addressing this by adding the unwind and undo operations. As for wanting to reuse parts of expressions, I think that part of the problem was that the first prototype cleared out the calculator after an expression was used. If the expression had just been left in the calculator, then it could be used in a later instruction. This assumes that the appropriate safe-guards are taken against stale values.

6.3 The second prototype

The second prototype was limited in the kinds of programs that it could be used to produce because the I/O devices and calculators were not fully implemented. The second prototype was not evaluated with the same tasks as the first prototype. It was evaluated with several programs that required programmer-defined subroutines and/or programmer-defined types (`ARRAY` based ones). The prototype was evaluated primarily on the `RecursiveFactorial` program (see section 3.7), and array manipulation programs such as reversing a list and selection sort (see section 3.8). The main things that I noticed were:

- The ability to undo actions helped correct the simple mistakes. However, the environment still needed editing and deletion, just not as much as the first prototype.
- Constructing complex expressions was cumbersome when I could only invoke functions in the order that they would be carried out (see section 4.4.5). I wanted to be able to construct expressions with a calculator like the first prototype's calculator.
- The comparator seemed to be quite useful, but I really wanted to see how it would work with the other calculators (which weren't implemented).
- The ability to unwind the state helped enormously, especially for programs with nested `IF` statements. The "Output the largest of three numbers" program was less of a problem to produce with this prototype than it was with the first prototype (see section 6.2.3).
- Worksheets worked well for displaying the state of the program and the invocation, filling in parameters, and execution processes.
- The prototype was quite slow to carry out an action. The demonstrator often had to wait a couple of seconds for the prototype to finish updating everything.
- With more than about ten worksheets on the clipboard at once, the prototype would slow down even further. Stepping forwards and backwards through a program become an intolerable process because of the amount of time it took to update the clipboard.
- I wanted to be able to set breakpoints, and run the program forwards and backward at full speed.

Overall, the second prototype was not as pleasant to use as the first prototype. This was surprising given that the second prototype had full unwind and undo operations, and supported the machine model (see chapter 2) better than the first prototype. It is not clear whether the second prototype is not pleasant to use because it is slow, because of its inconsistencies (see all the "issues" sections in chapter 4), because of its poor user interface, or because the way of programming it supports is too cumbersome. The fact that the prototype is so slow makes it difficult to establish whether the machine model and the demonstration model are any good for programming.

The prototype was slow due to some of the short comings in its implementation. The prototype was implemented in a combination of `iTcl/iTk` and `C++` — `iTcl/iTk` for the interface, and `C++` for the state engine. There were three main problems with its implementation: inefficient communication between the user interface and the state engine (command encoding and decoding), an inefficient display algorithm that naively updated every part of the machine after each action, and the number of objects that had to be kept consistent between the interface and the state engine. The prototype's implementation will be discussed in more detail in chapter 7.

I think that the lack of speed is the primary reason why this prototype is not pleasant to use. Certainly, this and the issues of user interface usability need to be addressed before the models and way of programming can be properly evaluated.

6.4 Secondary spatial notation

While neither prototype has any way for the demonstrator to comment their programs, both prototypes allow the demonstrator to position places on the worksheets, and to move them around. However, neither prototype provides any more support for spatial reasoning or any secondary spatial notation. It may be helpful to allow some secondary spatial notations such as lines, arrows, and doodling in future implementations of the demonstrational machine,[3] though these may also cause more problems than they are worth if they confuse novices by detracting from the model (Petre 1995). I suspect experts may find it more useful to spatially annotate their programs, perhaps turning the demonstrational machine into a workbench in a similar vein to Pygmalion (Smith 1990a, Smith 1993).

[3] This suggestion was made by Alan Blackwell.

Chapter 7

Implementation Issues

The prototypes of the demonstrational machine are complex pieces of software and implementing them was not straight-forward. This chapter describes how the prototypes were implemented, and the some of the issues that were encountered during their implementation. The purpose of this chapter is two-fold: to briefly describe the implementation of the prototypes (especially the second one), and to inform those who want to implement this kind of system of some problems and pitfalls to watch out for.

7.1 The first prototype

The first prototype was implemented in `Tcl/Tk`, a combination of a string-based interpreted language (`Tcl`) and a graphical toolkit (`Tk`, Ousterhout (1994)). It was intended as a throw-away prototype that would cover only a subset of the language (see section 3.3.5). As such, it was simply built rather than designed. The prototype was about 5800 lines of code. It ran reasonably quickly as long as the program was small. Most of the first prototype's implementation was straight-forward, except for the single large calculator (see section 4.4.2). It did not have to deal with the issues that complicated the second prototype, such as unwind and the worksheet stack.

The calculator stored an expression as a tree, and allowed the expression to be edited in two ways: by replacing parts of the expression, and by wrapping functions around a part of the expression. The calculator also enabled the demonstrator to select different parts of the expression by traversing up and down the expression tree (see section 4.4.2). The algorithm that I used to traverse the expression tree had to keep track of the current node, the previously visited node, the direction travelled from the previous node, and the new direction. It was not a straight-forward tree traversal.

7.2 The second prototype

The second prototype was implemented in a mixture of `iTcl/iTk` and `C++`. `iTcl/iTk` is a version of `Tcl/Tk` which has been extended to provide some support for object-oriented programming. `iTcl/iTk` is a string-based interpreted command language like `Tcl/Tk`; the `C++` program sends and receives strings from the `iTcl/iTk` interpreter (in fact, the `C++` code is compiled into the `iTcl/iTk` interpreter). The implementation consisted of 9200 lines of `iTcl/iTk` code and 14800 lines of `C++` code.

The second prototype was intended to test as much of the machine and demonstration models as possible. While it was still a prototype system, some effort[1] was put into designing it well. The prototype was built with object-oriented programming techniques and several design patterns described in (Gamma, Helm, Johnson & Vlissides 1994). The design was heavily influenced by the requirement to allow unlimited unwind and undo in a programming by demonstration environment.

7.2.1 Structure of the second prototype

The second prototype's implementation is divided into two parts (see figure 7.1): the user interface, and the state engine.

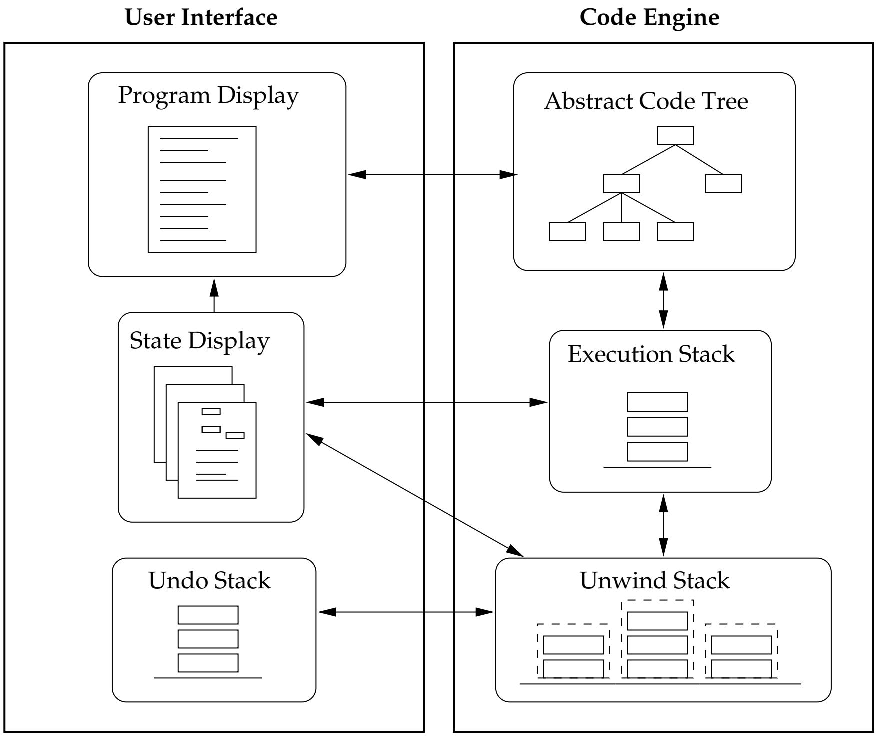


Figure 7.1: The parts of the prototype.

The user interface was responsible for getting the demonstrator's commands, keeping track of the undo stack, and displaying the state of the program and code. The state engine was responsible for the state of the code, the state of the execution stack, executing the code, and storing previous execution states for unwind. Each component of the prototype was responsible for a specific function:

[1] Nowhere near enough.

- Abstract Code Tree — stored the program code.
- Program Display — displayed the program code.
- Execution Stack — stored the state of the program.
- State Display — displayed the state of the program.
- Unwind Stack — stored previous states of the program.
- Undo Stack — stored all the programmer's (demonstrator's) commands.

The user interface was written in `iTcl/iTk` because it required graphical objects. The state engine was written in `C++` because it required processing large and complex data structures.

7.2.2 Abstract Code Tree

The program code was separated from the execution state to make is easy to store previous execution states (necessary for unwind and undo). The program was stored in a complex data structure called the Abstract Code Tree (ACT). Each `Pascal` statement was a node in the tree, as was each of the statement's sub-components that would be carried out in more than one step. Each node had a number of children which corresponded to the sub-components of the statement (see figure 7.2). For example, an assignment statement node has two child nodes: one for the code to determine the place (a variable reference), and one for code to work out the value (an expression of some sort).

