MONASH UNIVERSITY
THESIS ACCEPTED IN SATISFACTION OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
ON..... **3 August 2001** .....
.....[redacted]
Sec. Ph.D. and Scholarships Committee

Syntactic and Semantic Issues in Introductory Programming Education

Linda Kathryn McIver
B Comp (Hons)

Supervisor: **Dr Damian Conway**

School of Computer Science and Software Engineering
Monash University
Australia

Submitted for the degree of Doctor of Philosophy
January, 2001

Syntactic, Semantic and Social Issues in Introductory Programming Education

Abstract

Learning computer programming is difficult. Students often have trouble coming to terms with the fundamentals of programming. At the same time, they are forced to tackle the complexities of a particular programming language. There is considerable debate about the choice of programming language for introductory programming courses, but little empirical evidence for the efficacy of any particular language.

This thesis examines why learning to program is difficult. Programming languages are discussed as user interfaces, and hence analysed using usability principles. This analysis is used as the basis for the construction of a theory of pedagogical programming language design, which is then applied to the design of a new pedagogical programming language, GRAIL.

A new framework is presented for the evaluation of pedagogical programming languages. Using this framework, GRAIL is evaluated, and subsequently redesigned. The results of the evaluation provide evidence of the impact of introductory programming languages on the learning process, and show that the choice of introductory programming language is important.

In loving memory of Dianne Louise Wapling
10/11/1971 – 22/9/1996

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university or other institution. Furthermore, to the best of my knowledge, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Acknowledgements

I would like to thank many people for helping me survive the last few years. In particular:

- The admin and technical staff in CSSE have provided invaluable support throughout my candidature. In particular, I am indebted to Karen Fenwick, Jamie Scuglia, Glen Pringle, and Steve Welsh.
- Andrew M'Iver, Steve Welsh, David Chatterton, Kevin Lentin, Tony Jansen, and Darren Platt all provided excellent proof-reading services.
- Many thanks are due to Ricky McConachy and Kevin Lentin for technical help with the Jump Start Programme, and to Kate Chatterton, Kate Nairn, and Jane Sykes for beta testing the course.
- The Department of Computer Science, and its successor, the School of Computer Science and Software Engineering, especially Trevor Dix, have been exceptionally generous, both with scholarship funding (extended throughout prolonged illness) and funding for Jump Start.
- Darren Platt and Andrew Davison got me into this mess in the first place – thanks guys.
- My supervisor, Damian Conway, got me deeper into this mess, and turned me into an academic. He also provided support above and beyond the call of duty, especially when I was ill. "Thankyou" just doesn't seem adequate!
- The muffin-klatch and my second family in the Power Electronics Group (PEG) helped keep me sane – quite a feat! Thanks guys.
- Thanks to Kerryn Davison, for regular phone calls and support.
- Special thanks are due to Tony Jansen for endless cups of tea, strong support, and lots of hugs.
- Extra-special thanks are due to Darren Platt, for proving that it can be done, and for always being there for me, despite being on the other side of the world!

Finally, and most importantly: without my husband, Andrew M'Iver, I would never have survived, much less produced a thesis. Maybe one day I'll be able to make it up to you!

Table of Contents

1 Syntactic, Semantic and Social Issues in Introductory Programming Education

1.1 Introduction

Learning to program is an unnatural act. It requires novices to codify their notions of specification and process, to acquire an understanding of abstractions for which they may have no prior referents, and to express these concepts in a formal style of language they have never previously encountered. In other words, it is like trying to alter their belief system by quoting theoretical philosophy to them in hieroglyphics.

In addition to the problems posed directly by computer programming itself, there are social issues that can throw sizeable spanners in the works of the learning process. Rarely addressed or even acknowledged, these social and environmental hurdles can have considerable impact on student confidence, and on the way students handle their interaction with computers. Students' past experience with computers, together with their perception of computers and computer programming, can have a substantial effect on the learning process, and on how students handle errors and setbacks.

Given the nature and number of problems facing students learning computer programming for the first time, there is a case for minimizing the obstacles and maximizing student confidence in the early stages of an introductory programming course.

This thesis describes the difficulties students face when they learn to program, and details a method for reducing these difficulties and encouraging students to experiment and feel confident. Based on this method, a new programming language has been designed, and put to the test with a group of first year students who had no programming experience. The results show that the choice of introductory programming language can have a significant impact on the type and quality of interaction students have with the machine.

1.2 Thesis Outline

The particular social issues that impact on learning computing are often not directly addressed in programming courses. Chapter 2 provides an introduction to the field of usability, and describes some of the usability issues facing computer

users. Chapter 3 discusses programming languages as a special class of user interface, with their own unique usability issues as well as many that are common to the whole field of usability and human computer interaction.

Chapter 4 provides an introduction to the literature on learning and teaching programming, including such issues as why learning to program is difficult, and whether there is a problem that can be addressed – is it possible or even desirable to make learning to program easier?

The overwhelming trend in current programming courses is to teach so-called "real world" languages: languages that are used in industry. Chapter 5 describes the languages used in teaching today, as well as some of the languages that have had a significant impact on language design. The usability of these languages is analysed for novice programmers.

The issues that face language designers, and educators responsible for the selection of an appropriate introductory language, are discussed in Chapters 6 and 7. Specifically, Chapter 6 deals with flaws in pedagogically targeted programming languages, and other types of languages used to teach programming, and discusses how these flaws impede student learning. Chapter 7 constructs a theory of language design that seeks to avoid these flaws, and to provide rules for the construction of a pedagogical programming language that actually has a positive impact on the learning process.

The theory of language design constructed in Chapter 7 is applied in Chapter 8, which details the design of a new pedagogical programming language, GRAIL – Genuinely Readable And Intuitive Language. This chapter describes the features that were considered for inclusion in GRAIL, and presents the rationale for the acceptance or rejection of each.

Chapter 9 describes the implementation o GRAIL together with a testing process and results. Chapter 10 discusses the impact of the results on the theory, design, and evaluation, highlights the contributions of this thesis, and suggests appropriate directions for further research.

2 User Interfaces and Usability

An exploration of the issues involved in teaching computer programming logically begins with an examination of the technical, psychological, and social issues inherent in dealing with computers in general. Students approach introductory programming courses with a variety of backgrounds and experiences with computing. This thesis focuses on the particular needs of students with limited computing background[1], no programming experience, and low confidence.

In order to understand the issues which impinge on these students' learning in an introductory course, it is necessary to understand their prior experiences with computers and technology. The study of usability sheds some light on the attitudes with which people approach computers, and hence the way they interact with them. These experiences combine to form an attitude to computing which may, in the worst cases, significantly impede the learning process. This chapter examines the technical, psychological, and social issues created by existing computing hardware, software applications, and social attitudes to technology.

2.1 User Interfaces

The term *interface* refers to communication and control. A *user interface* is a way of communicating with, or using, a tool. Part of the user interface of a door is a door handle. The user interface of an electric kettle includes the handle, switch and spout. For a more complicated machine, the user interface is typically a complex system of interacting parts. For example, the user interface of a bicycle includes the pedals, gear levers, handle bars, and brake levers. Each part of the interface depends upon the other parts, and the overall state of the system. It makes no sense to apply the brakes firmly while still trying to pedal. Changing gears requires temporary decrease of pressure on the pedals, but the pedals must still be moving.

The keyboard, screen, mouse, and on/off switch are all parts of the hardware interface of a computer. Peripherals such as CD drives, speakers, and printers can also be viewed as parts of a computer's user interface. An additional part of the user interface of a computer is located behind the machine, where various cables

---

[1] While students may have used computers extensively, it is frequently in a limited sense, restricted to email and web browsing. Such students are familiar with a very limited subset of computers' capabilities.

and switches are located. Users might not interact with this level of interface very often, but it is still part of the machine that must be dealt with from time to time, such as when connecting a new printer, or reattaching a cable that has worked loose.

Another layer of interface is the operating system. This is where the software interfaces with the hardware and peripherals, determining the way the whole system works together. It affects the types of actions that users may perform, determining such details as whether two programs may be run simultaneously, and how peripheral devices are accessed. Many of the functions of the operating system are ones the user never directly experiences, for example determining for how long, and in what order, processes may use the CPU.

For most users, the important part of the interface with the computer is the user interface of the operating system. This is where users spend much of their time interacting, and it is where many of the usability problems described later in this chapter originate (see section 2.2). The interface of the operating system is distinct from the operating system itself, although it may not be entirely clear where the line is drawn. While the operating system deals with process scheduling and device management, and maintains the file system and network connections, the interface to the operating system gives users access to the programs they wish to run. The interface is often intended to shield the user from the low level technicalities of dealing with the hardware, and such interfaces are increasingly likely to be graphical in nature. Even novice users who only use the machine for reading email and browsing the web must still interact with the operating system interface in order to do things like start web browsers and switch between applications. For many users, especially novices, the interface to the operating system *is* the machine.

The idea that the user interface *is* the machine is not unique to the world of computing. Many motorists have little, if any, idea about the mechanical workings of a car, beyond basic details such as where and when to add petrol and oil. To these motorists, the car is not a complex internal combustion engine and a collection of detailed mechanical specifications (such as the number and arrangement of cylinders, the type of braking system, and the capacity of the engine). Instead, it is a collection of immediately visible details, such as the colour, the style of the seat covers, and the type of sound system, as well as the interface

itself: the steering wheel, the gear lever, the handbrake, and the clutch (if there is one), brake, and accelerator pedals.

Similarly, the user interface of a software application is that part of the program with which a user interacts directly.

The user interface of a program or application usually consists of menus, windows, buttons, scroll bars, dialog boxes, and similar objects[2]. This is the top level of the user interface, and it is the part of the application that the user can see immediately when the application is run. Many applications also possess hot keys or shortcuts which allow the application to be manipulated from the keyboard without the use of graphical tools.

Hardware interfaces possess certain characteristics which tend to make them innately easier to deal with than software. Hardware is visible, solid and sometimes audible as well (for example when a hard drive spins up). It is easy to determine whether a simple television is switched on and working by checking whether there is a picture on the screen. Today's computer screens are not so straightforward – if there is a picture on the screen, all is well. If there is a blank screen, the screen saver may be on, the screen may be in power-save mode, or the screen may be on but receiving no signal from the computer. If hardware is not working, there are obvious physical attributes to check, such as whether all the cables are fully plugged in, and whether the power is on. If software is not working, even users who know enough to check cables and connections may have little recourse.

2.1.1 Levels of interface

Computers possess several levels of interface. The hardware interface, which includes the keyboard, mouse, screen, off button, reset switch, disk drives, other peripherals etc, is the first level most users see ("how do I turn this thing on?"). The second level of interface with which users must contend is usually the operating system interface, which is likely to contain windows, icons, menus, buttons etc. Above the operating system lie the applications, which possess their own interfaces, often with their own individual peculiarities. Some applications

---

[2] Experienced computer users frequently use command line interfaces, but it is increasingly rare that a novice computer user would be required to deal with a command line interface.

even possess extra layers of interface, by way of macro languages which allow the user to access more sophisticated data processing within the program.

Each layer or level of interface adds to the mental burden imposed on the user. Although some levels are designed to protect the user from the complexity and technicalities of the underlying levels, it is rarely possible to make this protection complete. For example, while the user interface of the operating system generally protects the user from the need to know details such as precisely which brand of video card is in use, any change to the system configuration, for example the installation of a new piece of hardware, or a change to the display properties, may require the user to know these details, even though they have deliberately been concealed up to this point. This unexpected requirement to deal with the mysterious technicalities inside the machine may make it more difficult, psychologically, for the user to deal with the problem.

2.2 Usability

2.2.1 Definition of usability

The usefulness of a system can be divided into two components: *utility* and *usability* (Nielsen, 1993). *Utility* refers directly to the functionality of a system. A system scores well in utility if it is capable of performing the tasks it was designed for. *Usability* refers to how effectively, easily, and efficiently users can access this functionality. A system scores well in usability if users find it simple, efficient, and pleasant to use, and if it makes users more productive. While utility is easily measured, usability tends to be more subjective and elusive (Nielsen, 1993). Usability is usually discussed in reference to a specific user group, such as accountants, salespeople, teachers, etc.

Usability is user-dependent. In addition to varying between professions, age groups, gender, and physical capabilities (such as strength, eyesight, disabilities etc), usability varies with experience, so that a program which is extremely usable for a novice may be much less usable for an expert. Conversely, what is usable for an experienced programmer may not be usable for a person with no computing experience. For example, the unix text editor, vi, has a steep learning curve, and yet it can be used very efficiently by an expert. Vi can be confusing and frustrating for a novice who has not yet come to grips with insert and command modes. Nonetheless, an experienced user who is familiar with the modes and

with many of the available commands will often find it a powerful and fast way to edit text, often faster than a "user-friendly" graphical interface[3].

There is no proven technique for making a program usable. This is largely due to the subjective nature of usability. There is no single, universal target state of usability. Designers must first ask for whom they intend to achieve usability: novices or experts, men or women, children or adults, particular professions, and different cultures. For each of these groups the level of usability will be based on different aspects of a program or tool.

Usability is not easily quantified. Some aspects, such as task completion and speed of performance, are easily measured. Others, however, such as ease of recovery from errors and learnability, are more subjective and harder to measure.

Nielsen proposes some important usability heuristics (Nielsen, 1993), and Green's cognitive dimensions (Green, 1989), discussed further in section 2.5, also provide some insight into ways of analysing usability. It is significant, though, that there are few, if any, quantitative or universal usability metrics, despite considerable research activity in the area. Even the cognitive dimensions, some of which are measurable characteristics, are not black and white "this dimension is good, this one is bad" values. Instead, they are aspects of notations and interfaces which must be considered in tandem with the domain and the users for which the notation is intended. Different users and different domains will have different priorities among the dimensions, and different aspects of each dimension will be important in different circumstances. Both Green's dimensions and Nielsen's heuristics involve trade-offs between the different principals, such that maximising one dimension or heuristic may decrease another.

Nielsen (1994) discusses a wide range of usability heuristics, pointing out that the literature contains many lists of general usability principles, such as the ones found in Polson and Lewis (1990). Nielsen also notes that, while usability problems are fairly easily categorised as major or minor[4], some problems which fall into the minor category turn out to have a surprisingly substantial impact on usability. It is not sufficient, therefore, to focus on major usability issues at the expense of minor ones.

---

[3] In part, this is because a text based system does not require the "device switching" of a graphical interface – in other words, switching between the mouse and the keyboard.

General usability principles provide overall guidelines for software design and evaluation, but they often fall short of providing clear, unambiguous, and incontrovertible decisions on specific individual design questions. The principles which seem so clear in practice are often remarkably subjective. For example, one usability principle described by Polson and Lewis (1990) is "*Provide an obvious way to undo actions.*" Like many general usability principles, this one falls down on a single subjective word: "*obvious*". What is obvious to developers may not be obvious to users.

Dix et al (1998) list some general principles which support usability. These consist of three major categories: learnability, flexibility, and robustness. It is generally agreed (Nielsen, 1993; Norman, 1990; Landauer, 1995) that the only way to be sure of a system's usability is to evaluate it under real world conditions. In other words, to have it used by the intended users in their normal environment. In addition to the feedback such users provide of their own impressions of the systems, developers observing the users may detect problems of which the users themselves are unaware.

Although post-production usability evaluation can successfully provide a measure of a system's usability after it has been built, it would be more useful to have a measure of usability which can be incorporated into the early stages of the design phase. The growing discipline of usability engineering is working towards this goal (Nielsen, 1993). It is clear that there is no easily-applied panacea which leads to the design of a truly usable product.

2.2.2 Market forces

The principles of usability are incompatible, or at least partially conflicting, with many of the forces behind software development today. In the marketplace, features differentiate software packages from their competitors. The number of features is easy to measure, and readily compared with competing products, so it is used as a selling point (Norman, 1998). In addition, extra features can be used to sell new versions of software to existing customers. As a result, the number of features is often proportional to sales.

---

[4] where major problems have the potential to cause significant delays or prevent task completion altogether

Features accumulate because, as features are added to a product, old features are rarely removed, for the sake of backwards compatibility. A new version of a product must still be able to do everything the previous version did, even if there are now better ways to handle those tasks. The latest version of a word processor needs to be able to read the files produced by the previous version, in order to allow continuing access to existing data. The problem of accumulating features is often called *creeping featuritis*, and is discussed in greater detail in section 2.3.4.

Taken together, all of these problems result in systems which lack usability. Poor usability results in frustration, feelings of inadequacy and helplessness, and a fear of technology. Unusable systems require the user to adapt or conform to them. An ideal system would adapt and conform to the users' needs and style of working (Norman, 1993).

2.2.3 Essential Attributes of User Interfaces

A list of important attributes of user interfaces is relatively simple to enumerate, but often not so easy to measure, quantify, or even recognise. It is surprisingly difficult to design an application which possesses these general attributes, as indicated by the magnitude of research effort dedicated to improving usability development and analysis techniques (Green & Blackwell, 1998; Norman, 1992, 1993, & 1998; Landauer, 1995; Constantine, 1995; Nielsen, 1993 & 1994; Schneiderman, 1987). Nonetheless, it is important to achieve consensus on a fundamental set of qualities, even if agreement is harder to reach on precisely what they mean and how to ensure their prominence in the development process.

The characteristics listed below are by no means exhaustive. Rather they are a collection of the most critical qualities as generally agreed in the literature (Nielsen, 1993; Norman, 1990; Landauer, 1995). Each element in the list is interconnected with every other element. For example the stability and reliability of a system has a strong impact on usability, because a system which crashes all the time interrupts users' work and increases frustration. Nonetheless, they are still separate principles, each of which can have a significant and distinct impact on a user interface.

2.2.3.1 Useful

A *useful* user interface increases productivity and cuts down on wasted time and frustration. Usefulness is related to utility (as described in section 2.2.1), and

requires that a system has both the required functionality – ie that it can do everything the user needs it to do – and usability.

Ultimately the main goals of high productivity and high user satisfaction are dependent on the usefulness and usability of the software. Useful features map directly onto problems users need to solve, and tasks they need to perform. A large collection of features that are only marginally useful is likely to lead to decreased usability, as the number of options increases and makes locating a desired option more difficult and time-consuming (as described in section 2.3.4). The lack of standard usability testing, and the cognitive and experiential gulf between developers and users means that user satisfaction and productivity levels are rarely measured, and infrequently achieved. (Landauer, 1995; Nielsen, 1993; Norman, 1990)

2.2.3.2 Usable

Usability, as discussed in section 2.2.1, is a measure of how effectively, easily, and efficiently an application can be used. Usability is a key component of both increased productivity and user satisfaction. Even if the software contains all the necessary features to execute a particular task, it may still disrupt or delay the task substantially if it is not easy and efficient to use. Although there is broad agreement in the field of usability engineering on what constitutes usability, there remains disagreement on specific details. Few analytic methods for determining usability[5] based on a design are sufficiently broad to be applied to all user interfaces, and yet specific enough to give feedback on individual features and design decisions. As a result, usability analysis relies on usability testing involving genuine users, or a representative set of users.

In general, usability testing involves a high degree of subjectivity and variability. It is frequently difficult to apply usability testing to enough users, under a broad enough range of situations, to analyse an interface in a fully objective, statistically valid fashion. Test subjects are often not entirely representative of the ultimate user community, if only because user communities can be unmanageably large and varied. Furthermore, some features of an interface can be tested effectively by any user, other features need to be analysed by specific real users under real conditions, since some features will only be used by specific groups of users, and

---

[5] Green's cognitive dimensions, discussed in section 2.5 offer an analytic approach to usabiiity, but a full analysis is extremely time -consuming.

other features may be used in different ways by different users. For example, a class list containing assignment marks may be used in one way by those entering the marks, and in a different way by users who need to process the marks. If testing is only conducted using users entering the marks, the display may be biased towards the needs of those users, and a usability flaw for users trying to process the marks may not be detected until the product is released.

Applications – and the features that get added to them – are frequently more appealing to the programmer if they are easy to create. Unfortunately some of the most usable and helpful features of an interface are not the simplest to code. For example, a help system which describes what each button does is relatively simple to write – you simply list all the buttons and menu items, and describe them one by one. In contrast, a help system which relates directly to what the user wants to achieve, rather than to how the system works, is much more useful[6] and correspondingly more difficult to create. Building usable help utilities requires knowledge of how users work, what they typically want to get done, and what terminology they use, rather than knowledge of what the program does. As a result, applications often end up being easy to create rather than easy to use (Norman, 1998).

2.2.3.3 Human-centred

A usable interface is designed to suit the needs of people, rather than requiring people to adapt to the quirks and inflexibility of the machine. Because machines are less adaptable than people, it is simpler to design systems that place the burden of flexibility on humans. In this way, features that are difficult to implement need not be included, instead they are passed on to the user in the form of *work-arounds* (extra steps the user must take in order to circumvent the shortcomings of the system). A human-centred system requires careful attention to user's requirements and the way people think and work.

In contrast, machine-centred applications are designed around the capabilities and requirements of the machine or the technology, rather than those of the user. Technology-centred design led to the power switches on the first personal computers being at the back of the machine: awkward for users to reach, but

---

[6] ...because the user can look up how to create a cross-reference, rather than needing to know in advance that cross-references are created using "power fields", or whichever proprietary technique is used in a particular word processor...

easiest for the switch to connect to the hardware. User-centred design moved that switch to the front of the machine[7] where a user can reach it easily and conveniently.

Some examples of human-centred design are:

- Lengthening a power cord so that a machine can be positioned where it is most useful for the user, rather than restricting the location to within a small radius of a power point.
- Positioning a "close window" button on a software application away from the "make full screen" and "iconify" buttons, to make it less likely that users will inadvertently quit an application by hitting the wrong button.
- Attaching the power cord on an electric kettle to a detachable base, so that the kettle need not be unplugged in order to be taken to the tap and refilled.
- Video recorders which have an on-screen display on the television screen, rather than relying on the (often distant) digital readout on the front of the video recorder.

2.2.3.4 *Task-oriented*

A system which is task-oriented matches user needs closely, providing an efficient, intuitive, and effective way of achieving users' goals. Creating a task-oriented system requires the designer to be familiar with the problem domain, and with the tasks users need to be able to perform with the system. The system should not merely re-implement on a computer those tasks which users currently perform manually (Nielsen, 1993), rather it should enhance and support the task, using the power of the machine to cut down on the tedious and error-prone aspects of a job.

A truly task-oriented system will provide all the information users need when they need it, and in a useful and logical form. It will anticipate the actions users are likely to want to perform at each step in the process, and provide quick, easy, and efficient access to those actions, while maintaining flexibility. In cases where the anticipated action is not the action the user wants to perform, it will be simple

---

[7] On many machines the internal power switch is still at the back, but it has been connected to a front-facing switch.

to choose the desired action, thus allowing users to choose their own paths through the system where necessary.

Examples of systems which are not task-oriented can often be found in software help systems. Help systems are frequently laid out in the same fashion as the software, with a paragraph or two allocated to each button and menu option in the program. A task-oriented help system, instead of addressing program features directly, would address tasks the user might need to carry out.

Such a help system for a web browser might have a section on how to speed up the downloading of web pages (by turning off the automatic display of images), while a program-oriented help system might instead describe the Preferences item on the Edit menu. While both systems may eventually provide the same information, the program-oriented system requires the user to know in advance where the desired option is located, and how to solve the problem, while the task oriented system starts with the problem, and describes how to fix it.

While a good interface is designed around the tasks the user needs to perform, many existing interfaces are program-oriented. A program-oriented interface is designed to use the functions of the back-end program, compared with a task-oriented interface which is designed to make users tasks easier and more efficient. In the program-oriented situation, menu items generally map directly onto function calls on a one-to-one basis. The labels used in the interface reflect the underlying functions, rather than the terminology and expectations of the users.

For example, commercial databases are often set up to retrieve files based on membership or identification numbers rather than surnames, because some database systems only allow records to be retrieved based on a single key, and while identification numbers can be guaranteed unique, surnames cannot. Similarly, accounting systems often display information poorly, showing the information contained in a single record and not allowing easy comparison with other records, or with totals and averages, because they are stored in different places in the underlying program.

2.2.3.5 Consistent

Nielsen (1993) describes several different types of consistency that must all be taken into account in order to maximise the usability of a system. Consistency is important between features in a program; between programs; between the

program and the help system, manuals, and training materials; and between the program and what users already know.

Consistency requires information to be presented in the same way throughout the entire system. While this does not preclude the use of multiple terms in a searchable index (to maximize the chance of users finding the topic they are seeking), it does mean that the topics themselves should use terms which are consistent with the labelling within the program. For example, if a help file refers to a menu item on a submenu, the menu item, the submenu, and the main menu should all be referred to by the titles the user sees on the screen. These titles should also match the user's terminology rather than the terminology of the interface designer. Consistency with what users already know is more difficult to achieve, particularly when the likely set of end-users is not homogeneous.

User interfaces are frequently inconsistent in a number of different ways. A program may not be consistent with its online help systems or its hardcopy manual. For example, the help system may not use the same terminology as the interface itself, or the help may refer to an older version of the program, or to the proposed design of the program, rather than the current version.

The program may not be consistent with the operating system. For example the positioning of menu items may not be consistent with the operating system standards: "cut", "copy", and "paste" may be on a "text" menu rather than the more standard "edit" menu, or the labelling of buttons may not coincide with system standards (for example "accept" and "reject", rather than "ok" and "cancel").

Terminology used in software interfaces is frequently inconsistent with the terminology in the target domain, because it refers to the way the program operates, rather than to the way the user works. For example, a secretary trying to send his boss a document via email may be perplexed to find that his email program does not offer a "send document" option. Unless the secretary is aware of the relevant computing terminology (adding "attachments"), it may prove difficult to send the document.

2.2.3.6 Robust

Robustness has two facets. Firstly, the program itself must be internally robust: it should not crash or exhibit strange behaviour. A program with otherwise high usability which is prone to crashing will be frustrating and confusing for users, and is very likely to cause lost or wasted work. For example, instability can lead

to a program crashing just after a user performs a particular action. Human beings are accustomed to perceiving cause and effect relationships where effect follows cause closely (Philipchalk & McConnell, 1994), so that the action immediately preceding the crash is perceived as the cause of it. Users may subsequently be reluctant to perform that action, even if it is unrelated to the cause of the crash.

A software crash, even if the operating system is left unscathed, may corrupt the files being worked on at the time of the crash, sometimes irrevocably destroying information. This can happen even when users save their work regularly, particularly if the crash happens to occur during saving. In short, unstable systems mean that there are few, if any, ways in which users can reliably protect themselves from lost or corrupted data.

The second critical facet of robustness is the program's response to incorrect or unexpected behaviour on the part of the user. It is not possible to stop human beings making mistakes. Research has shown that even expert users make frequent errors in the course of their work (Norman, 1990). Interface designers need to take this into account and make errors easily reversible. Unexpected or erroneous behaviour on the part of the user should be handled gracefully – in other words, errors should not be catastrophic, or particularly intrusive. Robustness can be difficult to achieve in software without extensive testing, as it is rarely feasible to predict all possible user actions.

2.3 Technological barriers to usability

2.3.1 Lack of reliability

Reliable systems meet user expectations, and behave in a consistent, predictable manner. Lack of reliability means that actions do not produce the same response every time. In the case of a simple product such as a light bulb, it is fairly trivial to define what is meant by reliability – the bulb should light up when switched on, consistently and with a predictable intensity. There is some latitude, however, in determining how long a bulb should last in order for it to be truthfully called reliable.

Software, though, exhibits many different forms of unreliability. Failing to function at all under some conditions is clearly unreliable (although some programs are reliably unreliable, in that they usually fail under the same

conditions). However, a tool that does function, but not necessarily in the same way every time, is also unreliable, even though it may still get the job done – eventually.

Software sometimes crashes, ceasing to work entirely. It may disappear from the screen unexpectedly, or simply freeze, failing to respond to user commands. Software crashes often result in lost or corrupted data, sometimes to the extent that even saved work is lost.

Many software products contain inconsistencies and interface quirks which lead to unexpected behaviour. In most Windows 95 dialog boxes, the "cancel" button undoes any actions performed in that box. In some, however, such as the "Display Properties" dialog box, the "cancel" button sometimes performs the same function as "ok" (generally when the "apply" button has been used), and actions are not undone. Because this happens silently, it is not always easy to tell whether actions have been cancelled or not, which can lead to confusing and unexpected outcomes.

Unexpected behaviour can cause lost data or settings. In addition, it can result in considerable time spent locating the cause of the problem, as the results of the behaviour may not be recognised or even visible at the time. In consequence the origin of the problem can be difficult to trace, since any recent action could be the cause.

Sometimes inconsistencies and unexpected behaviour are the result of errors in the software, rather than poor design. These errors can be even more difficult to anticipate and allow for, as they are often dependent on the state of the system, and can appear random. They can be influenced by how much memory is being used, which other programs are also being run, whether there are any external influences such as the state of the network, and whether any local printers are behaving correctly, etc. If the system appears unpredictable, users cannot be confident of the results of their actions, even if they perform exactly the same actions each time.

Different programs, for example two different word processors, are often unable to exchange data without unwieldy and error-prone procedures to force files into the correct format. Even different versions of the same program may not be able to communicate effectively. Version 2 of a word processor may not be able to read files created by versions 3 and beyond. This can lead to problems between a user's home machine and the systems in the workplace, which may not be running the

same software, or the same versions of applications and operating system. It can also make exchanging data between colleagues hazardous and error-prone. Ultimately, users cannot be sure of being able to transfer data between applications, to other colleagues, to other machines or to other platforms.

All these problems combine to mean that the user cannot rely on being able to use programs successfully. It may be difficult, if not impossible, to achieve the desired outcome when a bug in the application is encountered, or when two different applications prove to be incompatible, or when data is irretrievably lost or corrupted. If the end result is achievable, it is often through compromises and work-arounds which are time consuming and frustrating. For example: printing a file one page at a time until the page which crashes the system is encountered, and then editing that page until the offending section has been removed.

It also becomes difficult, if not impossible, to determine the behaviour of an interface element based on other, familiar objects. Inconsistent interface behaviour can lead to incorrect, though plausible, assumptions. In some cases, opening an icon in Windows 98 requires a single click, in others a double click. Without knowing which icons are "special" (and hence require only a single click), the user has no way of knowing how many clicks a new icon requires. To make matters worse, double clicking on an icon which requires a single click may start an application twice, or have some other unintended effect if the second click is interpreted by the application as happening somewhere inside its borders.

In the short term, these usability issues lead to disruption and lost work. In the long term they may lead to false cause-and-effect relationships in users' minds, and to a perception that technology is impossible for the individual to master.

Human beings naturally try to make connections between incoming stimuli, or perceived events (Philipchalk & McConnell, 1994). In order to make sense of the world, we try to interpret things and construct cause-and-effect relationships which explain the events we perceive. When a machine or application crashes, the user typically asks "What did I do?" This can lead to false cause-and-effect relationships being built up, based on the last action that was performed before the problem occurred.

Applications and operating systems form such complex interactions that a system can become unstable and crash due to a problem that actually occurred some considerable time earlier. Worse still, problems can sometimes occur in an application layer the user never sees. Nonetheless it is common, and

understandable, for the user to assume some responsibility for the problem, and to believe that the most recent action caused the crash.

When only one person is using the machine, then if a user is to blame, the identity of that user is obvious. Because "computers don't make mistakes", it must clearly be something the user did wrong. Depending on exactly when the problem occurs, it can appear that some valid and reasonable action by the user was the cause. Not only can this lead to the avoidance of valid behaviour, it can also further erode users' confidence and faith in their own abilities and experience. Because the reason for a problem is frequently impossible to detect, as far as users can see, actions which seem valid and logical cause significant problems in some situations, for no apparent reason.

2.3.2 Lack of standardization

Lack of standardization can cause problems at many different levels. At the hardware level, few personal computers are truly identical. The use of many different components manufactured by a range of different companies can lead to differences in configuration that can in turn impact on the way the operating system and applications behave. In practice, this means that users who have come to grips with their own machines may not be able to transfer those skills to dealing with colleagues' or friends' machines. In the same office, printing to the same printer from a different machine running the same operating system may require a subtly different approach. Installing new software successfully on one machine does not mean that it will be possible to install that software successfully on the next.

At the operating system level, the lack of standardization affects the behaviour of similar objects within a single operating system. For example, it is not always clear whether a "tab" can be clicked on, and whether clicking on one tab in two rows of tabs will change the order of the rows. Although these are nominally standardized within an operating system, in practice applications frequently do not adhere to the standard, even when produced by the same software company that wrote the operating system.

The behaviour of objects on different versions of the same operating system may also be affected. For example, icons that require double clicking in one version but only require single clicking in the next, or pull down menus that only remain visible when the mouse button is held down, versus menus which stay down until

explicitly dismissed. Small individual differences may be relatively easily detected and allowed for. Collectively, though, they can be frustrating and time-consuming to deal with. The difference between menu behaviour in the various Macintosh operating systems and Windows, for example, may be considered trivial, but because it is an oft-repeated action that becomes well-known and habitual, it can be disruptive and awkward to switch systems.

At the software level, lack of standardization is an even greater problem for users. Switching between two different software packages, say a word processor and a spreadsheet, can often entail handling different commands for the same simple operations. Menu structures frequently differ from one application to the next, so that finding a particular item, such as "insert page break", may be in the "insert" menu in one application, and the "text" menu in another.

2.3.3 Inherent complexity

Computers are inherently complex. This is, in part, because they are sophisticated calculating machines which have no visible mechanism from which their functions may be deduced. They are classic "black boxes", where the user performs some action, akin to pulling a lever on a magic box, and the mysterious machinery produces an impressive result, with no clues as to how the result was obtained.

In addition to the "black box phenomenon", today's operating systems and software attempt to be general purpose devices. Instead of focussing on one task and fitting each machine perfectly to its assigned task (Norman, 1998), computers and software are designed to be able to do every possible task that marketing departments believe might sell.

Although the idea of computers has been around for over 100 years, it has been argued that computers are sufficiently different from anything else with which people are already familiar that there exists no effective conceptual model (Dijkstra, 1985). Some metaphors work well in limited circumstances, such as the notion of a computer being like a brain, a calculator, or a desktop, but such models invariably cause problems where the metaphor breaks down. For example, a "brain" would truly understand a question posed in English rather than some

artificial query language[8]. A calculator only covers the basic computational abilities of the machine. A real desktop cannot find you a file in Sweden when your desk is in Australia, and will not lose your work when the power fails.

2.3.4 Imposed complexity (featuritis)

Imposed complexity, often dubbed *creeping featuritis*, is prevalent in the computer world. Features sell. Number of features is an easy (if not meaningful) way of comparing different software packages for the same task. Reviewers, marketing departments, and developers all perpetuate this form of measurement, as it is easy to calculate, easy to publicize, and easy to compare.

New features are also an important way of selling "upgrades" – new versions of the software which are marketed as essential due to the new and impressive features they contain. Selling upgrades is a cheap and effective way to retain market share. This approach reduces the need for the research, development, and innovation that would be necessary for the successful production of entirely new software. It also allows the vendor to capitalise on the marketing that was done for the previous version, and gives access to an existing client base. Compared with starting from scratch, selling upgrades is a simple way for software vendors to stay in business.

Users also want new facilities. A large user base means large numbers of requests for different features. Even if a small percentage of the requests are attended to, that can add up to many new features. Features are often added because they are easy to implement, rather than because they are useful.

Careful design is required to organise features in a usable, logical way, but features are often added piecemeal, in a haphazard fashion (Constantine, 1995). This is frequently because features are added after the design process, or for reasons of marketing rather than usability.

More features lead to greater complexity. The more features there are in a single piece of software, the more difficult it becomes to locate particular features because a large number of facilities leads to a larger, more confusing search space each time the user needs to track down a particular feature of the program.