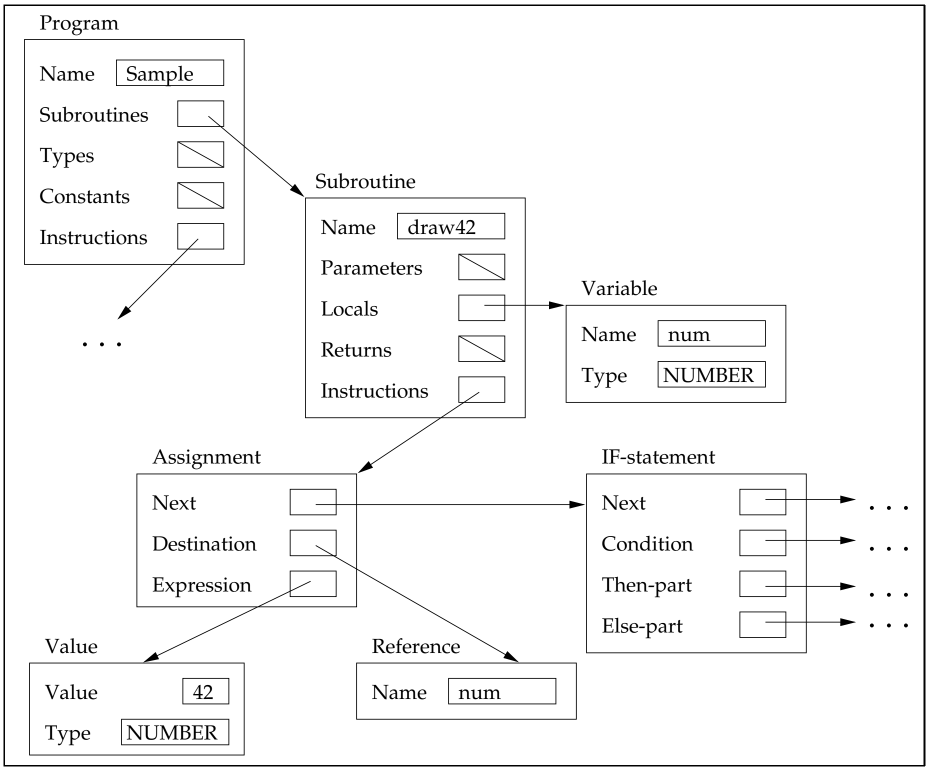


Figure 7.2: Part of a sample code tree.

The abstract code tree was implemented in `C++`. Each class of node in the tree was inherited from the same base class. Although the nodes in the code tree represent particular statements and pieces of code in the program, they are not stored in a language specific way. The nodes in the abstract code tree did nothing other than record the details of the code. None of the nodes have any kind of display subroutines at all, displaying the code was part of the interface.

7.2.3 Program display

The program was separated from the way it was displayed in order to make it easier to display the program in a variety of ways. We wanted to be able to display the program using the syntax of several traditional programming languages. This was to support our bias of focusing the novice on the ontology and semantics of the programming language, rather than the syntax (see section 2.1). Even though the language was `Pascal`-based, it is really representative of a class of languages with the same core ontology (see section 2.6). The display algorithm for the second prototype was designed to allow new ways of displaying the program to be added with reasonable ease.

The obvious way to display a program represented by an abstract code tree is to write a display method for each class of node. Starting at the root, each node would display itself and call the display methods of its children. Thus, the whole program would be displayed appropriately. The trouble with this method is that there are many different classes of nodes in the tree, and to add a new way of displaying the program means adding a new display method to *each* class. A better approach is to encapsulate the thing that you want to change, and separate it from the things that depend on it, or use it. In this case, that thing is the way of displaying the program.

The display algorithm used in the second prototype encapsulated the way of displaying the program into a class, and made use of two design patterns: the Visitor, and the Strategy. A Visitor is a variation on an iterator. Its job is to traverse a data structure and carry out some operation on each node (in this case displaying the node). A Strategy is a family of encapsulated interchangeable algorithms that some other objects use to carry out a task, rather than implement the task themselves (in this case displaying the program). Combining these two patterns allows the code to be displayed in a way that is independent of the code itself. It also makes it straight-forward to alter the syntax of the code — apparently changing the language that the program is written in — while still keeping the same program underneath.

In order to carry out its operation, a Visitor has to know that there are a variety of different kinds of node in data structure, how to traverse from one node to another, and what information it can expect in each kind of node. A properly implemented Visitor can traverse the data structure without breaking the encapsulation of the various node classes by using double dispatch[2] (or a simple implementation of it for languages without it, i.e. `iTcl/iTk` and `C++`, see figure 7.3).

in a sample Visitor class

```
...
VisitNode (Node * node)
{ node → AcceptVisitor(this); }

VisitAssignment (Assignment * assign)
{ ...
   printf(":=");
   value = assign → GetValueCode();
   VisitNode(value);
   printf(";");
   ...
}
...
```

in a sample Node class (an assignment)

```
...
AcceptVisitor (Visitor * visitor)
{ visitor → VisitAssignment(this); }
...
```

Figure 7.3: The interaction between Visitor and Node.

The Visitor implements the display operations for the various nodes. It works out what kind of node it is up to (*e.g.*, an assignment statement), queries the node for the information it requires (*e.g.*, the destination reference and the value expression), and displays it in the appropriate way (*e.g.*, $num := 42;$ for `Pascal`). An abstract base class, `DisplayVisitor`, was created to define the interface for the various Visitor classes that were planned, but only a `PascalDisplayVisitor` was actually implemented for the second prototype.

A range of classes inherited from `DisplayVisitor` could be implemented to display the program in any suitable language or format, *e.g.* `C`, `C++`, Assembler, `Java`, Nassi-Shneiderman diagrams (see figure 7.4). The major limiting factor is the kinds of language constructs (ontology) that the underlying code tree contains. For example, though we could display the program in `C++` syntax, it would not contain any classes.

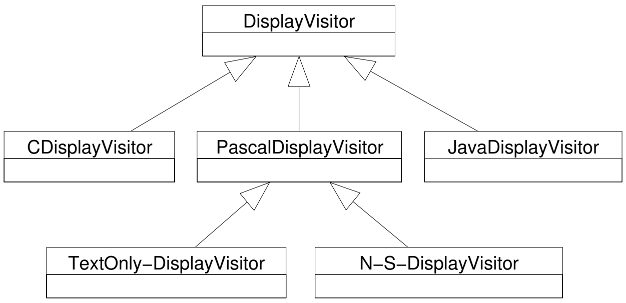


Figure 7.4: A family of Visitor classes.

[2] A special case of multiple dispatch.

The `DisplayVisitor` and its subclasses are used as a Strategy pattern in the second prototype. Program code is displayed at two points in the second prototype's interface: the program window, which displays the entire program, and the worksheets, which display their own subroutine's code. Using the Strategy design pattern, the program window and each worksheet contains a reference to a `DisplayVisitor` derived object that they use to display code, *e.g.* a `PascalDisplayVisitor`. When the program window or worksheet wants to display a piece of code, they pass that code to off to the `DisplayVisitor`, and it displays the code appropriately. By altering the particular `DisplayVisitor` that the program window and worksheets use, the program can be displayed in different formats as the prototype is running.

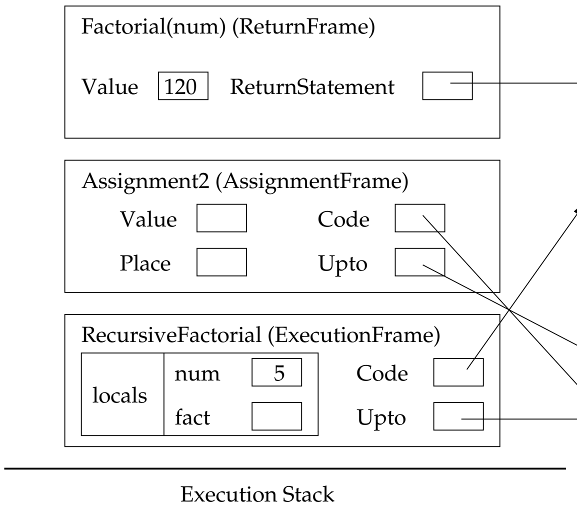
The `DisplayVisitor` and its subclasses are part of the interface and as such were implemented in `iTcl/iTk`. This compounded some of the second prototype's efficiency problems that will be discussed in section 7.3. Further information on the advantages *and* disadvantages of these design patterns can be found in (Gamma et al. 1994).

7.2.4 Storing the program state: frames

The state of the program's execution is stored as a stack of 'frames' that models a runtime stack. Each 'frame' is similar to, but not identical to, an activation record in a typical runtime stack. In a typical runtime stack, there would usually be one activation record per subroutine invocation, here there is one frame per compound statement because the program is executed in small meaningful chunks. In this context, a compound statement is one that contains code that requires at least two execution steps to be carried out. An execution step is the smallest chunk of code that is carried out in one step during the execution of the program (see section 4.1.2).

There are several kinds of frames, one for each compound statement: "invocation frames", "execution frames", "return frames", which correspond to carrying out subroutines during their worksheets' folded, unfolded, and re-folded stages, "assignment frames" for carrying out assignments, "if frames" for evaluating conditionals, "reference frames" for finding a variable that is being referenced, and special "calculator frames" for the comparator (see sections 3.7.7 and 4.4.4). Invocation, execution, and return frames are separate frames, rather than stages of a single frame, because this reduces the amount that needs to be stored as history for any point in a subroutine's execution. There are no frames for loops because the `LOOP` statement has no condition associated with it.

Each frame stores a pointer to the piece of code it represents, and stores information about the values and stages involved in carrying out that piece of code (see figure 7.5). Each frame contains its own "position counter", which is a localised "program" counter that records how far the frame's associated code statement is complete. For example, an invocation frame has a position counter that records how many of the subroutine's parameters have been evaluated. The state was recorded this way, rather than using a more traditional activation stack, because breaking the activation records up into smaller frames made it more efficient to store the previous states on the unwind stack. This will be described in more detail in the next subsection.




```
PROGRAM RecursiveFactorial;

FUNCTION Factorial(n:NUMBER):NUMBER;
 BEGIN
  IF(n = 1) THEN
   RETURN 1;
  ELSE
   RETURN (n*Factorial(n-1));
  ENDIF;
 END;

VAR
 num:NUMBER;
 fact:NUMBER;

BEGIN
 num:=5;
 WriteNUMBER(num);
 WriteSTRING(' factorial is ');

 fact:= Factorial(num);

 WriteNUMBER(fact);
 WriteLINE();
END.
```

Figure 7.5: The frame stack near the end of the `RecursiveFactorial` program.

For example, an assignment is carried out like this (see figure 7.6): an assignment instruction is reached, an new assignment frame is created, the value to assign is worked out, the destination is worked out, the assignment is carried out and the assignment frame is removed. Working out the value to assign and the destination can sometimes take several steps, if either contains an expression. For example, the value part of the assignment '`fact := Factorial(num);`' contains the expression '$Factorial(num)$', which will take several steps to work out.

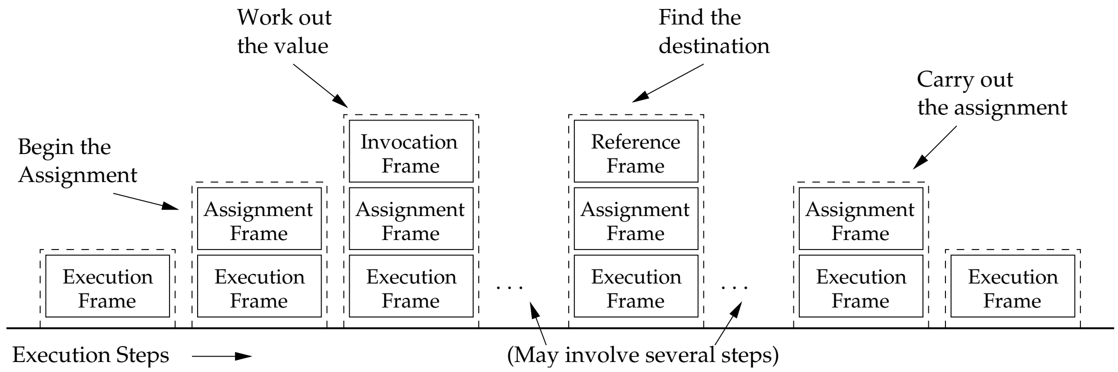


Figure 7.6: The steps in carrying out an assignment.

All three stages in carrying out a subroutine are represented by the same worksheet in the user interface. This creates some problems for keeping the display synchronised with the state. The state contains three separate objects that refer to a single worksheet. It would have been easy to coordinate the state and display if each frame had a corresponding graphical object, but that would have created more computation in the interface to add, remove, and store three worksheet objects that looked almost the same.

The code and the state are separate and each consists of many parts. Most parts of the code and state have graphical counterparts in the interface. All the state information is kept in the code engine (`C++`), and the user interface (`iTcl/iTk`) has to be updated constantly to keep it displaying consistent information.

7.2.5 Storing previous states for unwind

The previous states of the program are stored on the unwind stack, which is a stack of states. The previous state of the program includes the state of the the typing buffer, input buffer, output window, and graphics window as well as the execution stack. Before any execution step is carried out, the execution stack is copied, it is put onto the unwind stack, and the copy is used in its place.

The naive approach would be to copy the entire execution stack. This wastes a lot of space because most of the execution stack doesn't change between execution steps. The second prototype uses a reference counting mechanism to reduce the amount of data duplicated at every step. Instead of copying the entire state, only frames that might change as a result of the next execution step are copied. The frames that stay the same between the two steps are "copied" by taking another reference to them, *e.g.* the mainline of the program when the computer is in the middle of another subroutine (see figure 7.7). However, because of the demonstrational way of programming, the frames that *might* change during one execution step were all the frames down to and including the topmost "execution frame" (unfolded subroutine call) because of the way certain statements were demonstrated, *e.g.* assignment. Why this is so will be described later, in section 7.4.2.

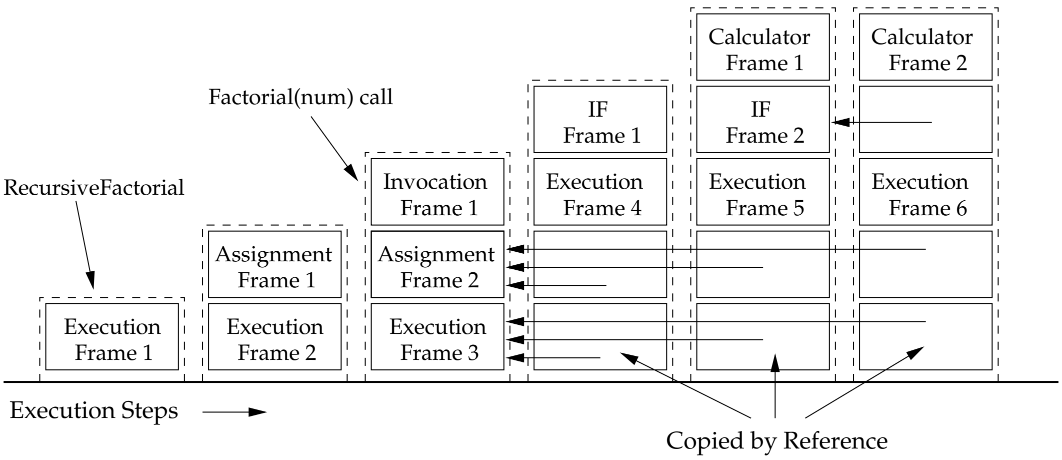


Figure 7.7: "Copying" frames by taking references during part of an assignment.

Most of the time, the rest of the frames did not change. So ideally, only the frames that *will* actually change need to be copied. However, the second prototype stored more than it needed to. This was done to simplify the implementation and to ensure that everything that might change was duplicated.[3]

To unwind the program to the previous state, the current program state is discarded and replaced by the top state on the unwind stack. The display is then updated to reflect the new state. The worksheets on the clipboard were updated rather than restored from an unwind stack of their own because of the size of graphical objects.

7.2.6 Implementing undo

The second prototype implemented unlimited undo and redo using the Command design pattern (see Gamma et al. (1994) for details) to make each command completely reversible. Each kind of command was turned into a class that could be initialised with the appropriate values for a particular action, *e.g.* each assignment command needs a destination, and a source for the assigned value (the destination's previous value is not stored in the command because it will be stored in the previous state on the unwind stack). Any commands that were carried out were put onto the undo stack, including undo and redo commands. A command was undone by carrying out its reverse. The undo stack was part of the interface because that was where the commands were being given.

Making each command seamlessly reversible is difficult on a system of this size, especially for commands involving the execution of code (and hence a change to the program state). However, this must be done to make sure that the system's state does not become inconsistent during undo and redo operations. This is particularly tricky and is the cause of most of the bugs in the second prototype. The commands need to be designed carefully to ensure that each can be seamlessly reversed without any side effects. Designing all the commands up front would also allow for greater code reuse than I was able to achieve with the "incremental" design of the second prototype.

7.3 Speed of the second prototype

The second prototype was slow. The response time for some operations made it irritating to use (see section 6.3). There are two major contributing factors for this lack of speed:

1. **Inefficient communication between the interface and engine (the language barrier).**

   The `iTcl/iTk` and `C++` components of the prototype communicated with each other using commands made from strings because `iTcl/iTk` is string-based. When the `iTcl/iTk` interpreter sent a command to the `C++` engine, the string would be decoded to find the function and objects. The decoding was done by splitting up the

[3]Better to be safe than sorry.

string and hashing its parts. The matching `C++` function would be carried out, and the result converted into a string and returned to the `iTcl/iTk` interpreter. Also, when the `C++` engine sent a command to the `iTcl/iTk` interpreter, the command would have to be encoded into a string, and the result decoded from one.

All of this resulted in a lot of encoding, decoding, splitting, and concatenation of strings, which reduced the performance of the system.