---

[8] Although natural language query systems are present in many current software packages, such as word processors, they are as yet very primitive, and lack robustness compared to human understanding of natural language

Shortcuts become less intuitive because the obvious shortcut keys are all taken. Control-i, control-b and control-u for italics, bold, and underline respectively, are simple shortcuts which are also reasonably mnemonic (once the concept of the control key has been mastered). However, the use of control-u for underline means that it is no longer available for "undo" (which is often an arbitrarily assigned control-z[9]). Menus become longer, and submenus proliferate, leading to numerous different "logical" locations for a single feature, always assuming that there is logic to the placement of features, rather than mere haphazard accumulation.

More features lead to greater strain on human cognitive abilities – remembering which menu contains a particular function, interpreting command names (short names are often given to complex functions), etc. Learning to use a package is therefore more difficult.

2.4 Psychological barriers to usability

2.4.1 Bad experiences with technology in general

Technology can be complex, hard to use, unreliable, and intimidating. Devices such as video recorders, photocopiers, microwaves, washing machines, and stereos are notoriously difficult to use (Norman, 1990). This leaves users in the embarrassing position of requiring help to do conceptually simple things. ("I just want to copy this one page!") To add to this discomfort, the helper is frequently someone who *does* find it simple (or has done it so many times before that she makes it look simple), reinforcing the message that the user is stupid for not being able to do it. Such helpers will often inadvertently compound this impression by using accusatory language such as "You pressed the wrong button!", implying that the user is at fault.

2.4.2 Bad experiences with computers in particular

Like other technologies, computers can be complex, hard to use, unreliable and intimidating. Working with computers often leads to lost work, wasted time and frustration, even for experienced and highly proficient users. Less experienced

---

[9] Control-z for undo, while not particularly mnemonic, is at least a widely-used standard on many systems.

users can find that the novelty and value of computers are quickly negated by the drawbacks, both physical and emotional. Computers are often set up in an unergonomic fashion, leading to uncomfortable working positions that can cause serious pain long after the work has ceased.

In addition to seeming complex and difficult to use, computers appear to make random, unpredictable errors, often due to a problem which occurred some time ago and made the operating system or application unstable (see section 2.3.1). This reinforces the idea that understanding the computer is at the very least beyond the user's abilities, if not utterly impossible. Those who do seem able to understand and tame the beast are accorded "wizard" status and treated with considerable awe. By using jargon, typing fast, and failing to explain their actions, wizards compound this impression, effectively waving a magic wand and making the problems disappear.

These experts whose hands fly over the keyboard are particularly intimidating – it seems to take them no time to fix apparently insurmountable problems. The user is forced to choose – it may be possible to see the keystrokes, **or** see what is happening on the screen **or** interpret what the wizard is saying, but it is rarely possible to do all three at once. Even if the user recognizes this and decides in advance to concentrate on a single facet of the performance, it is rarely possible to identify *and* remember exactly what is happening.

Experienced computer users forget what was difficult to learn (or may never have had the same problems) for two main reasons. Firstly, experienced computer users have, by definition, been using computers for some time, and are temporally far removed from their novice days. Secondly, the computers that experienced users learnt on may bear little or no resemblance to the computers in use today. Today's Windows expert may not have learnt to use Windows from the position of a novice, having already been familiar with other interfaces that were related, possibly ancestral, to the current version.

Computers introduce many unfamiliar concepts with which users must grapple – such as file formats (Word documents, text files, Rich Text Format, PostScript...), directories and file systems, the web and the internet. File formats are particularly troublesome, since every time a user looks at a file, it is clearly readable. Because applications hide the details of the file format from users deliberately, the concept of different formats can be difficult to grasp – users know that they can read the

file, and can't understand why others may not be able to, particularly when they have what seems like the same software (but is, perhaps, a different version).

Manuals are written by experienced computer users[10]. Support staff are experienced computer users. In addition, computer systems abound with accusatory error messages which imply the user is at fault, such as "User Application Error", and "Windows was not properly shut down..."[11]. Manuals are product-centric rather than task-centric (they describe what each menu item does, rather than how to accomplish a certain task). Manuals, support staff, and well-meaning friends assume certain "basic" capabilities, such as the mysterious ability to "double-click". Some software upgrades change the functionality of familiar packages. Users are told "This will be better" but it often proves to be worse. All of these difficulties combine to make novice computer users feel inadequate and intimidated.

2.4.3 Intimidating jargon

The world of computers is, naturally, rife with jargon. It is necessary for any new discipline to find convenient ways to express new concepts. Jargon is a useful kind of shorthand which enables experts to communicate with each other accurately and succinctly. Used carelessly, though, it can alienate those who are unfamiliar with it, excluding them from the discussion.

Jargon comes in two main forms: new and unfamiliar words, and old words used in new and unfamiliar ways. While strange new words can be disconcerting, frustrating, and difficult to deal with, familiar words can be even more confusing when used in unexpected ways.

Jargon makes computers even less accessible. What does "double click" mean? What's an "icon"? A "font"? A "style"? A "modem"? An "intranet"? A "browser"? An "applet"? What's a "window"? What's a "menu"? What's the difference between a "button" and a "tab"? What's "scrolling"? What's "dragging"?

---

[10] The technical writers who write software manuals may not be experienced in the particular application they are documenting, but they are usually experienced computer users, who may be out of touch with the knowledge and skill levels of novice computer users

[11] Which, frustratingly, often appears after Windows has crashed and become incapable of being properly shut down.

Help systems, manuals and support people often incorrectly assume familiarity with basic terms, such as "right click", "select", "screen saver", "spreadsheet", "scroll bar", etc.

Jargon used carelessly can be demoralising, as users become aware that they are expected to know terms that they do not, and as attempts to learn lead to more jargon, sometimes in a circular fashion. Seeking explanations of the jargon can be intimidating, as it requires the confession of ignorance. Experts can often be impatient with users who do not understand apparently simple (to the expert) words. Even given a patient and understanding expert, the response may inadvertently contain extra jargon, which can be even more demoralising – to ask for an explanation and receive one that is unintelligible is likely to discourage further requests.

2.4.4 Learned helplessness

Continually encountering problems that one is unable to fix (or even understand) can lead to the expectation of personal helplessness. Users cease to try new things or to solve problems because they "know" they will fail. This phenomenon is known as *learned helplessness* (Norman, 1990), because people learn that they are helpless under certain circumstances, and subsequently assume that they are helpless under all similar circumstances. As a result, they stop trying.

Continual problems with computers that require remedies the user does not understand can lead to the assumption that all problems with computers will be insoluble without expert help. If printer problems repeatedly lead to a user being unable to print without seeking technical support, it is not uncommon for the user to seek technical support before even trying to print.

Many aspects of the computing world encourage learned helplessness. Incomprehensible error messages leave the user confused and aware of their own "ignorance". Knowledgeable support staff, who fix problems with a rapid series of commands that the user does not understand, compound users' impressions that solving the problem is beyond them. Software that is unintuitive and unusable makes it difficult for users to work out how to do new things on their own, leaving them with the impression that they will always have to ask for help in order to achieve something different.

When "user-friendly" software, marketed directly at novice computer users, proves difficult to understand, users develop the impression that they are "not good with computers". Help systems which are couched in jargon, or which fail to contain the answers users are looking for, also compound learned helplessness. They teach users that attempting to solve the problem with the resources available to them is doomed to failure, so there is no point in trying (Norman, 1990).

Learned helplessness erodes user confidence and motivation. It produces users who call for assistance every time something goes wrong, or even when they see something unfamiliar on the screen.

2.4.5 Cognitive dissonance

New information which conflicts with prior knowledge causes a feeling of discomfort known as *cognitive dissonance* (Corsini, 1994). This psychological discord can cause the new information to be rejected or to be altered to fit the existing cognitive structure. Sometimes dissonant information will simply be ignored, due to "belief bias" (a tendency to discount the significance of dissonant information (Evans et al, 1983)). Facts which fit our preconceptions are more likely to be remembered and accepted than facts which conflict with our view of the world. As a result, learning something which conflicts with what is already known is particularly difficult. It will be an extra effort to process and remember it correctly.

In general, people seek to minimise cognitive dissonance, so as to feel less uncomfortable. This means either making the new information "fit" existing knowledge, or avoiding the anomalous situation and forgetting the inconsistent information.

Cognitive dissonance arises in computing when familiar concepts are represented in unfamiliar ways - for example, * used for multiplication, or = used for assignment rather than equality.

2.4.6 False consonance

In contrast to the phenomenon of cognitive dissonance, sometimes new information looks familiar when it is, in fact, subtly different. This type of situation can be referred to as *false consonance,* and can also make learning particularly difficult. The very familiarity is misleading, suggesting that there is nothing new or different to contend with, when in fact there is.

False consonance can occur when metaphors break down. For example, consider the metaphor of a file. Computer files behave in many ways which are different to the paper files to which they are allegedly analogous. Paper files do not need special applications in order to be accessed or read. A single sheet of paper can easily be removed from a paper file, and different files may be recognisable from their colour, shape, and any visible attributes of the contents, such as size or quantity. On a computer, the only visible attributes of files are the names, and sometimes the types of application with which they were created.

False consonance can raise false expectations, leading users to believe that the system works in familiar and manageable ways. Creating and then violating reasonable expectations leads users to believe that they cannot rely on their own knowledge, expectations, or assumptions. Continually behaving in an inconsistent and unpredictable manner encourages learned helplessness (see section 2.4.4), by convincing users that they are not equipped to interpret the machine's behaviour on their own.

2.4.7 Fear of breaking something

Many new computer users are afraid that, if they do something wrong, the consequences will be catastrophic, not just for their data, but for the entire machine, or even beyond, now that computers are frequently universally networked. The recent global difficulties with email viruses, such as Melissa (Melissa, 1999), have shown that innocent computer users can unintentionally bring down their friends' (or even the worlds') systems.

Even without the help of viruses, email can have unintended consequences. For example, sending large files (executables or images are common culprits) can crash some mail systems.

Although the fear of breaking the data, the machine, or the entire network is reasonable, and often based on realistic scenarios or users' own experiences, this fear is unproductive, and can impair productivity and learning.

2.5 Cognitive Ergonomics

2.5.1 Cognitive Dimensions of Notations

> *"A notation is never absolutely good, therefore, but good only in relation to certain tasks." (Green, 1989)*

In 1989, the fledgling field of Human Computer Interaction (HCI) lacked powerful generalisations, and practical techniques for analysing usability (Green, 1989). Green's paper on cognitive dimensions of notations (Green, 1989) describes one of the first techniques for an objective and rational system of analysis.

The cognitive dimensions of notations (CDs) are a collection of usability aspects of notations. Each dimension has an independent impact on user experience. Green and Blackwell (1998) describe 13 dimensions which can be used collectively to develop a profile of a notation. Each dimension is inherently neither good nor bad, since different tasks require different profiles. In addition, trade-offs are likely, where an ideal profile with respect to a particular dimension may require sacrifices in other dimensions. Different dimensions will have different priorities depending on the task to be tackled.

Cognitive dimensions deal with the notation alone, not the environment for manipulating the notation, although the environment clearly also impacts on user behaviour.

A full list of the cognitive dimensions can be found in chapter 3, where those dimensions most relevant to programming languages will be discussed in more detail.

2.5.2 Knowledge-in-the-world

Norman (1993) defines knowledge-in-the-world as knowledge which need not be remembered, since it can easily be extracted from the world around us. Knowledge-in-the-world is defined in the context of concrete appliances that can be manipulated physically, such as door handles, stereos or water taps. For example, a door which has no handle, that has a flat, hand-sized plate where the door handle would normally be, clearly indicates that it should be pushed, not pulled. In contrast, a door which has handles on both sides, but which can only be pulled from one side, is misleading.

The same principle can be applied as a useful usability metric for software user interfaces. The more information a user must remember in order to use a program effectively and efficiently, the more difficult and stressful it becomes to perform the necessary tasks.

Knowledge-in-the-world, in this context, can be defined as clues given by the user interface which indicate how to perform particular actions. In the simplest case, an interface containing a large button marked "RUN" in the centre of the screen, whatever its other merits or drawbacks may be, does not require the user to remember how to run the program. This concept will be discussed in more detail in Chapter 7.

2.6 Summary

The usability issues discussed in this chapter provide insight into the social, psychological, and technical issues confronting students learning to program for the first time. The technology students use every day is frequently not well designed, not usable, nor user friendly. Chapter 3 discusses how these usability issues apply to programming languages.

3 The programming language as interface

3.1 What is a programming language?

3.1.1 Definition

A programming language is a set of symbols and rules for writing computer programs. In other words, a programming language is a means of controlling the machine, to make it perform some particular task.

Programming languages are very diverse, ranging from macro languages within spreadsheets and word processors, to mark-up languages designed for document formatting, such as LaTEX (Lamport, 1986) and HTML (Raggett, 1998), as well as more formal and common programming languages such as C (Kernighan & Ritchie, 1988), FORTRAN (Bellamy, 1989), Java (Arnold & Gosling, 1998), or Haskell (Hudak & Fasel, 1992). Programming languages are usually text-based, although there are also visual programming languages that are primarily graphical in nature, such as Hank (Mulholland & Watt, 1998).

Programming languages have evolved rapidly from punch cards in the early days of computing, through assembly and machine languages, to high level and very high level languages, often known as 4GLs (fourth generation languages). Much of the evolution of programming languages was strongly influenced by the hardware available at the time. For example, characters in programming languages, with a few exceptions such as APL (Iverson, 1962), have long been limited to those available on conventional keyboards. Part of the reason programming languages are traditionally text-based is that early computer display screens had limited, if any, graphics capabilities. Some languages, even relatively high level languages, retain constructs and keywords that refer directly to machine architecture, such as **`car`** and **`cdr`** in LISP (Contents of Address Register and Contents of Decrement Register, respectively).

Almost every domain into which computers extend has its own domain-specific programming languages. From visual languages designed to help psychology students develop programs to assist them in their research such as Hank (Mulholland & Watt, 1998), or languages with visual tools whichencourage children to interact in creative ways with computers, such as LOGO (Papert, 1993), to purely textual languages, such as assembly language, used for maximum speed

and efficiency when programming micro-controllers, programming languages have been developed for a wide range of programmers, environments, and domains.

Domain-specific languages are generally designed to possess the features that are most useful in the intended domain. A programming language that is designed for creating simulations of electrical waveforms is likely to include built-in graph drawing functions. A language designed for creating graphical user interfaces will have its own built-in techniques for creating windows, buttons, menus and dialog boxes.

3.2 *The connection between language and interface*

Programming involves several levels of user interface. The development environment, language, and compiler/interpreter are each separate levels of interface that the user must master in order to program successfully.

3.2.1 Language

As discussed in Chapter 2, a user interface is a mechanism for communicating with, or controlling, a tool. A programming language is a way of communicating with and controlling a computer, and therefore fits the definition of a user interface.

A programming language differs from a typical user interface in that it is less immediate and more complex – a programming language allows the programmer to create a list of commands to be executed, while a graphical user interface enables the user to manipulate objects directly. In many graphical interfaces, when a button is clicked on, some visible changes happen immediately, so that the button appears to have been physically depressed – although some actions may have less immediate feedback, such as changing a setting on your window manager, or creating a new text paragraph style in a word processing document.

Only some statements in a programming language have clear and immediate feedback (mostly output statements), and the feedback is only immediate if the

language is interactive. On the other hand, visual languages contain some of the direct manipulation and feedback aspects of graphical interfaces[1].

Thus a programming language is not, in essence, different from a typical user interface. Languages may have received less attention as user interfaces because they were originally intended for advanced computing experts and the focus was on speed, efficiency, and power, rather than on usability and cognitive ergonomics.

In order to program, the student must learn to manipulate the symbols in the language correctly, in much the same way as a word processor requires a beginner to master its symbols: menus, buttons, mouse control, and shortcuts.

3.2.2 Development environment

The first thing a programmer must interact with is the development environment. This may be a simple text editor, an interactive interpreted line-by-line interface, or a fully integrated development environment.

The operating system also forms a small part of this level of interface, as it must be dealt with on some level, if only to access the development environment. In the case of graphical environments, the operating system impacts on the entire editing process, dictating the operation and sometimes arrangement of menus, menu items, buttons and scroll bars. Some interactive programming environments, such as those commonly used for Prolog (Clocksin & Mellish, 1981), where the language is interpreted line by line, have a simpler and arguably less powerful development environment which is an integral part of the interpreter.

3.2.3 Compiler/Interpreter

In addition to the language and the development environment, learning to program requires mastering the language compiler or interpreter. For the purposes of this thesis, the term "compiler" will be used collectively to denote whatever compilation or interpretation systems students must interact with, unless the difference is significant. Mastering the compiler involves learning how to pass it code, and working out how to interpret the responses given as result codes, error and warning messages, etc. The student must also know what files, if

---

[1] such as the ability to manipulate objects with the mouse, creating data flow and object dependencies without using text

any, are created by the compiler, and how to invoke a program once it has compiled successfully. Once a program can be run, any runtime feedback must also be interpreted and dealt with.

3.3 The programming language as interface

3.3.1 What programming languages should be

As user interfaces, programming languages should possess the desirable attributes of user interfaces described in section 2.2.3: useful, usable, human-centred, task-oriented, consistent, and robust. The particular ways in which these attributes apply to programming languages are described in detail in the following sections.

3.3.1.1 Useful

Like other user interfaces, programming languages need to be useful, increasing productivity and efficiency. A useful programming language contains all the capabilities necessary for the domain(s) to which it will be applied. It is well suited to its target user group, and appropriate to the problem domain. While extra features in a programming language can increase the power and flexibility of the language, they can also slow down compilation, increase the complexity of the compiler, and make the language harder to use. There are many such trade-offs, and the appropriate design decisions depend on the intended users of the language (novices versus experts, domain specialists versus expert programmers, etc), and the target problem domain.

For example, Hank (Mulholland & Watt, 1998), a programming language intended for psychology students with little or no programming background, emphasizes the sort of statistical computations that psychologists are likely to need. Hank also uses a visual format that allows the programmer to fill in the required constructs, rather than requiring that the syntax of the language be memorized.

This technique has been very effective for use in Hank's intended environment (Collins & Fung, 1999), but probably would not be as effective for the use of, say, engineers programming microcomputers. For example, an engineer programming a chip with only 256 bytes of on-chip RAM must ensure that the code is as small and efficient as possible, particularly if the application is time critical. Here the emphasis shifts away from usability and ease of programming and on to

producing code that will be small when compiled into assembly code, and therefore will fit into the on-chip memory.

3.3.1.2 Usable

The usability of a programming language inevitably impacts on the productivity and efficiency of the programmer. As a programming language has two parts – the syntax/semantics of the language itself, and the usability of the development environment – the usability of both these parts impacts on the overall usability of the language. A truly usable programming language should facilitate both program development and program maintenance, balancing readability with "writeability".

As is the case with all user interfaces, the only way to be sure of a programming language's usability is to test it using real programmers, and in particular on the types of programmers who will be expected to use the language in real life. Spreadsheet macro languages should therefore be tested using spreadsheet users, introductory programming languages will require a process of testing and revision using real students without programming experience, a programming language for mathematicians should be tested using mathematicians, and so on.

3.3.1.3 Human-centred

Just as a usable application must be designed around the needs of the user, a usable programming language should be designed around the needs of the programmer, rather than the needs of the machine. While there may need to be trade-offs between making a language efficient and making it human-centred, the needs of the programmers should be considered at some level.

For example, in languages such as C and C++, array indices start at 0, rather than 1, as is more traditional in mathematics. This is a very efficient technique, as array variables in C are pointers to the start of the array, and the index is actually an offset into the array. For programmers, however, it can be difficult to remember (and to take into account in all appropriate calculations) that a 10 element array actually has indices 0...9, rather than 1...10.

3.3.1.4 Task/Domain-oriented

No programming language can be all things to all programmers and all domains. Designers of domain-oriented programming languages are able to focus on the

particular constraints that apply in their own domain, and may be able to make their languages smaller and more powerful as a result. For example, a programming language designed purely for teaching introductory programming to computer science students probably does not require built-in advanced statistics functions, while a language designed for psychologists would be more useful if it made such functions easily accessible.

Like software that is designed to be able to perform as many tasks as possible, languages that are designed for a wide range of possible domains, rather than one specific domain, tend to be over-featured, and hence harder to use (Norman, 1998). The more features a language possesses, the more syntax and semantics must be mastered before the language can be used successfully. Larger languages must, of necessity, possess a larger cognitive search-space, so that it can be harder for a programmer to find the appropriate construct for each situation. Books, tutorials and on-line help systems must all be larger and more complicated in order to deal successfully with a larger and more complicated language.

3.3.1.5 Consistent

In the same way that applications must be consistent in numerous ways, a programming language should also aim for several different types of consistency.

- Consistency with what programmers already know. This enables programmers to capitalise on their existing knowledge about programming, programming languages, and computers.
- Consistency within the language's own constructs. Constructs should behave in a predictable fashion. Once a programmer has learned how to use one language construct, other constructs should behave in the same manner as far as possible. For example, if a language contains two string manipulation functions (such as concatenation and copy) that both take a source string and a destination string as parameters, the order of the strings should be the same in both functions – eg:

```
copyString(source,destination)
catString(source,destination).
```

- Consistency with domain knowledge and terminology. Domain specific languages should use terminology and concepts that are appropriate to their domain.

- Consistency with other programming languages, where it does not conflict with other forms of consistency, or other more significant aspects of usability. Consistency with other programming languages is useful, but does not take priority over consistency with what the target users or programmers already know, or over consistency with domain knowledge and terminology.

Unfortunately, consistency is not always easily defined. Precisely what the design should be consistent with is open to argument. Ideally a language should be consistent with the way its users think. Unfortunately, users' thought processes are notoriously difficult to model (Mayer, 1992), and when there is a large and varied user group, their ways of thinking and working are likely to vary substantially.

For example, the designers of the Turing programming language (Holt et al, 1988) sought consistency in syntax where there were similarities in underlying semantics. The notation **A(B)** possesses several distinct meanings, including "return the value found at index **B** of array **A**", "return the value of function **A** called with parameter **B**", and "return the value found in collection **A** at position **B**". This syntax was chosen deliberately (Holt et al, 1988) to highlight the similarity between each construct - they each return a value from **A** that is somehow related to **B**.

Unfortunately, this type of consistency can cause problems for student programmers, who are unable to determine the nature of **A** and **B** from context, and must instead refer back to the beginning of the program where **A** and **B** are declared. In addition, this type of syntactic linking of constructs that, in fact, have distinctly different semantics and uses, can lead to difficulty when a student tries to determine the differences between the constructs.

3.3.1.6 Robust

Programming languages should both be robust in themselves, in terms of the available compilation and development tools, and they should facilitate the production of robust software. This means that the constructs of the language should support reliable error detection. A robust programming language is also tolerant of er · in other words, small slips on the part of the programmer, such as inadvertently using the wrong operator, do not have disastrous consequences for the program.

3.3.2 What programming languages usually are

3.3.2.1 Easy to process (compile/interpret)

Parsing a programming language and producing machine code from it can be a complex and difficult process. A language that resembles machine code closely, for example one whose commands map directly onto machine code commands, is much easier to process, since it only requires direct translation. Higher level languages tend to have more complex constructs that do not translate easily into machine code. As a result, it is often more difficult to write compilers for these languages. Complex compilers are also likely to be somewhat slower than simpler compilers, and hence cause the development cycle (write, compile, test, modify, compile, test, etc) to be slower.

Language designers have not only their end users to consider, but also those who will be required to write and maintain compilers for the language. Often, language designers are the first to write compilers for their languages. It is not surprising, then, that languages are often designed to be easy to compile, or at least easy to write compilers for.

3.3.2.2 Hard to use

Languages that are easy for a computer to process are not necessarily easy for humans to read and write. The goals of ease of compilation, speed and efficiency are not necessarily compatible with ease of use. As with much of today's software, difficult language constructs are often propagated for historical reasons rather than out of current necessity. Sometimes these constructs are deliberately included in order to maintain compatibility with an existing language, sometimes they are a product of the language designers' familiarity and expertise with existing languages, to the point where once-confusing constructs seem familiar and logical. For example, experienced C programmers are familiar and comfortable with the idea that array indices start at 0, rather than 1. In consequence, the idea of arrays starting at zero propagated into Java[2].

---

[2] There are also other reasons for starting indices at zero, for example, some algorithms, such as hashing, are easier to code with indices which start at zero.

3.3.2.3 *Hardware-centred*

For a language to be maximally efficient, hardware-centred design is a necessary evil. Language constructs that are hardware-centred are designed to take maximum advantage of hardware features. For example, the separation of numbers into floating point and integer types is effective because it takes advantage of the fact that computer chips typically have floating point and integer mathematics implemented separately, and integer maths is often faster. It is more efficient to separate out the integer maths that can be done fast and efficiently, rather than doing all numerical calculations together, and thus losing time on integer calculations.

3.3.2.4 *Paradigm oriented*

A programming paradigm is a style of programming. The paradigm has a substantial impact on the form of a language that is designed to adhere to it. The paradigm of a language affects the type of constructs available (for example, in its strictest form the functional paradigm does not have assignment statements), the way constructs are used (for example, the object oriented paradigm treats variables as objects that have their own defined behaviours), and the way the entire program is constructed. Each different programming paradigm requires a different way of looking at a problem: functional programming focuses on the actions required to solve a problem; object oriented programming deals with the entities involved in a problem; constraint programming focuses on the requirements that define a problem; logic programming deals with what facts are known about a problem.

In an attempt to achieve paradigmatic purity, programming language designers sometimes make decisions that can lead to awkward features in a language. For example, pure functional languages, such as Miranda (Turner, 1985), have no input or output capabilities, because I/O has the side effect of altering the input and output streams of the program, and pure functional languages have no side effects.

3.3.3 Cognitive dimensions of programming languages

As described in Chapter 2 (section 2.5.1), cognitive dimensions provide an important list of points to be aware of during the design of any notation. The full

list of cognitive dimensions is shown in Figure 3.1. The most relevant cognitive dimensions to introductory programming languages are discussed below.

"Closeness of mapping" deals with the domain for which the notation is to be used – how well does the notation represent the domain for which it is intended?

There are many forms of "consistency", and programming languages require several of those forms in order to be maximally usable. Consistency requires that similar semantics are expressed in similar syntactic forms. "Similar", however, is somewhat subjective in this context. This issue will be discussed further in chapters 5 and 7.

"Diffuseness", or "verbosity", is an example of a dimension that is neither good nor bad, but dependent on the target domain. Different levels of verbosity are appropriate for different applications, and in different situations. For example, a command that must be typed frequently (and hence is easy to commit to memory, by repetition), but is rarely read, would be more usefully concise and easy to type, whereas a command that is rarely used (and hence easy to forget) could usefully be made more verbose and expressive.

Some programming constructs and concepts, such as starting array indices at 0 rather than 1, seem to invite and encourage errors, making them almost inevitable. This is an example of an error-prone construct.

"Hard mental operations" refers to cognitively challenging constructs. For example, a programming language that requires all numbers to be represented in hexadecimal format requires hard mental operations of the programmer who wishes to write a calendar program, since all dates will need to be converted from decimal to hexadecimal and back. Of course, the level of difficulty of this mental operation is entirely dependent on the programmer's familiarity with hexadecimal, and how much experience he or she has in converting between decimal and hexadecimal numbers. This is another example of how the ideal cognitive dimensions for an application will vary depending on the application and its intended users.

"Role-expressiveness" refers to the degree to which information about the function of a construct can be inferred from its structure or form. In other words, is it easy to determine what a component of a notation actually does, just from the component itself – does the word, or symbol, indicate its function, or role? In a programming language, high role-expressiveness allows for greater readability,

Cognitive Dimension	Description
Abstraction Gradient	types and availability of abstraction mechanisms
Closeness of mapping	closeness of representation to domain
Consistency	similar semantics are expressed in similar syntactic forms
Diffuseness	verbosity of language
Error-proneness	notation invites mistakes
Hard mental operations	high demand on cognitive resources
Hidden dependencies	important links between entities are not visible
Premature commitment	constraints on the order of doing things
Progressive evaluation	work-to-date can be checked at any time
Provisionality	degree of commitment to actions or marks
Role-expressiveness	the purpose of a component is readily inferred
Secondary notation	extra information in means other than formal syntax
Viscosity	resistance to change
Visibility	ability to view components easily

Figure 3.1 Full list of Cognitive Dimensions of Notations (taken from Green & Blackwell, 1998)

and simple constructs that are easy to remember and to use. For example, the Ada keywords **`in`** and **`out`**, used for passing parameters to subroutines, are highly role-expressive. An **`in`** parameter is used for passing a value *in* to a subroutine, and an **`out`** parameter is used for passing a value *out* of a subroutine.

"Secondary notation" provides extra information using notation that is not a part of the formal syntax. One example of secondary notation in programming languages is the use of code indentation in addition to explicit block delimiters. Another is the provision of comments[3]. Secondary notation can be an extremely useful tool, and can be as simple as using white space to group lines of code into

[3] Although comment markers (such as `//` in C++, or `{ }` in Pascal) are part of the formal syntax, comments themselves are not.

related chunks. In a text-based programming language, most secondary notation is in the hands of the programmer rather than the language designer, although it may be provided by the environment, in the form of syntax directed editing and annotation capabilities.

3.4 Summary

Viewing programming languages as user interfaces allows usability principles and analysis techniques to be applied both to the design of new programming languages and the analysis of existing ones. Any analysis on usability terms, however, must first be put in context - as discussed in Chapter 2, usability of an application is logically discussed in terms of a specific use for a specific group of users. Students learning programming have particular needs, which vary quite dramatically from the needs of expert programmers. Chapter 4 discusses the problems with learning to program, leading into Chapter 5, which analyses some existing languages from a usability standpoint.

4 The trouble with learning to program

The learning and teaching of programming has engendered an extensive literature. At the computer science end of the literature spectrum are discussions of introductory programming courses, and their experiences with particular languages (Allen et al, 1996), software (Kölling, 1999b), or teaching techniques (Geitz, 1994). Such accounts of educational experience are undoubtedly useful as records of techniques that have been tried, and the successes and failures attendant upon each, but they do not, in general, provide fodder for comparison of different techniques, due to the ethical problems of conducting comparative experiments using live courses. At best, comparison takes the form of anecdotal evidence of observed differences between the course described and previous incarnations of the course.

At the psychological end of the spectrum there are empirical studies comparing individual programming constructs (Soloway, Bonar, & Ehrlich, 1989; Sime, Green, & Guest, 1973), analyses of student errors (Eisenstadt & Lewis, 1992; du Boulay & Matthew, 1984), and discussions of some of the problems experienced by novice programmers (du Boulay, 1989; Spohrer & Soloway, 1989).

There are many theories about what makes learning to program so difficult (Bruckman & Edwards, 1999; MIver & Conway, 1996; Brilliant & Wiseman, 1996; Brusilovsky et al, 1994; Murnane, 1993; Eisenstadt & Lewis,1992; du Boulay, 1989; Bonar & Soloway 1985; Brooks, 1977). Although theories abound, and courses are designed and redesigned in an attempt to circumvent the difficulties encountered, there remains surprisingly little empirical evidence to support the theories. New languages and tools (Kölling, 1999a; Meertens, 1981) are often designed based on experiences with existing languages, but without reference to learning theory or research into the psychology of programming.

Section 4.1 discusses the evidence that learning to program is difficult, and looks at some of the issues that must be considered when designing an introductory programming course. Section 4.2 deals with cognition and how people think, learn, and solve problems. Section 4.3 examines the cognitive requirements of learning computer programming, while section 4.4 looks at some of the ways introductory programming is currently taught at the tertiary level. Section 4.5 discusses the information still missing from the language debate.

4.1 Is there trouble?

4.1.1 Evidence

Du Boulay (1989) describes five main areas of difficulty with learning to program. The first, he terms *orientation* – what *is* programming, and what is it good for? Students sometimes have difficulty coming to grips with the various forms of programming, from using spreadsheets and word processing macro languages through to more formal programming languages such as Java. As programming becomes a widespread, common feature of popular software packages, the boundaries of "end user" computing and programming are blurring.

The second area of difficulty is the *notional machine* – how does the computer work? What form does the virtual machine take? Hence what sort of commands can it be expected to understand? How should those commands be issued? The mental models that students construct are crucial to their understanding of each new concept to which they are introduced. In addition, having a poor mental model can lead students to develop poor learning strategies and become unmotivated and discouraged (Kessler & Anderson, 1986).

The third category of difficulty du Boulay (1989) describes is trouble with the *notation* – problems arising from the programming language itself, including the syntax and semantics of the language. These include constructs that are error-prone (see section 3.3.3), that lack role-expressiveness, or that involve hard mental operations.

The fourth describes an essential element of the transition from novice to expert – the acquisition of *structures*, *cliches*, or *chunks* that can be slotted into a solution. Example of structures are: the computation of a sum using a loop; simple searching algorithms, and swap functions. Experts have a library of such structures at their disposal, developed through practice. Novices lack such a resource, and it hampers their efforts to solve problems.

The fifth area of difficulty is often neglected in programming courses, despite its profound impact. Du Boulay terms this area *pragmatics* – the auxiliary skills necessary to programming, from dealing with the operating system interface of the computer to editing, compiling and debugging a program. Students sometimes have difficulty coming to terms with the mechanics of the development environment, before they even start to deal with the programming language itself.

The levels of interface discussed in section 3.2 must all be dealt with in order to program successfully.

Du Boulay (1989) also notes that students are sometimes unaware of the appropriate response to an error, so they may overreact to a minor syntax error, for example, by throwing away an entire program, or by rebooting the machine.

Perkins et al (1989) studied the strategies of students learning to program. Like du Boulay, they found that negative experiences cause some students to give up. They describe such students as "stoppers" – stoppers stop trying when they encounter a problem. Significantly, Perkins et al (1989) found that this tendency could be countered by giving students a small positive experience to encourage them. This suggests that early experiences in programming are important when students are formulating their programming technique – positive experiences and successes achieved while learning could lead to better programming habits and more appropriate problem solving techniques.

The other class of students found was the "movers". Movers tend to keep trying different strategies for making a program work, although not always wisely (movers sometimes randomly change a program without understanding where the problem is).

Putnam (1986) studied high school students learning to program in BASIC. He found that that preprogramming knowledge incorrectly transferred to programming causes many errors. Students tend to make up for their incomplete knowledge of BASIC by assuming that the machine has natural language capabilities, incorrectly making inferences about the use of keywords based on their English meanings.

This study, together with Bruckman and Edwards (1999), suggests that these natural language errors are likely to occur regardless of how close the programming language is to a natural language. Bruckman and Edwards found that natural language errors when using a natural language style programming language made up a small proportion of total errors (10.6%), and of those, 61.8% were corrected by students without outside assistance. They conclude that natural language errors are not sufficient reason to avoid taking advantage of natural language knowledge in programming language design. Indeed, it may be that making the language *more* natural, rather than less, might help to minimise natural language errors, by making students' assumptions more likely to be valid.

Putnam (1986) also notes that students sometimes use meaningful variable names in the mistaken belief (or hope) that the computer will automatically make the variable contain the appropriate value, for example, by calling a variable **smallest** they assume it will contain the smallest value. Another point noted about BASIC is the lack of directionality of the statement **Let A = B**. Does **A** receive **B**'s value, or vice versa?

The significance of mental models in learning programming has been highlighted by the findings of Kessler and Anderson (1986), who undertook a study of students' errors and understanding when learning recursion and iteration. They found that students have poor mental models of the functionality of programming, which in turn causes them to develop poor learning strategies when learning to program. Van Someren (1990) concurs, looking at errors made by beginners learning Prolog. A clear understanding of the underlying virtual machine is critical for learning programming. For example, the **A=1 B=A** problem[1] arises from referring to variables as boxes, which does not accurately reflect the mechanisms of the underlying machine. Van Someren (1990) also found that unification and depth-first search in Prolog complicate the virtual machine substantially, making the language more difficult for novices to grasp.