2. **Inefficiencies in updating the display.**

   The display of the interface is updated after every action. There are two inefficiencies in the second prototype's display routines:

   - The *entire* display, the code and the state, is updated after every command. Every item was checked, even if it hadn't changed. Worksheets and other graphical entities were not deleted and redrawn unless they had changed. However, the entire program was redisplayed when a new statement was added, instead of just inserting the new statement at the appropriate point. This update scheme is inappropriate for a usable system, but more acceptable for a research prototype that was subject to constant change.
   - The implementation of the Visitor strategy compounded the language barrier problem because it required a large number of commands to be sent between the `iTcl/iTk` interpreter and the `C++` engine. The abstract code tree was implemented in `C++` because `C++` was much better for implementing large data structures, but the display algorithm was implemented in `iTcl/iTk` because that was where the display would happen. This meant there are an awful lot of passing strings back and forth between the two languages. This was a mistake. Either the DisplayVisitor classes should have been implemented in `C++` and sent one big update command to the `iTcl/iTk` interpreter, or the code display algorithms should have been implemented to insert new code statements without redisplaying the whole program (see above item), and preferably both.

There are obvious solutions to these problems: use one language or at least minimise the command encoding/decoding, only update objects that change, and use an implementation of the Observer design pattern described in (Gamma et al. 1994).

7.4 Unwind related problems

There were two main problems related to unwind: ensuring that a new variable is created in every frame that it should be in, and splicing new frames into the middle of the program's execution stack. These problems also apply to undo because undo can cause the state to be unwound. These problems are both consequences of the fact that demonstrating an action can affect previous states of the program. This comes about because there is an inconsistency between carrying out an instruction by demonstration, and carrying it out as an interpreter would (in the machine model of the computer) — the demonstrator does not communicate actions to the recorder in the same order that the clerk carries the actions out from the instructions. These problems are examples of the kinds of thing to watch out for when implementing a system like this.

7.4.1 New variables and the "missing variable" syndrome

A difference between when the *computer* can declare variables and when the *demonstrator* can declare them is the cause of a problem related to the declaration of new variables. The computer only declares variables when a subroutine is first called, i.e. when it's worksheet is first printed. However, the demonstrator can declare a new variable during the subroutine's execution (though not during its invocation).

Declaring a variable causes the declaration code to be created, and affects all the execution frames associated with the current subroutine, each of which may lie at any point in the program stack and the unwind stack. The new variable must be added to all these states, to avoid the "missing variable" syndrome. For example, if the demonstrator declares a variable at some point in the middle of carrying out a subroutine, uses that variable in an assignment, then *unwinds* to a state *prior* to the declaration of the variable, and tries to execute forward over the assignment involving that variable, then the system may crash because the variable may not exist in the current frame for the subroutine's state (see figure 7.8). Unwinding the state changes the program stack, but does not change the program's code (see section 3.3.3.2).

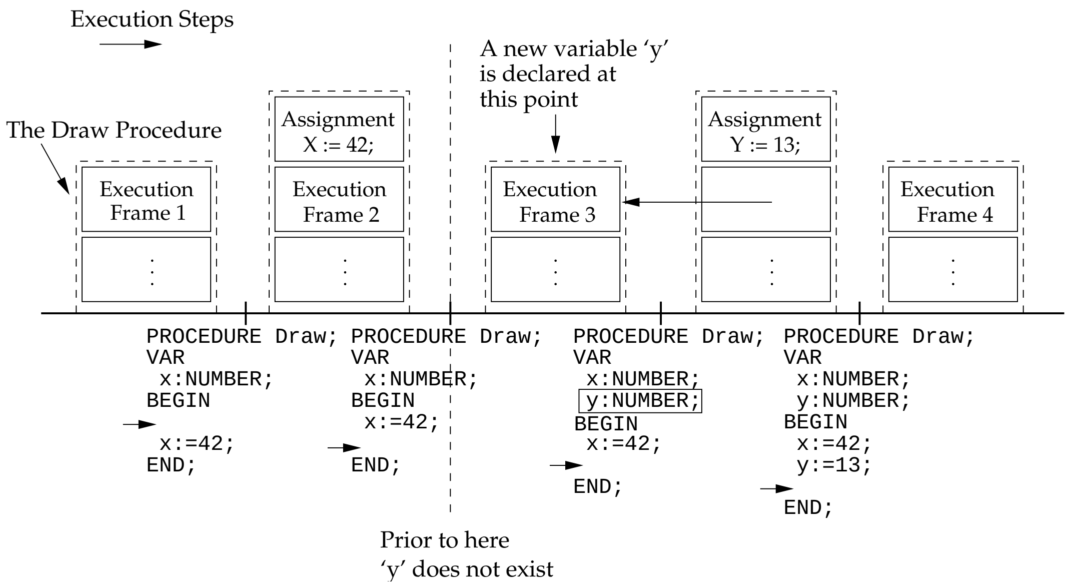


Figure 7.8: An example of how the missing variable syndrome is setup.

In the second prototype, carrying out those actions will cause the system to crash because the prototype only creates new variables in the current frame associated with the subroutine. A newly declared variable will not exist in the copies of the frame that are already on the unwind stack, or those that are further down the execution stack. When the demonstrator unwinds, the variable creases to exist (the graphical variable place also disappears, though the code does not change), and on stepping forward the system reaches the assignment to a now non-existent variable. Similarly, if the program is executed until the subroutine is completed and returns to an earlier call to the same subroutine, then the system may reach the same assignment to a non-existent variable. There are at least three solutions to this problem:

- Restrict variable declaration to the first time a subroutine is entered. This is much too constraining.
- When a new variable is declared, the system should go back through all the previous states and create the new variable in all the previous calls (including recursive ones). Effectively the system rewrites history so that the variable always existed in the previous frames for the subroutine. For a large number of previous states, this will require a lot of work.
- Whenever a state encounters a reference to a variable that doesn't exist, a new variable should be created immediately. The subroutine's definition can be consulted to find the appropriate type of place to create because it contains the variable declaration. This will not introduce any discrepancies because only variables that have actually been declared (in the future) can be recreated (in the past), and because it is a just-in-time solution it will require much less computation that the previous option.

The last solution is the most preferable.

7.4.2 Frame splicing

Frame splicing is inserting a new frame into the execution stack in order to carry out some instruction. Frame splicing is necessary in the second prototype because of a subtle difference between demonstrating an instruction and the clerk carrying it out. The *computer* knows which instruction it is in the process of carrying out before it calculates any values required for the instruction. However, in some cases, the *demonstrator* calculates the values required before carrying out the instruction because the value has to be calculated in order to demonstrate with it.

For example, when the clerk begins to carry out an assignment statement such as '$fact := Factorial(num)$', the clerk knows that it is about to carry out an assignment. Thus when the clerk is calculating the value of '$Factorial(num)$', an assignment frame is on the stack (see figure 7.5). However, for the demonstrator to demonstrate the assignment, '$Factorial(num)$' must be calculated first, so that its value can be dragged to a place for the demonstration of the assignment (see section 3.7.14). In effect, the demonstrator gets the value and the place before deciding to carry out an assignment. Whereas the clerk knows that it has to carry out an assignment, so it goes and gets the value and the place. This is significant in the second prototype because of the way the execution stack was built up of frames.

The particular demonstrations where this occurs are: `IF` statements, `RETURN` statements, assignment statements, and direct demonstrations of text output (*e.g.*, `WriteNUMBER`). Demonstrating these instructions requires a new frame to be spliced into the program stack so that the instruction can be properly executed. A frame for the instruction is inserted underneath the frame that holds the value that was used in the demonstration. Splicing a frame into the program stack can mess up the frames stored on the unwind stack, unless those frames have been properly duplicated. The value's frame is altered during the splice and because of the reference counting, this can affect several "versions" of the frame (see figure 7.9). A typical alteration is that the value has now become a "parameter" of the instruction that the newly spliced frame corresponds to, *e.g.* the assignment frame. If the state is messed up, then the program state cannot be unwound past the point of the splice.

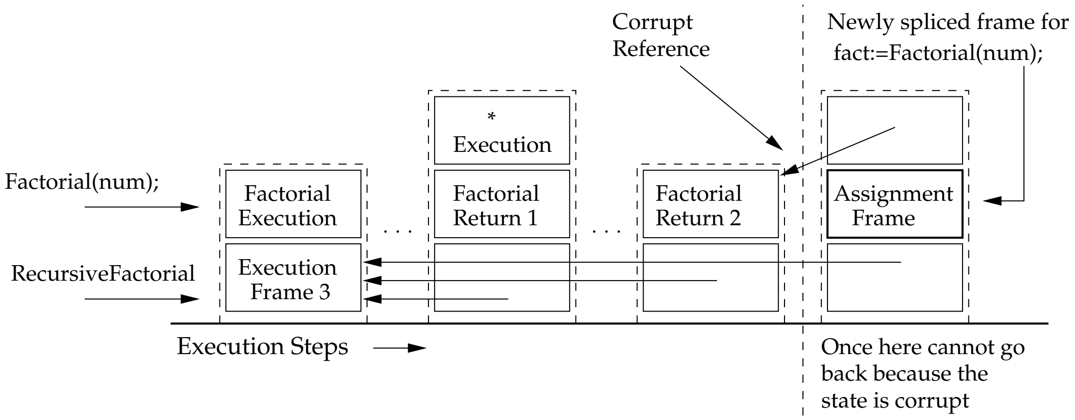


Figure 7.9: A frame splice for an assignment.