Van Someren's (1990) discussion of the importance of the virtual machine is supported by the results of Dyck and Mayer (1985), who compared student response time for comprehension of statements in BASIC with that for comparable statements in English. Subjects tested for comprehension of BASIC statements were required to know BASIC. Subjects tested for comprehension of English statements were required to have no knowledge of BASIC. The study found a high correlation between English and BASIC performance - response time for comprehension of BASIC statements for students who knew BASIC was strongly correlated with response time for comprehension of English statements by students who did not know BASIC. Correlation was also found between macrostructure (the total number of statements in the program), the microstructure of each statement (the number of actions the statement performs), and the response time for both English and BASIC statements. Students took longer to comprehend more complex statements that performed multiple actions,

---

[1] where students assume that **A** is now empty, because the value that was in **A** has been physically transferred to **B**

regardless of whether they were in English or BASIC. This suggests that more complex statements and implicit behaviours, where multiple actions take place as a result of a single statement, are likely to be more confusing to students. This is consistent with the findings of Van Someren (1990), described above.

4.1.2 Choice of language

The selection of a programming language for introductory programming courses is often politically motivated (McIver and Conway, 1996). There is pressure to teach industry relevant[2] languages from day one. Many institutions that have switched to teaching an industry relevant language such as C++, Ada, or Java as their introductory language report that students respond well to the idea of learning a language that they can use in industry directly (Allen, Grant, & Smith, 1996). Time spent on learning a language that will not be immediately and obviously useful outside the classroom or laboratory can be resented and perceived as wasted, even if fundamental concepts are being well conveyed using that language (Conway, 1993a).

As a result of these perceptions, overall student motivation may well be lower in a course using a special purpose teaching language than one using an industry relevant language. It is possible, although it has not been tested, that those students who prefer using an industry relevant language are distinct from those students who have no computing or programming experience and who find learning to program a difficult, stressful and intimidating process. The selection of a language that maximizes motivation for students whose first concern is industry relevance may not be compatible with the aim of maximizing motivation for those students in the latter group, who may require a simpler, friendlier language and environment in order to achieve the best learning outcomes.

The usability of the first programming language a student encounters can impact on motivation and enthusiasm in many ways. If the language is misleading, in that the intuitive interpretation of the syntax leads students to false conclusions, they may learn to mistrust their own judgement and intuition, making the task of learning to program more difficult and less enjoyable. If the language is difficult to read and hard for students to interpret, that may also impact on motivation,

---

[2] Languages that are commonly used in industry, as opposed to so-called "toy" languages used only for teaching concepts.

leading those students who lack confidence to believe that they are not sufficiently experienced or talented to be successful in learning to program.

4.1.3 Choice of tools

The choice of tools for teaching programming can be crucial to the success of an introductory programming course. Important tools in the programming course include compilers, development environments, text books, examples, hardware, operating system, and demonstration tools used in lectures and laboratory classes. Some of the questions to be considered when choosing tools include the following:

- Is the tool intended for novices?
- Is it industry relevant?
- Is it easy to use?
- Is it reliable?
- Is it functional over a network?
- Can students use it at home?

Du Boulay and Matthew (1984) discuss the problems inherent in compilers that are built for one purpose but used for another. In particular, compilers are frequently built to produce efficient executables for experienced programmers, rather than for training novices. Such compilers are often guilty of a mismatch between reported information and actual errors (such as "**`unexpected end of program`**" being reported because a comment was not closed somewhere earlier in the program.) These error messages direct the students' attention to the wrong place, and the wrong kind of error. Du Boulay and Matthew (1984) describe a prototype error checker for Pascal which they point out broke no new ground, even in 1984. The technology required for sophisticated and accurate error checking has existed for some time. Developing dedicated development environments for teaching programming is expensive, however, and not nearly as profitable as developing commercial tools. Many of the same arguments against using real world languages to teach programming (namely that they are unnecessarily powerful and complicated, and that they are designed for experts, not for teaching novices) also apply to using real world tools, with similar consequences.

The software used in programming courses typically suffers from the same usability problems as any other software. Integrated Development Environments

(IDEs) in particular are designed for industry, not school, and have large numbers of features, leading to high levels of complexity. Most compilers have a large range of options that can be configured, which can make it difficult to find important options when changes do need to be made. This high degree of configurability often places a heavy burden on the programmer, for which beginners are ill-prepared.

For instance, some environments require the size of the memory model to be explicitly set by the programmer. In some systems, this can quickly become a critical issue, particularly with ill-chosen default settings. A novice programmer, who does not yet understand what a memory model is, will find it difficult to determine the best memory model to use. In addition, problems caused by an poorly chosen memory model may be difficult to identify and correct, since they generally lead to unexpected behaviour (such as variables being overwritten), rather than meaningful error messages.

Producing effective error messages is notoriously difficult (but far from impossible – see Du Boulay and Matthew (1984)). Error messages suffer from the same usability problems as software, in that they are designed by experts who are thoroughly familiar with the subject area, but often targeted at novices with no experience or understanding of the necessary topics. While an error such as "**`L-value required on left hand side of assignment`**" may be helpful and meaningful to a programmer who is familiar with the term "**`L-value`**", it is not much use to a novice who has not yet encountered, or understood, the concept. Similarly, the error message "**`<<function>>`**" may be meaningful to an experienced Haskell programmer, but it does not convey much useful information to a beginner[3].

Many of the usability problems of the software in use in introductory programming courses are compounded by the use of applications that are designed for expert programmers. The political pressure to teach so called "real world" languages is intense, and when teaching such a language, it makes sense to use a popular commercial compiler, in order to equip students with as many commercially marketable skills as possible. This leads to software (that may be

---

[3] The return result "`<<function>>`" means that the return value is another function call that cannot be evaluated

quite usable for expert programmers) being inflicted upon beginners who lack the skills and knowledge to manipulate it effectively.

There is a vast gap between tools that best facilitate the expert programmer's daily work, and those that are well suited to the particular challenges faced by students learning to program for the first time. The fact that teachers must, of necessity, be experts once again counts against them, this time because their favourite tools are likely to be inappropriate for teaching purposes. In order to choose the right language for teaching, the right development environment for the language, and the best textbook for both language and underlying concepts, teachers must be able to put themselves in the shoes of their students, remembering what it was like to face all these tools for the first time.

As well as the tools that students use directly, the choice of teaching tools in the lecture theatre is also important and equally challenging. Some tools, such as HyperLecture (Conway, 1993b) support the display of animated versions of the code. Such tools make changes to the value of a variable immediately visible, and display data structures such as linked lists graphically. This is intended to foster the development of an appropriate mental model of the structure. These tools, while useful, require technical support in the lecture theatre, and can be difficult to fit into a lecturer's personal teaching style.

As well as the hardware support for teaching in the lecture theatre, lecturers, demonstrators, and tutors need to devise metaphors and analogies that help the students to create the right cognitive models of their code and algorithms. Once again it is necessary to start with something that all the students are familiar with, so an example from everyday life is ideal, if it fits the topic in hand. Unfortunately the wide cultural mix in many classes means that everyday life is substantially different from student to student, and even a simple example such as making a sandwich may not translate. In addition, the best of metaphors breaks down eventually, so it is necessary to determine how far the example may safely be taken, and to ensure that the breakdown of the metaphor does not cause an associated breakdown of the fledgling cognitive models that students are developing.

4.2 Cognition

4.2.1 Requirements for learning

The basic requirement for learning is often taken for granted: in order to learn effectively, students need a safe environment that they can explore with confidence. Some care is necessary to turn computers into such an environment, especially if students have had disastrous experiences with them in the past, such as losing hours of work through a system crash, or by pushing the wrong button and being unable to recover from the error.

Once this basic need is met, there are more complex requirements for meaningful learning to take place. Mayer (1989) describes meaningful learning as a process by which new information is connected to existing knowledge – in other words, learners learn best when they can tie incoming information to what they already know. Existing knowledge is the "cognitive framework" to which we attach new, but related, ideas. This process of linking is important. In its absence, students "rote learn", which, in contrast to meaningful learning, occurs when students memorize information, but fail to understand it.

Students can reproduce rote-learned knowledge, but they may not be able to use it effectively. Rote learning also makes it substantially more difficult for students to transfer their knowledge. Where meaningful learning has taken place, solutions to known problems can more readily be adapted for new, but related, problems. With rote learning, related problems are less likely to be identified as related, and solutions cannot readily be adapted to new situations, because the solutions are memorised rather than understood.

Norman (1993) points out that accretion - the accumulation of facts - is easy, painless, and efficient when the proper conceptual framework is already in place. That is, adding to existing knowledge is simple if the proper conceptual framework already exists. New information that conflicts with what is already known disrupts this process. Every item of new data that conflicts with existing knowledge impedes accretion.

Norman (1993) describes reflective and experiential mode cognition. Experts spend more time in "experiential mode" when problem solving, where action is automatic and instinctive. Novices spend more time in "reflective mode", where problem-solving is a prolonged and laborious effort. The process of "tuning" allows the novice to assimilate new knowledge, and begin the transition from

reflective to experiential mode. The goal of learning can be characterized as moving the learner from reflective to experiential cognition. New knowledge that conflicts with existing knowledge anchors the novice in reflective mode, hampering the tuning process.

Both Norman (1993) and Reason (1990) agree that experts work with larger chunks of information than novices. For example, in programming, a novice may concentrate on small details of syntax as part of the problem-solving strategy, while an expert will match a problem with similar problems, and slot in, say, a particular sorting algorithm. If the number and complexity of syntactic details a student must remember can be minimised, thus minimising the cognitive overhead, students may move more easily into experiential cognition.

Mental models have been shown to be crucial to learning and understanding – students who have appropriate mental models are likely to do better on tests, and show greater understanding than students with poor or inappropriate mental models (Mayer, 1992). Mayer (1983) found that the provision of pictorial models alongside explanatory text improved students' performance dramatically (by an average of over 60 percent) over the explanatory text alone (although not for all students, as top students showed better performance without the models).

4.2.2 Errors

Reason (1990) divides errors into two categories: "slips and lapses", and "mistakes". Slips and lapses occur in situations where the correct action is known, and the plan, or intention is correct, but an attentional check fails, causing a mistake to be made. Attentional checks are required to keep plans on track, and to ensure that the correct action is carried out. They fail when attention is distracted - when a person is thinking about something else, or when an external distraction occurs, such as the phone ringing. Slips and lapses occur during skill based performance, which consists of routine actions carried out in a familiar environment. Occasional checks are required to maintain correct behaviour during skill based performance, and it is when these checks are disrupted that slips occur.

Mistakes can be divided into two further categories: rule based errors and knowledge based errors. Rule based errors occur during rule based performance, where situations are pattern matched against the individual's experience, and analogous situations are used to form appropriate solutions. Errors occur here

when a situation is incorrectly matched, due to a difference that has either not been noted, or has been disregarded inappropriately. Knowledge based errors occur during knowledge based performance, when the problem at hand is outside the individual's experience, and a new solution must be designed from scratch. Knowledge based errors arise from incomplete or inaccurate knowledge.

Bonar and Soloway (1989) studied errors made by beginners learning to program, and proposed a model that explains novice behaviour. In particular, they examined the link between "preprogramming knowledge", or what the students knew before learning programming, and the kinds of misconceptions students have. The proposed model suggests that natural language knowledge accounts for many of the misconceptions students experience when learning to program. For example, Bonar and Soloway found that some students incorrectly assume that a **`while`** loop in Pascal terminates as soon as the condition becomes false[4], rather than when the condition is false at the beginning of the loop.

Bonar and Soloway also discuss "bug generators", which they describe as *"patches generated in response to an impasse reached by the novice while developing a program."* When a student encounters a problem she doesn't know how to solve, the solutions attempted often cause more errors. These bug generators are often natural-language related. In the absence of sufficient knowledge of the programming language to solve the problem, students extrapolate from their knowledge of natural language in an attempt to move beyond the impasse. This is consistent with the behaviour of "movers" described by Perkins et al (1989) (see section 4.1.1).

Eisenstadt and Lewis (1992) conducted an analysis of errors made by novice programmers using SOLO[5]. The study proposes causes for each type of error, although it does suggest that errors may have multiple causes. Eisenstadt and Lewis compare their error analysis with Du Boulay's (1979) analysis of errors in LOGO, and find the same primary culprits in the same relative ordering – spelling errors, wrong number of arguments, no line number, and call to an undefined

---

[4] Because the language implies it : **`while condition do stuff`**. It is natural to assume that when condition becomes false, the loop is immediately terminated. The statement more accurately translates as **`if condition is true now, do stuff, then go back to the start and test again`**.

procedure. They suggest two approaches to solving these problems – trapping errors in an intelligent fashion after they have been made, or pre-empting the possible occurrence of errors using syntax-directed editing tools *before* the errors have been made.

It should be noted that these solutions are predicated on the notion that the syntax of the language is fixed and cannot be changed to eliminate some errors. Without that constraint, an obvious third approach is to redesign the syntax to eliminate common problems. For example, the third type of error, missing line numbers, can clearly be dispensed with by eliminating the requirement for line numbers (which LOGO has subsequently done), perhaps including them in the editing environment instead.

Eisenstadt and Lewis suggest that eliminating unnecessary sources of confusion *"fosters more acceptance and participation by marginally motivated novices"*. They put the case for a programming environment that uses "pre-emptive design" to create an environment in which it is not possible to make syntax errors. The SOLO interpreter already does some error-correcting inference. For example, typing errors generate warnings such as "**`When you typed FIOD did you mean FIDO?`**".

Spohrer and Soloway (1989) examined the popular belief that misunderstandings of language constructs are the primary source of errors in novice programming. They found that a few types of errors account for a high proportion of total errors, but that misapplication or misunderstanding of language constructs were not represented among the most common types of errors. Instead, they argue that the majority of bugs are "*plan composition problems*", or difficulties putting the pieces of the program together correctly. It is significant that Spohrer and Soloway focussed exclusively on bugs in syntactically correct programs, so that syntactic problems with language constructs were not considered. Bugs were categorized as "definitely due to a construct misunderstanding", "maybe", and "definitely not". Over the three exercises studied, 9% of errors were definitely caused by construct misunderstandings, 40% were categorized as "maybe", and 52% as "definitely not". Spohrer, Soloway, & Pope (1985) also examine novice programming errors. However, they focus on "planning errors", no syntax errors are considered.

---

[5] SOLO is a programming language developed at Open University in the UK for teaching programming to psychology students

4.2.3 Problem solving

Although problem solving is difficult, it is something at which we get considerable practice in our day to day life. "How will I get to work now that the car is broken down?" "How much food can I buy before I need to go to the bank?" "Where shall I hide this body?"

Instructing or teaching is more difficult. To begin with, instructing someone (or something) on how to solve a problem requires not that the problem be solved, but that a solution be described (Dalbey & Linn, 1985). While we may be adept at problem solving, delineating a solution is considerably more difficult. Human problem solving is full of assumptions and intuitive leaps – we often do not actually know *how* we solved a problem. Even simple problems can be challenging to describe in precise detail – for example it is not often necessary to describe how to find the largest of ten numbers, rather than simple finding it directly.

In order to give instructions on how to solve a problem, some common ground must first be found between instructor and pupil. Background knowledge is an important factor in the assimilation of new information (Mayer, 1989). As a result, effective instruction and communication require some awareness of the existing knowledge that may be assumed. For example, when describing the best way to reach a particular destination, it helps to know whether the listener knows the area and can recognise important landmarks. It is not helpful to suggest a left turn at the Civic Centre if the listener does not recognise the Civic Centre as a landmark.

Instructing, or programming, machines is made particularly difficult by the necessity for a high and unfamiliar level of precision. When instructing humans, assumptions can be made, and steps can be skipped and interpolated. While this can lead to errors in person to person communication, it also makes communication robust in the presence of errors. For example, when giving directions it is not uncommon for a person to leave out a step without realizing it, and a person who has taken a wrong turn is probably capable of detecting the error eventually, and correcting for it, given sufficient time. It is not necessary for the person giving directions to provide directions for every possible circumstance. A computer given incorrect instructions, however, cannot usually correct for them and calculate the correct response.

In every day life we make assumptions without being aware of them. We assume shared knowledge and skills. We assume interpretive abilities. We assume certain basic physical abilities and skills, unless it is clearly inappropriate. For example, when giving directions, we assume that the recipients of the directions can see where they are going, unless we know that one of the recipients is vision impaired.

When instructing computers rather than humans, the range of valid assumptions is dramatically different. Because we are unaware of many of the assumptions we make, recognising them and eradicating them from our instructions is particularly difficult. In order to give instructions effectively, a common base of shared knowledge must be established. To apply this to programming, a programmer must know exactly what the machine "knows". In other words, a detailed knowledge of the instructions a computer can interpret is necessary. The programmer must remember not only what instructions are valid, but how to use and combine those instructions effectively. This becomes the base of shared knowledge upon which the instructions are built.

4.3 Programming-Specific problems

4.3.1 Cognitive demands of programming

Learning to program requires students to hold a wide range of information in working memory. In addition to the details of syntax and semantics specific to the programming language being used, students must determine how to solve each problem, and be able to differentiate between *solving the problem* and *specifying the solution*.

Pennington (1987) characterises the types of programming knowledge used by experts in order to comprehend programs written by others:

- data flow – what happens to the data between input and output?
- operations – low level individual actions, eg "set x to 0"
- control flow – sequence of execution
- state – state of the system at various points – effectively the impact of operations, and
- function – what does the program actually do? what do bits of the program do?

Novices must acquire all of these different types of programming knowledge, at the same time as they accumulate syntactic and semantic knowledge about the programming language, and general knowledge about how computers work. As early as 1977 it was well established that errors made while programming are clearly related to particular features, constructs, or properties of programming languages (Brooks, 1977).

Murnane (1993) discusses programming from the natural language standpoint, looking at Piagetian and Chomskyan theories of natural language acquisition, and how they apply to programming. He suggests that, initially, students require solid, tangible objects to work with. They need good feedback, and a clear path from cause to effect - "when I do x, it causes y". This is consistent with Van Someren's (1990) finding that a good understanding of the underlying virtual machine is vital for students learning programming.

4.3.1.1 Fear of the unknown

As Dijkstra (1985) has pointed out, computers are difficult to learn to use because they are very different from anything we encounter in the normal course of our lives. Although this is gradually changing as computers proliferate in everyday household devices, operating a personal computer is still substantially more complex and difficult than operating a microwave, for example (even though operating a microwave is itself often surprisingly difficult and frustrating). New users are often acutely aware of this complexity, if only second-hand, and using the computer can be intimidating and frightening, particularly for a user who has trouble operating the video recorder.

Learning to program a computer is significantly more difficult, complex and intimidating than simply using one. A user who has trouble making a word processor produce the desired output may be justifiably afraid of programming, with its concomitant reputation for arcane and mystifying symbols and extreme complexity. Not only is programming a new and unknown feat (intimidating enough in itself), it also has a reputation for being exceptionally difficult. This can lead to nervous, timid students who are already half convinced that programming is beyond them. Subsequent errors and stumbling blocks, of which there are many whenever something new is being learnt, compound and exacerbate these feelings of helplessness and inadequacy.

4.3.1.2 *Precision*

Human beings are not precise and accurate by nature. They make assumptions, skip steps in instruction books, and filter their world for meanings that make sense to them. Norman (1998) uses the following example: "If an airplane crashes on the border between the United States and Canada, killing half the passengers, in which country should the survivors be buried?" It is not uncommon for this question to be considered seriously, on the assumption that it makes sense and is a meaningful question. Many people fail to notice the problem with the question: the survivors should not be buried at all.

While this implicit error-correction means it is easy to fool a person into considering a nonsensical question as though it were serious, it also means that communication can be extremely robust. People often make small slips in conversation that go entirely unnoticed. Errors in written or spoken language are often unimportant, as the reader or listener uses context to interpret the meaning correctly, often without being aware of it. For this reason, proof reading of a work by its author is likely to be ineffective, particularly immediately after it was written, as the author will tend to read what he or she intended to write, rather than what is actually on the page (Norman, 1993).

A further aspect of precision is the need in programming to consider all eventualities, and take every possible alternative into account. In communicating with each other, it is usually not necessary to do this – it is common to take some things for granted (Philipchalk & McConnell, 1994). For example, in instructing someone to pick up a book from a table, it is reasonable to assume that if there is no book on that table, the attempt to pick up the book will cease. In programming instructions for a robotic arm, however, it would be necessary to include some sort of backup response that catered for the absence of the book.

In a programming context, when reading in numbers some languages will fail if text is encountered rather than digits, some will silently convert text to digits, and some will behave in an undefined fashion. If, on the other hand, a person were asked to add two numbers, "3" and "dog", it could reasonably be assumed that a problem would be detected. In most cases the problem would probably be solved using a request for clarification. It would not be necessary, in giving a person instructions on how to add the two numbers, to specify a separate procedure for handling unexpected input.

Not only are human beings unaccustomed to the need to state explicit responses to every possible input or eventuality, we are also accustomed to assuming that some eventualities are not possible, when they are merely highly improbable. As such, we do not even assume that these situations will be handled appropriately should they arise, we do not consider them at all. When leaving instructions for the care of house plants during an extended absence, it is uncommon to leave precise instructions for the eventuality of a meteor strike. If we were to attempt to plan for every remote possibility, we would be unable to deal with everyday life. We naturally filter the world to deal with situations that are likely and imminent, in order to survive (Norman, 1993; Philipchalk & McConnell, 1994). Nonetheless it is likely that, if a meteor does strike, the caretaker will make *some* appropriate response (such as buying replacement plants).

As a result, the level of precision required in programming, where every eventuality must be catered for, is likely to be challenging for the novice programmer. Operating at this level can be extremely taxing for those who are not accustomed to it. A high level of precision and specification must be maintained at all times, requiring a level of attention to detail that does not come naturally to human beings. At the same time, the programmer must deal with the limitations of language, machine, and compiler, all of which frequently require the inclusion of details that are not directly related to the problem, such as function prototypes when a function is to be used before it is declared, inclusion of libraries, memory management routines, etc.

4.3.2 Expert programmers

Mayer (1992) describes four main areas of expert-novice differences in computer programming – syntactic, semantic, schematic, and strategic knowledge. Novices struggle with syntactic knowledge because they have trouble recognising incorrect grammar, while experts are be quick to recognise grammatical mistakes. Experts have effective mental models of the virtual machine, while novices have yet to build these models – this is the difference in semantic knowledge. Schematic knowledge describes the structure of a program – experts use deep structure to categorize programs based on the type of routines required, while novices use surface characteristics for categorization. The planning process during problem solving uses strategic knowledge. Novices tend to use low-level plans, and are unskilled at problem decomposition. In contrast, experts keep the overall problem in mind while decomposing problems into small, manageable sub-problems.

Experts also consider many more alternatives than novices, and are more capable of comparing different solutions and considering the merits of each. Novices frequently do not consider alternative solutions at all (Jeffries et al, 1981).

A person who is experienced at problem solving in a particular domain tends to build up a repertoire of problem solving techniques and strategies (Mayer, 1992). When confronted with a new problem, such an expert compares it with past problems and, where possible, constructs a solution out of familiar, reliable techniques. Wiedenbeck (1985) argues that experts have automated low-level programming skills, thus freeing cognitive resources that can be used to tackle the higher level aspects of a problem. Experts can thus recognise grammatically incorrect code with a minimum of conscious effort. Novices, in comparison, require considerable conscious effort and cognitive resources to determine whether a piece of code is grammatically correct, before they can begin to try to determine what the code actually does.

For example, an experienced doctor who encounters a familiar set of symptoms for the tenth time is likely to diagnose the problem quickly and easily. While this can lead to problems such as tunnel vision, (where the familiar, expected solution, turns out to be incorrect in occasional cases (Norman, 1993)), in general it is likely to make the task of problem solving quicker, easier, more efficient, and more accurate.

A novice with no experience at problem solving in a particular domain, or in a related domain, must deal with problems individually, effectively solving them all from the ground up. Rather than adapting a new solution from existing algorithms, the novice, having no bank of algorithms to draw upon, must construct each necessary algorithm from scratch. A problem requiring a simple sorting routine necessitates the laborious construction of a sorting function from first principles, where an expert would simply slot in, say, an existing **`quicksort`** function with minor modifications to suit the current problem. Even when experts do not have access to files of code, they retain mental libraries of algorithms that can be reproduced at need. Novices have yet to build these libraries.

Details of the language that need to be remembered include: Which types of lists are sorted? What is the default step size in a **`for`** loop? What types of values may be returned from a function? How are parameters passed to a function? (by reference, by value, both depending on syntax) What is the difference between an array and a list? How does pattern matching work? What is a wild card? What

data structure is most appropriate for the current problem, and how is that structure accessed? Is there a built-in operator suitable for use with this problem?

In addition to remembering details of syntax and semantics, the novice must also endeavour to keep track of the current problem and solution. Any non-trivial task (and most tasks are non-trivial to beginners) requires the programmer to keep track of details such as the maximum size of the data to be manipulated, the precise manipulations required, and the necessary algorithms. This is a manageable task for an expert, as much of the information required is familiar to an experienced programmer and may be "chunked" effectively into one unit that can easily be recalled. But the information is new and unfamiliar to a beginner, and is remembered in separate pieces that fill up working memory and leave little space for algorithms and problem solving.

4.4 Existing solutions

4.4.1 Subsetting

Three of the most common approaches to teaching programming involve somehow shrinking the language to a manageable size. Brusilovsky et al (1994) call these approaches the incremental approach (teaching small subsets of the language and gradually expanding them), the sub-language approach (teaching only a subset of a real language), and the mini-language approach (creating a small, clean, new language exclusively for teaching). The incremental and the sub-language approaches stem from the same basic theory, and teach a small part of the language first, so that students don't have to deal with too much straight away. These two approaches are discussed in this section, while the third is covered in section 4.4.4.

Subsetting can be challenging to integrate properly with support materials. Text books are unlikely to be confined to the same subset of the language unless they are purpose-written for the course, which can be prohibitively expensive. Compilers are also unlikely to correspond purely to the relevant subset. Features of the development environment and error messages issued by the compiler may therefore refer to parts of the language that have been deliberately concealed from students.

4.4.2 Language overlays

Another approach to teaching programming is to put together a supporting library of "novice-friendly" functions and data structures - an overlay that is less intimidating and more intuitive than the language hidden underneath. This is particularly feasible in an object-oriented language, where the overlay may be used without reference to the underlying language.

However, this approach suffers from many of the problems of subsetting, in that compilers, teaching materials, and text books often don't confine themselves to the sections of the language being taught. In the case of language overlays, text books may not contain the new functions and data structures at all, referring instead to the underlying language which, again, has been deliberately concealed from students.

4.4.3 Supporting environments

Unfriendly, difficult programming languages can be mitigated by using a supporting environment designed to minimise the impact of obscure syntax and complex semantics on students. One such environment, BlueJ (Kölling and Rosenberg, 1996 a&b)) was first developed as a language and programming environment, but was later modified to be a development environment for Java. BlueJ provides a visual environment where objects may be created using menus, and skeleton code is produced that novices may then edit, for constructors and object behaviour. Students can define methods and attributes of objects without needing to focus on complexities of the syntax. The major advantage is that students can focus on object-orientation, working with objects and dealing with a visual representation of their program that enables them to see how objects interact and interrelate.

Empirical analysis of the environment remains to be done, to compare its performance with typical Java programming environments, and to answer questions such as how well students learn the syntax of a language when presented with code skeletons or templates, and how well they subsequently transition to using the language without the environment.

Miller et al (1994) describe a collection of different structure editors developed and used at Carnegie Mellon University. These editors, and others like them, are designed to avoid problems of syntax by detecting syntax errors while the code is being written, rather than waiting until compile time. Some structure editors

allow only correct code to be written, which can cause problems where partial code (for example, a function without a fully specified parameter list) is written and then edited later. Others provide code templates which may be inserted, which prevents students from making most syntax errors, since the templates need only be filled in, and semi-colons and other details of syntax are automatically provided.

These approaches successfully avoid many syntax errors, and reduce some of the cognitive demands on students. However, as students still need to be able to read, understand, and modify the code, not all of the cognitive overhead imposed by syntax can be avoided in this fashion.

4.4.4 New languages

Many languages have been designed explicitly for teaching programming, including Pascal (Wirth & Jensen, 1975), ABC (Meertens, 1981), Turing (Holt & Hume, 1984), and Blue (Kölling & Rosenberg, 1996a). Despite their pedagogical purpose, these languages were generally not designed based on psychological and learning theory. In fact, the principles of good user interface design, and what is known about how people learn in general, have rarely been applied to programming language design (although there is some movement in that direction, for example Pane & Myers (2000), and Myers (2000)).

Nonetheless, languages such as those listed above have made significant pedagogical improvements on commercial languages. Such languages are discussed in more detail in the next chapter.

There are also a number of mini-languages used for teaching introductory programming, often to children. Brusilovsky et al (1997) describe "mini-languages" as "small, simple languages to support the first steps in learning programming", and reports that many existing mini-languages use an actor to engage the enthusiasm and interest of students - for example, the LOGO turtle, and Karel the robot . Such languages are usually trimmed-down, simplified versions of existing languages (Lisp for LOGO, Pascal for Karel), often, like Karel, intended to serve as an introduction to the parent language. Much of the development work involved in these mini-languages has been invested in the actor and its environment. What can the robot/turtle do? What objects can be manipulated, what actions are available, and what control structures should be retained from the parent language? A crucial benefit of mini-languages such as

these is the visual feedback students receive as the actor visibly carries out their instructions.

These important considerations have usually taken priority over matters of syntax and semantics, in a successful bid to maximise student motivation. It is difficult to study the effects of actors, syntax, and semantics all at once. In order to focus on issues of syntax and semantics, this thesis leaves aside the study of such actors. Future work resulting from the GRAIL project could combine the resulting awareness of syntactic and semantic issues with the clear positive effects of mini-languages using actors, in order to achieve an optimal learning environment.

4.4.5 Acceptance

One approach to the difficulties of learning to program is to accept them. Students who wish to learn programming must eventually come to grips with real world programming languages, and some educators take Van Someren's (1990) view that syntax is only a problem in the early stages of learning a language. Given that students will need to deal with real world languages sooner or later, it could be assumed that sooner is better, so as to give students maximum exposure to languages they are likely to use in industry after they graduate.

This same argument, though, when applied to the exercises students are given initially, is clearly flawed: "Given that students will need to write large programs sooner or later, sooner is better, so that they get as much experience as possible in programming in the large." Few educators would expect students to write a large program from scratch as their first programming exercise. In addition, not all concepts students must master in order to become competent programmers are taught at the very start of the first programming course, or even in that first course at all.

4.5 Missing links

Despite enthusiastic, and occasionally heated, debate about the best language to use for introductory programming courses, there are still gaps in the empirical evidence gathered to support the various arguments. In part this is due to the ethics of experimentation using real students – if a cohort were divided into groups and each group was taught a different language, students could quite justifiably claim unfair advantage depending on which group achieved better results. Even if the process were not ethically questionable, developing two

courses instead of one (or at least altering the materials of a course to handle two separate languages) is prohibitively expensive, particularly if the impact of development environment is to be minimised by providing the same environment for each language.

The following questions remain to be answered:

- **Does the programming language influence the types of interaction and hence the learning process?** Do students make different types of errors, or alter their problem solving strategies depending on the language they are learning?
- **Does the development environment influence the types of interaction and hence the learning process?** Does student behaviour change according to the type of development environment in use? What sort of impact does an interactive environment have, where each line of code receives immediate feedback? Do the error messages have a significant impact on the students' motivation, understanding, and overall learning process?
- **Does syntax impact on cognition in a significant way?** Does the syntax of a programming language have a significant effect on the cognition and learning of novice programmers? Does the Piagetian view that language shapes thought apply to programming languages? How does students' first programming language shape their mental model of the virtual machine?
- **Does usability of programming language, environment, and support materials affect learning?** Does it matter what tools are used in teaching programming? Does the usability of the tools impact on student learning, and, if so, is it necessary to create tools that are specifically designed for teaching programming, rather than using commercial tools for purposes for which they were not designed.
- **Do programming skills transfer well to new languages?** Is there a valid argument for teaching a "toy" language first rather than an industry relevant one? Does it work to teach concepts first in a simple language - do they transfer to an industry relevant language?
- **Does language choice really matter?** Is the impact of a student's first programming language truly significant?

Chapters 6 through 10 of this thesis address these open questions.

5 Usability of Programming Languages

There are thousands of programming languages in existence today (Language, 2000). New languages are constantly being designed and implemented for a variety of purposes. Some, such as Java[1], are developed for commercial reasons, while others, such as SOLO[2], are aimed at academic environments. Few languages are designed with strict attention to usability principles. The decisions made in schools, universities, and industry to use particular languages are also rarely made with usability in mind.

However, the cognitive dimensions of notations, and other usability principles discussed in Chapter 2 can be used to analyse existing programming languages, and subsequently to support the design of new programming languages which provide greater support for specific activities, such as learning to program.

Most of the languages discussed here are commonly used as introductory programming languages. While some languages, such as ALGOL and FORTRAN, are no longer as popular for introductory teaching as they once were, it is appropriate to include them for historical perspective. Such languages give some insight into the background of today's language designers, and provide the basis for some of the pervasive features and techniques used in programming today.

Other languages have been included as being representative of a particular style or paradigm of language. For example, Hypertalk is included although according to the Reid report (1999)[3] it is only taught in one school as an introductory language. Languages such as Turing, Logo, and ABC are significant in having been designed for pedagogical purposes.

The focus of the following discussions is solely on the use of the languages as introductory or pedagogical programming languages. As discussed in Chapter 2, usability is only meaningful when discussed in terms of a certain user group. An

---

[1] Java was designed at Sun Microsystems to be a platform-independent language (Arnold & Gosling, 1998)

[2] SOLO was designed at Open University for teaching computer programming to psychology students (Eisenstadt & Lewis, 1992)

[3] The Reid Report (1999) is a somewhat inaccurate measure of a language's popularity for introductory programming courses, since the survey relies on schools to volunteer the information. Reid does not actively query any institutions, and is no longer maintained. Nonetheless, it is the only available survey of language choice in tertiary teaching institutions.

application or programming language which is highly usable for an expert may not be usable for a novice, and vice versa. This thesis focuses on the needs of novice programmers with no prior programming experience, particularly those students whose negative experiences with computers and technology have led to a fear of computers, and consequent learning difficulties.

5.1 Analysis of programming languages

5.1.1 Criteria

Comparison and analysis of programming languages is easiest to do when based around a set of measurable attributes. For an analysis based on usability, the cognitive dimensions of notations provide a useful set of attributes by which programming languages can be described and compared. For the purposes of this thesis, the set of cognitive dimensions will be reduced to those most applicable and important to introductory programming (see Table 5.1). Although other dimensions are certainly relevant, and may be mentioned for individual languages, those chosen to provide the basis of this discussion are the most critical, for reasons discussed below, and will be considered for all languages included here.

One dimension which is not covered here is visibility. Visibility refers to the amount of cognitive effort required to make information visible. It is particularly important to expert programmers who need to see large amounts of code simultaneously in order to understand and manage large programs. In this sense it is less relevant to novices writing small programs. Visibility also tends to be consistent from one text-based language to another. It is much more relevant to visual programming languages, or programming languages with a visual element, such as mini-languages with actors (see section 4.4.4). Visibility can be enhanced by the use of supportive environments which allow different views of the code (one such example is described in Miller et al (1994)).

5.1.1.1 Closeness of Mapping

"Closeness of mapping" is particularly important for introductory programming. When students solve a problem, how closely does the programming language fit the language and concepts of the students' solution? How many steps does the notation require in order to complete one conceptual step in a student's algorithm? How well does the language fit the way students think? Closeness of mapping

refers to the closeness of the language to knowledge students already possess, to the types of problems students are required to solve, and to the concepts students are attempting to acquire.

Closeness of mapping	closeness of representation to domain
Consistency	similar semantics are expressed in similar syntactic forms
Diffuseness	verbosity of language
Error-proneness	notation invites mistakes
Hard mental operations	high demand on cognitive resources
Role-expressiveness	the purpose of a component is readily inferred

Table 5.1 Cognitive Dimensions of Notations most relevant to introductory programming

5.1.1.2 Consistency

Consistency requires that similar concepts are expressed in similar ways – that related constructs work in recognisably analogous ways, in order to maximise the predictability of the language. Students who have mastered one construct can then rely on similar constructs working in similar ways. At the same time, though, consistency can be taken too far. Careful differentiation between semantically distinct constructs is also necessary. There is a balance to be struck between underlining semantic distinctions and emphasising similarities.