The problem is that you cannot know in advance when a frame needs to be duplicated because you cannot predict what the demonstrator will do. Fortunately, this problem is easily avoided if the previous states are properly stored. There are at least three way of doing this:

- Duplicate the entire state before each execution step. This requires a lot of space.
- Copy any part of the state that could be changed by one action. This requires less space, but does duplicate more than might be necessary. The second prototype uses this method (see section 7.2.5).
- Use more intelligent frame proxies instead of simple references to frames. The proxies should filter all requests for a frame, and whenever a request that will alter the data in a frame is received, the proxies duplicate the frame and returns the duplicate instead of the actual frame. If the request will not alter the frame, then

the proxy can happily return a reference to the frame. This uses the minimum amount of space required to store the previous states because a frame will only be copied when it changes.

The last method is the most efficient and safest way of implementing the unwind stack.

Chapter 8

Conclusions

This thesis presented an exploration of programming by demonstration in the domain of novice programming. The high level goal was to create and explore a programming by demonstration environment that would support novice programmers as they learned to program in an imperative language. An important contribution of the thesis is to throw light on the issues, distinctions, and trade-offs that were discovered during this exploration of programming by demonstration. The work was built up in three layers: the abstract machine model, the demonstration model, and the environment itself.

The aim of the abstract machine model is to provide a set of analogies that will help novices scaffold from what they already know to a better understanding of programming. The model describes what a computer does when it carries out the instructions in a program. The computer in the model is based on a tidied up `Pascal` abstract machine. The model of computation is general enough to be representative of the general class of non object-oriented, non parallel, imperative languages. The abstract machine model provides the basis for the demonstration model, the demonstrational machine, and the way of programming that they support.

The demonstration model consists of the demonstrational machine and the way of programming supported by the machine. The demonstrational machine is a theoretical computer based on the abstract machine with two extra roles added to enable programming by demonstration: the demonstrator and the recorder. The model describes how the demonstrator can demonstrate programs using the demonstrational machine. The demonstrational machine provides the basis for the novices' programming environment. Two prototypes of this environment were built: the first implemented only a small part of the abstract machine and demonstration models, the second was more complete. Both prototypes were used to explore the models and various techniques and issues for programming by demonstration in this domain.

The conclusions and contributions described in the rest of this chapter are split into four sections: the abstract machine model, the demonstration model, the prototype environments, and the other issues. The final section in the chapter discusses some areas for future work.

8.1 The Abstract Machine Model

The abstract machine model was a model of computation that was designed to provide a clear picture of what a computer does when it carries out a program. It brought together a whole collection of usually separate analogies for the way that parts of the computer work when carrying out instructions, and combined them into a coherent and reasonably consistent model.

The model personifies the flow of control in the computer as a clerk working in a room. The activation stack is the clerk's clipboard and each subroutine activation record is represented by a worksheet stacked on the clipboard. The stages in carrying out a subroutine are clearly distinguished by the way a worksheet is folded. Invocation is represented by the printing of a new folded worksheet that only shows its parameters. The subroutine's execution is represented by the clerk unfolding the worksheet and carrying out the instructions on it. Returning from a subroutine is represented by the clerk re-folding the worksheet so that only the return value (if any) is visible.

The model had to trade-off between different analogies for places. One analogy was less accurate for describing places, but more compatible with the rest of the analogies in the model. The other analogy was more accurate, but less compatible. The model used the less accurate analogy because it *was* more compatible with the rest of the model. The consistency of the model as a whole was traded off against the accuracy of one of its component analogies. As a result, the abstract machine model does not provide a good analogy for the way arrays are accessed by the computer. However, the more accurate analogy does provide a good analogy for arrays, and that analogy was used in the prototypes.

The abstract machine model with its coherent collection of analogies is one contribution of this thesis. It has been used in the lectures of two offerings of Comp 102, an introductory programming course at Victoria University of Wellington, and preliminary evidence suggests that it helps novices learn to program.

8.2 The Demonstration Model

The demonstration model is based on the abstract machine model and programming by demonstration. The model does not make use of any inferencing as part of its programming by demonstration. It makes use of a visualisation of the abstract machine model as the basis for the environment, and direct manipulation to allow the programmer to demonstrate instructions by manipulating the components of the visualised machine. The demonstration model describes the demonstrational machine and the way of programming that the machine supports.

The demonstrational machine extends the abstract machine to allow programming by adding two roles: the demonstrator and the recorder. The demonstrator (the novice programmer) takes over running the computer, and demonstrates what the program should do by carrying out the actions that will make up the program. The recorder watches the demonstrator and writes down the code that the clerk would need to carry out those actions. A contribution of the demonstration model is the distinction between the clerk and the recorder. In other programming by demonstration systems, the role of the recorder is not distinguished from that of the clerk. The distinction is made here so that the two roles played by the computer are clear (to the novice). The clerk is the computer when it is carrying out the program — fundamentally robotic and naive. The recorder is the computer when it is managing the program's construction — intelligent enough to translate demonstrator's actions into a program.

The recorder also records the machine's previous states so that they can be restored, and all the demonstrator's actions so that they can be undone. It was essential for the demonstrational machine's usability that the demonstrator was able to get the recorder to unwind the computer's state, undo the demonstrator's last action(s) entirely, and ignore demonstrator's actions so that the state could be set up appropriately for demonstrating particular cases.

Two spectra of methods for demonstrating declarations were explored: one for type declarations, and one for subroutine declarations. A contribution of the thesis is that it identified two related dimensions that the various methods range across. The first dimension determines how demonstrational the method is in the way declarations are constructed. It ranges from less demonstrational to more demonstrational depending on whether a method is based on defining the general case first or based on defining a particular instance first. The second dimension determines how flexible a method is for demonstrating a declaration. It depends on the relation between: (a) the point in time that the definition of the declaration began, (b) the point in time that the declaration could be used to create an instance, and (c) the point in time that the details have to be complete. The extremes of each dimension did not seem to produce the most usable methods.

Decisions made in the demonstrator's head as part of demonstrating actions need to be made clear to the recorder. The options used by programming by demonstration systems for determining the demonstrator's intentions range from full inference, to guided inference, to forcing the decisions to be made explicit. Inference was not part of the demonstrational machine so decisions made in the demonstrator's head had to be made explicit. This was particularly important for control structures where the demonstrator decides on the next instruction to carry out based on a particular value. The thesis described a spectrum of methods for demonstrating control structures. The methods used a range of techniques from the explicit communication of intent to full inference of the structure.

In the demonstrational machine, the order that the components of certain instructions are demonstrated is different from the order that the clerk will carry out those components when executing the instructions. Coupled with the capability to unwind the state and the requirement to carry out partial instructions, the different "execution" orders greatly increase the complexity of the machine. There is a trade-off between a flexible programming by demonstration environment and the complexity of the underlying system. Forcing the demonstrator to complete all instructions before moving on, or to demonstrate instructions using the clerk's order simplifies the implementation, but reduces the usability of the environment. The increased complexity is important for the environment's usability, and essential for promoting programming by demonstration.

8.3 The Prototype Environments

The thesis described two prototype implementations of the demonstrational machine. Each prototype differed slightly from the demonstrational machine defined in the demonstration model. The prototypes used interaction methods for demonstrating some instructions that were different from the method of acting like the clerk that was described by the demonstration model. The second prototype used the stricter, more accurate analogy for arrays that was excluded from the machine model. The differences between the prototypes and the models helped shed light on many hidden issues for programming by demonstration in this domain. Most of the issues were related to a trade-off between novice and expert programmers, which I believe to be roughly equivalent to a trade-off between ease of understanding and ease of use. However, for many of the issues, it was not clear which side of the trade-off gave the better solution.

Keeping the state consistent in a programming by demonstration environment is expensive. There is a trade-off between the interaction speed of the environment and ensuring that the state is kept correct. The state is difficult to keep consistent because demonstrating an action can affect the previous states as well as the current state. This is caused by the inconsistency between the order of demonstration and the order that the clerk carries out the instructions.

Dealing with expressions is problematic. The abstract machine model states that the clerk uses a calculator to work out the values of expressions. The demonstration model states that the demonstrator should use the calculator like the clerk does to build expressions. In practice, this is problematic with novices because:

1. The order that the clerk will evaluate the expression is not the same as the order in which a person would typically write out an expression.
2. The complexity of some expressions often makes it hard for novices to construct them without making any mistakes.

3. Expressions on the calculator still have to interact with programmer-defined functions that are carried out with worksheets and may have side effects.

There seem to be two ways that people approach expressions: mathematically as objects to manipulate, and computationally as a means for working out a value. The prototypes explored different methods for demonstrating expressions. The first prototype took the first approach and method was based on specifying the structure of the expression. It used a (hopefully) familiar calculator and formula metaphor, showed the entire expression as it was being built, and allowed reasonable flexibility for constructing an expression (though not full editing). However, it was not particularly demonstrational, dealt poorly with side effects, and did not show the order of evaluation well. The second prototype took the other approach and its method was more demonstrational: it restricted the construction order of an expression to the clerk's order of evaluation. However, it was quite cumbersome and tedious to use, though it made it clear how the clerk would evaluate the expression. Despite their good points, neither prototype's method was satisfactory. There is a trade-off between the ease of specification and showing the evaluation. A better method combining the best points of both prototypes' methods was presented, along with the problems that would have to be overcome to implement it.