5.1.1.3 Diffuseness

As discussed in Chapter 3, the appropriate level of diffuseness, or verbosity, is dependent on the target domain. In introductory programming, a careful balance must be struck between verbosity used to enhance the readability and clarity of the language, and brevity used to increase the visual structure of the language. Very verbose languages, such as Hypertalk (Shafer, 1991), often possess a less obvious visual structure and rely more on careful reading of the code to elicit the start and finish of code blocks and functions.

5.1.1.4 *Error-Proneness*

Error-proneness is a common problem in programming languages. Many languages possess constructs which make errors almost inevitable. One example is the use of similar symbols in semantically distinct situations, where there is no obvious way to remember which symbol is appropriate in which situation, for example = for assignment and == for equality testing. Another is the use of familiar symbols in unfamiliar ways, for example the way Java uses = to test for equality of reference rather than equality of value.

5.1.1.5 *Hard mental operations*

Hard mental operations are unavoidable when learning to program. As discussed in the previous chapter, there are many concepts in programming which are difficult for students to master, and which require them to think in unfamiliar and complex ways. However, a notation which avoids requiring students to perform unnecessarily challenging cognitive operations may be able to minimise these hard mental operations, and hence make coming to grips with programming easier and less stressful.

5.1.1.6 *Role-expressiveness*

Role-expressiveness is particularly important to first-time users of a notation, because it allows them to interpret the notation without reference to other resources such as teachers, and manuals. It decreases the learning curve, and increases the accessibility of a notation. At the same time, role-expressiveness is helpful to experts, because it facilitates memorisation of components of the notation – acting as a reminder of the function of each component. Clearly, role-expressiveness is vital to introductory programming, allowing students to read programs and form some impression of what they do, rather than needing every component explained to them. For example, the code

```
cout << "hello world";
```

is less role-expressive than

```
print "hello world"
```

due to the use of the keyword **cout** with which students are unlikely to be familiar, compared with **print**, whose English meaning is sufficiently analogous to its programming meaning that the statement may well be correctly interpreted without prior programming knowledge.

5.2 The languages

5.2.1 Pascal

5.2.1.1 Background

Pascal (Wirth & Jensen, 1975) was a popular choice of introductory programming language for many years. It is an Algol-like language first designed in 1968, first implemented in 1970. By the mid-seventies it was in use as an introductory programming language, and it continued to gain in popularity throughout the Seventies and Eighties. Its popularity as an introductory language is a compelling reason to study it here, but it is perhaps equally significant for its impact on subsequent languages such as the Modula family (Wirth, 1988), and, perhaps less obviously, object-oriented languages such as Ada (US Department of Defence, 1981), Smalltalk (Goldberg & Robson, 1983), and Java (Arnold & Gosling, 1998). Many generations of computer programmers learned Pascal as their first structured programming language, if not their first language[4], with the result that the language has had both tangible and intangible influence on language design.

5.2.1.2 Usability

Pascal maps closely onto the virtual machine, so that students with a good mental model of the machine's architecture and functionality should be well positioned to program in Pascal. However, it does not map as closely onto the way students solve problems and think about the world. Details such as the separation of numbers into floating point numbers and integers are likely to be unfamiliar to novice programmers.

Another feature which is particularly restrictive and frustrating involves the parameter passing mechanism as it relates to arrays (and hence to strings – arrays of characters). The type of an array includes its size, and the precise type of a parameter must be specified in the declaration of a procedure or function (subroutine). As a consequence, an array may only be passed as a parameter to a subroutine if it precisely matches the dimensions specified in the subroutine

[4] For many years BASIC was a common choice of first programming language for programmers learning programming at home, rather than as part of a formal course

declaration[5]. This limitation makes the creation of a string processing routine clumsy and inefficient. The only solution is to assume a standard size of string which is large enough for any purpose, and then to declare all strings to be this size, regardless of how much space they truly need.

Pascal scores well in consistency overall, although there are some minor syntactic issues where, although the rule may technically be consistent, it is unlikely to appear consistent to a beginner. One such issue is the requirement for a semi-colon after some **END** statements, but not others. The final **END** statement in a program requires a full stop, which is simple enough to remember in isolation, but when combined with the need to remember which **END** statements throughout the program require a semi-colon and which do not, the cognitive overhead increases.

Although many Pascal implementations ignore excess semi-colons, there are places where an extra semi-colon can change the meaning of the code. For example, the **else** in the following code is actually not considered part of the nearest **if**, because the semi-colon used in the preceding line ends the **if** statement.

```
if A < B then
     A:=B;  { this is the offending semi-colon}
else
     B:=A
```

Depending on the preceding code, this would either produce an error, or attach the **else** to a preceding **if** statement.

Pascal has low to moderate verbosity – it is not as terse as C, and not as verbose as Hypertalk. The problems with consistency translate into some error-proneness, which can also be a problem with memory management in Pascal, when using pointers and dynamic memory allocation. Pascal does not contain many hard mental operations, although keeping track of dynamic memory can be a problem.

---

[5] conformant arrays (which allow arrays of different sizes to be passed as parameters, as long as they "conform" to the same number of dimensions, and the same type of the array contents, as the formal parameter) are actually part of "Standard Pascal" but are optional, and frequently not implemented, perhaps because they are not necessarily easy or efficient to include

Pascal scores fairly well in role-expressiveness. Its control structures are, in general, self-explanatory, from **for** statements (eg. **for i := 1 to 10 do**) to procedures (eg. **procedure p (x : integer)**), however some of its syntax is less role-expressive than it could be. For example, the role-expressiveness of := versus = is not clear. It is useful to differentiate assignment from equality testing, but the use of := is somewhat arbitrary, and does not convey any useful information, other than "this is not equals".

Many of the built-in functions in Pascal are well named and very role-expressive, for example **read** and **write** for I/O, and **abs** for absolute value. However, there are subtleties to some of these functions which can be unexpected – for example, reading a real value into an integer variable will read everything up to the decimal point. The next **read** statement will read the decimal point (if reading in a character) or the digits after the decimal point will be read as a new value (if reading in a numeric value).

5.2.2 ABC

5.2.2.1 Background

ABC (Geurts, Meertens, & Pemberton, 1990) is a language designed specifically for introductory programming. It is tightly integrated with its environment, so that any study of the usability of the language must also discuss that environment. Although it was moderately popular for teaching programming when it first appeared, it does not appear in the Reid Report (1999), and seems to have fallen into disuse. ABC is a strongly typed procedural or imperative language, somewhat similar to Pascal, although it has several significant differences from traditional imperative languages, including a wider range of abstract data types, persistent values (which persist permanently, even between executions of the interpreter, until explicitly deleted), and a single, arbitrary precision numeric type.

5.2.2.2 Usability

ABC is very closely mapped onto the types of programs novice programmers are likely to write, and onto the language students can be expected to use. Many functions more typically carried out in traditional programming languages using symbols are done in ABC using keywords, making for a very readable (but consequently verbose) programming language. Consider the following ABC assignment statement: **PUT x IN y**.

In addition, data structures in ABC map easily onto problems beginners understand – for example, the table type in ABC is effectively an array indexable by any type. Indices, known in ABC as keys, need not be consecutive, making the construction of sparse arrays as easy as a traditional non-sparse array with consecutive keys. This means that, for example, a telephone book can easily be constructed, mapping names onto telephone numbers or vice versa.

ABC is consistent and, in general, not error-prone, although some constructs may be confusing at times – for example, lists in ABC are automatically sorted, leading to potential confusion for students who have forgotten, or never grasped, that this is the case. Hard mental operations are not present in ABC, and role-expressiveness is generally high. Some exceptions to this are the surprisingly terse operators chosen for some functions – for example: **#** (length), **^** (concatenation), **^^**(duplication and concatenation), **/** (used in a **WRITE** statement to print a newline), and **|** (returns the head of a text variable, to a specified length).

5.2.3 Turing

5.2.3.1 Background

Turing (Holt & Hume, 1984) is an Algol-like language based on Pascal, of interest primarily because of its pedagogical orientation. The designers of Turing set out to create a language which facilitates program maintenance and formal verification, as well as a syntax which does not get in the way of programming with superfluous details. Turing was also intended to be an effective introductory programming language, although this was not its sole aim, as another of the design goals was generality – a general purpose programming language that overcomes many of the flaws in existing languages such as Pascal and C.

5.2.3.2 Usability

The designers of Turing eliminated many of the more troublesome aspects of Pascal, and in doing so achieved higher usability in some ways – for example, Turing drops the requirement for semi-colons as statement terminators, and possesses more convenient string handling than Pascal. Turing also does not require a main program statement – a Turing program can be as simple as "**put 4+3**". In these respects Turing achieves closer mapping to the way students think and solve problems than Pascal.

Turing also possesses high consistency, although some aspects of this consistency can be taken too far, for example the 5 distinct meanings of `A(B)` (Holt et al, 1988) discussed further in section 6.1.3. One inconsistency from a novice's point of view is that input and output statements (`get` and `put`, respectively) are not allowed inside functions, due to Turing's "no function side effects" rule.

Turing is moderately diffuse, opting to use full English words in certain cases to maintain readability, and maximise role-expressiveness (with considerable success). For example, the declaration of a pointer in Turing is `var p: pointer to c`. Clearly `p` is a pointer to `c`. The one problem with role-expressiveness in this case is that `p` does not point to `c`, but rather is an index into `c`, which is a `collection` of dynamically allocated objects of a single type (collections are further described in section 6.1.6). To compound the confusion, creating dynamic memory for `p` is done using the command `new c, p` even though `p` is not a new object of type `c`, rather it is a new index into `c`. To use `p`, the same syntax is used as an array access - `c(p)`. This problem is discussed in more detail in section 6.1.6.

Turing shares memory management problems with Pascal and C, although they are mitigated somewhat through the use of collections.

5.2.4 Modula-3

5.2.4.1 Background

Modula-3 (Harbison, 1992) is a procedural language with its roots in Pascal and Modula. It is a high level language with an emphasis on modularity and multi-programming. Modula-3 extends Modula-2 to handle object-oriented concepts as well as possessing automatic garbage collection. The syntax of Modula was designed to improve on the most commonly used introductory language of the time, Pascal. According the Reid Report (1999), 48 institutions use Modula, Modula-2 or Modula-3 for their introductory programming courses (8.8% of total institutions surveyed).

The designers of the Modula family of languages aimed at efficiency, improved learning and ease of use, as well as maintainability.

Because of the aim of efficiency, some hardware dependencies are maintained, and actually increased compared to Pascal:

> *"The major reason for strictly distinguishing between real numbers and integers lies in the different representation used internally. Hence, also the arithmetic operations are implemented by instructions which are distinct for each type. Modula therefore disallows expressions with mixed operands." (Wirth, 1988)*

5.2.4.2 *Usability*

The hardware dependencies mentioned above are a problem for Modula in terms of closeness of mapping. It is natural for a student to add `5.3` and `4` to get `9.3` without considering that `5.3` is a real number and `4` an integer value. The distinction in Modula between reals and integers, and the requirement that arithmetic operations be separated for reasons of efficiency, means that even such simple arithmetic statements require an extra step - generally an explicit conversion of the integer value to a real one, or vice versa, using a function such as **`TRUNC()`** or **`FLOAT()`**.

Control structures in Modula generally require a corresponding **`END`** statement, but, unlike Turing, the name of the construct is not paired with the **`END`** (as in **`if...end if`**). This decreases verbosity somewhat, but at the cost of increasing error-proneness, as it becomes increasingly difficult to determine which **`END`** statement belongs to which construct. Consider the following procedure, taken from Wirth (1988):

```
PROCEDURE search(VAR p: TreePtr; x: CARDINAL) :TreePtr;
BEGIN
     IF p # NIL THEN
          IF P^.key < x THEN
               RETURN search(p^.right,x)
          ELSEIF p^.key > x THEN
               RETURN search(p^.left,x)
          ELSE
               RETURN p
          END
     ELSE
          Allocate(p,SIZE(TreeNode));
          WITH P^ DO
               key := x; left := NIL; right := NIL
          END;
          RETURN p
     END
END search
```

Only careful indentation makes it clear which **END** relates to which control structure. The treatment of procedures is also inconsistent with respect to **END** statements, in that they are the only constructs which also require a **BEGIN**, and which add the name of the construct to the **END** (as in **END search**). Modula also shares its semi-colon behaviour with Pascal, with similarly confusing results.

5.2.5 Prolog

5.2.5.1 Background

Prolog was first designed in 1971 as a tool for processing natural languages (specifically French) (Colmerauer & Roussel, 1996). A member of the family of logic programming languages, Prolog allows programmers to create a collection of clauses which contain both statements of fact (for example: Spot is a dog, expressed by **dog(Spot).**) and rules (for example: dogs eat meat, expressed by **dog(X) :- EatsMeat(X).**). Prolog can then use backtracking and unification to determine a result for a query (for example: Does Spot eat meat? Expressed by **EatsMeat(Spot)?**, which will give the result **Yes**.)

Although Prolog is not commonly used as an introductory language, it is useful to discuss it here as an example of the logic programming paradigm, which is quite different to the imperative and object-oriented paradigms more commonly taught in introductory programming courses.

5.2.5.2 *Usability*

For logic or constraint programming, Prolog maps very closely onto the problem domain: specify what is known about the problem, and then query the system for inferences which can be drawn from that knowledge. However, as discussed in Chapter 4, Van Someren (1990) found that the Prolog concepts of backtracking and unification are difficult for novices to grasp. These concepts require hard mental operations of novice programmers, and sometimes of experts, and mean that more complex operations in Prolog probably don't map well onto the way students naturally solve problems. In addition, Van Someren (1990) addresses closeness of mapping:

> *The notion of* plans *may be less useful with Prolog than with imperative languages, because Prolog has no built-in primitives that correspond closely to the building blocks of plans, e.g., "repeat an operation until some condition holds" or "hold this object" have no direct counterparts in Prolog." (Van Someren, 1990)*

Prolog is somewhat terse. It uses symbols rather than words in many instances (',' for a conjunction, '!' for cut, ':-' to signify a rule definition, '|' to separate the head of a list from the tail, etc). Prolog is likely to be error-prone for students who do not fully understand backtracking, although the language scores well in consistency.

5.2.6 C

5.2.6.1 *Background*

C is an imperative, Algol-like language which provides support for systems programming and hardware device access. While it contains relatively high level features for structured programming, it also provides low level features such as bit manipulation and pointer arithmetic. These features add to the power of the language, and make it suitable for systems programming and direct memory

manipulation, however they also add complexity and tend to be hazardous and error-prone constructs.

Noted for its terse and potentially obscure code, C is nonetheless extremely popular, in part because of the many highly efficient implementations available for a wide range of platforms, and because of its power and expressiveness. In some ways C is extremely close in nature to assembly language, with many C instructions mapping directly onto single assembly language instructions. C is a powerful and flexible language that allows direct access to hardware-level features of the machine, making it a useful language for writing device drivers and other low-level, device specific programs.

5.2.6.2 Usability

C does not score well in closeness of mapping. Although it provides some higher level data types, such as **`unions`** and **`structs`**, as well as some support for abstraction (using the aforementioned data types as well as enumerated types), it is not closely mapped overall to the way students think and solve problems, due to its low level features. For example, some features of C, such as the size of an integer, are hardware dependent, in that they vary depending on the native word size of the platform on which the code is compiled and run. This problem can directly impact even on simple novice programs, as an integer that is 16 bits can easily overflow. This means that in addition to students needing to come to terms with the basic programming concepts required to write a simple mathematical program, they must be able to understand why adding **`4000`** to **`30000`** results in a negative number. This complicates the debugging process and requires an understanding of the details of the implementation that might better be left until later in the course. Features such as this increase the error-proneness of C substantially.

Other features which also increase error-proneness are dynamic memory management in C, and its terse and unforgiving syntax. Many crucial errors in C compile legally, in some cases without even a warning. For example, the `&` and `*` characters for addressing and dereferencing respectively, are not particularly mnemonic or memorable, and are frequently exchanged inadvertently, sometimes producing legal code with surprising results. `=` and `==` are a source of considerable confusion, with the assignment operator often being used in an expression where a comparison is required.

There are many ways in which C falls down in terms of consistency. For example, all variables passed as parameters to the input function, **`scanf`**, require an ampersand character in front of them (eg. **`scanf("%d",&i);`**) except for character strings, which do not require the ampersand. While the reasons for this are consistent in a technical sense[6], they are generally out of students' conceptual reach until much later in the course, so that the special case must be rote-learned as a mystifying exception to the rule, rather than understood, and hence remembered.

Hard mental operations are a problem in C, like Pascal, where dynamic memory is concerned. In addition, C array subscripts start at 0, requiring surprisingly hard mental gymnastics when designing a **`for`** loop to iterate through an array. This is compounded by the slightly odd syntax of a **`for`** statement, which operates *while* the central expression is true, rather than *until it becomes* true. This means that iterating through a 10 element array requires either **`for(i=0;i<10;i++)`** or **`for(i=0;i<=9;i++)`**, which can be translated, respectively, to "start with i equal to zero, continue while i is less than 10 (but what about the 10th element?)" or "start with i equal to zero, continue while i is less than or equal to 9 (where did 9 come from? There are 10 elements in my array)", rather than the more intuitive **`for(i=1;i==10;i++)`** - "start with i equal to 1, continue until i equals 10".

Many of the constructs described above are also very poor in terms of role-expressiveness, most notably the * and & operators. This is, in part, a function of C's terseness.

5.2.7 C++

5.2.7.1 Background

C++ is an object-oriented language built on top of the imperative language C. It was first proposed by Stroustrup in 1979 as "C with classes" (Stroustrup, 1994), which was intended to provide the advantages of Simula's class mechanisms with the efficiency and suitability for systems programming of C. It is effectively a super-set of C, offering all of C's features with additional object-oriented mechanisms. As a result, C++ suffers from many of the difficulties experienced

---

[6] **`scanf`** takes a pointer as its argument, and as a character string is an array, the name of the array is a pointer to the start of the array.

with C, and adds a new set of quirks and problems. As is typical of object-oriented languages, it combines the necessity of knowing the principles of the imperative coding style (which is used at the lowest level, to define methods or member functions) with the complexities of object-orientation, including inheritance, polymorphism, and genericity.

C++ is not a pure object-oriented language, and may in fact be used in a completely imperative style. It has the stated aim of being backwards compatible with C[7]. This introduced some complications to the C++ design process. As Stroustrup himself acknowledges:

> *Over the years, C++'s greatest strength and its greatest weakness has been its C compatibility. This came as no surprise. The degree of C compatibility will be a major issue in the future. Over the coming years, C compatibility will become less and less of an advantage and more of a liability. (Stroustrup, 1994)*

5.2.7.2 *Usability*

All of the usability issues which apply to C also apply to C++. The addition of object-orientation arguably helps somewhat with closeness of mapping, since the language facilitates better encapsulation and design. In addition, some functions such as input and output are handled more cleanly in C++, so that much of the confusion surrounding the **`printf`** and **`scanf`** statements is disposed of. Many of the new keywords that have been added to C++ are very role-expressive, for example **`public`**, **`private`**, **`class`**, **`template`**. Some of the new concepts, however, are sufficiently complex that role-expressiveness is a considerable challenge. Many of these are represented with mnemonic keywords which, while they cannot convey the full complexity of the construct, are at least memorable once the construct has been understood, for example, **`friend`**, **`protected`**, **`virtual`**.

Error-proneness and hard mental operations are at least as problematic for C++ as they are for C, with similar problems with the terse and unforgiving syntax, and dynamic memory management. To add to these problems, C++ has default and implicit behaviours, such as the calling of constructors and destructors, which can make code even more difficult to decipher, since each step in the code is no longer traceable without a good knowledge of the underlying execution model.

---

[7] with some exceptions

Additional hard mental operations may be required, such as to determine which version of an inherited function is called in a particular situation.

5.2.8 Ada

5.2.8.1 Background

Ada is a powerful object-based language, created and used by the US defence department to handle large software projects, including some with a realtime component. While it is primarily used for defence projects in the US and elsewhere, Ada is also used for some industry projects, and has been widely used as an introductory object-oriented programming language in tertiary courses (Allen, Grant, & Smith, 1996).

5.2.8.2 Usability

Some of the Ada features designed to encourage or enforce good programming practices add significant complexity to even the simplest programs, which leads to problems with closeness of mapping for beginners. For example, it is necessary to include the "**`with package`**" statement in order to access standard functions such as input and output. This statement specifies which version of the function should be used (for text I/O, numerical I/O, file I/O etc). Consider the following example:

```
with Ada.Text_IO;
procedure Hello is
begin
     Ada.Text_IO.Put_Line("Hello world");
end Hello;
```

Even coming to grips with this simple program requires dealing with the **`with`** statement, as well as the rather cumbersome **`Ada.Text_IO.Put_Line("Hello world");`**[8]

Ada is quite diffuse, which allows it to be, in general, effectively role-expressive. For example, the Ada equivalent of declaring function parameters to be passed by value or by reference is to declare variables **`in`**, **`out`**, or **`in out`**. An **`in`** variable is

---

[8] This cumbersome syntax can be avoided at the cost of introducing another concept: **`use Ada.Text_IO;`**, which automatically searches the **`Ada.Text_IO`** library.

used for passing a value into a function. An `out` is used for passing a value out, and `in out` is used for both. This is more role expressive than C's use of `&` and `*`, and even than Pascal and Turing's `var` parameters.

Hard mental operations are a problem in Ada, as even simple programs require students to grasp some fairly sophisticated aspects of object-orientation, and of the Ada class hierarchy.

5.2.9 Smalltalk

5.2.9.1 Background

Smalltalk is particularly noteworthy for its contribution to the birth of object-oriented programming (OOP). Whereas many subsequent object-oriented languages, such as Java, C++, and Ada are hybrid languages, sacrificing paradigmatic purity for efficiency, convenience, power, or application-oriented features, Smalltalk is paradigmatically pure. Programs are objects, data structures are objects, and objects communicate by passing messages to each other. Messages are analogous to functions or methods, in that a message generally requests that an object perform one of its repertoire of actions.

5.2.9.2 Usability

Smalltalk does not score well in closeness of mapping for introductory programmers, since even relatively small programs require students to grasp the complexities of object-oriented programming, and the Smalltalk class hierarchy (in addition, this is a source of hard mental operations). It is an extremely consistent language, adhering precisely to the object-oriented paradigm, and treating all objects alike. However, it is not entirely consistent with what students already know, as it does not define arithmetic or boolean operator precedence. This means that students cannot rely on arithmetic working the way they expect it to. For example `2+3*4` will produce `20`, rather than `14`.

Smalltalk is moderately verbose, using words rather than obscure symbols for most built-in messages, although it does possess some unexpectedly terse operations, such as "^", which means "return". Smalltalk is not overly error-prone, although its strict adherence to the object-oriented paradigm does require hard mental operations of beginners.

5.2.10 Java

5.2.10.1 Background

Java is an object-oriented language with a syntax which is loosely based on C and C++. Java owes its popularity in part to the popularity of the World Wide Web, and the development of Java Applets, which are small programs which may be embedded in web pages. Apart from the execution model, which is independent of the actual language itself and not of interest here, Java differs substantially from C++ in its approach to memory management. While C++ uses pointers and references, and has no garbage collection, Java has references but no pointers, and employs garbage collection. This simplifies memory management for the programmer, possibly at a cost of poor runtime speed, and eliminates pointer manipulation, which has proven to be a large source of errors in C++.

5.2.10.2 Usability

In order to program in Java for the first time, there are many concepts that students must either understand or take on faith. Due to the module-based, object-oriented nature of the language, even the simplest Java program requires a complex structure, as illustrated by a basic "Hello World" program:

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World");

    }

}
```

This hampers Java in terms of closeness of mapping - novices endeavouring to solve programming problems in Java must go to considerable lengths to translate their algorithm into correct Java code. Some simple constructs in Java are surrounded by complexity, due to the structure of the language. For example, the definition of a simple constant requires four keywords before the identifier, as shown in this example:

```
public static final double PI=3.1415926;
```[9]

The distinction between "reference types" and primitive types passed by value creates the potential for confusion when using assignment. The effect of the statement `i=j;` is dependent on the types of `i` and `j`. If they are values of a primitive type, such as `int`, then the above statement merely copies the value of `j` into `i`. A subsequent change to `j` would not change `i`, and vice versa. If they are objects or arrays, however, then they are handled by reference rather than value, and the above statement makes `i` refer to `j`. A subsequent change to `j` will also show up in `i`. Although this is likely to be made clearer by subsequent statements, it is a subtle point that is likely to confuse beginners.

In addition to the confusion created by assignment, testing for equality is also made more complex by the handling of objects by reference. If `i` and `j` are objects, the expression `i==j` tests not whether both `i` and `j` contain equal contents, but whether `i` and `j` refer to the same object. To test whether the contents are the same, a specially defined function must be used for comparison.

The use of references does eliminate several sources of confusion in C-like languages: the difference between pointers and references, the need for dereferencing, etc. However, it also introduces a new source of confusion, by not syntactically differentiating objects handled by reference from those handled by value. This leads to a lack of consistency (from the novice perspective), and increases error-proneness.

Java is terse rather than diffuse, and the language requires hard mental operations of students trying to come to terms with the class hierarchy. For example, it can be difficult to keep track of which version of a function is being invoked, where inheritance has been used. Java also has some problems with role-expressiveness, largely due to its terseness, and the complexity of object-orientation. The meaning of the keywords **`static`** and **`final`**, for example, are not intuitively obvious

---

[9] **`public`** means it is visible and accessible outside its class, **`static`** means there is only one copy of **`PI`** regardless of how many instantiations of the class exist (it is a class variable, rather than an instance variable), **`final`** means it is a constant which must have a value assigned in the declaration, and that value may never change, **`double`** refers to a double precision floating point number

(although **final**, at least, is probably memorable once the meaning has been explained - the object has received it's "final" value, and hence cannot be changed).

5.2.11 Scheme

5.2.11.1 Background

Scheme was developed in 1975 by Gerald Sussman and Guy Steele. It is a dialect of Lisp, and was originally designed for teaching and research purposes. As a result, Scheme is substantially smaller than Lisp, with a much reduced (and simplified) syntax (Dybvig, 1987). The designers of Scheme aimed to produce a clean, elegant, and consistent language. Scheme is widely used for teaching introductory programming.

5.2.11.2 Usability

Although smaller than Lisp, the base definition of scheme possesses over 150 procedures and operators, not including features declared optional in the official report (Abelson et al, 1998). Scheme is a high level language which is relatively platform independent, although some features, such as numeric representations, will vary depending on the implementation and underlying hardware. Scheme also retains some Lisp features which refer to hardware features unlikely to be familiar to novice programmers, namely **car** and **cdr** and their variants (Contents of Address Register and Contents of Decrement Register respectively). Together with prefix notation for arithmetic and boolean operators, as well as functions, this means that Scheme does not map closely onto the way students have previously learnt to solve problems.

Consider the following Scheme function to calculate the factorial:

```
(define factorial
     (lambda (n)
          (let fact ([i n])
               if (zero? i)
                    1
                    (* i (fact (- i 1)))))))
```

This example highlights the terseness of Scheme - many parts of this function are left implicit, for example the return of a value, and the behaviour of the **if**

statement[10]. Due in part to its terseness and the elements of the syntax which are left implicit, Scheme lacks role-expressiveness. The prefix notation and difficulty matching parentheses in large programs make Scheme somewhat error-prone, and lambda calculus is the source of hard mental operations for novice programmers.

5.2.12 Logo

5.2.12.1 Background

Logo was originally designed by Seymour Papert in the 1960s (Papert, 1993). A dialect of Lisp, Logo was created to encourage learning and creativity, especially for children. The aim was to create an accessible language which would be easy to use, with a small learning curve, but nonetheless powerful enough to avoid limiting users who were ready for more advanced exercises. Early versions of Logo included a robotic turtle which could be directed to move around the floor, so that students could see direct and physical results of their programs. With the development of graphics-capable computer monitors, the turtle moved onto the screen and became "turtle graphics".

5.2.12.2 Usability

Although it is technically a dialect of Lisp, Logo has a much simpler and more readable syntax, having done away with the need for large numbers of nested parentheses, and included infix arithmetic and prefix function invocation. It is somewhat terse, although many commands have a long form which can be used initially, and the short forms are reasonably mnemonic. For example, to move the turtle forward 50 paces, the command is **`forward 50`**, which may also be expressed as **`fd 50`**.

LOGO achieves a high closeness of mapping for simple programs, and for turtle graphics, although more complex programs are further from the sort of language and problem solving students are accustomed to. Logo is consistent and not overly error-prone, although some of the syntax can cause confusion, for example the difference between **`"name`** and **`:name`** (**`:name`** is used to access the value of **`name`**, while **`"name`** is used to access the address). LOGO does not require hard

---

[10] the **`if`** statement does not explicitly indicate that "if the expression is true, then the return value is the first statement, otherwise it is the second"

mental operations, and most of its constructs are very role-expressive. Consider a LOGO loop:

```
repeat 4 [forward 20, right 90]
```

The repeat statement takes a single parameter - the number of repetitions required. The block of code to be repeated is inside the square brackets, and **forward** and **right** are self-explanatory.

The function definition syntax is similarly role-expressive:

```
to square
     repeat 4 [forward 20, right 90]
end
```

5.2.13 Haskell

5.2.13.1 Background

Haskell is a functional language designed to be used in teaching, research, and the building of large scale applications. Its designers aimed to provide a single, general purpose, and powerful language which would reduce the need for the wide variety of functional languages currently available. While Haskell is a purely functional language, intended to be general purpose, rather than specifically aimed at any particular programming domain, it includes some features more commonly associated with object-oriented languages, including polymorphic types, classes, and inheritance.

5.2.13.2 Usability

Closeness of mapping is an issue in Haskell. While features such as pattern matching map very nicely onto some problems, Haskell's adherence to strict functional programming makes some problems rather more cumbersome to solve. This is largely because the lack of assignment means that temporary variables cannot be stored - everything is done using function calls and their return values.

This also leads to some hard mental operations, keeping track of, and interpreting, nested function calls. Some relatively trivial syntactic issues are potentially error-prone, and inconsistent with what students already know - for example, a function call in Haskell is not written '`f(x)`', which would be consistent with mathematical notation, instead it is written '`f x`'. This fails to take advantage of something

students already know - ie that `f(x)` is a call to apply the function `f` to the value `x`.

Haskell is fairly terse, but can maintain high readability even so. More than many imperative languages, readability in a Haskell program is highly dependent on the programmer, since so much of the code consists of user-defined functions. Where keywords are used, their role-expressiveness is generally high, but Haskell tends to use symbols rather than keywords, for example '++' for string concatenation, '.' for function composition, and '&&' for boolean 'and'.

5.2.14 Hypertalk

5.2.14.1 Background

Hypertalk (Shafer, 1991) is a scripting language developed for the Apple Macintosh. Hypertalk is an integral part of Hypercard, a Macintosh application for building programs and interfaces. Despite rarely being used as an introductory programming language in a formal setting[11], it is worth including here because of its general popularity and its "Englishy" appearance. In other words, the language designers intended programming in Hypertalk to be as close to programming in English as possible, without the ambiguity of natural language. Hypertalk's original target audience was novice programmers.

5.2.14.2 Usability

As a result of its closeness to natural language, Hypertalk is easy to read and maps very closely onto students' algorithms, but it lacks visual structure and clarity. For example, the following code shows equivalent statements in C and Hypertalk:

---

[11] Although Hypertalk has been used as a lead-in to an introductory programming course (Katz & Porter, 1991)

C

```
y = 42;
myBool = 1;
if (myBool) {
   x = x + 1;
}
else {
   x = x - 1;
}
```

Hypertalk

```
set y to 42
set myBool to true
if myBool is true then
     set x to x + 1
else
     set x to x - 1
```

This is a trivial example, but it does highlight some of the disadvantages of verbosity. In the terse C code, the indentation is more obvious than the verbose Hypertalk equivalent, because there is less text on the page, and the visual structure is clearer. In this case the braces in the C example are redundant, but a useful way of enhancing the visual structure.

Some Hypertalk commands silently evaluate in unexpected ways. If an uninitialized variable is used, or an identifier is left out accidentally, no error message will be produced, but the statement will not be executed as expected. For example, the statement

```
sort by "Date"
```

will not give rise to an error message, but no **sort** will take place unless the word "**field**" appears in front of "**Date**". Such constructs are inconsistent with students' expectations, and are somewhat error-prone.

In general, though, Hypertalk is highly readable and unsurprising, with high role-expressiveness and a lack of hard mental operations.

5.2.15 Visual Basic

5.2.15.1 Background

Visual Basic is perhaps the most widely used programming language today. Despite its name, Visual Basic is primarily a text-based language with a tightly-coupled integrated development environment. Derived from BASIC, a popular programming language developed in the 1960s, Visual Basic is commonly used for the creation of graphical user interfaces (GUIs) for Microsoft Windows. The

"Visual" in Visual Basic refers to the facilities for the creation of GUIs. The development environment allows the creation of various GUI objects - such as buttons, menus, windows, and text boxes - at the click of a button, without the need for any code written by the user. However, these objects nearly always require code in order to perform the desired functions, so the programmer must ultimately come to grips with the underlying textual programming language.

5.2.15.2 Usability

Visual Basic scores well on closeness of mapping for creation of visible elements of user interfaces, as objects may be dragged and dropped, and the form of an interface can be created quickly and easily. The creation of such an interface parallels its use, and is simple and intuitive. Unfortunately the underlying textual language is not as closely mapped onto user experience and expectations. For example, the code:

```
Private Test, Amount, J as Integer
```

declares **Test** and **Amount** to be of type **Variant** (the default data type) and only **J** is an **Integer**, although the code seems to imply that all three variables have been declared as **Integers**. The array declaration syntax is also problematic. When an array is declared, bounds are specified rather than size. This is a positive step if both bounds are declared, as it makes it clear what the legal bounds of the array are. Unfortunately the lower bound need not be specified, and it defaults to zero. Thus the array declaration **Dim Counters(15)** specifies an array of size **16**, going from **0** to **15**. This also increases the error-proneness of the language.

Consistency is a problem in Visual Basic. For example, the built-in value conversion function to convert to a **Long** is **CLng**, to a **Currency** is **CCur**, but to a **Boolean** it is **Cbool**. Why the '**o**' is omitted from **Long**, or the "**b**" in **Cbool** is lower case, is not clear. Consistency is further threatened by the existence of multiple syntactic forms for single constructs. For example, the following three statements all call a procedure, **F**, with parameters **x** and **y**:

```
F x,y
Call F(x,y)
F a:=x,b:=y
```

Role expressiveness could be better in Visual Basic with better choice of keywords: for example, the keyword `Dim`, for declaring a variable, is rather obscure. Other keywords, such as `Call`, for calling a subroutine, are much more role-expressive.

Visual Basic is effective for event-driven interface programming, but does possess hard mental operations when the user is trying to keep track of the many different components of a single Visual Basic program: forms, modules, classes, objects, etc.

5.3 Conclusion

Table 5.2 summarises the usability of the programming languages analysed here. The optimal values listed are purely from the perspective of novices learning introductory programming.

The usability issues discussed in this chapter give rise to the question - how do usability problems in programming languages arise, and how can we prevent them? Chapter 6 discusses some of the common problems in programming language design, examining the origins of many popular programming constructs which cause problems for beginners and even expert programmers.

Language/ Dimension	*Optimal*	Pascal	ABC	Turing	Modula-3	Prolog	C	C++	Ada	Smalltalk	Java	Scheme	Logo	Haskell	Hypertalk	Visual Basic
Closeness of mapping	*high*	med	high	med-high	med	med-high	low	low-med	low-med	low-med	low	low	high	med	high	low-med
Consistency	*high*	low-med	high	high	med	med - high	low	low	med	high	low-med	med-high	high	med-high	low-med	low
Diffuseness	*med-high*	low-med	med-high	low-med	low-med	low	low	low	med-high	med	low	low	med	low	high	med
Error-proneness	*low*	low-med	low-med	low-med	med	med-high	high	high	med	low-med	med-high	med-high	med	low-med	med	med
Hard mental operations	*low*	low-med	low	low-med	med	med-high	high	high	high	med-high	med-high	med-high	low	med-high	low	med
Role-expressive-ness	*high*	med-high	med-high	med-high	med-high	med	low	low-med	med	med	low	low	high	med-high	high	low-med

Table 5.2 Cognitive Dimensions of Programming Languages

6 A theory of bad pedagogical programming language design[1]

Building on the usability analysis of programming languages presented in Chapter 5, this chapter describes serious pedagogical problems which are common to many languages used as introductory programming languages (McIver and Conway, 1996).

6.1 Language Traps

6.1.1 Less is more

The "less is more" principle appears in many forms, almost all of which seem to be ultimately detrimental to the learning process. Perhaps the most obvious examples are the Scheme language and other LISP variants. Scheme has effectively only one data type – the list – and one operation – evaluation of a list. While this abstraction is very simple to explain, and not difficult for the beginner to grasp superficially, it does result in code that is difficult to read because of large numbers of nested parentheses and the absence of other structuring punctuation.

A "less is more" approach is usually justified in terms of paradigmatic purity: strict adherence to a single functional, logical or object-oriented paradigm. While this "orthodoxy" can make for a certain conceptual simplicity and elegance (which can be of considerable advantage in teaching concepts such as scoping, recursion, and encapsulation), in practice it can also lead to extremely obscure and unreadable code. In some cases, relatively simple programs must be substantially restructured to achieve even basic effects such as input and output.

The underlying pedagogical difficulty is that students are not used to solving problems in a single pure paradigm. Much of the problem solving they do in the real world is procedural in nature (cooking a meal, totalling a restaurant bill, etc.), but other problems with which they are familiar are more amenable to constraint solving (dispute resolution, holiday planning, budgeting), a functional or pipe-line approach (collaborative tasks, various types of component assembly), or even object-oriented methods (using an automatic teller machine, learning physical skills).

---

[1] Material in this chapter was presented in McIver & Conway (1996)

The results of enforcing paradigmatic purity can be as simple as the annoying requirement in Turing that functions have no side-effects, or as far reaching as the lack of I/O in pure functional languages such as Miranda (Turner, 1985).

6.1.2 More is more

It is equally true that many languages are based on design philosophies which err in the other extreme. Powerful, real world languages (for example C++ and Ada) are amongst the prime culprits here. Often such languages are taught by subsetting – teaching a small but usable part of the language whilst ignoring its more powerful features.

At first glance this approach seems quite reasonable, but two pedagogical problems frequently sabotage it. The first is that textbooks and other reference materials rarely confine themselves to the selected subset. The second is that, even if the textbook does limit itself to the required subset, the compiler almost certainly does not. As a result, error messages produced by the compiler, syntax directed editing provided by the environment, and examples used in the textbook inevitably refer to parts of the language from which the student is being "protected" – i.e. have deliberately not been explained or described. The result is often worse than if the full language was taught: students must still contend with the full semantics of the language, but much of it has deliberately not been explained to them. See section 4.4.1 for a more detailed discussion of subsetting.

Writing programs in a subset of a language still requires a working knowledge of, at the very least, the entire set of reserved words which may not be used as identifiers, and reserved characters which may not be defined as new operators (except in the case of deliberate overloading). While attempted compilation may detect errors such as these, they can be demoralising and frustrating, as they are errors which the student had no way of preventing, short of checking a list of reserved words and operators every time a new identifier or operator is declared. The larger the set of reserved words and operators, the more onerous and impractical this task becomes, and the greater the likelihood of such errors arising.

Even where these errors are detected, the error messages may not be meaningful to the student. An error message about a redefinition of something previously defined, or, worse, simply a syntax error on a particular line, does not provide clear guidance to a student who knows she has only defined an object once, or who cannot detect the syntax error on a line which looks perfectly legal (and

indeed probably is perfectly legal, except for the use of a reserved word of which the student was unaware).

C++ is certainly one of the most popular languages in "real-world" use and (for that very reason) is also increasingly widely taught as an introductory language. One of the justifications typically cited for teaching C++ (Conway, 1993) is the range of low- and high-level features it provides (from bit manipulation of raw pointers to templated abstract classes with polymorphic member functions.)

Beginners, however, are notoriously poor at dealing with two or more conceptual perspectives simultaneously (Hofstadter, 1979). Dichotomies of perspective (such as syntax *vs* semantics, static *vs* dynamic structure, process *vs* data) complicate the teaching of any programming language. The availability of very low-level implementation-oriented constructs and high-level solution-oriented features in a single language only serves to increase substantially the already considerable cognitive demands placed on the student.

As well as the obvious concerns regarding learning curves, confusion of levels, and the difficulties of adequate error detection, a wide range of features necessitates a commensurately complex syntax and often also entails a host of implicit operations and function calls, automatic conversions, type inferences, and resolutions of overloaded functions, varia and function scoping.

Examples of this "creeping featuritis" are easy to cite: C++ provides over 50 distinct operators at 17 levels of precedence, Ada9X has 68 reserved words and over 50 pre-defined attributes, Modula 3 reserves over 100 keywords, and some commonly-used LISP dialects ((Wilensky, 1984) for example) define over 500 special functions. Because most textbooks and compilers attempt to cover the full language, novice programmers are forced to contend with all of these features, even if they are not using them.

6.1.3 Grammatical traps

Another class of pedagogical problems stems from various kinds of "unfortunate" syntactic and semantic constructs which are present in most introductory languages. Some of these constructs arise from the constraints of the ASCII character set, whilst others are the result of a deliberate "less-is-more" design policy. The common feature of these problems is that they are analogous to certain sophisticated grammatical constructs in natural languages, and result in the same types of learning problems as are seen in natural language acquisition.

One such construct is the *syntactic synonym*, in which two or more syntaxes are available to specify a single construct. An common example of this is dynamic array access in C, wherein the second[2] element of an array may be accessed by any of the following syntaxes, some of which are legal only in certain contexts:

```
array[1]
*(array+1)
1[array]
*++array
```

Less well-known examples include list construction in Haskell (where `[1,2,3]` is synonymous with `1:(2:(3:[]))`) and Prolog (where both `.(a,.(b,.(c,[])))` and `[a,b,c]` produce the list `(a,b,c)`).

In themselves, synonyms are a minor irritant. However, they can have a more serious and insidious effect by blurring the underlying programming concept in the student's mind, because that concept is no longer associated with a single clear syntax.

Syntactic homonyms are constructs which are syntactically the same, but which have two or more different meanings depending on context. Such constructs are perhaps a more serious flaw in a language, and are unfortunately common. An extreme example of this[3] may be seen in the pedagogically-oriented language Turing, in which the construct `A(B)` has five distinct meanings:

- call function `A` and pass parameter `B`,
- dereference pointer `B` to access an object in collection `A`,
- access the `B`th element of array `A`,
- construct a set of type `A` with a single element having the value of `B`.
- create a one-letter substring of the string `A` consisting of its `B`th character,

The student, armed with only a fuzzy understanding of the differences between these concepts, finds no support from the syntax. It should be noted that the decision to overload this construct was taken quite deliberately and on pedagogical grounds:

---

[2] The fact that `array[1]` refers to the second element of array is itself a grammatical trap.

[3] But not as extreme as LISP and its variants, which could be viewed as one massive homonym.

> *"Notice that referencing an element of array* **a** *with subscript* **i** *as in* **a(i)** *is notationally equivalent to* **c(p)**. *This is an example of uniform referents, which means that analogous ways of accessing data should be notationally equivalent." (Holt et al, 1988)*

Ada and Turing share a common notation for array indexing and parameter passing, based on the notion that the expressions **myArray(index)** and **myFunction(parameter)** are analogous, in that they each return a discrete value. While it draws attention to the similar features of these constructs, using the same notation for both also hides their differences, and makes it impossible to tell at a glance whether the statement **y:=g(x)** ; is a function call or an array access, without checking back to find the type of **g**. The notion of returning a value applies in both cases, but it is necessary for the programmer to distinguish between a function call and an array access in order to determine whether a statement such as **g(x) :=y;** is valid.

Careful and mnemonic choice of names for identifiers addresses this problem to some extent, but if the names reflect the problem space rather than the language constructs, the difficulty can still exist. Consider the expression **classroom(fred)** - is this an array holding the classroom in which **fred** can be found (hence the statement **classroom(fred) :="4B"** ; is valid)? Or is it a function which calculates the appropriate classroom based on **fred**'s year level and subject profile (in which case **classroom(fred) :="4B"** ; would probably be an error)?

Another difficult grammatical construct which frequently appears in languages is *elision* (the omission of a syntactic component). C is well known for its default integer return value and its curious string literal concatenation behaviour, but default behaviours occur in many languages, for example:

- Type inference in Haskell. Function definitions in Haskell need not include type declarations. A function which is defined without a type declaration will have its type inferred by the Haskell compiler/interpreter.
- LISP super-brackets. The symbol '**]**' can be used in LISP to close any remaining open parentheses. This is a convenient notation, as it avoids long lists of closing parentheses which can be hard to balance with open ones. However, it is dangerously error-prone, since an open parenthesis which

should have been closed earlier in the code may not be detected by the interpreter once a super-bracket has been used.

- Switch fall-through in C++. The switch statement in C++ is a notorious source of errors. Each case must be explicitly terminated by a **`break;`** statement, otherwise subsequent cases will also be executed, until either a **`break;`** or the end of the switch is encountered.
- Automated sorting of lists in ABC. Lists in ABC are discussed in more detail in section 6.1.7.

6.1.4 Hardware dependence

In addition to battling the various syntactic and semantic levels of an introductory language, the novice programmer is often forced to contend simultaneously with the constraints of the underlying hardware This burden is often imposed merely for the convenience of the compiler writer, or in the name of efficiency.

This "closeness to the metal" is particularly noticeable in the design and implementation of basic numerical and character string types. There seems no convincing reason why the novice student, already struggling to master the syntax and semantics of various constructs, should also be forced to deal with the details of representational precision, varying machine word sizes, awkward memory models, or a profusion of conceptually-equivalent but semantically-distinct data types.

The semantics of arrays in Pascal, in which the novice must grapple with the fundamental type difference of arrays of different lengths, is a notable example (see section 5.2.1.2). The presence of thirty-two distinct numerical data types in C/C++[4] is another. These types are particularly problematical in C as they are generally not portable (see section 5.2.6.2). Modula-3 is explicitly and deliberately hardware dependent, disallowing expressions with mixed operands (integer and real, for example) for reasons of efficiency.

---

[4] **`int`**, **`short`**, **`long`** and their **`unsigned`** equivalents; **`float`** and **`double`**; and **`const`** and/or **`volatile`** variants of each.

6.1.5 Backwards compatibility

Backwards compatibility is a useful property from the experienced programmer's point of view, as it promotes reuse of both code and programming skills. Novices, however, can take no advantage of these benefits (they have no such resources to reuse) and must instead bear the pedagogical costs they entail.

Backwards compatibility comes in two major forms: *genetic* and *memetic*. Whilst both forms can lead to pedagogically suspect decisions during the design of a language, genetic compatibility is generally the result of a conscious decision on the part of the language designers, whereas memetic compatibility is frequently inadvertent.

6.1.5.1 Genetic compatibility

Genetic compatibility is exemplified by the relationship between languages such as C++ and C (Stroustrup, 1994), Scheme and LISP (Springer & Friedman, 1989), and Turing, Euclid and Pascal (Holt et al, 1988). It results from the decision to retain the semantics and often the general syntactic "look-and-feel" of an ancestor language.

Genetic compatibility need not of course imply the near-complete backwards compatibility as seen in the C/C++ relationship (Turing and Scheme differ significantly from their ancestors), nonetheless languages that attempt a significant degree of historical consistency inevitably perpetuate some problematical constructs.

> *Complex arguments are invented to justify features of BASIC that were originally included because the primitive technology demanded them or because alternatives were not well enough known at the time the language was designed." (Papert, 1993)*

Language designers occasionally acknowledge the problems that their quest for genetic compatibility produces:

> *"At this point, the reader may be confused at having so many ways to define a function! The decision to provide these mechanisms partly reflects historical conventions, and partly reflects the desire for consistency (for example, in the treatment of infix vs. regular functions)." (Hudak & Fasel, 1992)*

"Over the years, C++'s greatest strength and its greatest weakness has been its C compatibility. This came as no surprise." (Stroustrup, 1994)

As well as the confusion-of-level problems alluded to in section 6.1.2, the addition of new concepts to an old language often leads to the creation of the sorts of synonyms or homonyms discussed in section 6.1.3, as well as the perpetuation of poorly-designed constructs (such as **`char*`** strings in C++) or awkward syntax (for example, the inexplicably-named[5] **`car`** and **`cdr`** which Scheme inherits from LISP).

6.1.5.2 *Memetic compatibility*

Not all syntactic or semantic inheritance stems from deliberate decisions regarding backwards compatibility. Dawkins (1989) defines *memes* as ideas which propagate themselves (unconsciously) from mind to mind. For example, many people unconsciously imitate their parents' way of doing something – making a cake, weeding a garden bed, or washing a car. The technique is observed many times during childhood, and the meme is picked up without the "carrier" even being aware of it.

Some constructs and symbols seem to propagate memetically across language family boundaries, and have become de facto standards within the programming community. This is often despite the fact that such constructs may originally have been viewed by their progenitors as unsatisfactory compromises and may indeed have no discernible connection with the concepts they are intended to represent. Memetic compatibility is surprisingly pervasive and may be seen in the widespread use of "standard" symbols such as `*` for multiplication, `=` or `:=` for assignment, **`array[index]`** for indexing.

The major pedagogical problem with the presence of such syntactic memes is that they significantly reduce the degree to which the novice, an outsider to the programmer culture, can rely on existing knowledge (such as × meaning multiply, or the use of subscripts to represent indexing of a collection).

Unfortunately memetic compatibilities can also be particularly difficult to identify (and their pedagogical effects correspondingly hard to analyse), precisely because

---

[5] "Inexplicable" in the sense that explaining that they derive from "Contents of Address Register" and "Contents of Decrement Register" respectively, rarely assists the student's comprehension or recall.

both the language designer and the programming teacher are so familiar with them.

Keywords in programming languages are frequently memes propagated through from past programming languages. Worse situations arise when memes are abbreviated, since the novice must then extrapolate an unfamiliar contraction into an equally unfamiliar term, and then determine the meaning of that term. One example of this is the keyword **var** in Turing. Experienced programmers mentally expand **var** to **variable** without effort, but novice programmers, unfamiliar with the concept of a variable, may find this more difficult, and may first translate **var** to variety. Similarly **const** is probably obvious to anyone with programming experience, but even knowing that it expands to **constant** may not immediately help the novice - "constant" means, among other things, "continuous", and "occurring frequently". Nonetheless, the keywords **const** and **constant** are taken for granted in programming.

6.1.6 Excessive cleverness

Instances of "excessive" cleverness can be difficult to spot, precisely because the "excess" often exists only relative to the knowledge level of the novice. Indeed, frequently the only way to detect excessive cleverness is to witness a novice programmer's complete misunderstanding of an "obvious" concept.

The premier example of the adverse effects of cleverness in programming language design (and one which is obvious to programmers at all skill levels) must surely be the C/C++ declaration syntax (Ritchie, 1996). On the surface, it seems like an excellent notion: let declarations mirror usage. Unfortunately, the very concept undermines the principle of visually differentiating semantic differences. Students have enough trouble mentally separating the concepts of declaration and usage, without the syntax conspiring to blur that crucial difference. Consider the following C declarations:

```
int i,j,k;      /*  i, j, and k are all the type at the
                start of the declaration - int     */
int* i,j,k;     /* only i is an int*, j and k are still
                ints  */
```

Sometimes a genuinely clever idea can be sabotaged by its own syntax. For example, in Turing dynamic memory may be partitioned into strictly typed "collections", which are then capable of storing dynamically allocated instances of a single data type. Pointer variables may be associated with a particular collection and can only be used to refer to data within the collection. This approach provides strong type checking of dynamic memory and enables the compiler to catch and accurately diagnose the majority of common pointer manipulation errors.

Unfortunately, this genuinely clever idea is disastrously undermined by poor choice of syntax:

```
% Declare a collection to store instances of SomeDataType
    var collectionName : collection of SomeDataType
% Declare a pointer
    var ptr : pointer to collectionName
% Instantiate a new object of SomeDataType
    new collectionName, instance
```

Students immediately (but erroneously) conclude that:

- **collectionName** is a variable. It's not, it's actually a partition of dynamic memory and does not have the full semantics of a variable. Unlike variables, collections cannot be assigned, compared or passed as parameters. They cannot be named in a **const** declaration, nor can they be renamed using a **bind** statement.
- **ptr** can be used to point to **collectionName**. It can't, it can only be used to point at instances of **SomeDataType** allocated within the partition accessed via **collectionName**.
- **instance** is a new instance of type **collectionName** . It isn't. Rather, **instance** points to an instance of type **SomeDataType** newly allocated within **collectionName**.

6.1.7 Violation of expectations

As the last example in the previous section indicates, violating a reasonable expectation is probably the worst pedagogical sin that an introductory programming language designer can commit. Some examples are very well-known, such as the C++ expression:

```
if (x=1 || -10<y<10)
{
     /* WHATEVER */
}
```

in which the condition always evaluates to true (regardless of the values of **x** and **y**) whilst, as a side effect, the value of **x** is silently reset to **1**.

The expression **x=0**, rather than being a test to see if the value of **x** is **0**, actually assigns the value **0** to **x**. Even experienced programmers who are familiar with this trap often still write = where they intend to write ==. This is a natural response to the spoken pronunciation of the expression: "x equals zero". Particularly insidious is the fact that this code is perfectly legal and compiles without even generating a warning under many compilers.

The expression **-10<y<10** does not, as one might reasonably expect, evaluate to the equivalent mathematical expression and test whether **-10** is less than **y** which, in turn, is less than **10**. Instead, C defines that the first comparison should be evaluated, with the result of that comparison then being used in the second comparison. In other words, if **-10<y** evaluates to *false*, the final expression to be evaluated is *false*<**z**. Due to the definition of *true* (non-zero) and *false* (zero), the expression effectively becomes **0<10**. This can be particularly difficult to detect since, depending on the value of **y**, it will not always produce the wrong answer, and it does not trigger an error or a warning at compile time.

A less obvious example of syntax violating expectations is the use of **%** as a comment introducer in Turing. The following code is syntactically correct and semantically valid, but will result in unexpectedly low pass rates, since the **%** makes the rest of the line a comment:

```
passMark := bestMark * 50%
```

Semantic violations of expectation are even less excusable, but regrettably more common. For example, consider the **list** type in the ABC programming language. A novice may have written a seemingly straight-forward program to store a list and then print the first element:

```
PUT {"first", "second", "third", "fourth", "fifth"} IN list
WRITE list item 1
```

He may well be considerably puzzled and disheartened when the result appears:

```
"fifth"
```

The blame for this minor failure can hardly be laid on the novice, who may simply have forgotten (or perhaps never grasped) the fact that ABC lists are automatically sorted on input. The fault lies squarely with the language designer. Although a sorting function is an extremely useful capacity in a language, hidden side effects such as this can be highly confusing for the inexperienced user, especially when the "magic" gets in the way of the programming task.

Even the semantics of fundamental and nearly universal programming memes, such the **`while`** loop and the finite precision integer, can be surprisingly difficult for students to comprehend. A **`while`** loop doesn't execute "while" its condition is true, but rather until its condition ceases to be true at the end of its associated code block. Finite precision integers don't obey the familiar rules of whole number arithmetic and can also cause much confusion when overflow, underflow or truncation produce consequential errors (which may manifest well after the actual numerical error occurred).

The Prolog equality operator (**`X=Y`**) violates expectations in another way, in that it implies an assignment of reference as a side effect:

> *"If* **`X`** *is an uninstantiated variable and if* **`Y`** *is any object, then* **`X`** *and* **`Y`** *are equal. As a side-effect,* **`X`** *will be instantiated to whatever* **`Y`** *is. [When* **`X`** *and* **`Y`** *are both uninstantiated,] the goal succeeds, and the two variables share. If two variables share, then whenever one of them becomes instantiated to some term, the other one automatically is instantiated to the same term." (Clocksin & Mellish, 1981)*

Ada possesses a convenient notation for accessing partial arrays, known in Ada as "slices".

For example:

```
Value_Record(1..20) := Value_Record(21..40);
```

While this notation is extremely convenient, it has a potential drawback if a slice is passed as a parameter – the formal parameter possesses the same index range as the slice. In other words, passing a slice from **`21..40`** bestows the index range **`21`** to **`40`** on the formal parameter inside the subroutine. For example:

```
File_Name : String(1..32);
procedure Open (The_File : in String);
```

and then, in the body of Open:

```
if The_File(1) = 'Y' then...
```

would raise an exception in the case of the following call

```
Open(File_Name(16..32));
```

because index **1** is out of bounds in the array slice passed in (**File_Name(16..32)**).

Attributes can be used to avoid this problem (**The_File'First** instead of **The_File(1)**), which reinforces a more general and arguably less error-prone syntax. Nonetheless, it is unusual, and potentially error-prone, for the range of the formal parameter not to be independent of the range of the actual parameter. The use of the term **First** in the attributes of the parameter acknowledges that **The_File** clearly has a first element, but it cannot be accessed as **The_File(1)** unless the range of the actual parameter can be guaranteed to start from **1** every time.

6.1.8 Dangerous side effects

Many otherwise effective and usable constructs possess dangerous side effects which can prove surprising and problematic for beginners. These side effects tend to arise for two reasons

1. The side effect was deliberately added to the language as a means of circumventing some other problem, or to provide a more convenient and powerful mechanism for solving a particular programming problem, or
2. The side effect is an unintentional result of the pursuit of efficiency or power.

For example, Pascal contains some side-effect traps for unwary novice programmers (and even for experts). The Pascal **for** loop, eg **for i := 1 to num do**, has a clear syntax - it is reasonably obvious that **i** takes values from **1** to **num** (although the step size (**1**) is not explicitly stated here, it is a reasonable default). But Pascal allows the programmer to modify the value of **i** inside the loop.

For example:

```
for i := 1 to num do
   BEGIN
    i:=num;
    writeln(message);
   END;
```

This is useful where the programmer wishes to break out of the loop before the specified number of iterations has been performed – just set the value of **i** to **num** and the loop will terminate – but if the modification of **i** is inadvertent (**i** might be passed as a **var** parameter to a subroutine), or simply ill-thought out, the loop will exhibit unexpected behaviour and cause confusion for the novice programmer.

6.1.9 Fitting the wrong model

Sometimes programming languages cause problems for programmers because the design of certain features is based on the wrong model of computation. For example, the C requirement for an **&** before the variable in a **scanf** statement is a logical way to indicate that the address of the receiving variable is to be passed to **scanf**, rather than the value. Given an understanding of the difference between calling by reference and calling by value, this makes sense.

However, students using **scanf** for the first time rarely possess such an understanding. Moreover, they are likely to be ill-equipped to understand the concepts involved, having just begun to learn programming. Asking the students to accept the necessity for the **&** without understanding its use causes problems as soon as arrays of characters, or strings are used. To read in a string of text into a character array the **&** is not used, since the value of a **char *** variable is already the address of the start of the array. The use of I/O in C is not easily deferred until later in the course, as it is the only way for students to see the results of their programs, so this problem is not easy to handle.

As described in section 5.2.6.2, C also has a potentially troublesome **for** loop. Most versions of a **for** loop go from a starting point to an end point, as in **for i:=1 to 10**, where **i** starts off with the value **1**, and is incremented by one each time through the loop, until **i** is equal to **10**, which is the last time through the loop.

C, however, uses different semantics for the **for** loop, using three statements to set up the loop. Traditionally the first statement is an initialization (such as **i=1;**) the second statement is an expression which must evaluate to true (non zero) for the body of the loop to be executed (such as **i<10;**) and the third statement is an increment (such as **i++;**). Even in this traditional usage, the **for** loop poses some traps for unwary novices. Rather than the common model "**for index:=x to y;** execute for all values of **index** from **x** to **y** inclusive, incrementing by steps of 1", the typical C version uses a different model:

"**for(statement1; statement2; statement3)** : On the first time through the loop, execute **statement**[1], then if **statement2** is true, execute the body of the loop, and then execute **statement3**. On subsequent passes through the loop, do not execute **statement1**, but while **statement2** is true, repeatedly execute the body of the loop followed by **statement3**."

Because the second statement acts like the condition of a while loop, the loop **for (i=1;i==10;i++)** will not be executed even once, because the test **i==10** fails on the first pass. Instead, the statement **i<=10** is needed to achieve the same result as **for i:=1 to 10**.

6.2 Summary

This chapter has described some pedagogical problems which are frequently found in languages used for introductory programming. Chapter 7 takes the next step on the road to the design of a pedagogically sound introductory programming language – the construction of a theory of good pedagogical programming design. This theory will provide a positive framework of design goals and principles which can be used in the design of a usable introductory programming language.

7 A theory of good pedagogical programming language design[1]

As discussed in Chapter 4, there are some basic requirements to be met in order to achieve meaningful learning. Meaningful learning takes place when new knowledge can be attached to an existing cognitive framework, and learning is most difficult when new knowledge actually conflicts with what is already known. As a result, an introductory programming language will be most effective if it can match what students already know wherever possible. Where matching existing knowledge is not possible, sometimes the best the language can do is to be *so* different as to avoid conflicting with it.

An effective tool or application allows the user to focus on the task to be performed, not on the tool itself (Norman, 1998). By extension, an effective programming language facilitates programming, rather than getting in the way. The goal is to create a programming language that adheres to usability principles: one that maps closely onto the way users think and solve problems, is consistent, does not require hard mental operations, is not error-prone, and is high in role-expressiveness (see section 5.2).

7.1 The issues

7.1.1 Concepts to be taught

Programming is, of necessity, a practical subject. The theory of programming is difficult, if not impossible, to teach in isolation from the practice. An introductory programming course needs to involve a programming language, although it may aim to teach the fundamental concepts behind programming, not specifically a particular paradigm or a particular language. Programming must be experienced and practised in order to be comprehended fully.

There is some consensus on the concepts that need to be taught when teaching programming, although there is little agreement about how those concepts should be taught (Brilliant & Wiseman, 1996; Brusilovsky et al, 1994; M'Iver & Conway, 1996), or indeed at what point in the course each concept should be introduced. There is some weight to the argument that pointers and references, for example, should be postponed until novices have built some confidence and gained more

---

[1] Material in this chapter was presented in McIver & Conway (1996)

basic skills. Others argue that pointers take a long time to come to grips with, so students should start the process as early as possible.

Some of the fundamental programming concepts involved in an introductory course are listed here:

- algorithms
- data structures
- sequence/flow of control
- selection
- iteration
- functions
- recursion
- abstraction

7.1.2 Motivation

The style of language can impact on the motivation level of students. Motivational tools can be "added extras": For example, Blue (K*lling, 1996a), although it is a relatively complex object-oriented language, has achieved considerable motivational success with students through the inclusion of an adventure game project where students can alter an existing game, changing existing modules and adding new parts to the game. Similarly, LOGO (Papert, 1993) uses its turtle graphics concept to encourage experimentation and creativity, and some packages have graphics and interface libraries that give students an easy way to create interesting interfaces to their programs.

While it is often "gimmicks" such as those listed above that increase motivation levels most dramatically, the language itself can be a drain on motivation if it is too frustrating and difficult to learn. A language that is hard to read and uses obscure operators and keywords can drain student motivation, particularly for those students who lack a strong computing background.

7.1.3 Implementation issues

The challenges of implementation should not be considered in the design process. Implementation issues have coloured the design of many existing programming languages, affecting everything from the size of an integer to the inclusion or

exclusion of features purely for reasons of speed and efficiency, such as the exclusion of automatic garbage collection in C++. In an introductory programming language, speed and efficiency are not primary considerations. The language must certainly compile and run within a reasonable time frame, to avoid frustration and the loss of motivation and concentration, but this requirement is easily met with existing technology, without requiring extremes of efficiency.

7.2 General design imperatives

The theory of pedagogical programming language design constructed as part of this thesis hinges on three high-level design imperatives: facilitate learning, maximise readability, and minimise unproductive errors. These imperatives guided the development of the lower level, specific design imperatives used to direct the design process.

7.2.1 Facilitate learning

A teaching language should facilitate learning to program. This means not getting in the way of learning by causing unnecessary syntax errors and having excessively complex semantics. Facilitating learning requires support for the concepts students need to learn (see section 7.1.1), informative and non-threatening feedback on errors, and a minimum of distraction from the development environment.

7.2.2 Maximise readability

There are two primary aspects to the learning of programming - one is the ability to write a program, the other is the ability to read code. While both reading and writing programs require a certain amount of skill and familiarity with the programming language, it has long been acknowledged (Green, 1989) that reading a program written by someone else can be extremely difficult, unless the program is deliberately written in a readable way.

One goal of an introductory language, then, is to support and encourage readability as a product of the nature of the language, rather than simply to allow readability as a possible alternative for those programmers who wish to make the extra effort. Ideally an introductory language would actually make it difficult to produce unreadable programs.

7.2.3 Minimise unnecessary errors

While errors are an important part of the learning process, not all errors are productive errors that facilitate learning. Trivial, frustrating and disruptive errors can actually hinder the learning process, leading to phenomena such as learned helplessness (see section 2.4.4). Small syntax errors that disrupt the flow of work without leading to meaningful feedback, such as leaving out a semicolon, or using the wrong type of bracket, can impede learning and make learning to program a frustrating and unhappy experience. The design of an effective pedagogical language minimises the incidence of these unnecessary errors.

7.3 Specific design imperatives

The example design process carried out for this thesis has lead to the formulation of a number of low level design rules. Not every rule can be satisfied by every construct, and some rules have higher precedence than others. Taken together, these rules provide a framework for the design of a programming language that possesses high usability in the context of teaching introductory programming. More generally, many of these principles apply equally well to the design of a usable programming language for expert programmers.

7.3.1 Start where the novice is.

A fundamental aspect of learning is the process of assimilating new concepts into an existing cognitive structure (Thorndike, 1932; Ausubel, 1963; Rumelhart & Norman, 1978). This process, known variously as *connecting*, *accretion*, or *subsumption*, is made all the more difficult if parts of the existing structure have to be removed (*unlearning*) or restricted (*exceptions*).

Hence, the novice who must unlearn that × or • means multiply, and then substitute * in a programming context, faces a harder learning task than the student who can continue to put their knowledge of × to use. Similarly, students have a large corpus of knowledge regarding integer and real arithmetic, that cannot be capitalised upon if they must disregard it to cope with finite precision representations.

Another example of this type of difficulty is the use of = variants for assignment. Many students, when confronted with this operator, become confused as to the nature of assignment and its relationship to equality. For example, seeing the following statements :

```
X = 3
Y = X
Y = 7
```

such students often expect the value of `X` to be equal to `7`(Du Boulay, 1989). The equivalent sequence:

```
X ← 3
Y ← X
Y ← 7
```

seems to evoke less confusion, possibly because the syntax reinforces the notion of procedural *transfer* of value, rather than transitive *equality* of value.

As part of this thesis, over one thousand novice programming students were shown the C/C++ expression:

```
"the quick brown fox" + "jumps over the lazy dog"
```

and asked what they believe the effect of the + sign is. None of them has ever (correctly) suggested that the + sign is illegally attempting to add the address of the locations of the first characters of the two literal strings. Without exception, they believed that the + should concatenate the two strings.

Introductory languages should be designed so that such reasonable assumptions based on prior non-programming-based knowledge remain reasonable assumptions in the programming domain. In other words, the constructs of a teaching language should not violate student expectations. Note that this principle has both syntactic and semantic implications in the selection and definition of operators, functions, and inbuilt data types.

Existing knowledge should not be contradicted. New knowledge that conflicts with existing cognitive structures causes cognitive dissonance, and is much harder to understand and retain than knowledge that either fits into existing cognitive structures or requires the creation of entirely new structures. In this respect it is preferable to introduce an entirely unknown operator, rather than "misuse" a known one. For example, "`eq`" might make a better choice for equality testing than "`==`". The more features in a language that conflict with what students know or the way they think, the more difficult it will be for them to learn, and the more error-prone it is likely to be.

7.3.2 Avoid jargon

Jargon is difficult to avoid, in part because it is difficult for an expert to identify jargon that will be unfamiliar to a beginner. Jargon in programming languages applies to both keywords and operators, as well as to the environment and support tools such as help systems and text books. Avoiding jargon involves avoiding using operators and keywords simply because they are memetic programming "standards", or because they are familiar to the language designer, and instead choosing the most novice-appropriate words and symbols for the construct. This is particularly challenging because it requires designers to identify their own cultural biases.

Avoiding jargon requires careful analysis of the reasons for selecting each and every construct. Keywords are particularly difficult to analyse, because they are often so familiar to experienced programmers. Consider the Modula-3 keywords **INC** and **DEC**. In English, the abbreviation "inc." usually means "including" or "incorporated", while "dec." usually means "deceased" or "declared". Experienced programmers, even those who have no experience of Modula-3, are likely to recognise **INC** and **DEC** effortlessly as increment and decrement, but these meanings may not be so obvious to novices.

7.3.3 Favour simplicity over power

A feature should never be included simply because it is powerful - it must also be intuitive, readable, and easy to understand and use. Powerful features are useful for experienced programmers, making complex programs simpler and easier to write, but for novice programmers, an excess of powerful features can be counter-productive.

Some powerful features obviate the need for students to learn how to solve a particular problem themselves, for example sort functions, or lists that are automatically sorted. Others can actually be obstructive and dangerous, with unexpected consequences. For example, equality testing in Prolog, which has the side-effect of assignment of reference, if one of the variables is uninstantiated (see section 6.1.7).

7.3.4 Make features self-explanatory

It is important that constructs should stand alone without requiring extra explanation. It should be possible for a novice to determine the workings of a

construct from code examples. In other words, the role-expressiveness of each construct is important to the overall usability of the language.

This means that any construct that does not possess clear and logical semantics should only be included where there are strong reasons for doing so, which outweigh the disadvantage of the extra explanation required. Though it is unlikely that a construct will be fully understood without experimentation and or some instruction from a teacher, a fellow student, a textbook, or a help system, these aids should never be relied upon to make the construct usable.

The Turing **`collection`** construct is a good example of a feature that is not self-explanatory, largely due to the lack of correspondence between syntax and semantics (see section 6.1.6). The complex semantics of a collection require considerable explanation.

7.3.5 Avoid unexpected results

Automatic and default behaviours can be extremely confusing for a novice. Implicit behaviours that silently change the state of a program or variable can be responsible for the violation of reasonable expectations. The behaviour of a construct should be able to be anticipated or assumed. Where behaviour is more complex than the syntax expresses, the complexity should be made explicit, with extra syntax if necessary.

Implicit calls to functions, such as constructors and destructors, are undoubtedly powerful and useful constructs for expert programmers who are aware of the rules for their use, and the situations when they will be called. Novices, however, do not yet possess this understanding, and may be puzzled when trying to debug a program that appears to be doing things they have not told it to do.

Unexpected results can challenge students' mental models of the virtual machine, and lead them to believe that the machine has powers that it does not, or does not have powers that it actually does. For example, lists that are automatically sorted can lead students to believe that all data structures are automatically sorted, or that the computer "knows" more than it does, so that variables named **`maximum`** will automatically contain the largest value, etc. By the same token, an attempt to read in a number from the keyboard that fails due to a decimal point (as may happen if a real value is mistakenly read in to an integer variable in Pascal) may lead students to believe that real values cannot be read in from the keyboard.

7.3.6 Never complicate the simplest programs

Any construct that introduces changes to the most basic operations, such as I/O or simple assignment and arithmetic, should be approached with extreme caution. The priority for a pedagogical language aimed at under-confident beginners is an easy, painless, and confidence-building experience, particularly initially. The introduction of a relatively advanced feature should be viewed with caution, particularly if that feature forces a simpler construct to become more complicated in order to preserve consistency.

For example, if the introduction of file input and output requires the specification of streams for keyboard and screen I/O, then the first few simple programs students write have been complicated unnecessarily.

7.3.7 Maximise "knowledge-in-the-world"

As discussed in section 2.5.2, knowledge-in-the-world is information that can be extracted from the tangible and visible parts of a structure, and which therefore need not be remembered. Knowledge in the world is closely related to role expressiveness (see Chapter 5). In a programming language, knowledge-in-the-world can be defined as syntax that clearly indicates the nature of the construct it represents. Maximum knowledge-in-the-world means that the amount of details a programmer must remember can be minimised. In other words, each construct should be carefully crafted to contain as much information as possible in its keywords and general structure.

7.3.8 Use differing syntax to differentiate semantics.

Novices experience great difficulty in building clear and well-defined mental models of the various components of a programming language. Syntactic cues can be of significant assistance in differentiating the semantics of various constructs.

Constructs that may, to the accomplished programmer, seem naturally similar or analogous in concept, functionality, or implementation[2] still need to be clearly syntactically differentiated for the novice. It is unhelpful to novices to highlight the similarities of such constructs with similar (or worse, identical) syntaxes.

---

[2] for example: using the integer subset {0,1} as a substitute for a true boolean type; arrays being analogous to discrete functions of finite domain and range; arrays being implemented via pointers

Initially, such syntactic overloading blurs the crucial differences by which students can first discriminate between programming concepts; later it robs them of the opportunity to consolidate understanding by identifying these underlying conceptual connections themselves.

The Turing and Ada construct **A(B)**, discussed in section 6.1.3, is an example of semantically different constructs that are insufficiently distinguished syntactically.

7.3.9 Make the syntax readable and consistent.

Novice programmers, like all novices, have a weak grasp of the "signal" of a new concept and are particularly susceptible to noise. This suggests that an introductory programming language should aim to boost the conceptual signal and reduce the syntactic noise. One obvious means of improving the S/N ratio is to choose a signal with which the recipient is already familiar. For example:

- **if** rather than **cond**
- **head/tail** rather than **car/cdr**
- × rather than *

Another approach is to select signals that are consistent, distinct, and predictable. For example, delineating code blocks within constructs by **<name>...end<name>** pairs:

```
loop
     if isValid(name)
          exit loop
     end if
     output name
     name ← getNextName()
end loop
```

It can be difficult to steer an appropriate path between the syntactic extremities of "less-is-more" and "more-is-more". On one hand, reducing syntactic noise might involve minimizing the overall syntax, for example:

```
if  x < y                          if (x LT y) {
     y ← x                              y := x;
else                    rather than  } else {
     x ← y                              x := y;
                                   }
```

Alternatively, it may be better to *increase* the complexity of the syntax in order to reduce homonyms that blur the signal. For example, the meaning of the various components of the Turing expression[3]:

```
f(C(p).A(I))(N)
```

might be better conveyed with the syntax:

```
f(C::p->A_I)[N]
```

The second form, whilst regrettably no more mnemonic than the first, does at least provide adequate visual differentiation between pointer dereference, array indexing, function call, and substring extraction.

7.3.10 Provide a small number of powerful, non-overlapping features.

It can be argued that a large language that contains many features and multiple ways of performing each task is more likely to be able to provide the best possible way of solving any problem. This is useful for an expert who has already come to grips with the concepts underlying programming, and who is experienced in the use of different types of language constructs. For a beginner, though, it is better to be able to associate each part of a problem with a single clear solution.

In the same way, it is easier to remember each possible operation if it has a single clear construct associated with it. In the short term, while students are becoming familiar with the fundamental concepts of programming, the fewer constructs they must come to grips with the better. A smaller, simpler language will be easier to master than a larger more complex and powerful one. The needs of the brighter, more experienced students (who will run up against the limitations of the language sooner) must be balanced against the needs of those students with no computing background.

Homonyms and synonyms are an acute problem in the design of a teaching language. One way to contend with these pedagogical impediments is to select a small set of orthogonal language features and to assign them distinct (and mnemonic) syntactic representations.

---

[3] "Create a substring consisting of the `N`th letter of the string returned by the function `f` when passed the `I`th element of the array member `A` of the object within collection `C` that is pointed to by `p`".

A side effect of this approach is that, as the number of language constructs is restricted, those that *are* chosen must inevitably become more general and probably more powerful. In particular, it is important to provide basic data types at a high level of abstraction, with semantics that mirror as closely as possible the "real-world" concept those data types represent.

For example, it seems reasonable that an introductory language should not provide separate data types for a single character and a character string. Rather, there should be a single "variable length string" type, with individual characters being represented by strings containing a single letter.

Likewise, an introductory language need only provide a single numeric data type that stores rational numbers with arbitrary precision integer numerators and denominators. The restriction to rationals still allows the educator to discuss the general issue of representational limitations (such as the necessary approximation of common transfinite numbers such as π and e), but eliminates several large classes of common student error that are caused by misapplication of prior mathematical understanding to fixed precision arithmetic. A single, arbitrary precision numeric type has the additional benefit of eliminating many hardware-dependence problems.

Other features that might be provided include:

- A single non-terminating loop construct, possibly modelled on the Turing or Eiffel **`loop`** statement, with an associated **`exit loop`** command.
- A single generic list meta-type, allowing the user to define homogeneous or heterogeneous lists, indexed by any well-ordered type (integral numeric, boolean, string).
- A single, consistent model and syntax for I/O (see below).

7.3.11 Be especially careful with I/O.

With growing awareness of the importance of software usability, it is natural that students should be encouraged to engineer the input and output of their programs carefully. Too often, however, they are hampered by "more-is-more" programming language I/O mechanisms that are needlessly ornate or complicated.

The essence of I/O is very simple: send a suitable representation of a value to a device. The complexity frequently observed in the I/O mechanisms of

introductory languages often stems from a desire to provide too much control over the value conversion process.

Somewhat surprisingly, the C++ language, not otherwise known for its friendliness towards the novice[4], provides a reasonable if somewhat over-featured model of I/O. Turing also offers a very straightforward I/O model and syntax.

The I/O mechanism for an introductory language should be defined at the same high level of abstraction as the other language constructs. The basic features of a good pedagogical I/O model might be:

- **a simple character stream I/O abstraction, with specific streams (for screen, keyboard, and possibly files) represented by variables of special inbuilt types.** I/O is often one of the first constructs students use, so it is important that it should be simple and logical from the students' point of view. For example, the commands "**`print x to file`**", and "**`read x from file`**", while somewhat verbose, are role-expressive.
- **a single input syntax and a single output syntax for all data types** (for example, infix "input" and "output" operators that may be applied between a stream variable and a heterogeneous list of values and/or references). Novices are unlikely to recognise any conceptual difference between printing a number, and printing some text. A simple syntax which treats all data types alike will be easy for students to grasp, and to use.
- **a default idempotent[5] I/O format for all data types (including character strings and user defined types).** Idempotence ensures that there are no unexpected results when students use I/O. It maximises consistency and makes I/O constructs more predictable.
- **Choice of appropriate formatting defaults for justification, output field width, numerical precision[6], etc.** Specification of formatting details for

---

[4] or indeed towards the expert!

[5] Idempotency of I/O implies that the input and output format for any type are exactly equivalent. That is, if the output of one variable is used as the input to another, the effect is exactly equivalent to assignment between the two. Input/output of character strings is notoriously non-idempotent in most programming languages, because strings are typically written in their full length but read only to the first whitespace character.

[6] Note that idempotent I/O implies that output of rational numeric values should be at maximal precision by default, and may require special formats for recurring decimals.

output, in particular, complicates the syntax of an otherwise simple I/O statement. The Turing output statement, **put x**, is a good example of a very simple output statement that novices can use immediately, without explanation of any complex formatting syntax. Sensible default formats, such as putting a space between each output value, minimise the details students must struggle with in order to view the results of their programs.

- **A reasonable, automatically deduced output format for user-defined data types.** For example, output each globally accessible data member of a user-defined type, one value per line.
- **A simple and explicit syntax for specifying non-default output formatting.** For example, a generic **`leftjustify`** function to convert any value to a left-justified character string of specified field width.

However, when it comes to including these features in a language, it turns out that some apparently ideal features do not lend themselves to an obvious and intuitive syntax, or they clutter and complicate the simpler features more than their usefulness justifies. For example, including an automatically deduced output format for user defined data types, such as structures, necessitates a similar automatically deduced input format. This leads to assumptions about which member of a structure will be read in first. Although this can be verified by checking the definition of the structure, it may not correspond to the order of use used in user-defined functions, and could be a source of confusing and hard-to-find errors in novice programs. This is probably an example of excessive cleverness (see section 6.1.6), as well as a default, implicit behaviour that may prove confusing to a novice.

7.3.12 Provide better error diagnosis.

There is a widely cherished belief amongst educators that one of the ways students learn best is by making their own mistakes. What is often neglected is that this mode of learning is only effective when the student's otherwise random walk through the problem space can be guided by prompt, accurate, and comprehensible feedback on their errors.

Making and correcting an error is certainly a useful experience for expert and beginner alike, but the process of correction can be tortuous without meaningful guidance. Compiler error messages are often couched in unnecessarily terse and technical jargon that serves to confuse and distress the student. By the time the

messages have been deciphered and explained by a teacher or tutor, any useful feedback that might have been gained has been largely negated by the delay and stress involved.

The type of feedback that students receive when compiling their programs is typically along the lines of these real world examples:

```
Syntax error on line 4   (which may not be on line 4!)
Not implemented: & of =
No subexpression in procedure call
Application of non-procedure "proc"
```

Even should they manage to compile their program, run time errors typically produce unhelpful feedback like:

```
Segmentation violation: core dumped
Application "unknown" has exited with error code 1
<<function>>
```

Error diagnosis is a weak point of most compiler technology, yet it is the compiler feature that most novices spend most of their time interacting with. Although well-designed error feedback is not unknown (Turing is exemplary in this respect) many language implementations, particularly interpreters, have little or no error diagnosis. In these cases, errors are detected when the program executes in some unexpected way. Detecting and correcting errors in these implementations can be extremely difficult, particularly for a beginner, who may be uncertain what the expected behaviour of the program actually was.

For an introductory language, error messages should be issued in plain language, not technical jargon. They should reflect the syntactic or semantic error that was discovered, rather than the behaviour of the parser. Error diagnosis must either be highly reliable or, where this is infeasible, error messages must be suitably non-committal. For example, given the statement:

```
int F(UnknownType);
```

a widely-used C++ compiler emits the error message:

```
')' expected
```

rather than explaining that:

An undefined type 'UnknownType' was used in declaring the parameter list of function 'F'.

In this case even a vague message like:

There seems to be a problem in the declaration of the parameter list of function 'F'

would be of more use than the actual error message.

A clear, simple and accurate error reporting mechanism is clearly critical to the usability of an introductory programming language. Such a mechanism must mandate plain language error messages and should ideally provide multiple levels of detail in error messages (possibly through a "tell-me-more" option).

Common compilation errors (such as forgetting end-of-statement markers, or mismatching brackets) should be accurately diagnosed and clearly reported. Cases where the root cause of the error is not easily determined should be reported as problems of uncertain origin, with one or more possible causes offered in suitably non-committal language.

Run-time errors should likewise be clearly and accurately reported, at the highest possible level of abstraction. It may be sufficient for the expert to be informed that a segmentation fault has occurred, but the novice needs a hint as to whether the event was caused by an array bounds violation, an invalid pointer dereference, an allocation failure, or something else entirely.

7.3.13 Choose the appropriate level of abstraction.

When first introduced to programming, students often have trouble finding the correct level of abstraction for writing algorithms. Some expect a very high level of understanding from the computer, even to the extent of expecting variable names to affect the semantics of the program. For example, they may assume that naming a function **max** is sufficient to ensure that it computes the maximum value of its arguments. Others attempt to code everything, including basic control structures, from scratch. To require algorithms to be coded in languages with extreme levels of abstraction (for example: high-end logic, functional, or pure object-oriented languages, or low-level assembler) merely compounds the students' already abundant confusion.

It is critical for an introductory language to approximate closely the abstraction level of the problem domain in which beginners typically find themselves

working. Hence it is appropriate to provide language constructs suitable for dealing with basic numerical computing, data storage and retrieval, sorting and searching, etc. For most introductory courses, language features that support very low-level programming (for example: direct bit-manipulation of memory) or very high-level techniques (such as continuations) will merely serve to stretch the syntax and semantics of the language beyond the novice's grasp.

7.3.14 Use a sensible, unsurprising type system

Type systems are usually categorised as either strong or weak. Language definitions mandate strict or semi-strict type checking in the compiler (or, occasionally, no type checking at all). Rather than discuss typing in these technical, computing terms, it is appropriate here to examine the consequences of the type system for the novice. As with the selection of paradigm, the choice of type system should be based on making the system as unsurprising, consistent and usable as possible, rather than on purity of concept.

It is often argued that a strict type system is safer for novice programmers, as it is more likely to detect and correct errors in the compilation phase. This example from Haskell, though, demonstrates the danger of strict type checking on a strongly typed language.

As noted by Hudak & Fasel (1992), the type system can cause confusion for a novice, for example the following code, although it looks legal, will not compile:

```
main = print (average[1,2,3])
average xs = sum xs / length xs
```

Which, using the Haskell compiler hbc (hbc, 1998) produces the error message:

```
Errors: "average.hs", line 3, [56] Not an instance
Prelude.Fractional Int in (/) (((sum)::a2950)
A1_average) (length A1_average)in average
```

The problem is caused by the numeric type system, in which numerals are overloaded rather than implicitly coerced to the correct type. Because the / operator requires **Fractional** arguments and **length** returns an **Int**, an explicit cast must be used:

```
average   :: (Fractional a) => [a] -> a
average xs     = sum xs / fromIntegral (length cs)
```

Modula-3 imposes similar type strictness upon programmers (see section 5.2.4.2), for reasons of efficiency.

Like everything else in an introductory language, the type system should not spring these types of nasty surprises on novice programmers. Novices expect to be able to carry out the kinds of calculations they are accustomed to, in a manner as close as possible to the way they would normally do them. In a sensible, unsurprising type system, numbers can be manipulated in standard ways - integers can be added to real numbers without the need for explicit conversion routines, etc.

7.3.15 If it's not obvious, leave it out

Any feature that does not have an obvious and clear representation, particularly when the possible representations each have significant drawbacks, must be considered suspect. Unless there are compelling reasons for the inclusion of the feature, bearing in mind the introductory nature of the language and the short time it will be used, the feature should be excluded. As well as avoiding the inclusion of confusing features that do not add significantly to the functionality of the language (in its intended setting), this rule helps keep the language small and the syntax minimal.

7.4 Summary

This chapter has described the construction of a theory of good pedagogical programming language design. A language which adheres to this theory will not conflict with beginners' existing knowledge, and will support the learning of concepts fundamental to programming. It will, as far as possible, be consistent with students' expectations, and map closely onto their problem solving techniques.

Just as no battle plan survives contact with the enemy, no pedagogical language design (no matter how sound its design principles or clever their realization) can hope to survive contact with real students. Yet the outcomes of such encounters are the only meaningful measure of the success of an introductory language. This implies that the most important tool for pedagogical programming language design is usability testing, and that genuinely teachable programming languages must be evolved through prototyping rather than springing fully-formed from the mind of the language designer.

Chapter 8 describes the application of the design theory outlined here to the design of an introductory programming language. Subsequent chapters describe and analyse the evaluation and testing of the language.

8 Case study: Applying the design process: GRAIL

The definition of the GRAIL programming language proceeds from the hypothesis that it is the unfamiliarity of the "hieroglyphics" (i.e. the language syntax) and the sheer complexity of the full theory that are the primary stumbling blocks for the novice. Hence GRAIL is designed to provide a means for the novice programmer to explore the most fundamental concepts of programming, without the need to wrestle with the arcana of real-world syntax or the full power of complex semantics.

The GRAIL language is extremely small (its full grammar fits comfortably on three pages (see Appendix A)), and designed to present imperative programming concepts through a syntax that is consistent with the student's prior (mathematical) experience. The syntax also maps isomorphically to the small number of language constructs.

GRAIL is specifically designed as a short-term introductory language. Typical problems tackled by students using GRAIL include real world tasks such as calculation of annual tax payable based on weekly salary rates, building databases of music or student marks, and solving simple mathematical problems such as the computation of Fibonacci numbers and factorials.

The language described in this chapter is exactly as it was tested by students during the evaluation process described in Chapter 9. Changes made subsequent to the evaluation are described in Chapter 10. The next section discusses the design principles that have supported the above goals.

8.1 Design of GRAIL

The main aim of GRAIL is to facilitate the teaching and learning of programming. Existing introductory languages (including real world languages not designed for teaching) often impede the learning process by forcing novice programmers to spend their time focusing on obscure details of syntax and unnecessarily complex semantics. On the premise that students' time could more profitably be spent concentrating on algorithms and problem-solving skills, GRAIL was designed to be intuitive and unsurprising to a novice programmer.

Unlike many languages used for introductory programming courses (McIver & Conway, 1996), GRAIL's principle design rationale is strictly pedagogical in

nature. Every decision made during the design of the language was made on pedagogical grounds, following the principles of pedagogical language design outlined in Chapter 7. Implementation issues were not considered, nor were political or historical considerations taken into account. Inevitably GRAIL bears some resemblance to existing languages (in part because some programming languages have features which meet the aims of GRAIL very well), and the biases and programming experience of the designer have inevitably had some influence on the language. However a process of testing and revision has been undertaken, in order to minimise the deleterious effects of such influences. No feature was included in GRAIL simply because it exists in some other language or programming paradigm.

Many programming languages were considered during the design process (see Chapters 5, 6, and 7), and these languages covered many paradigms - some of them were intentionally paradigmatically pure, while others were powerful hybrid languages that include features from a number of different paradigms. While the procedural paradigm was selected for GRAIL as being closest to the style of instruction that students were most likely to be familiar with (from recipes, instruction manuals etc), there was no attempt to adhere strictly to a particular paradigm, and features from many different paradigms were considered for inclusion in the language.

It is interesting, then, that the final language design is purely procedural, despite the decision having been made that paradigmatic purity was not a design criterion. Many of the cleverest and most interesting features that were considered for adoption from other languages and paradigms were rejected on the grounds that they complicated the language unnecessarily. These included pattern matching, sophisticated list constructs with indices of arbitrary type, classes, and functions as first class objects. In order to retain truly clean and consistent syntax and semantics, it was necessary to restrict the number of features in the language to only those that were deemed absolutely essential.

It is true that this approach limits the power of GRAIL considerably - it is not difficult to come up with functions that cannot easily be implemented in GRAIL, and typical first programming exercises that cannot be set. For example, the lack

of pointers and references makes a swap procedure[1] difficult to write (although a version *can* be written using structures).

At the same time, however, it is also no challenge to produce a wide range of programming tasks that are easily implemented in GRAIL, and that allow students to create substantial programs whilst avoiding the complexities and confusion of more powerful languages. For example, students can write programs to keep track of their music collections: adding, deleting, sorting, and searching for the titles and artists of their CDs. They can build databases that keep track of the marks of all students in a class, and search to find the student with the best marks, or all the students who are failing the subject.

Bright students will indeed encounter the limits of the language. These students can be redirected into other tasks that are not limited by the language, and they can be given the task of designing ways around the limits. It is not necessary (in fact it may be counter-productive) for the language to be easy and well-suited to the bright students, but it is critical that it be well suited to the struggling students for whom it is intended. If further testing revealed that advanced students quickly become frustrated with the limitations of the language, libraries of graphics functions could be added to stimulate and retain their interest, or the course could be streamed so that advanced students finish with GRAIL sooner than those who are struggling to come to grips with computer programming.

It is intended that GRAIL would be taught for a single semester at most, and that, having laid a solid foundation of understanding of the fundamental programming concepts, the course would subsequently move to a more complex, powerful, and "difficult" language. GRAIL is intended as a means for students to quickly acquire programming concepts, without getting bogged down in complex semantics or ornate and unfamiliar syntax. In other words, GRAIL is designed for *teaching* programming, rather than for *doing* programming.

8.2 General Design Principles

The theory of pedagogical programming design outlined in Chapter 7 can be distilled into three guiding principles: syntactic predictability, memetic compatibility, and minimalism.

---

[1] that swaps the values of two variables – the traditional example is `swap (a,b) {tmp=a; a=b; b=tmp;}`, where `a` and `b` are passed by reference

These three principles and their application to the design of GRAIL are discussed in this section.

8.2.1 Syntactic predictability

The need for "consistency" is a rare point of consensus in the literature on introductory programming language design (Brilliant & Wiseman, 1996; McIver & Conway, 1996; Budd & Pandey, 1995; Kölling et al, 1995; Levy, 1995; Conway, 1993; Murnane, 1993; Meertens, 1981). Consistency is, however, a slippery concept, particularly in the context of programming language syntax. As discussed in Chapter 6, certain kinds of apparent consistency (such as syntactic synonyms and homonyms (McIver & Conway, 1996), or declarations that mimic their subsequent usage (Ritchie, 1996)) actually make learning harder for the novice, by muddying semantic and conceptual distinctions between language components.

Such self-defeating forms of "consistency" may be avoided by, instead, striving for "predictability". For example, in GRAIL:

- Every construct that forms an aggregation (such as a structure declaration, the body of a subroutine, or a loop block) is bracketed in a ***keyword*...end *keyword*** pair. Hence it is immediately possible to determine which aggregation a "closing bracket" refers to. Contrast this with the use of "anonymous" **begin...end** blocking in Pascal (Feuer & Gehani, 1984).
- In all types of declarations, the keyword **is** denotes the declaration of a type equivalence (i.e. between two types), whilst the keyword **holds** denotes the declaration of a type relationship (i.e. between a type and an instance). This contrasts with the use of : for both these purposes in Eiffel (Meyer, 1992).
- The input and output syntaxes are identical for all types. This contrasts with the specificities of **scanf()** and **printf()** in C (Feuer & Gehani, 1984).
- GRAIL's syntax and semantics are isomorphic. That is, every syntactic component of GRAIL has a single meaning and every GRAIL construct has a single syntax. This contrasts with the *five* distinct, context-sensitive meanings of **A(B)** in Turing (Holt & Hume, 1984), the numerous reference-generating mechanisms in Perl (Wall et al, 1996), or the multiple ways of accessing arrays in C (see section 6.1.3) (Ritchie, 1996).

8.2.2 Memetic compatibility

As noted in section 2.4.5, "Cognitive dissonance" (Corsini, 1994) is the discomfort engendered by information that is incompatible with existing knowledge. Apart from the obvious negative reinforcement it produces in the student, cognitive dissonance also sets up a "belief bias" (a tendency to discount the significance of dissonant information (Evans et al, 1983)) that impairs learning.

The design of GRAIL addresses these impediments to the acquisition of programming skills by grounding the syntax of the language in terms and symbols likely to be familiar to the average novice programmer, and by deliberately avoiding widely-used programming "memes" (McIver & Conway, 1996) that have meanings inconsistent with everyday usage.

Thus GRAIL uses + and × for multiplicative operations, since they are likely to be more familiar and "intuitive" to the novice than the computing standards / and *. Conversely, GRAIL does not use **`while`** as a keyword for repetitions, since mapping the usual **`while`** ***`condition`*** **`do`** ***`statement(s)`*** syntax back to English often results in the mistaken assumption that the condition is tested at *every* point in the loop (see section 6.1.7).

8.2.3 Minimalism

GRAIL has been designed to be used in a short, preliminary "programming concepts" course. For this reason, the overriding design principle has been Occam's Razor (Thorburn, 1915). When a useful but non-essential construct was overly complex, or did not have a single obvious syntax, as well as intuitive and exception-free semantics, it has been omitted from the language.

These omissions include:

- direct access to individual characters of a string,
- a distinct syntax for extracting characters
- string concatenation
- dynamic arrays or lists
- structured loops
- case statements
- pointers and references

- reference parameters in subroutines
- explicit initialization
- object-orientation
- first class functions or closures
- nested functions

Specific rationales for each of these decisions are presented in the next section.

Although this policy has led to the exclusion of many constructs that programmers will eventually encounter, the resulting simplification makes the remaining (and more fundamental) concepts of programming, which GRAIL *does* convey, that much easier for the beginner to grasp.

8.3 Significant Features

8.3.1 Imperative

GRAIL is an imperative language. This paradigm was selected, after much deliberation, because it represents a style of algorithmic expression that students are already familiar with (from previous interactions with instruction books, recipes, navigational directions, shampoo bottles, etc.), and because it is the paradigm most widely favoured in actual teaching. The Reid Report (Levy, 1995) shows that 48.7% of surveyed institutions use an imperative language in their introductory course, 36.2% use object-oriented or object-based languages, 12% use the functional language Scheme, another 2.4% use functional languages such as ML etc, and 0.7% use other languages. The Reid report is somewhat inaccurate, because it relies on institutions to update their entries in the report, rather than actively polling for current languages in use. Nonetheless, it remains the widest survey of introductory language use in tertiary teaching institutions.

It is widely argued (Kölling & Rosenberg, 1996; Kölling et al, 1995; Meyer 1992; Booch, 1991) that, given the increasing popularity of object-oriented languages in industry, an object-oriented introductory language eliminates the need for a later paradigm shift.

However, it might also be argued that, rather than removing the paradigm shift, this approach merely moves it to the beginning of the students' computing education, when they are *least* equipped to deal with it.

Moreover, nothing is lost by teaching GRAIL ahead of an object-oriented language, since (with the possible exception of non-method subroutine calls) all the concepts GRAIL teaches – type vs instance vs value, aggregation, repetition and selection control structures, I/O, scoping, etc. – are also used in object-oriented languages.

8.3.2 No pointers or references

Pointers and references are notoriously difficult for students to grasp. While they are extremely powerful programming constructs, they are also hazardous and error-prone (M'Iver & Conway 1996). Novice programmers are ill-prepared to make use of the power, but fall easy prey to the pitfalls. In a language that possesses sufficiently powerful data structures, there is little reason to use pointers and references early in the course.

For a teaching language that is not intended for long-term use, "sufficiently powerful" need not imply significant complexity. A language with numbers, structures, strings and arrays is powerful enough for the implementation of a range of relatively complex algorithms, without introducing the confusion and difficulty of pointers and/or references. Even dynamically-linked lists can be implemented by using array indices instead of pointers[2].

Other options were considered for GRAIL in order to retain the power and flexibility of passing by reference, and still avoid the difficulties of pointers.

One option considered was the Ada parameter passing syntax, which allows the creation of **`in`**, **`out`** and **`in out`** parameters (see section 5.2.8.2).

This is a useful idea that reflects the problem-space more than the underlying machine code, in contrast to the traditional pass by value or reference models. The Ada model encourages programmers to think about the use of parameters in advance, and to specify the precise way each parameter will be used in the subroutine.

The addition of this syntax, however, can clutter a function definition:

```
function factorial (n: in Integer) return Integer;
```

---

[2] Using an array of structures, where the structure has a number field which holds the index of the next item in the list.

and is not intuitive to read. Spoken out loud, "function factorial takes n in integer" may not be entirely clear.

Most importantly, the addition of extra calling modes in addition to call-by-value requires students to come to grips with the difference between the different modes. Although the **in/out** syntax is clearer than some of the more obscure specifiers in use (such as * &), it is still an added cognitive burden for relatively small gain. Allowing call-by-reference does broaden the range of subroutines that can be written in the language (making traditional examples like the swap function possible), but the range of subroutines that can be written without this feature is sufficiently large to sustain a moderately complex introductory course.

8.3.3 Non-ASCII characters

The non-ASCII Unicode characters

- ← (\u2190) (assignment)
- × (\u00d7)
- ÷ (\u00f7)
- ≤ (\u2264)
- ≥ (\u2265)
- ≠ (\u2260)

are used as operators in GRAIL , rather than the more familiar (to experienced programmers) symbols = or :=, /, *, >=, <=, and != or <>. This decision was made in order to remain as consistent as possible with students' existing knowledge. While much of students' educational background cannot be assumed, it is reasonable to rely on a working knowledge of basic mathematical notation.

Given a user-friendly coding environment, there is little reason for adhering exclusively to the ASCII character set. While the symbols for multiplication, division, etc. are a relatively trivial part of the language, learning new symbols adds further complexity and confusion to an already difficult task.

The reasons for choosing ÷, ×, ≤, ≥, and ≠ are self-evident, but the choice of ← (rather than some "=" variant) is discussed in section 7.3.1.

8.3.4 A single numeric type

In most languages, the distinction between reals and integers is usually made for reasons of efficiency, and the specific details (such as the size of an integer) are dependent on the underlying hardware. In languages such as C, the difference between an **int** and a **long int** can be critical. (see section 6.1.4)

Because novices rarely differentiate between natural and rational numbers, neither does GRAIL. Instead, the language provides a single numeric type (**number**), thereby avoiding the extra conceptual level of hardware-based distinctions between floating-point and integer (McIver & Conway, 1996). The **number** type provides arbitrary precision rational numbers, which avoids confusing overflows and other mysterious fixed-precision side-effects.

8.3.5 Static arrays

Arrays in GRAIL are indexed with integers starting from **1** and may contain any single type, including user-defined types. The size of an array is fixed, having been specified in the variable's declaration. However arrays used as subroutine parameters may be declared without a fixed size, thereby allowing a single subroutine to accept arrays of any size.

At first glance, requiring bounds specifications in variable declarations, but not in parameter declarations may seem inconsistent. However, when designing a subroutine it is logical to specify the type of the array without the bounds, as in "an array of numbers" or "an array of texts", as the bounds are rarely relevant to the type signature of the subroutine. In other words, whether a subroutine can usefully be applied to a specific array is rarely dependent on the bounds of that array, rather than on the contents[3]. All array accesses are bounds-checked at runtime.

Dynamic arrays were considered for inclusion in GRAIL, but were rejected because they introduce a surprising level of semantic complexity. Consider the statement:

```
array1[i] ← array2[j]
```

---

[3] Except in extreme examples, such as where an array is of zero length (and hence has bounds 0 and 0).

It is unclear what the appropriate behaviour of this statement is under the following conditions:

- `i` equals `length(array1)+1` Assigning a value to a position in the array that is out of bounds by 1 position has unclear semantics if arrays are dynamically extensible – should it assign the value to a newly created position, effectively extending the array by 1 element? Should it produce an error?

- `i` equals `length(array1)+2` Assigning a value to a position that is out of bounds by more than 1 position is even less clear – if a new element is created dynamically, are the intervening elements also created, and if so with what values?

- `j` equals `length(array2)+1` Reading the value of an array position that does not yet exist logically produces an error – but what if writing to that position of an array automatically extends the array? Does consistency demand that reading and writing have matching semantics for matching situations? If so, an explicit "**`undefined`**" value could be returned, but it is still unclear whether the position should be instantiated as a result of the read, particularly if the index is out of bounds by more than one position.

Some of these dilemmas are addressed by making arrays only explicitly extensible, in other words requiring an explicit statement extending the array before any new positions can be accessed. The use of non-contiguous arrays (that is, arrays that may have uninitialized gaps) also solves some of these problems but only by greatly increasing the complexity of the array concept. This "cognitive cost" may outweigh the potential benefits of dynamic arrays. Overall, the benefits of dynamically extensible arrays did not justify the cost.

Also considered were user-defined arrays with indices of any strictly-ordered type. For example, since character strings may be strictly ordered, character strings were considered as legal indices for an array. This allows a range of data structures such as a database of telephone numbers, indexed on surname:

```
phonebook["mciver"] ← 99055210
phonebook["jones"] ← 99055555
```

While this is a powerful and useful addition to the language, it has several attendant difficulties:

- Appropriate declaration is not intuitively obvious (and difficult to make concise and readable):

```
item phonebook is array of 10 number indexed by text
item phonebook is text indexed array of 10 number
item phonebook is array of 10 number with text index
item phonebook is array of 10 number[text]
```

- Unlike an array indexed numerically (and contiguously), specifying an array of 10 items with strings for indices does not specify the indices. Assigning something to an array position therefore means initialising the index as well as the value of the array at that position.
- There is no obvious way of cycling through an array indexed with strings, unless it is also mapped by default onto integers, in which case position **1** may change depending on what items are created. This problem could be solved using an "**indices**" function which returns a new array containing the indices of the first array, this time indexed with integers, but this complicates the construct somewhat.
- What should be the appropriate behaviour when a new item is created and the array is already full? Does the first item get overwritten? Does the last item get overwritten? Is it a runtime error?

Once again the ambiguities and problems associated with indexing arrays with strings outweigh the potential benefits.

8.3.6 Line-based strings

Real-world manipulation of character-strings requires powerful and complex mechanisms, that are inappropriate for a teaching language. GRAIL provides a simple string type (**text**), that is defined as a line of characters. There is no separate character type; a single character is a text of length 1.

The choice of the line as a unit of text is unusual, but fits well with the common novice uses of strings – as prompts, labels, output decoration, and simple data. More importantly, using an entire line as an atomic string makes it possible for GRAIL to provide fully idempotent I/O (see below).

More unusual still is the fact that GRAIL **text** values are truly atomic. That is, it is not possible to access an individual character in a **text** directly. This decision was taken because it reduces the syntactic complexity of the language, by eliminating syntactic puns (McIver & Conway, 1996) such as:

```
var[10] ← "a"
```

where the semantics of this statement depends entirely on the (possibly distant) declaration of **var** (as either a **text** or an array of **text**).

Many such syntactic puns are particularly relevant to dynamically resizable strings. A string type that cannot be dynamically resized is cumbersome in situations where the same string variable is used for different length strings at different points in a program. The simplest technique for handling this situation – making the string large enough to deal with all possible strings, engenders poor programming habits, as this is a most inefficient use of space, particularly in larger programs.

Removing direct access to characters also eliminates the need to define semantics for assignment to invalid indices or assignment of multi-character strings to single elements, for example:

```
var ← "123456789"
```

Assuming **var** has been declared as a text variable, the above code makes **var** a 9 character string.