8.4 Other Issues

The demonstration model and the prototypes all but ignore any mistakes that the demonstrator makes while demonstrating the program; they assume that undo will be enough. However, the demonstrational machine needs editing capabilities beyond demonstrating, undoing, and redoing. The need for editing capabilities such as deletion and modification became apparent after using and evaluating the prototypes. The thesis discussed the difficulties associated with integrating deletion and modification into the demonstrational machine.

There is a trade-off between the real world understandability and the computer world accuracy of the models and analogies. The models focused primarily on the understandability, while the prototypes leant more towards the accuracy. The analogies used in the abstract machine and the demonstrational machine were chosen to fit together in a way that was plausible in the real world. Some of the more accurate analogies for components of the machines were discarded because we did not want to resort to hand-waving and magic in order to explain how all the components could fit together in the real world. However when it came to the prototypes, we found that the less computationally accurate analogies were not good enough to describe what was going on inside the computer. To get an accurate description, the very analogies that we had discarded from the models were used as the basis for implementing some components of the prototypes. In the prototypes, we did not have to explain how the analogies could fit together because it was apparent that they did: the medium of the computer contained sufficient magic to make it technologically plausible for the components to fit together.[1] However, the advantages of the analogies start to be lost when the models have to be patched up to make them accurately match the computer. Perhaps it would have been better for the models to have been more "magical" in their make up.

8.5 Future Work

The most important area for future work is to carry out a proper evaluation of the environment. However, before that can be done, the usability problems of the second prototype must be fixed, and that will require implementing another prototype of the environment. Once the implementation is complete, an evaluation can be carried out to discover how helpful the models and environment are for novice programmers. The next implementation and the evaluation are discussed in the following sections, along with other areas for future work, such as extending the models and environment.

8.5.1 The next prototype

The aim of the next prototype would be to improve the environment's usability (and reliability) to the point where it could reasonably be used in an evaluation with novice programmers. The biggest problem with the previous prototypes is that almost no effort was put into making them user friendly.

The new prototype needs to be fast enough to be usable. It should not slow down once the program reaches a certain size, like the first prototype did. Nor should it fall into the same display and inter-language communication inefficiency traps that made the second prototype slow. This probably means using a single language, such as `Java`, or paying careful attention to the communication between the interface and the back-end.

The appearance and layout of the environment should be improved. The environment should make more use of the real world analogies for the components of the machine provided by the abstract machine model. Adding a look and feel that is in line with the machine model will be an important part of how well the environment supports the machine model. The easier it is to make the connection between the machine model and the computer environment, the more benefit novices should be able to gain from both.

The next version should fix the following short comings of the previous prototypes:

- Ensure that the I/O devices use the machine model's method for demonstrating built-in subroutines. The direct-demonstration method should be relegated to a proper short-cut.
- Split the parts of the I/O devices up and lay them out to clearly delineate the parts of each device that the user interacts with from the parts that the demonstrator

[1] Apologies to Arthur C. Clarke for mangling his law.

and clerk interact with.

- Change the interaction methods for declaring places from "drag and drop" to something more like stamping out a place.
- Resolve the issues with array index places so that assignments to array elements can be carried out like other assignments.
- Display the instructions on the worksheet as Nassi-Shneiderman diagrams instead of plain text.
- Add more controls for the machine's execution, *e.g.* fast-forward, rewind, and breakpoints.
- Provide context sensitive help and more feedback. A brief tutorial of the environment might be useful.

The next prototype should implement a more flexible method for demonstrating expressions that combines the best of both previous expression construction methods (see section 4.4.7). The method needs to support demonstration in any order, it needs to cope with a mixture of calculators and worksheets, and it needs a way of coping with stale values.

Central to improving the demonstration order for expressions is being able to keep the state consistent at all times. The next prototype should keep the state of the machine consistent and avoid stale values and side effects (see section 4.3.6).

8.5.2 A proper evaluation

A proper evaluation of the models and environment needs to be carried out. The evaluation should be aimed at determining how well the models help novices understand programming, and how helpful the environment is for supporting novices as they learn to program.

Various techniques on the spectra of demonstration techniques for declarations and control structures could be implemented, and then further evaluations could be carried out to help resolve some of the unresolved issues from this work, such as:

- Which is better for demonstrating calls to built-in subroutines: the machine model method or the direct-demonstration method?
- What is the best method for declaring a new type?
- What is the best method for declaring a new subroutine?
- What is the best method for demonstrating a control structure?

The evaluations should be carried out with both novice and expert programmers in order to get a range of views.

8.5.3 Extending the models

The analogies for files, units, abstract data types, and pointers could be incorporated properly in the abstract machine model. The appropriate actions for manipulating these objects and demonstrating the related code could be added to the demonstration model. From there, support for these concepts could be added to the environment.

The models could be extended to other kinds programming languages and concepts, such as: object-oriented programming, functional programming, parallel programming, and event based programming. Each new language or set of concepts may introduce major changes to the model, and it may be better to have more than one model rather than an all encompassing model. The most important concept to add to the model is that of objects.

Programming by demonstration in the demonstration model (and the environment) could be extended to incorporate inferencing. Type inferencing could be the first step, and control structures the following step.

8.5.4 Extending the environment

The ability to modify and delete pieces of code would greatly improve the environment. Ways to properly integrate these operations into this kind of programming by demonstration environment would have to be explored because it is not clear how to incorporate them. It would also be useful to have some way of duplicating parts of the code and then having them carried out again without having to re-demonstrate them, *e.g.* use the same expression in another place — a kind of demonstrational copy and paste.

The current prototypes lack any means for producing any kind of secondary notation. Another way to extend the environment would be to provide the demonstrator with the ability to comment and annotate their programs. It is unclear whether adding comments directly to the code would be the best form of secondary notion in this environment. Perhaps it would be more appropriate to allow the demonstrator to draw lines, arrows, doodle, and add text annotations to the worksheets and clipboard. The demonstrator could then create their own secondary notations that made use of the spatial relationships between the components of the machine in the environment (see section 6.4).

If a version of the environment was to be used to support a programming course for novices, it would be useful to allow parts of the environment to be hidden and gradually revealed. Novices could be introduced to the components of the machine slowly, giving them time to come to grips with the complexities of programming. By the same token, integrating the environment with the facilities of a more traditional programming environment would allow the novices to be weaned off the demonstrational machine by allowing them to create more and more of their code by typing it in.

8.5.5 Exploring algorithm animation

The next version of the environment could use animation to support the models, provided the animations do not detract from the purpose of the environment. The operation of the clerk carrying out instructions could be animated by having values move between places, array carousels could rotate as array elements are indexed, and worksheets could be shown being printed, folded and unfolded.

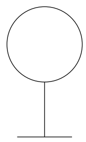
It would be interesting to explore this environment as a tool for program and algorithm animation. The environment is based on visualising the abstract machine model. By using it to write several programs that contained typically taught algorithms, the current version of the environment could be used to produce animations of the algorithms and programs in action. The animations could be used as teaching aids, debugging aids, for displaying performance, and for code walk-throughs.

Appendix A

Evaluation programs

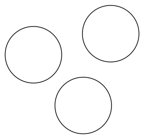
This appendix contains sample code for the programs used to evaluate the first prototype. Other examples of programs constructed with the prototype demonstrational machines can be found in sections 3.5, 3.7, and 3.8. It is important to note that the graphics coordinate system used in the first prototype has positive x going from left to right, and positive y going from bottom to top (*i.e.* it uses cartesian coordinates to map the screen, and not the VDU/scanline coordinate system).