```
var[7] ← "ABC"
```

There are several possible interpretations of this statement. The code could:

- place the start of the new string **"ABC"** at position **7**, overwriting positions **7**, **8**, and **9**, and thus create the string **"123456ABC"**
- replace the current value of **var[7]** (**7**) with the whole three character string, and thus create the string **"123456ABC89"**
- simply produce an error.

Similarly, the following code has several associated problems:

```
var[10] ← "A"
```

It might:

- produce the string **"123456789A"**
- be an error to assign to an uninitialized position in the string, in which case there is the problem of determining valid code for extending an existing string, or initializing a new one
- initialize the string correctly, in which case, since assigning a literal string to a text variable is valid initialization in the first instance, it is unclear whether

it should be valid subsequently, or if it is only valid to assign to a complete text variable, rather than to a specified position in the text.

The following statement is also problematic:

```
var[100] ← "A"
```

- If it is legal to assign to an uninitialized position in a string, thus initializing it, then it must be determined whether it is only legal for the next available uninitialized position, or if a string can be sparse and have characters only at selected positions. Only assigning to the next uninitialized position requires the programmer to know the length of the string in advance, which is likely to lead to error-prone code when novices use hard-coded numbers instead of ascertaining the length of the string each time.
- If it is not legal to have a sparse string, the code may initialize the intervening positions in the string with some default character, such as a space, or it could produce an error.

Substring extraction possesses similar ambiguities. If it is valid to refer to a substring, perhaps using syntax such as `myString[4...6]`, then a number of ambiguities arise. Assuming `myString` begins with the value `"ralph"`, in the case of `myString[1..3]←"fred"`, where the substring on the right is longer than the substring specified on the left, the appropriate behaviour in this case could be to:

- produce a runtime error
- insert "`fred`" in place of characters 1 to 3, and move the subsequent characters along to make space for the fourth character, to produce "`fredph`"
- insert "`fred`" at the start of the string to produce "`fredralph`"
- overwrite characters 1 to 4, producing "`fredh`"

Similar problems arise if the substring on the right is shorter than the substring on the left, as in the case of `myString[1..3]←"A"`, which could mean:

- produce a runtime error
- insert "`A`" in place of characters 1 to 3, producing "`Aph`"
- insert "`A`" in position 1, but leave the rest of the string the same, producing "`Aalph`"

These problems can be avoided by making it illegal to assign to a substring, but since in all other ways a substring is identical to a string – it can be printed out, passed to a subroutine as a parameter, etc – this would be inconsistent, and potentially confusing.

As strings are dynamically resizable, treating them as special types of arrays also leads to the logical assumption that arrays are also dynamically resizable. In turn, this leads to all the problems described above - whether arrays can be extended simply by assigning a value to the next sequential element, even if it doesn't yet exist, etc. In addition, the declaration of an array includes its size, whereas the declaration of a string does not. If strings are arrays, this is inconsistent. If strings are not arrays, but are instead an atomic type – a line of text – then they need not be treated the same way as arrays in order to maintain consistency.

String concatenation creates similar problems. String concatenation done using a function call is somewhat ambiguous: `concatenate(string1,string2)` could mean:

- concatenate `string1` and `string2` and store the result in `string1` or
- concatenate `string1` and `string2` and return the result without changing the values of `string1` or `string2`

Although this is arguably less ambiguous than the indexing problems discussed above, there were insufficient strong reasons for including string concatenation to outweigh the ambiguity. The rule in section 7.3.15 was invoked: If it's not obvious, leave it out. Since there is not an unambiguous representation for string concatenation, and its presence in the language would not offer substantial gains, it was not included. Should it be subsequently proven that string concatenation is necessary to the language, a function could be added without disturbing the existing syntax.

8.3.7 Idempotent I/O

As suggested in (McIver & Conway, 1996), the input and output operators in GRAIL are idempotent in effect. That is, an input followed by an output is "value-preserving" in all cases.

Most languages provide non-idempotent I/O, which frequently leads to significant confusion for novices. For example, in most languages a character string is read in to a variable until the first whitespace character is encountered.

However, when the same variable is written out the entire string it contains (including any embedded whitespace) is printed.

GRAIL's use of line-based strings (**text**) and arbitrary-precision rationals (**number**) makes idempotent I/O of its basic data types automatic, and hence unsurprising.

8.3.8 Associative comparisons

GRAIL provides associative comparisons, so that the GRAIL expression **1<x<10** is valid and behaves as students naturally expect, resulting in a boolean value (**true** if **x** is between **1** and **10**, and **false** otherwise).

8.4 Language Overview

8.4.1 Comments

Comments in GRAIL are prefixed by "|" and run to the end of the line. For example:

```
write factorial(x)   | call factorial and print result
```

The "|" character, whilst not particularly mnemonic, at least has no other meaning which might confuse the students. The vertical line has the advantage of symbolically dividing the code from the comments.

8.4.2 Types

GRAIL provides three fundamental data types: **number** (arbitrary-precision rationals), **text** (atomic line of characters), and **boolean**.

Type aliases may be created:

```
type scoretype is number
type a_name is text
```

Array types, implementing collections of elements of simpler type, may be defined:

```
type marklist is array of 10 number
type daily is array of 52 × 7 marklist
```

Automatic pluralisation[4] was rejected for being too complex. Students must learn what can and can't be expected of a computer programming language, and automatic pluralisation of types, especially if it were to take into account user-defined types, requires fairly sophisticated processing, although it can be done for English words (Conway, 1998).

However, consider the case of a student for whom English is not the first language, who wishes to create a type named in his or her native tongue. If GRAIL handles automatic pluralisation for types named in English, students may reasonably expect that it also handles automatic pluralisation in other languages - it may seem to be all part of the "magic" by which GRAIL "knows" that an array to hold more than one ox actually holds oxen, or that an array to hold more than one student holds students, but an array to hold more than one fish holds fish.

In addition, English is sufficiently complex that the algorithm described in Conway (1998) is not perfect. For example, an earlier version of the program incorrectly inflected human to humen, because the algorithm specified that any word ending in "man", such as "man", "workman", or "caveman" should, in the plural form, end in "men". Although this particular example has been corrected, more unusual cases come to light from time to time. The cases where the algorithm behaves unexpectedly could have disastrous consequences for a novice trying to develop a model of the computer's behaviour.

Structured types, which aggregate heterogeneously typed data, may also be defined:

```
type student_data has
     field name holds a_name
     field mark holds scoretype
     field ID   holds text
end type
```

The first version of structured types in GRAIL omitted the keyword **field**, to look like this:

---

[4] such that first declarations would become: `type marklist is array of 10 numbers`

```
type student_data has
     name holds a_name
     mark holds scoretype
     ID   holds text
end type
```

The extra keyword was added to be consistent with the **constant**, **item**, and **type** declarations.

Also considered was the addition of another keyword to specify the structure definition, such as **structure**, so that the structure definition would look like this:

```
structure student_data has
     ...
end structure
```

This was rejected in favour of making the declaration of these user-defined types explicitly type declarations. This makes the contrast between **item** declarations and structure declarations clearer.

Type inference, as found in languages such as Haskell, was considered for GRAIL. When considering this feature it was necessary to examine the cognitive aspects of type specification. Type inference obviates the need for explicit declaration statements, and makes polymorphic functions simple and elegant to implement, since a function can be defined with no explicit type. The type is then inferred at runtime from the function invocation, based on the combination of the actual parameters and the formal function specification. For example, the simple Haskell function:

```
first(a,b,c) = a
```

can be invoked on any tuple containing three items. There is no restriction on the types of the three items – they need not even all be the same type[5].

---

[5] The function can be rewritten for homogeneous lists rather than heterogeneous tuples using square brackets: `first[a,b,c]=a` Interestingly the type of the list version of the function is inferred as `[a]->a`, meaning the function takes a list of objects and returns a single object of the same type. The inferred type implies that the function can take a list of any length, but in fact it can only take a list of length 3, as specified in the function definition.

The following invocations are all valid and work as expected:

```
first("fred","ralph","arnie")    -- prints "fred"
first("fred",2,3)                -- prints "fred"
first(1,2,3)                     -- prints 1
```

However, there is some value in forcing students to consider (and declare) the types of their variables explicitly. This often helps in tracking down errors, since if the type declaration mismatches with the use of the variable, it is a compile time error and easily detected and corrected. Haskell allows variables to change type during the life of a program (so that `let a="ralph"; let a=1;` is quite valid code). This allows what Bonar and Soloway (1985) term "multirole variables". These were found to be "bug generators"[6] in novice code, because the multiple uses of a variable sometimes clash (for example where a variable is simultaneously used as a loop counter and a place to store a sum being calculated in the loop).

Type inference could be used where the type of a variable is fixed once it is first determined, which is the way ABC handles types, but again the advantage of explicitly declaring the type, and forcing students to state their intentions explicitly, is lost.

8.4.3 Values

Literal values of type **number** are decimal values of arbitrary length. Note that there is no exponential notation available.

Literal values of type **text** consist of any sequence of characters (except '"') on a single line, enclosed in matching quotation marks.

For example:

- `"enter new value"`,
- `"the data you entered was not valid"`,

but not:

- `"a line containing the " character"`, or
- `"text which, within a single set of quotes, spans two or more lines"`.

---

[6] A source of errors. See Chapter 4 for more detail.

Literal values of type **boolean** are **true** and **false**.

Neither arrays nor structures have an associated mechanism for specifying compound literals. In the case of structures, this is because specifying values for individual fields of a structure without naming the fields is error-prone, as it requires knowledge of the order of declaration of the fields (something which is not required anywhere else in a GRAIL program).

Specification of compound literals for single dimensional arrays is straightforward, using syntax along the following lines:

```
myArray ← {1,2,3,4,5,6,7,8,9,10}
```

Multi-dimensional arrays are more complex, however. Among the possibilities for the specification of a 2 by 4 array are:

```
myArray ← { {1,2} {3,4} {5,6} {7,8} }

myArray ← { {1,2,3,4} {5,6,7,8} }
```

This form of array initialisation is of limited use, as it quickly becomes unwieldy with larger array sizes. In addition, the mechanism for iterating over an array is a technique students will need to learn, and array initialisation is a simple way to introduce it.

8.4.4 Variables

All variables in GRAIL are specified with the **item** keyword:

```
item name holds text
item nextmark holds student_data
item marks holds array of 100 student_data
```

Many options were considered for variable declarations, among them

```
item x is number
item x stores number
object x holds number
obj x holds number
box x holds number
```

Ultimately, **item x holds number** was chosen for several reasons. **item** was chosen over **obj** and **object** because **obj** and **object** have a distinct meaning in the object-oriented paradigm, and may cause some confusion if students

subsequently attempt the transition to an object-oriented language. **box**, although it is a useful metaphor for a variable, was rejected because of the confusion that can be caused when the metaphor breaks down (Van Someren, 1990). For example, the translation of **a=b;** as "Take the value in box b and put it into box a" implies that box b is now empty. A more accurate translation: "take a copy of the value in box b and put that into box a" makes the metaphor much clumsier and less clear.

holds was chosen because **is** implies equality rather than a container (and as such **is** is more aptly used for type equivalence: **type salary is number**), and **holds** is shorter than the other equivalent considered, **stores**, and hence somewhat more useful in maintaining visual structure in the code.

Mandatory variable initialization at declaration was considered and rejected, on the grounds that there is frequently no useful value to put in a variable at declaration time, especially if the variable is to be read in from the keyboard or a file. Variables are automatically initialized to the "null" value for their type (**0** for **number**, **""** for **text**, **false** for **boolean**). There is no explicit initialization syntax.

Access to the elements of an array is provided by the usual **[]** indexing notation:

```
bestmark[1] ← marks[best]
```

Unicode subscript characters were considered for array indexing, in order to be consistent with the subscripting notation students are familiar with from mathematics:

$$\mathbf{bestmark}_{1} \leftarrow \mathbf{marks}_{best}$$

However, while Unicode subscript characters exist for digits, representing variable names as subscripts is more difficult. In addition, subscripts can be somewhat hard to read in the case of 2 dimensional arrays, especially if the comma is not subscripted (there is no Unicode subscript comma character):

$$\mathbf{student}_{1,3} \leftarrow \mathbf{marks}_{1,3}$$

Implementation difficulties in this case outweighed the advantage of the more familiar syntax. This is the only case where that holds true, and where the syntax chosen conflicts somewhat with students' prior experience.

Access to the fields of a structure is provided by the **'** or **'s** operator:

```
item student holds student_data
```

```
student's name ← "Lee"
student's ID ← "L123456"
student's score ← 99
best ← student's name
```

"**'s**" was chosen for its English connotation of possession – students are accustomed to seeing "Linda's book" or "Sarah's piano", and knowing that the apostrophe and the **s** indicate belonging. Since a structure possesses its fields, this syntax is role-expressive and consistent with what students already know.

The "**s**" was made optional in order to allow typical English usage where a noun already ends in s. Automatic pluralisation was rejected for the reasons outlined in section 8.4.1. [7]

8.4.5 Operators

GRAIL has 17 operators at 10 levels of precedence, as shown in Table 8-1.

Precedence	Operator	Operand type	Result
highest	( )	any	Same as operand
	unary +, unary -	**number**	**number**
	× +	**number**	**number**
	+ -	**number**	**number**
	< ≤ ≥	**number** or **text**	**boolean**
	= ≠	any	**boolean**
	unary **not**	**boolean**	**boolean**
	and	**boolean**	**boolean**
	or	**boolean**	**boolean**
lowest	←	any	-

Table 8-1: GRAIL Operator Summary

[7] Although the "**'s**" was chosen independently, it has been a part of various scripting languages, including Applescript, for many years.

8.4.6 Constants

GRAIL supports the definition of constants:

```
constant pi is 3.15159
```

Since the type of a constant is immediately obvious when it is defined, it is not necessary for the programmer to specify the type in this case. The keyword **is** was chosen over **holds** (which is used for variable declaration) in an attempt to highlight the difference between a constant and a variable.

8.4.7 Assignment

All assignments in GRAIL use the ← operator:

```
name ← "Sam"
height ← 1.754
mark ← nextmark
```

Note that aggregate types (arrays and structures) can be directly assigned, provided both variables are of the same type.

8.4.8 Control structures

GRAIL provides a single unconditional iteration control structure:

```
loop
     count ← count - 1
     if count ≤ 0 then
          exit loop
     end if
     factorial ← factorial × count
end loop
```

Other iterative structures were considered, including a **foreach** statement, **while** loops, and **for** loops. All were ultimately excluded, on the basis that a single, simple loop structure would be easy for students to remember, and that it is important to provide a single, clear mechanism for each concept (see section 7.3.10).

The loop structure described above, with an **exit** statement which may be positioned anywhere within the loop, may map most closely onto the way students think, because it allows loops of three kinds:

- "while something is true, do stuff",
- "do stuff until something is true", and
- "do stuff, if something is true then stop, otherwise do more stuff".

All of the alternative loop constructs restrict students to a single loop model. It should be noted, however, that the syntax of loops caused some problems when the language was evaluated. This is further discussed in Chapters 9 and 10.

GRAIL also offers a single selection mechanism:

```
if 80 ≤ mark ≤ 100 then
     grade ← "distinction"
else if 0 ≤ mark < 50 then
    grade ← "fail"        | conditional alternative
otherwise
     grade ← "pass"       | unconditional alternative
end if
```

A **case** or **switch** statement was also considered for inclusion, but, like other forms of iteration, did not provide sufficient advantages to justify extending and complicating the syntax. Such decisions must always be viewed in the context of GRAIL's intended purpose as a short-term, throw-away, introductory language with a deliberately limited life span.

Case statements are useful for long lists of alternatives, and are a convenient way to introduce pattern matching to the language, but they generally have moderately complex semantics. Inclusion of a switch or case statement would contravene rule 7.3.10 - *Provide a small number of powerful, non-overlapping features,* since the case overlaps with an **if** statement.

8.4.9 Subroutines

GRAIL subroutines may be defined before or after the main program section (and hence before or after any call to them). The more common approach requires subroutines used before their declaration to have a prototype preceding their first use. This is to allow one-pass compilation, so that the compiler always knows the

type of subroutines before they are used. There is no need for GRAIL to be compiled using a one-pass compiler, since the resulting gain in efficiency is not necessary for an introductory language.

Subroutines do not have access to any information outside their own scope apart from that which is passed via parameters (i.e. there are no global variables or lexical closures). Recursive subroutines are supported.

Subroutines may take zero or more parameters of any type(s). Each parameter is specified using the standard **item**...**holds**... variable definition syntax. An empty parameter list must still be specified (as empty brackets) both in the declaration and all calls, in order to make it explicit that a function is being called. All parameters are passed by copy, since the language does not contain pointers or references.

Functions may return single values of any type (including structures and arrays). There is only one exit point for any subroutine: before its **end function** marker. Although there are some situations where having multiple function exit points would be convenient, it does make functions somewhat more complex. It can be difficult to explain to a student that a function is terminated by its **return** statement, when it seems to be terminated by the **end function** statement.

A typical GRAIL function looks like this:

```
function factorial (item n holds number) returns number
     item result holds number
     if n≤1 then
          result ← 1
     otherwise
          result ← n × factorial(n-1)
     end if
     return result
end function
```

The return statement is defined as the last statement in the function before the **end function** statement. This was intended to remove the confusion which can be caused by multiple return points from a function, however it caused some confusion for students during evaluation, and has subsequently been redesigned (see Chapter 10).

A function with no parameters and no **return** statement looks like this:

```
function haiku()
    write "late frost burns the bloom"
    write "would a fool not let the belt"
    write "restrain the body?"
end function
```

Formal division of subroutines into functions and procedures (as in Pascal) was considered, but ultimately discarded as complicating the syntax unnecessarily. Students rarely seem to make the distinction themselves, unless forced to do so by the language they are using. It was subsequently realised, however, that the use of the keyword **function** in this context is inconsistent with mathematical functions (which always return a value), and as such may cause students some confusion. This is further discussed in Chapter 10.

Nested functions were rejected largely due to the complexities that they introduce to scope rules. A nested function typically inherits the scope of the enclosing function, meaning that variables local to the enclosing function may be accessed within the nested function. This may well be confusing to students who have just come to grips with the idea that a function can only access its own parameters and local variables. The difference between a nested function definition and a nested function call is also difficult to explain to students meeting functions for the first time.

8.4.10 I/O

All input in GRAIL is performed using the **read** statement:

```
read name, rank, serial_number
```

To ensure idempotence, each item is read in from a separate line. Only variables of GRAIL's three basic types may be read in. That is, array elements and structure fields must be read in individually, for the same reasons that compound literals cannot be assigned directly to arrays and structures (see section 8.4.3):

```
item student holds student_data
read student's name, student's ID, student's mark
```

The elided form where the structure name is implicit after the first use was also considered:

```
read student's name, ID, mark
```

however, this was rejected on the grounds of ambiguity. A student may legitimately use temporary variables with the same names as structure fields (eg. **name**, **ID**, **mark**), so eliding the structure name could lead to confusion and ambiguous code.

Output is performed with the **write** statement:

```
write name, rank, serial_number
```

Once again, each item is written to a separate line of the output. This is to remain consistent with input – since text is read in to the end of a line, in order to be idempotent.

GRAIL does not contain file I/O. Several variations on file I/O were considered for inclusion, such as the addition of **from**, **to**, and **file** keywords, leading to the following syntax:

```
read name, rank, serial_number from file "data"
write serial_number, name, rank to file "outfile"
```

However, this raises numerous questions. Are files automatically opened, or is there further syntax needed to open them (and hence to close them)? If files are automatically opened, consistency demands that they should also be automatically closed. If they are automatically closed, it is not clear precisely when they should be closed. At the end of the program? When the end of the file is reached for reading, and after the last **write** statement for writing? What mode should be default for writing to a file? Append or overwrite? Overall, the complexity of file I/O outweighs the benefits of introducing it to a simple, short-term language such as GRAIL.

8.5 Summary

The design theory described in Chapter 7 has been applied here to create GRAIL. Although it provides all the fundamentals of programming in the commonest paradigm, GRAIL also departs in significant ways from existing programming languages. It is smaller, its semantics are simpler, and its syntax is grounded in the students' prior experience.

The success of the design process and the resulting language cannot be determined without testing and evaluation with students who are new to programming. To that end, the next chapter describes the evaluation and testing process, together with the results of the testing.

9 Testing and Evaluation

The evaluation process for a programming language typically involves years of experience with the language in its intended environment, be it industrial software engineering, computer science education, research computing, or recreational programming (Allen, Grant, & Smith, 1996; Brusilovsky et al, 1994; Collins & Fung, 1999). Formal evaluation programmes are few and far between, and most evidence gathered is anecdotal in nature. In an educational setting, the demands of courses and curricula make it difficult, if not impossible, to compare different languages in the same course, and different courses generally have sufficiently different curricula to make language comparisons meaningless. Financial constraints limit opportunities for formal comparisons in industry. Where comparisons can be made between courses or projects, the number of different parts of the course which vary between the different settings frequently obscure the results.

It is has been more common to compare single attributes, for example single language constructs, rather than whole languages (Soloway, Bonar, & Ehrlich, 1989; Sime, Green, & Guest, 1973). This approach is useful to the field of language design, as it gives firm indication of the value and impact of individual features, where comparison of entire languages does not easily lend itself to analysis of particular features within the languages. However, this technique leaves the question of which language is best for a particular task unanswered, and the interaction between language features is often neglected.

As Chapter 4 illustrates, the question of which programming language to use for introductory programming is not easily settled. Although much discussion has taken place in computer science literature (for example, Allen, Grant, & Smith, 1996; Budd & Pandey, 1995; Conway, 1993), to date there has been little or no comparative formal evaluation of student interaction with introductory languages. To some extent, comparing disparate languages does not answer the larger question of which language is better from a pedagogical perspective, but formal comparison can answer smaller questions.

Psychologists term our beliefs about the way we think *metacognitive knowledge* (Flavell, 1979). There is a large body of metacognitive knowledge about programming and the impact of programming languages on learning, and on programming style (Blackwell, 1996). However, there is little conclusive evidence

that the choice of language actually makes a difference to the interaction that takes place in an introductory setting, and hence to the quality of learning. The evaluation of GRAIL provides some evidence that choice of language is important, and that it does impact directly on the types of interaction taking place in introductory programming classes.

9.1 Questions

The testing and evaluation of GRAIL was designed to answer the following questions:

1) Are the number and type of errors students make when learning to program affected by the programming language used? Do students make more, less, or the same number of errors using GRAIL compared with some other language?

2) Are syntax and logic errors correlated? While correlation does not imply causality, it may hence be worthwhile aiming to reduce the number of syntax errors students make in order to make learning to program easier, and to allow students to focus on solving the problem rather than battling the syntax?

3) Has the design of GRAIL been successful in minimising the cognitive overhead for students who have no prior programming experience? Although this question is difficult to settle directly, it is closely tied to Question 1. The more details students have to remember, and the more cognitive overhead they must cope with in order to learn a language, the more slips and mistakes they are likely to make while programming. If students make less errors in GRAIL, presumably the cognitive overhead of GRAIL is lower than a language in which students make more errors.

4) Which parts of GRAIL did students have trouble with, and hence need to be redesigned? Which parts of GRAIL worked as intended? As noted in Chapter 2, usability can only be definitively tested using contact with real users. Using careful design, pedagogical and psychological theory, and learning from the successes and failures of other programming languages, GRAIL was designed to be as simple and easy to learn as possible. Without contact with students with no programming experience, however, there is no way of knowing whether the design technique was effective, and how usable GRAIL really is.

Ultimately, all of the above questions explore the overall question: *Does it matter which programming language is used for introductory programming courses?*

9.2 Testing

9.2.1 Outline

To answer the questions outlined in section 9.1, GRAIL needed to be compared with another language, in a standardised environment, using students who had never programmed before.

9.2.2 Choose a language for comparison

Questions 1 and 2 above require that GRAIL should be compared with another language, in order to gather data on relative error rates in two different languages. The following sections discuss the issues involved in choosing a language to compare with GRAIL.

9.2.2.1 Issues

GRAIL was designed for students learning programming for the first time, who have no prior programming experience. To examine error rates among a group of such students, GRAIL is best compared with a language that was also designed for students with no prior programming experience. A language with very different syntax and semantics from GRAIL could be expected to produce the most conclusive results, and hence the strongest indication of whether syntax and semantics do make a difference.

The choice of language for comparison purposes will only impact (if at all) on the scale of the difference.

9.2.2.2 Alternatives

Languages considered for comparison with GRAIL included Turing, Pascal, and LOGO. All three languages were designed for teaching programming, and hence required no prior knowledge of programming. The resources involved in teaching the course and analysing the results were considerable, and there was a limited number of student volunteers to participate in the programme. As a result, it was only possible to compare GRAIL with one other language. As the most syntactically and semantically different from GRAIL, LOGO was chosen, in order to achieve the most definitive results possible.

LOGO was designed to help children learn, to stimulate their mental development and creativity (Papert, 1993). LOGO has the clear visual feedback of the turtle,

which not only makes it very obvious what programs are doing (especially what they are doing wrong), but also provides a motivational lift for students, because it is fun to use. LOGO is a very different language syntactically and semantically from GRAIL, despite having some design aims in common.

9.2.3 A standard interface

LOGO is an interactive language that is typically used by typing one line at a time and receiving immediate feedback from the interpreter. GRAIL is a compiled language in which whole programs must be written before the compiler is invoked. In order to minimise the number of variable elements in the evaluation, it was desirable to create an interface and programming environment that was standard for both languages.

Rather than using an existing text editor with a wide range of features, an interface was designed with minimal features and a very simple layout. The interface contains standard text and file features (cut, copy, paste, open a file, save a file, save as, new file, and quit), and a help function. In the lower right hand corner of the screen is a large button marked "RUN" that runs the program. The user interface was built on DECStation 5000 machines. Stickers with pictures of operators on them were attached to the keyboard to allow students to insert the non-ascii characters in GRAIL using labelled function keys.

Running code that contained syntax errors in either language caused a small image of a bug to appear next to the appropriate lines of code. Clicking on each bug produced the relevant error message.

Code with no syntax errors was run in an input/output window, which showed any textual output from the program, and allowed text to be input from the keyboard when necessary. This window also contained basic text editing facilities. In addition, code that made use of the LOGO turtle caused a graphics window to be displayed, where the turtle could be seen.

No sophisticated error processing was included in the GRAIL compiler, since the LOGO interpreter's error messages were rudimentary. Including more intelligent error checking and reporting would have increased the number of variables in the test, and possibly skewed the results.

9.2.4 Course Design

The syllabus was chosen to encourage learning of the basic concepts of imperative programming, such as algorithms, variables, selection, iteration, primitive data types, simple data structures, and functions.

The teaching materials for both groups were made as similar as possible. The major problems set in each language were the same, with some small differences in initial programs - the LOGO groups, for example, initially made use of the turtle to draw simple shapes. The problems tackled included simple examples of I/O (reading in a name such as "Fred", and printing out "hello Fred"), a simple calculation of the age of a person based on the month and year of birth, a banking program to keep track of savings and stop when the desired target is reached, up to a student database that records students names, id numbers and marks, and prints out the details of the top student in the class.

The short time frame (8 hours per group) necessitated a mixed lecture/practical class style, where a topic was introduced, some examples were covered on the board involving the whole class, and they were then encouraged to test out the theory by coding something on their own. The notes are included in Appendices B & C.

9.2.5 Student recruitment

Students beginning the Monash University CSC1011 Introduction to Programming course were offered the opportunity to take an 8 hour introductory programming course (entitled the Jump Start Programme) before the start of First Semester. All students were surveyed on their prior computing experience, and any form of programming experience, including macro languages in applications such as spreadsheets, excluded students from the programme. Students who volunteered and had no prior programming experience were split randomly into four groups, each containing 6 or 7 students. Two groups were taught GRAIL, the other two LOGO. In total, 13 students completed the GRAIL course and 13 students completed the LOGO course.

9.2.6 Data collection

Each time the students pressed "RUN", a copy of their code was saved. Fifty-five thousand lines of code were collected. In addition, the students were observed during the classes, and qualitative observations were recorded. For example, how

much the students experimented with the language, writing programs that were not set as exercises in class.

9.3 Analysis

9.3.1 Analysis of code

The method of data collection, collecting a copy of every program each time it was run, meant that there were many duplicate programs collected, due to students running the code multiple times to try different inputs. In addition, many errors were run multiple times – sometimes while other errors were fixed, and sometimes due to failed attempts to solve the problem. In order to analyse the code, duplicate programs were removed, and errors were only counted once, regardless of how many times the program was run/compiled before the problem was solved. However, if an error was removed and was later reintroduced, the reintroduction was counted as a new error.

Due to the difficulty of detecting logic errors automatically, analysis of the code was done by visual inspection.

Two broad categories were used in the analysis of the code - syntax and logic errors. Errors classified as syntax errors were generally simple, language-based errors that did not indicate a serious misunderstanding, an incorrect algorithm, or an incorrect translation from algorithm to code. A list of the most common syntax errors for LOGO and GRAIL can be found in tables 9.1 and 9.2. Syntax errors included:

- typographical errors
- use of incorrect keywords (eg "**`end`**" instead of "**`exit`**" in GRAIL, "**`loop`**" instead of "**`repeat`**" in LOGO)
- incorrect punctuation (eg '**`"`**' instead of '**`:`**' in LOGO, or a missing "**`,`**" in a "**`write`**" statement in GRAIL)
- missing keywords (eg missing "**`end`**" statements in both languages)
- incorrect operators
- missing quotes or brackets

Errors classified as logic errors generally involved a flawed algorithm. These included:

- failure to increment a loop counter
- accessing an array position that doesn't exist (ie violating the bounds of an array)
- prompting for the user to enter a value after the value has been read eg