A.1 Draw a tree



```
PROGRAM DrawATree;

CONST
  ROOTS : 5;
  GROUND : 10;
  TRUNK : 50;
  HEIGHT : 10;
  LEAVES : 7;

VAR
  Size : NUMBER;

BEGIN
  WriteSTRING('Please enter the size of the tree: ');
  Size := ReadNUMBER();

  DrawLine((TRUNK - (ROOTS * Size)), GROUND, (TRUNK + (ROOTS * Size)), GROUND);
  DrawLine(TRUNK, GROUND, TRUNK, GROUND + (HEIGHT * Size));
  DrawCircle(TRUNK, (GROUND + ((HEIGHT * Size) + (LEAVES * Size))),
             (LEAVES * Size));
END.
```

A.2 Draw three circles



```
PROGRAM DrawCircles;

CONST
  RADIUS : 15;

VAR
  count : NUMBER;
  x : NUMBER;
  y : NUMBER;

BEGIN
  count := 1;

  LOOP
    WriteSTRING(‘Please click the mouse on the screen.’);
    WriteLINE();

    WaitForMouseClick();
    x := MouseX;
    y := MouseY;
    DrawCircle(x, y, RADIUS);

    IF (count < 3) THEN
      count := count + 1;
    ELSE
      LEAVE;
    ENDIF;

  ENDLOOP;
END.
```

A.3 Output the largest of three numbers (version 1)

```
PROGRAM OutputLargestV1;

VAR
  first : NUMBER;
  second : NUMBER;
  third : NUMBER;

BEGIN
  WriteSTRING(‘Please enter three numbers, separate each by a space: ’);
  first := ReadNUMBER();
  second := ReadNUMBER();
  third := ReadNUMBER();
  WriteSTRING(‘The largest number you entered was: ’);
```

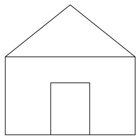
```
  IF (first > second) THEN
    IF (first > third) THEN
      WriteNUMBER(first);
    ELSE
      WriteNUMBER(third);
    ENDIF;
  ELSE
    IF (second > third) THEN
      WriteNUMBER(second);
    ELSE
      WriteNUMBER(third);
    ENDIF;
  ENDIF;

  WriteLINE();
END.
```

A.4 Output the largest of three numbers (version 2)

```
PROGRAM OutputLargestV2;

VAR
  num1 : NUMBER;
  num2 : NUMBER;
  num3 : NUMBER;

  num1GTRnum2 : BOOLEAN;
  num1GTRnum3 : BOOLEAN;
  num2GTRnum3 : BOOLEAN;

BEGIN
  WriteSTRING('Please enter three numbers, separate each by a space: ');
  num1 := ReadNUMBER();
  num2 := ReadNUMBER();
  num3 := ReadNUMBER();

  WriteSTRING('The largest number you entered was: ');

  num1GTRnum2 := (num1 > num2);
  num1GTRnum3 := (num1 > num3);
  num2GTRnum3 := (num2 > num3);

  IF (num1GTRnum2 AND num1GTRnum3) THEN
    WriteNUMBER(num1);
  ELSE
    IF (secGTRthird) THEN
      WriteNUMBER(num2);
    ELSE
      WriteNUMBER(num3);
```

```
    ENDIF;
  ENDIF;

  WriteLINE();
END.
```

A.5 Draw a house

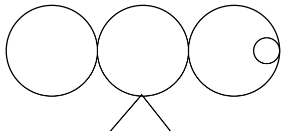


```
PROGRAM DrawHouse;

CONST
  CORNER_X : 10;
  CORNER_Y : 10;
  HEIGHT : 80;
  WIDTH : 100;
  DOOR : 50;
  DOOR_WIDTH : 20;
  DOOR_HEIGHT : 65;
  ROOF_HEIGHT : 30;

BEGIN
  DrawRectangle(CORNER_X, (CORNER_Y + HEIGHT), (CORNER_X + WIDTH), CORNER_Y);
  DrawRectangle(DOOR, (CORNER_Y + DOOR_HEIGHT), (DOOR + DOOR_WIDTH), CORNER_Y);

  DrawLine(CORNER_X, (CORNER_Y + HEIGHT),
           (CORNER_X + (WIDTH / 2)), (CORNER_Y + (HEIGHT + ROOF_HEIGHT)));
  DrawLine((CORNER_X + (WIDTH / 2)), (CORNER_Y + (HEIGHT + ROOF_HEIGHT)),
           (CORNER_X + WIDTH), (CORNER_Y + HEIGHT));
END.
```

A.6 Multiply two numbers

```
PROGRAM MultiplyTwoNumbers;

VAR
 number1 : NUMBER;
 number2 : NUMBER;

BEGIN
 WriteSTRING('Please enter two numbers, separate them by a space: ');

 number1 := ReadNUMBER();
 number2 := ReadNUMBER();

 WriteSTRING('The product of those numbers is: ');
 WriteNUMBER((number1 * number2));
 WriteLINE();
END.
```

A.7 Draw an insect



```
PROGRAM DrawInsect;

CONST
  BODY : 10;
  LEG_HEIGHT : 12;
  LEG_WIDTH : 9;
  EYE : 2;

VAR
  size : NUMBER;
  x : NUMBER;
  y : NUMBER;
  bodysize : NUMBER;
  eyesize : NUMBER;
  legx : NUMBER;
  legy : NUMBER;
  leftlegx : NUMBER;
  leftlegy : NUMBER;
  rightlegx : NUMBER;
  rightlegy : NUMBER;

BEGIN
  WriteSTRING(‘Please enter the size of the insect: ’);
  size : = ReadNUMBER();

  WriteSTRING(‘Please click the mouse where you want the insect’s head to go.’);
  WriteLINE();
  WaitForMouseClick();
  x := MouseX;
  y := MouseY;

  bodysize := (BODY * size);
  DrawCIRCLE(x, y, bodysize);
  DrawCIRCLE((x + (2 * bodysize)), y, bodysize);
  DrawCIRCLE((x + (4 * bodysize)), y, bodysize);

  legx := (x + (2 * bodysize));
  legy := (y - bodysize);

  leftlegx := (legx - (LEG_WIDTH * size));
  leftlegy := (legy - (LEG_HEIGHT * size));
  DrawLINE(legx, legy, leftlegx, leftlegy);

  rightlegx := (legx + (LEG_WIDTH * size));
  rightlegy := leftlegy;
  DrawLINE(legx, legy, rightlegx, rightlegy);

  eyesize : = (EYE * size);
```

```
  DrawCIRCLE(((x - bodysize) + eyesize), y, eyesize);
END.
```

A.8 Calculate the properties of a number

```
PROGRAM NumberProperties;

VAR
  num : NUMBER;

BEGIN
  WriteSTRING(‘Please enter a positive number: ’);
  num := ReadNUMBER();
  WriteLINE();

  WriteSTRING(‘The square of the number is: ’);
  WriteNUMBER((num * num));
  WriteLINE();

  WriteSTRING(‘Half of the number is: ’);
  WriteNUMBER((num / 2));
  WriteLINE();

  WriteSTRING(‘The number is ’);
  IF (((num / 2) = 0)) THEN
    WriteSTRING(‘even’);
  ELSE
    WriteSTRING(‘odd’);
  ENDIF;
  WriteLINE();
END.
```

A.9 Click in the button


```
PROGRAM ClickInTheButton;

CONST
  LeftX : 50;
  RightX : 160;
  TopY : 100;
  BottomY : 75;

VAR
  x : NUMBER;
  y : NUMBER;
  message: STRING;

BEGIN
```

```
  WriteSTRING('Enter the words on the button: ');
  message := ReadSTRING();

  DrawLine(LeftX, TopY, RightX, TopY);
  DrawLine(RightX TopY, RightX, BottomY);
  DrawLine(RightX, BottomY, LeftX, BottomY);
  DrawLine(LeftX, BottomY, LeftX, TopY);
  DrawSTRING(((LeftX + RightX) / 2), ((TopY + BottomY) / 2), message);

  WriteSTRING('Please click the mouse in the button.');
  WriteLine();

  LOOP
    WaitForMouseClick();
    x := MouseX;
    y := MouseY;
    IF (((x > LeftX) AND (x < RightX)) AND ((y > BottomY) AND (y < TopY))) THEN
      LEAVE;
    ELSE
      WriteSTRING('You clicked outside the button. Please try again');
      WriteLine();
    ENDIF;
  ENDLOOP;

  WriteSTRING('Well done! You clicked in the button.');
  WriteLine();
END.
```

A.10 Print a message ten times

```
PROGRAM PrintTenTimes;

VAR
  message : STRING;
  counter : NUMBER;

BEGIN
  WriteSTRING('Please enter a message:');
  WriteLINE();
  message := ReadSTRING();

  counter := 1;
  LOOP
    IF (counter =< 10) THEN
      WriteNUMBER(counter);
      WriteSTRING(') ');
      WriteSTRING(message);
      WriteLINE();
```

```
      counter := counter + 1;
    ELSE
      LEAVE;
    ENDIF;
  ENDLOOP;

END.
```

Bibliography

Adobe (1996). *Using Adobe FrameMaker 5 (UNIX)*, Adobe.

Anderson, J. R., Boyle, C. F. & Reiser, B. J. (1985). Intelligent tutoring systems, *Science* **228**: 456–462.

Chang, S.-K. (1990). Visual languages: A tutorial and survey, *in* Glinert (1990b), pp. 7–17.

Cockburn, A. & Bryant, A. (1996). Do it *this* way: Equal opportunity programming for kids, *in* J. Grundy & M. Apperley (eds), *Sixth Australian Conference on Computer-Human Interaction*, IEEE Computer Society Press, pp. 246–251.

Cypher, A. (1993a). Eager: Programming repetitive tasks by demonstration, in *Watch What I Do: Programming by Demonstration* (Cypher 1993b), chapter 9, pp. 204–217.

Cypher, A. (ed.) (1993b). *Watch What I Do: Programming by Demonstration*, M.I.T. Press.

Cypher, A. & Smith, D. C. (1995). Kidsim: End user programming of simulations, *CHI 1995*, ACM Press.

diSessa, A. A. & Ableson, H. (1986). Boxer: A reconstructible computational medium, *Communications of the ACM* **29**(9).

Eisenstadt, M. (1997). My hairiest bug war stories, *Communications of the ACM* **40**(4): 30–37.

Eisenstadt, M., Price, B. A. & Domingue, J. (1993). Software visualisation as a pedagogical tool: Redressing some its fallacies, *Instructional Science* **21**: 335–365.

Ellershaw, S. & Oudshoorn, M. (1994). Program visualisation – the state of the art, *Technical Report 94–19*, Department of Computer Science, University of Adelaide.

Erickson, T. D. (1995). Working with interface metaphors, *in* R. M. Baecker, J. Grudin, W. A. S. Buxton & S. Greenberg (eds), *Readings in Human-Computer Interaction: Toward the Year 2000*, 2nd edn, Morgan Kaufmann Publishers Inc., chapter 2, pp. 147–151.

Finzer, W. F. & Gould, L. (1993). Rehersal world: Programming by rehersal, *in* Cypher (1993b), chapter 4, pp. 78–100.

Ford, L. & Tallis, D. (1993). Interacting visual abstractions of programs, *in* E. P. Glinert & K. A. Olsen (eds), *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, pp. 93–99.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design Patterns*, Addison-Wesley.

Glinert, E. P. (ed.) (1990a). *Visual Programming Environments: Applications and Issues*, IEEE Computer Society Press.

Glinert, E. P. (ed.) (1990b). *Visual Programming Environments: Paradigms and Systems*, IEEE Computer Society Press.

Goldson, D. (1994). A symbolic calculator for non-strict functional programs, *The Computer Journal* **37**(3): 177–187.

Golightly, D. (1996). Harnessing the interface for domain learning, *CHI '96 Doctoral Consortium*, pp. 37–38.

Gonzalez, C. (1996). Does animation in user interfaces improve decision making?, *in* M. J. Tauber, V. Bellotti, R. Jeffries, J. D. Mackinlay & J. Nielsen (eds), *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*, ACM Press, pp. 27–34.

Halbert, D. C. (1993). Smallstar: Programming by demonstration in the desktop metaphor, *in* Cypher (1993b), chapter 5, pp. 102–123.

Hashimoto, O. & Myers, B. (1992). Graphical styles for building user interfaces by demonstration, *Proceedings of the 5th Annual Symposium on User Interface Software and Technology*, ACM Press, pp. 117–124.

Holst, S. J. (1996). Directing learner attention with manipulation styles, *CHI '96 Doctoral Consortium*, pp. 43–44.

Hyrskykari, A. (1995). Development of program visualization, *Technical Report A-1995-3*, Department of Computer Science, University of Tampere.

Koffman, E. B. (1995). *PASCAL*, 5th edn, Addison-Wesley.

Kurlander, D. (1993). *Chimera: Example-Based Graphical Editing*, in Cypher (1993b), chapter 12, pp. 270–290.

Lieberman, H. (1993a). *Mondarian: A Teachable Graphical Editor*, in Cypher (1993b), chapter 16, pp. 340–358.

Lieberman, H. (1993b). *Tinker: A Programming by Demonstration System for Beginning Programmers*, in Cypher (1993b), chapter 2, pp. 48–64.

Lieberman, H. & Fry, C. (1995). Bridging the gulf between code and behaviour in programming, *CHI '95*, ACM Press, pp. 480–486.

Madsen, K. H. (1994). A guide to metaphorical design, *Communications of the ACM* **37**(12): 57–61.

Maulsby, D. (1993). *The Turvey Experience: Simulating an Instructible Interface*, in Cypher (1993b), chapter 11, pp. 238–269.

Maulsby, D. & Witten, I. H. (1993). *Metamouse: An Instructible Agent for Programming by Demonstrartion*, in Cypher (1993b), chapter 7, pp. 154–181.

Mayer, R. E. & Bayman, P. (1981). Psychology of calculator languages: A framework for describing difference in users' knowledge, *Communications of the ACM* **24**(8): 511–520.

McDaniel, R. G. (1996). Improving communication in programming by demonstration, *CHI '96 Doctoral Consortium*, pp. 55–56.

McIver, L. & Conway, D. (1995). Seven deadly sins of introductory programming language design., *Technical Report 95/234*, Dept. Computer Science, Monash University, Australia 3168.

Miyashita, K., Matsuoka, S., Takahashi, S., Yonezawa, A. & Kamada, T. (92). Declarative programming of graphical interfaces by visual examples, *Proceedings of the ACM User Interface Software and Technology '92 (UIST'92).*

MountFord, S. J. (1994). Tools and techniques for creative design, *in* R. M. Baecker, J. Grudin, W. A. S. Buxton & S. Greenberg (eds), *Readings in Human-Computer Interaction: Toward the Year 2000*, second edn, Morgan Kaufmann Publishers Inc, pp. 128–141.

Myers, B. A. (1990). Visual programming, porgramming by example, and program visualization: A taxonomy, *in* Glinert (1990b), pp. 33–40.

Myers, B. A. (1992). Demonstrational interfaces: A step beyond direct manipulation, *IEEE Computer* pp. 61–73.

Myers, B. A. (1993a). *Demonstrational Interfaces: A Step Beyond Direct Manipulation*, in Cypher (1993b), chapter 26, pp. 485–512.

Myers, B. A. (1993b). *Garnet: Uses of Demonstrational Techniques*, in Cypher (1993b), chapter 10, pp. 218–236.

Myers, B. A. (1993c). *Peridot: Creating User Interfaces by Demonstration*, in Cypher (1993b), chapter 6, pp. 124–153.

Myers, B. A., McDaniel, R. G. & Kosbie, D. S. (1993). Marquise: Creating complete user interfaces by demonstration, *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, ACM Press, pp. 293–300.

Nardi, B. A. (1993). *A Small Matter of Programming*, M.I.T. Press.

Nassi, I. & Shneiderman, B. (1990). Flowchart techniques for structured programming, *in* Glinert (1990b), pp. 72–78.

Norman, D. A. (1988). *The Psychology of Everyday Things*, Basic Books, chapter 1, pp. 1–33.

O'Niell, E. J. (1995). Classroom presentation of memory usage with constants, variables, and subprograms in `Pascal`, *SIGCSE Bulletin* **27**(2): 18–24.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*, Addison-Wesley.

Pane, J. F. & Myers, B. A. (1996). Usability issues in the design of novice programming systems, *Technical Report CMU-CS-96-132*, School of Computer Science, Carnegie Mellon University.

Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming, *Communications of the ACM* **38**(6): 33–44.

Price, B. A., Baecker, R. M. & Small, I. S. (1993). A principled taxonomy of software visualization, *Journal of Visual Languages and Computing* **4**(3): 211–266.

Rader, C., Brand, C. & Lewis, C. (1997). Degrees of comprehension: Children's understanding of a visual programming environment, *CHI '97*, ACM Press, pp. 351–358.

Read, M. & Marlin, C. (1996). Generating direct manipulation program editors, *in* J. Grundy & M. Apperley (eds), *Proceedings of the Sixth Australian Conference on Human Computer Interaction*, IEEE Computer Society Press, Los Alamitos, California, pp. 177–183.

Repenning, A. & Citrin, W. (1993). Agentsheets: Applying grid-based spatial reasoning to human-computer interaction, *in* E. P. Glinert & K. A. Olsen (eds), *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, pp. 77–82.

Shneiderman, B. (1990). Direct manipulation: A step beyond programming languages, *in* E. P. Glinert (ed.), *Visual Programming Environments: Applications and Issues*, I.E.E.E. Computer Society Press, pp. 317–329.

Smith, D. C. (1990a). Principles of iconic programming, *in* Glinert (1990b), pp. 216–251.

Smith, D. C. (1993). *Pygmalion: An Executable Electronic Blackboard*, in Cypher (1993b), chapter 1, pp. 19–47.

Smith, R. B. (1990b). Experiences with the alternate reality kit: An example of the tension between literalism and magic, *in* Glinert (1990b), pp. 388–396.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations, *Communications of the ACM* **29**(7): 850–858.

Spohrer, J. C. & Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct?, *Communications of the ACM* **29**(7): 624–632.

Ungar, D., Lieberman, H. & Fry, C. (1997). Debugging and the experience of immediacy, *Communications of the ACM* **40**: 38–43.

Wilcox, E. M., Atwood, J. W., Burnett, M. M., Cadiz, J. J. & Cook, C. R. (1997). Does continuous visual feedback aid debugging in direct-manipulation programming systems, *Proceedings of ACM CHI'97*, ACM Press, pp. 258–265.

Wilde, N. P. (1993). A WYSIWYC (what you see is what you compute) spreadsheet, *in* E. P. Glinert & K. A. Olsen (eds), *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, pp. 72–76.

Witten, I. H. & Mo, D. (1993). *TELS: Learning Text Editing Tasks from Examples*, in Cypher (1993b), chapter 8, pp. 182–203.

Wolber, D. (1996). Pavlov: Programming by stimulus-response demonstration, *in* M. J. Tauber, V. Bellotti, R. Jeffries, J. D. Mackinlay & J. Nielsen (eds), *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*, ACM Press, pp. 252–259.

Wolber, D. & Fisher, G. (1991). A demonstrational technique for developing interfaces with dynamically created objects, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, ACM Press, pp. 221–230.

Young, R. M. (1981). The machine inside the machine: user's models of pocket calculators, *International Journal of Man Machine Studies* **15**: 51–85.