```
read myNumber
write "please enter my number"
```

- using the wrong variable
- incrementing a loop counter in the wrong place (ie outside the loop, or before the loop counter is used for the first time)

missing **end** statements
wrong keyword (eg **end** instead of **exit**)
missing comma in write (eg **write x y**)
typing errors
undeclared variables
missing "**'s**" (eg **read class[index] name**, rather than **read class[index]'s name**)
missing quotes round text

Table 9.1 Common syntax errors in GRAIL

incorrect or missing " or :
bracketing errors (missing brackets, wrong sort of bracket, etc)
incorrect keyword (eg **setitem** and **item** interchanged)
strings missing enclosing square brackets
incorrect list accessing syntax (eg **pr :mylist**, or **pr :index :mylist**, rather than **pr item :x :mylist**)
mismatch between function definition and call (eg call or definition missing parameters)
poorly constructed compound statement (eg **pr item :x triple :mylist**, rather than **pr (triple item :x :mylist)**)

typing errors (mistyped variable names, etc)
: or " used inappropriately (eg **make "output 2*:n**, instead of **output 2*:n** or **:square** where **square** is a function name, or on a number, eg **:1**)
print broken over multiple lines (eg **(pr [.....] :y[.....] :x))**
variable printed in **[]**, (eg **pr [sum]** instead of **pr :sum**)
problems with **ifelse** (eg **if month<3 [....] ifelse [....]** instead of **ifelse month<3 [....] [...]**, or missing argument to **ifelse**)
= instead of **make**

Table 9.2 Common syntax errors in LOGO

In addition to the data collected on error rates, mean-time-to-completion was analysed for both groups, in an attempt to determine whether problems were solved faster in one language than the other. In practice, however, these results proved unreliable, due to a number of interfering factors:

- Power failures and machine breakdowns during some classes artificially inflated completion times.
- Due to the format of the classes, where tutorials were interspersed with coding time, it was not always possible to tell whether a student was programming or listening to the tutor.
- Many exercises were carried over from one tutorial to the next, so it was not possible to tell how long students spent working on these problems in their own time.

9.3.2 Analysis of results

Error rates in GRAIL were significantly different from those in LOGO, for both syntax and logic errors, as shown in tables 9.1 and 9.2.

Syntax Errors (per student)	LOGO	GRAIL
Mean	31.08	13.62
Standard Deviation	8.33	7.98
Number of Students	13	13

Table 9.3 Syntax Errors per student

Logic Errors (per student)	LOGO	GRAIL
Mean	17.77	9.54
Standard Deviation	7.049	5.40
Number of Students	13	13

Table 9.4 Logic Errors per student

The frequency distributions in Figures 9.1 and 9.2 show that the distributions are, as expected, roughly normally distributed (taking into account the small sample size). For both syntax and logic errors, the distributions for GRAIL are clearly different to those for LOGO, with students making more errors in LOGO than in GRAIL. Figures 9.1 and 9.2 show that the error frequencies for the GRAIL groups were substantially lower than those for the LOGO groups. The mean number of syntax errors over the course of the evaluation for the LOGO group was 31.08, with a standard deviation of 8.33, versus a mean of 13.62 and standard deviation of 7.98 for GRAIL. Logic errors gave a less dramatic result, but still significant, with LOGO students making 17.77 errors on average (standard deviation 7.05), versus 9.54 (standard deviation 5.39) for GRAIL.

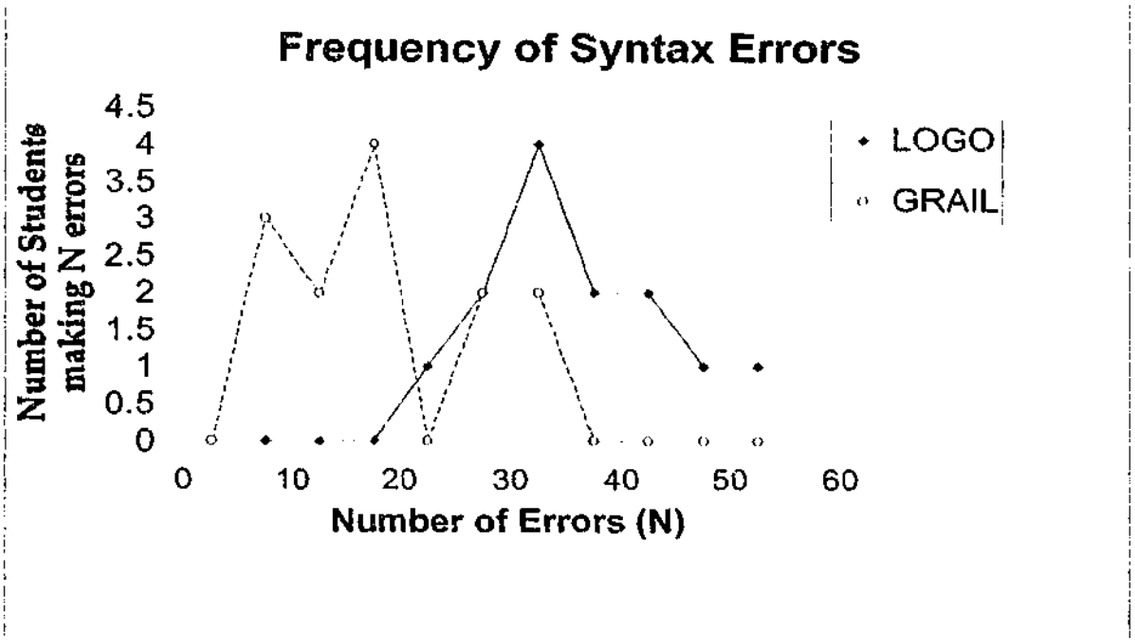

Figure 9.1 Frequency Distribution of Syntax Errors

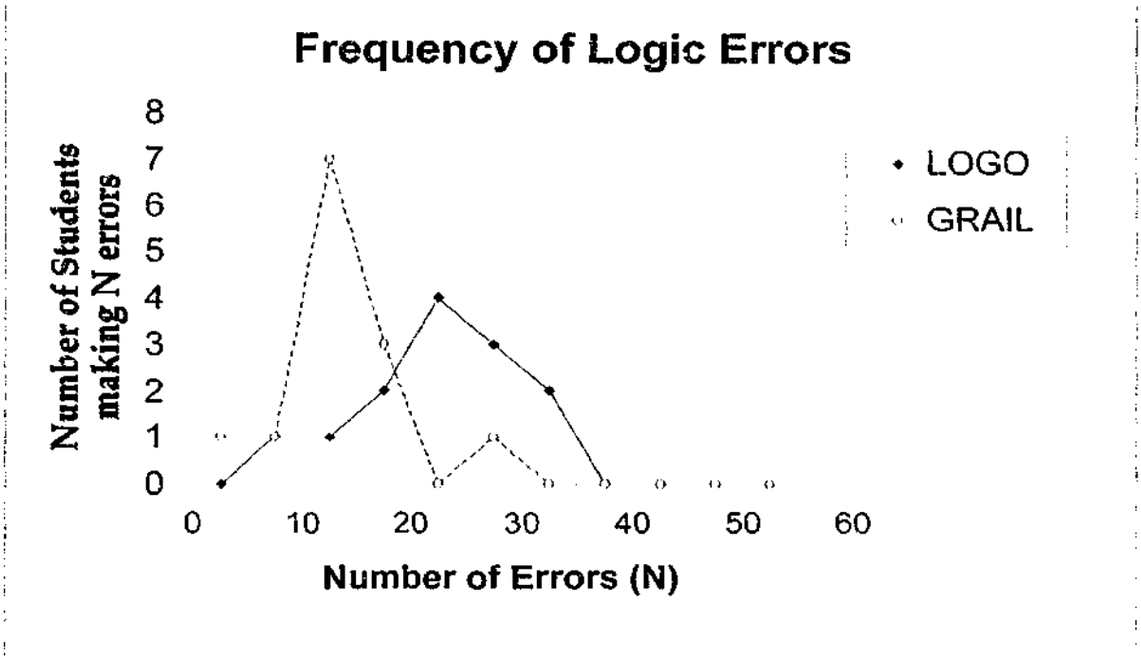

Figure 9.2 Frequency Distribution of Logic Errors

An independent t-test shows that the two groups are significantly different for both syntax errors ($t = 5.24$, $df = 24$, $p < 0.01$) and logic errors ($t = 3.21$, $df = 24$, $p < 0.01$). These results support the hypothesis that the choice of language, or at least the type of language chosen, does make a difference to the type and frequency of errors made by novice programmers.

9.3.2.1 *Correlation between syntax and logic error rates*

The mean error rates for logic and syntax errors in the two groups suggest a link between the number of syntax errors made and the likely number of logic errors. The correlation coefficient ($R=0.56$, $df=24$, $p<0.01$) suggests that syntax and logic errors are moderately correlated. This correlation could be due to student differences, as struggling students are more likely to make both types of errors. It could also mean that syntax errors interfere with problem solving and programming in such a way that an increase in syntax errors makes logic errors more likely. Although no firm conclusions can be drawn from this, the results suggest that minimising syntax errors may have an impact on logic errors, and may facilitate student learning.

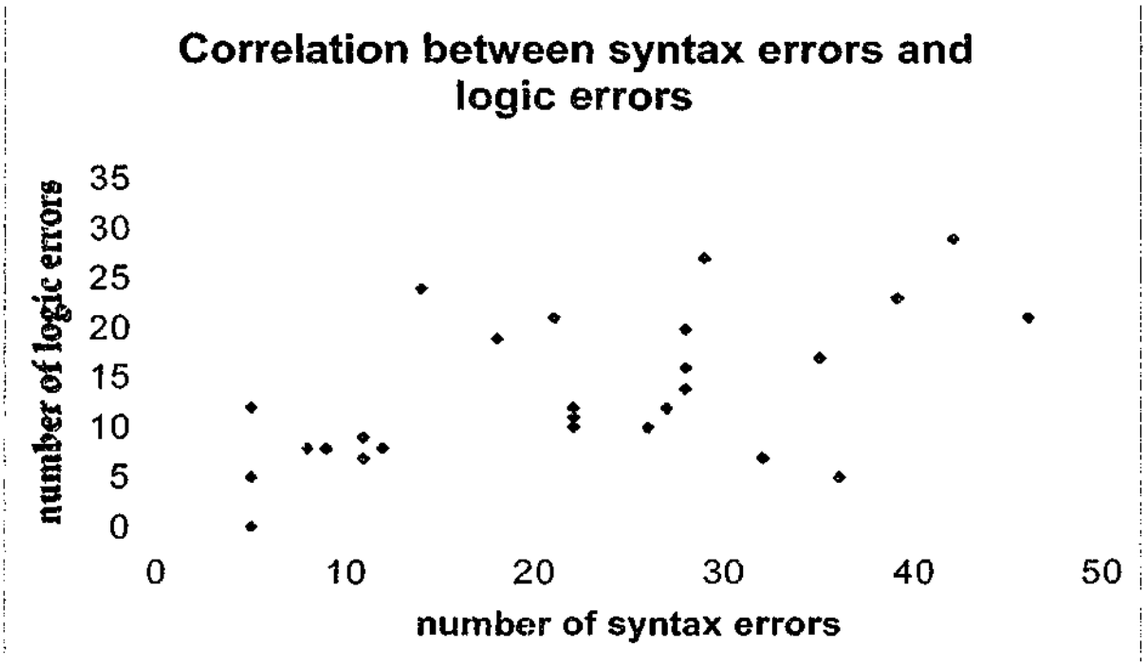

Figure 9.3 Correlation between syntax and logic errors

9.3.3 Observed qualitative results

9.3.3.1 *Willingness to experiment*

Observation of the students during the evaluation process showed that students using GRAIL were more likely to experiment with the language outside the bounds of the course material. Two students from the GRAIL groups (one from each group) also wrote substantial programs in their own time, despite having no access to the software outside class times. One student wrote a database to store friends' phone numbers, the other wrote an extended version of the student marks

database that printed a different message according to the range in which each student's mark fell, and allowed the user the choice of looking at all student details, selected students or just the best student. No students from the LOGO groups wrote programs in their own time, and although the LOGO groups did experiment somewhat with the turtle, when encouraged to do so, no substantial programs outside the syllabus were written in LOGO.

9.3.3.2 Specific errors

Some features of both languages caused students problems. GRAIL's closeness to English or natural language was expected to cause some problems when students slipped into English rather than code, or accidentally typed a grammatically correct sentence in English, rather than a syntactically correct piece of GRAIL code. For example, the following errors did appear, but surprisingly infrequently: **`item x is a number`**, instead of **`item x is number`**, or pluralisation of types, as in: **`item x is array of 10 numbers`**.

Pluralisation of types could be expected to occur where an aggregate variable, such as an array, is declared, as the natural language of the declaration might lead the students to think in English rather than code – "**`10 numbers`**" makes more sense in this context than "**`10 number`**". However, studies have shown that natural language errors are likely to occur regardless of how close the programming language is to natural language (Putnam, 1986; Bruckman & Edwards, 1999). Only 3 students incorrectly pluralised types in array declarations, and each student only made this type of mistake once. Similarly, only two students added an extraneous article to variable declarations, and neither student repeated the error.

Case insensitivity also proved to be a minor issue, with only one student using incorrect capitalisation. Once again the student never repeated the error.

Variable declaration may happen anywhere in a LOGO program. This caused students some problems when writing loops that processed arrays – some students declared their arrays inside the loop, causing the program to fail, since the variable was replaced by a new variable each time through the loop. The GRAIL syntax for loops caused more iterative errors, where students iterated 9 times instead of 10. This is further discussed in Chapter 10.

Operators caused the most errors overall in LOGO: confusion between the " and : operators, and mistakes involving placement of brackets, or use of the wrong sort of bracket, were some of the most common errors.

The GRAIL function syntax caused some problems, as the similarity between "`return`" and "`returns`" – caused confusion, with students interchanging the two and not understanding subsequent syntax errors. The function syntax is redesigned in Chapter 10.

9.4 Discussion of results

9.4.1 Cognitive Dimensions of GRAIL

Chapter 5 gives a description of 6 cognitive dimensions which are relevant to introductory programming languages, and discusses existing programming languages in relation to those dimensions. The results presented here allow the analysis of GRAIL with respect to those dimensions. The optimal values for an introductory programming language used for teaching novices are:

- Closeness of mapping high
- Consistency high
- Diffuseness medium-high
- Error-proneness low
- Hard mental operations low
- Role-expressiveness high

The results from the Jump Start Programme suggest that GRAIL scores well in closeness of mapping and consistency, and is sufficiently verbose to allow high role-expressiveness, but not so verbose as to obscure the structure of the code. Error-proneness and hard mental operations are low, although error-proneness could be lower if some problem constructs were redesigned (see Chapter 10).

The results from the Jump Start Programme suggest that GRAIL has achieved a high closeness of mapping. Students easily translated their algorithms into GRAIL code, and most GRAIL constructs seemed to fit naturally into the way students were trying to solve problems. Consistency is also high in GRAIL. This is partly achieved by only the use of a single construct to represent each individual

concept. Syntactic consistency is also assisted by the size and simplicity of the language.

Role expressiveness is high, which is again assisted by the small number of obvious constructs (and the design policy of leaving out any construct which does not have a single, obvious interpretation). Hard mental operations are not present in GRAIL, and error proneness is generally low. Some error-prone syntactic elements, such as the "**returns**" and "**return**" keywords, could be improved by redesigning these parts of the language (see Chapter 10).

9.4.2 Summary of findings

To summarise, the four questions raised at the beginning of this chapter are discussed below.

1) *Are the number and type of errors students make when learning to program affected by the programming language used?* The results presented in this chapter suggest that language does affect the number and type of errors that students make. Students learning GRAIL made less syntax and logic errors than students learning LOGO.

2) *Are syntax and logic errors correlated?* Syntax and logic errors are moderately correlated. This finding suggests further research would be valuable. The avenues of research that arise from this part of the study are further discussed in Chapter 10.

3) *Has the design of GRAIL been successful in minimising the cognitive overhead for students who have no prior programming experience?* The results suggest that cognitive overhead has been reduced as a result of less errors being caused by the syntax and semantics of the language. However, there is still scope for improvements in the language to reduce the overhead even further. This is further discussed in the next chapter.

4) *Which parts of GRAIL did students have trouble with, and hence need to be redesigned? Which parts of GRAIL worked as intended?* The results of the evaluation show that some parts of GRAIL need to be redesigned. At the same time, many parts of the language worked as intended, combining to reduce the number of errors students made. Those sections of GRAIL that have been redesigned are discussed in the next chapter.

9.4.3 Discussion

Does it matter that programming language is used for introductory programming courses? The original hypothesis, that a programming language is a type of user interface, and hence that usability principles apply to the design of a programming language in the same way as they apply to the design of any other user interface, suggests that the choice of programming language is, indeed, important. Usability demonstrably impacts on the productivity, efficiency, and satisfaction with which users can carry out their tasks. Different user interfaces are suited to different types of activity, and to different types of user.

The results presented in this thesis suggest that the design of a programming language based on usability principles can lead to a language that facilitates increased productivity, efficiency, and user satisfaction by decreasing the incidence of unnecessary and frustrating errors. This suggests that the choice of programming language for introductory programming courses can have a considerable impact on the number and type of errors that students make in the course of learning to program.

These findings support the conclusion that programming languages can usefully be regarded as user interfaces, and that they are subject to the principles of usability. This, in turn, supports the thesis that the language used for introductory programming education *is* important, and can have a significant impact on student learning and motivation.

10 Conclusions and Further Work

10.1 Redesign of GRAIL

10.1.1 Problem features

Some features of GRAIL were clearly shown to be problematic during the evaluation process. This section discusses possibilities for redesign of these features which may solve the problems discovered during testing.

The testing process showed that some students confused the keywords **return** and **returns**.

```
function factorial (item n is number) returns number
    ...
    return result
end function
```

This syntax has since been redesigned, so that the return is now a part of the **end function** statement, like this:

```
function factorial (item n is number) returns number
    ...
end function returning result
```

This new syntax is clearer and may be less error prone. Students tended to confuse the original version, due to the similarity between the "**return**" and "**returns**" keywords. Although it seemed clear during the design phase that "**function f returns number**" and "**return result**" were unambiguous due to the grammatical difference in the English expressions – "**returns number**" is a descriptive term, while "**return number**" is an imperative statement requiring action – it turns out that the visual distinction between the two terms is not sufficient to make the functional difference clear to the students.

There is an additional benefit to the new syntax, as it changes the affordance of the return statement, and makes it clear that there is only one way to exit the function – the **returning** statement cannot be placed anywhere inside the function, it is tied to the **end function** marker. This is more likely to be effective than simply mandating that the return statement must be the last line of the function body without providing any visual or semantic cues to apprise or remind students of this rule.

The loop structure also bears closer examination. Many students confused the **end loop** and **exit loop** statements, and it was not uncommon for students to execute loops over indices from **1** to **9** where they intended to go from **1** to **10** (due to incorrect positioning of the index increment, or use of an erroneous operator, such as < rather than ≤.

In retrospect, the loop was designed around an overly machine-centric concept. When using loops, students most often want to iterate over some list, such as an array of values. To make the loop more problem-based, rather than machine-based, it would seem appropriate to alter the loop construct to be more like a "**foreach**" statement, which takes a list of values and executes the loop once for each value. For example:

```
foreach value in (1..10) do
    sum ← sum + value
    write "progressive sum is", sum
end foreach
```

This also removes some of the complexity of the loop construct, as it is clear that in the above example everything inside the loop happens once for each item in the brackets, so there is no confusion about when the loop is terminated based on the condition (ie does the loop terminate as soon as the exit condition is true, or only when the condition is true at the position in the loop where it is tested).

One drawback with this approach is that it eliminates the possibility of terminating the loop based on input. For example, the following code is difficult to achieve with a **foreach** statement:

```
loop
    read word
    if word = "quit" then
        exit loop
    else
        process(word)
    end if
end loop
```

A common error identified during evaluation was the elision of **end** statements, such as **end program**, **end loop**, and **end if**. This suggests that the use of indentation for block specification might cause less syntax errors, although more investigation is needed to show whether there is a subsequent increase in more pernicious blocking errors.

For example:

```
if x<y then
     write "returning x"
     result ← x
else
     write "returning y"
result ← y
```

where **result** will be set to **y** regardless of the value of **x**.

Another prevalent error was the lack of commas in **write** statements. This could easily be rectified by using white space to separate parameters to I/O operators. For example:

write x y z rather than **write x,y,z**

This may cause some confusion, however, in the case of structures, where there are already spaces in the specification of fields:

```
write student's name student's id student's mark
```

The alternate version with commas is probably clearer:

```
write student's name, student's id, student's mark
```

Another alternative might be to allow only one value to be printed out per **write** statement. This is somewhat cumbersome, but arguably more consistent with the nature of the **write** statement, which only prints one value per line. For consistency, this would necessitate changing the read statement to behave in the same way – only allowing one value to be read in per read. It is not clear what the ideal solution is in this case – user testing of the alternatives is required to settle this question.

10.1.2 Successful features

As noted in Chapter 9, the closeness of GRAIL to English caused very few problems, and seems to have enhanced the role-expressiveness of the language considerably. There was considerably less confusion in GRAIL between operators than there was in LOGO, which suggests that the choice of operators in GRAIL was effective. Unicode characters were used without difficulty.

Line-based strings mapped closely onto the kinds of operations students wanted to carry out during the course – prompting for input was by far the most common

use of strings. Other uses for strings included storing students' names, or the titles and artists of CDs in their collection. Since these required strings to be read in containing spaces, line-based strings were more convenient than the more common model, where strings are read in up to the first whitespace character. Although that requirement can be handled using a separate function for reading in a line of text, the GRAIL model is simpler, in that it requires only a single input function for all types of input.

Idempotent input and output proved to be an effective tool. Apart from the problems with commas mentioned in the previous section, students had no problems with I/O in GRAIL. Values were read in and printed out in predictable ways, since the I/O process does not change the values, and behaves consistently between input and output. This provides further confirmation of the value of line-based strings, since, without them, fully idempotent I/O would be more complicated.

The single, arbitrary precision numeric type, **number**, was also effective. Students seemed to find it comprehensible and predictable. Similarly, the assignment operator, ← worked as anticipated. There was no evidence of the usual misconceptions which surround assignment statements (as discussed in section 7.4), nor any confusion between the equality operator (=) and the assignment operator.

The lack of a switch statement in GRAIL did not prove to be a significant drawback during the evaluation process, although further evaluation with a longer course would be useful.

10.2 Redesign of the Evaluation process

Despite the small number of students involved in the study, the results from the evaluation showed strong indications that error type and frequency are affected by choice of programming language. The study was of a type that has not previously been attempted, and the results suggest that further research of this type would be of value. The choice of LOGO as the comparative language had both positive and negative implications – the high degree of difference between the syntax of LOGO and GRAIL accentuated the results and made it clear that choice of language does impact on error rates and hence is also likely to impact on learning. Further study is warranted with a language closer in nature to GRAIL, perhaps Pascal, and also with a more contemporary language, such as Java.

The normalisation of programming environments for GRAIL and LOGO, which ensured that environmental factors did not contribute to differences in error rates and interaction with the language, also may have had mixed implications. While it laid the focus of the study clearly on language rather than environment issues, it may have counted against LOGO to some degree in the final results, as LOGO was designed as an interactive interpreted language where immediate feedback for simple statements is possible. Typical use of most, if not all LOGO environments is interactive, except in the case of procedures, or `to` statements, which must first be defined, and subsequently called, before the interpreter can proceed to deal with them. However, since LOGO is intended to be used to write many small procedures (Papert, 1993), the impact of the environment may not have been significant in this sense.

The turtle graphics component of LOGO programs was retained for the purposes of evaluation, despite the extra variable this introduced into the study. The appealing nature of the turtle, together with the visible evidence of algorithmic correctness (or lack thereof) is an important part of programming in LOGO, and could not be excluded without impacting on the usable and motivational nature of the language. As it has been suggested that entertainment value is a valuable tool for teaching programming, turtle graphics provide an extra difference from GRAIL that is worth investigating. As it was not possible to differentiate the impact of the turtle from the impact of syntax and semantics, this should perhaps have been studied in isolation, and could be a valuable direction for future study.

Implementation issues impacted on the evaluation process more than anticipated. The compilation speed of GRAIL programs was somewhat slower than the interpretation speed for LOGO. This may have led to students being more cautious before compiling a GRAIL program than when using LOGO, although such a response was not immediately apparent. The fact that, during the analysis, each error was only counted the first time it appeared in the code, no matter how many times the program was compiled containing that error, should have minimised the impact of this problem.

10.2.1 Progressive evaluation

Given time and resources, the language would, like any software, probably be improved by more testing and revision, especially earlier in the design phase. Repeated testing of the language using small groups of users with no computing

experience would probably have resulted in a language with higher usability. Strict application of usability engineering to the design of a programming language could produce interesting results.

10.3 Further work

10.3.1 Further evaluation

While the evaluation was effective and informative, the results do not lend themselves to the evaluation of the precise impact of individual language constructs. In order to determine the effectiveness of each distinct element of the language, more studies would be required comparing GRAIL with itself, but with minor changes. For example, a study of the original version of GRAIL that used a separate **`return`** statement, with the new version that uses the **`returning`** statement tied to the **`end function`** statement, would shed light on the impact of this single change to the language. A sequence of such studies, each comparing single elements of the language, could yield considerable information, and could have a significant impact on the design of future programming languages, both introductory and industrial.

Comparison with different languages would also be of interest, particularly languages more similar to GRAIL than LOGO is, such as Pascal or Turing.

The work in this thesis leads naturally into a similar empirical evaluation of integrated development environments, to determine the impact of the development environment on learning to program, and also on programming in general.

10.3.2 Implications for software engineering

Given that programming language impacts on error rates for novice programmers, what are the implications for software engineering as a whole? There is scope to examine whether the results also hold for expert programmers. If so, there are ramifications for program correctness, reliability, and robustness, as well as programmer efficiency and productivity. If the language used by an expert programmer makes a difference to the number of errors and bugs in the code, then the choice of language for software development is critical, and further empirical analysis of programming languages for experts is necessary to determine which languages are truly more error prone.

10.4 Contributions of this thesis

10.4.1 Programming languages as user interfaces

By treating programming languages as user interfaces, this thesis has successfully applied the considerable body of work on the usability of user interfaces to the study of introductory programming languages.

10.4.2 Usability analysis of programming languages

The application of usability principles to the analysis of programming languages used for introductory programming education has a number of important uses. Firstly, it provides insight into some of the reasons that programming is so difficult for some students to learn. Secondly, it gives educators a framework on which to build their courses, so as best to handle the usability problems in programming languages, as well as a greater insight into programming from the novice perspective. Thirdly, it leads to informed selection of programming languages for introductory courses, and, finally, it is a basis for the design of more usable programming languages for all purposes.

10.4.3 Design framework for introductory programming languages

Using the usability analysis as a basis, this thesis presents a framework for the design of introductory programming languages. The framework is a set of design principles that describe the usability considerations for introductory languages, giving reasons for each principle.

10.4.4 A new introductory programming language

Based on the design framework, this thesis describes the design of a new introductory programming language, GRAIL, designed to facilitate the teaching and learning of introductory programming.

10.4.5 Empirical evaluation of programming languages

An important contribution of this thesis is the empirical evaluation of two programming languages used for introductory programming. This evaluation provides the first strong evidence that choice of programming language for introductory programming does impact on student learning. The evaluation also provides evidence that syntax errors are correlated with logic errors - in other

words that syntax errors may impede students' problem solving and coding abilities. This provides support for educators responsible for choosing languages for use in introductory programming courses to give usability and pedagogical issues a higher priority than has been feasible in the past.

10.5 Conclusion

Despite the obvious advantages of an introductory language specifically designed to facilitate learning, existing "teaching languages" struggle to achieve sufficient pedagogical advantage to overcome the philosophical, political, financial and psychological arguments against them. These arguments are generally based on the ultimate usefulness of industry-relevant languages, on student perceptions of (and demand for) that usefulness, and on teacher familiarity with "mainstream" languages.

GRAIL is yet another attempt to overcome these barriers, but it has the advantage of being extremely simple and deliberately "throw away" (reducing the force of the Industry Relevance From Day One argument).

Although providing all the fundamentals of programming in the commonest paradigm, GRAIL also departs in significant ways from existing programming languages. It is smaller, its semantics are simpler, and its syntax is grounded in the students' prior experience.

This thesis presents strong evidence of the pedagogical advantages of teaching a language which facilitates learning, and does not impede novice programmers with extraneous details, poor syntax, and overly complex and dangerous semantics. This evidence puts educators in a position to make more informed choices about the languages they use for introductory programming, and the way they use them.

Bibliography

Abelson H., Dybvig R.K., Haynes C.T., Rozas G.J., Adams N.I. IV, Friedman D.P., Kohlbecker E.,Steele G.L. Jr., Bartley D.H., Halstead R., Oxley D., Sussman G.J., Brooks G., Hanson C., Pitman K.M., & Wand M. (1998) *Revised^5 Report on the Algorithmic Language,* Journal of Higher-Order and Symbolic Computation 11 (1):7-105, August.

Allen, R.K., Grant, Douglas D., & Smith, R. (1996) *Using Ada as the first Programming language: A Retrospective.* In Proceedings of Software Engineering: Education & Practice, 1996 (SE:E&P'96), IEEE Computer Society Press.

Arnold, Ken, & Gosling, James (1998) *The Java programming language.* 2nd ed. Addison-Wesley, Reading, Mass. USA.

Ausubel, D. (1963) *The Psychology of Meaningful Verbal Learning,* Grune & Stratton.

Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samuelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M. (1960) *Report on the Algorithmic Language ALGOL 60,* Peter Naur Ed., Communications of the ACM. 3(5), pages 299-314, May.

Barnes, J.G.P. (1994) *Programming in Ada,* Addison- Wesley.

Bellamy, C. J. (1989) *Computer programming in FORTRAN 77.* Longman Cheshire, Melbourne.

Ben-Ari, Mordechai, (1996) *Structure Exits, Not Loops.* In SIGCSE Bulletin, 28(3), September.

Blackwell, A.F. (1996). *Metacognitive Theories of Visual Programming: What do we think we are doing?* In Proceedings IEEE Symposium on Visual Languages, pp. 240-246.

Bonar, J. & Soloway, E. (1985) *Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers.* In Human-Computer Interaction, 1(2):133-161.

Booch, Grady (1991) *Object Oriented Design with Applications,* Benjamin/Cummings, USA.

Brilliant, S. S. & Wiseman, T. R. (1996) *The First Programming Paradigm and Language Dilemma*, Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, March, pp338-342.

Brooks, Ruven (1977) *Towards a theory of the cognitive processes in computer programming.* In International Journal of Man-Machine Studies 1977, 9, pp 737-751.

Bruckman, Amy & Edwards, Elizabeth (1999) *Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language.* In Proceedings Computer Human Interaction 1999 (CHI'99), Pittsburgh, USA, May.

Brusilovsky, P., Calabrese, E., Hvorecky, E., Kouchnirenko, A., & Miller, P. (1997) *Mini-languages: A Way to Learn Programming Principles.* In Education and Information Technologies, 2(1): 65-83.

Brusilovsky, P., Kouchnirenko A., Miller P., Tomek I. (1994) *Teaching programming to novices: A review of approaches and tools.* In T.Ottman, I.Tomek (eds.) Proc.of ED-MEDIA'94 – World Conference on Educational Multimedia and Hypermedia. Vancouver, Canada, June 25-30, pp 103-110.

Budd,T. A. & Pandey, R. K. (1995) *Never Mind* the *Paradigm, What About Multiparadigm Languages?* SIGCSE Bulletin, 27(2), June 1995, pp 25-30.

Clocksin, W.F. & Mellish, C.S. (1981) *Programming in Prolog*, Springer-Verlag.

Collins, Trevor D., & Fung, Pat (1999) *Cognitive Modelling for Psychology Students: The Evaluation of a Pragmatic Approach to Computer Programming for Non-Programmers.* In G Cummings, T Okamoto and Louis Gomez (eds), Proceedings of the 7th International Conference on Computers in Education (ICCE'99), Chiba, Japan, November, IOS Press, Volume 1, pp 216-223.

Colmerauer, Alain, & Roussel, Philippe (1996) *The Birth of Prolog.* In History of Programming Languages – II, Thomas J. Bergin Jr and Richard G. Gibson Jr (Eds), ACM Press, New York.

Constantine, Larry L. (1995) *Constantine on Peopleware.* Yourdon Press, Eaglewood Cliffs, New Jersey, USA.

Conway, D. (1993a) *Criteria and Consideration in the Selection of a First Programming Language*, Technical Report 93/192, Department of Computer Science, Monash University, December.

Conway, D. M. (1998) *An algorithmic approach to English pluralization*, in C Salzenberg (ed), Proceedings of the Second Annual Perl Conference, San Jose, CA, USA, 17-20 August, O'Reilly.

Conway, D. M. (1993b) *HyperLecture: a self-correlating lecture presentation and revision system*, Proceedings of the 1993 East-West International Conference on Human-Computer Interaction (EWHCI'93) Moscow, Russia, vol. 1, 11-24.

Corsini, R. J. (1994) *Encyclopedia of Psychology*, Wiley.

Dalbey, John & Linn, Marcia C. (1985) *The Demands and Requirements of Computer Programming: A Literature Review.* Journal of Educational Computing Research, Vol 1(3).

Dawkins, Richard (1989) *The Selfish Gene.* Oxford University Press.

Dijkstra, Edsger W., (1985) *Fruits of Misunderstanding.* Datamation, February 15.

Dix, A., Finlay, J., Abowd, G., & Beale, R. (1998) *Human Computer Interaction.* Prentice Hall.

Dix, T. & Lien, T. (1993) *Safe-C for Introductory Undergraduate Programming*, in Proceedings of the 16th Australian Computer Science Conference (ACSC), 1993.

du Boulay, Benedict and Matthew, Ian (1984) *Fatal error in pass zero: how not to confuse novices.* Behaviour and Information Technology, 1984, volume 3(2), pp 109-118.

du Boulay, J. B. H. (1989) *Some difficulties of learning to program.* In E. Soloway and J.C. Spohrer, editors, Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale.

du Boulay, J. B. H. (1979) *Logo learning by school teachers.* Edinburgh: Doctoral dissertation, Department of Artificial Intelligence, University of Edinburgh.

DuPuis, C. & Burnett, M. (1997) *An Animated Turing Machine Simulator in Forms/3*, Department of Computer Science, Oregon State University, Technical Report #97-60-08 July.

Dybvig, R. Kent (1987) *The SCHEME Programming Language*, Prentice Hall, New Jersey.

Dyck, Jennifer L., and Mayer, Richard E. (1985) *BASIC Versus Natural Language: Is There One Underlying Comprehension Process?* In Proceedings of the CHI'85 Conference on Human Factors in Computing Systems, San Francisco, April.

Eisenstadt, Marc and Lewis, Matthew W. (1992) *Errors in an Interactive Programming Environment: Causes and Cures.* In Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence (1992), Marc Eisenstadt, Mark T. Keane, and Tim Rajan, (eds), Lawrence Erlbaum Associates, Hillsdale USA.

Evans, J. St. B. T., Barston, J. L., and Pollard, P. (1983) *On the Conflict Between Logic and Belief in Syllogistic Reasoning*, Memory and Cognition 11, pp295-306.

Feuer, A. & Gehani, N., 1984 : *Comparing and Assessing Programming Languages: Ada, C, and Pascal*, Prentice-Hall.

Flavell, J.H. (1979) *Metacognition and cognitive monitoring*, American Psychologist 34(10), pp. 906-911.

Friendly, Michael (1992) *Advanced LOGO : A Language for Learning*. Prentice-Hall.

Geitz, R. (1994) *Concepts in the classroom, programming in the lab*, in Technical Symposium on Computer Science Education Selected papers of the twenty-fifth annual SIGCSE symposium Computer science education March 10 - 12, Phoenix, Arizona, USA .

Geurts, L., Meertens, L., and Pemberton, S. (1990) *ABC Programmer's Handbook*, Prentice Hall.

Goldberg, A., Robson, D. (1983) *"Smalltalk-80: The language and its implementation"*. Addison-Wesley.

Green, T. & Blackwell, A. (1998) *Cognitive Dimensions of Notations and other information artefacts.* Tutorial presented at HCI'98. Available at http://www.cl.cam.ac.uk/~afb21/publications/CDTutSep98.pdf

Green, T. R. G. (1989) *Cognitive Dimensions of Notations*, in People and Computers V: Proceedings of Human Computer Interaction 1989, (HCI'89), Cambridge University Press.

Green, T. R. G. (1990) *Programming Languages as Information Structures.* In Psychology of Programming, edited by J.-M. Hoc, T. R. G. Green, R. Samur◷ay, and D. J. Gilmore, Academic Press.

Harbison, S. (1992) *Modula 3*, Prentice Hall.

hbc (1998) *Haskell compiler* http://www.cs.chalmers.se/~augustss/hbc/hbc.html.

Hofstadter, D. (1979) *Gödel, Escher, Bach: an Eternal Golden Braid*, Part II, Chapter 10: "Similar Levels", Basic Books.

Holt, R.C. & Hume, J.N.P. (1984) *Introduction to Computer Science using the Turing Programming Language*, Prentice-Hall.

Holt, R.C., Matthews, P.A., Rosselet, J.A., Cordy,J.R. (1988) *The Turing Programming Language: Design and Definition*, Prentice Hall.

Hudak, P. & Fasel, J.H. (1992) *A Gentle Introduction to Haskell*, SIGPLAN Notices, 27(5), May, ACM.

Iverson, Kenneth E., (1962) *A Programming Language*, Wiley.

Jeffries, R. Turner, A., Polson, P., & Atwood, M. (1981) *The processes involved in designing software.* In J. Anderson (Ed.), Cognitive skills and their acquisition. Erlbaum, Hillsdale, New Jersey.

Katz, Elizabeth E. & Porter, Hayden S. (1991) *HyperTalk as an overture to CS1.* In Proceedings of the twenty-second SIGCSE technical symposium on Computer Science Education, March, San Antonio, TX USA. Pages 48-54

Kernighan, B. & Ritchie, D. (1988) *The C Programming Language, 2nd ed.*, Prentice-Hall.

Kessler, Claudius M., and Anderson, John R. (1986) *Learning flow of control: Recursive and iterative procedures.* Human-Computer Interaction, 1, 135-166.

Kölling, M. (1999a) *The Design of an Object-Oriented Environment and Language for Teaching*, PhD Thesis, Basser Department of Computer Science, University of Sydney.

Kölling, M. (1999b) Teaching Object Orientation with the Blue Environment. Journal of Object Oriented Programming, 12(2), May, pp14-23.

Kölling, M. and Rosenberg, J. (1996b) *An Object-Oriented Program Development Environment for the First Programming Course.* Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, March, pp.83-87.

Kölling, M., & Rosenberg, J. (1996a) *Blue - A Language for Teaching Object-Oriented Programming*, Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, March, pp.190-194

Kölling, M., Koch, B., & Rosenberg, J. (1995) Requirements for a First Year Object Oriented Teaching Language, SIGCSE Bulletin, 27(1), Mar, pp 173-177.

Lamport, Leslie (1986) *LATEX : a document preparation system.* Addison-Wesley, Reading, Mass., USA.

Landauer, Thomas K. (1995) *The trouble with computers : usefulness, usability, and productivity.* MIT Press, Cambridge, Mass., USA.

Language List (2000) *The Language List* http://cui.unige.ch/langlist

Levy, S. (1995) *Computer Language Usage in CS1: Survey Results,* SIGCSE Bulletin, 27(3), Sept 1995, pp21-26.

Mayer, Richard E. (1983) *Can you repeat that? Qualitative effects of repetition and advance organizers on learning from science prose.* Journal of Educational Psychology, 75, 40-49.

Mayer, Richard E. (1989) *The psychology of how novices learn computer programming.* In E. Soloway and J.C. Spohrer, editors, Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale.

Mayer, Richard E. (1992) *Thinking, Problem Solving, Cognition.* Second Edition. W. H. Freeman and Company, New York.

M'Iver, L. & Conway, D. (1996) *Seven Deadly Sins of Introductory Programming Language Design,* In Proceedings of Software Engineering: Education & Practice, 1996 (SE:E&P'96), IEEE Computer Society Press.

M'Iver, L. & Conway, D. M., (1999) *GRAIL: A Zero'th programming language,* in G Cummings, T Okamoto and Louis Gomez (eds), Proceedings of the 7th International Conference on Computers in Education (ICCE'99), Chiba, Japan, November 1999, IOS Press, Volume 2, pp 43-50.

M'Iver, L. (2000) *The Effect of Programming Language on Error Rates of Novice Programmers.* In Proceedings Twelfth Annual Meeting of the Psychology of Programming Interest Group, Corigliano Calabro, Italy, April, Edizioni Memoria.

Meertens, L. (1981) *Issues in the Design of a Beginners' Programming Language,* Algorithmic Languages, de Bakker/van Vliet (eds), IFIP North Holland Publishing Company, pp167-184

Melissa virus (1999)
http://support.microsoft.com/support/exchange/content/whitepapers/melissa.doc

Meyer, B. (1992) *Eiffel: The Language*, Prentice-Hall.

Miller, P., Pane, J., Meter, G. & Vorthman, S. (1994) *Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University.* Interactive learning Environments. 4(2) pp 140-158.

Mody, R.P. (1991) *C in Education and Software Engineering*, SIGCSE Bulletin, 23(3) September.

Mulholland, P. & Watt, S.N.K. (1998) *Hank: A friendly cognitive modelling language for psychology students.* In Proceedings of the IEEE Symposium on Visual Languages, VL'98, Nova Scotia, Canada.

Murnane, J. (1993) *The Psychology of Computer Languages For Introductory Programming Courses*, New Ideas in Psychology, 11(2), pp 213-228.

Myers, Brad (2000) http://www.cs.cmu.edu/~NatProg/langeval.html

Nielsen, Jakob (1993) *Usability Engineering.* Academic Press.

Nielsen, Jakob, (1994) *Enhancing the explanatory power of usability heuristics.* In Proceedings Human factors in computing systems: celebrating interdependence (CHI'94), Boston, United States, April 24 - 28, pp 152-258.

Norman, Donald A. (1990) *The Design of Everyday Things,* Doubleday.

Norman, Donald A. (1992) *Turn signals are the facial expressions of automobiles.* Addison Wesley.

Norman, Donald A. (1993) *Things that make us smart.* Addison Wesley.

Norman, Donald A. (1998) *The invisible computer : why good products can fail, the personal computer is so complex, and information appliances are the solution.* MIT Press, Cambridge, Mass.

Pane, John & Myers, Brad. (2000) *The Influence of the Psychology of Programming on a Language Design: Project Status Report.* 12th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2000, Corigliano Calabro, Italy. 10-13 Apr. 2000.

Papert, Seymour (1993) *Mindstorms: Children, Computers, and Powerful Ideas*, 2nd Ed., BasicBooks, New York.

Pattis, Richard E., Roberts, Jim, & Stehlik, Mark. (1995) *Karel the Robot: A Gentle Introduction to the Art of Programming.* 2nd Edition. John Wiley & Sons, Inc.

Pea, R. D. (1986) *Language-independent conceptual bugs in novice programming.* Journal of Educational Computing Research, 2(1), 25-36.

Pennington, Nancy (1987) *Comprehension Strategies in Programming.* In Empirical Studies of Programmers: Second Workshop, Ablex Publishing Corporation, New Jersey.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. (1989) *Conditions of Learning in Novice Programmers.* In E. Soloway and J.C. Spohrer, editors, Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale.

Philipchalk, Ronald P & McConnell, James V (eds) (1994) *Understanding Human Behaviour,* 8th edition, Harcourt Brace College Publishers, Fort Worth, Texas.

Polson, P. G., & Lewis, C. H. (1990) *Theory-based design for easily learned interfaces.* Human–Computer Interaction 5, 2&3 , 191–220.

Putnam, Ralph T., Sleeman, D., Baxter, Juliet A., and Kuspa, Laiani K. (1986) *A Summary of Misconceptions of High School Basic Programmers.* In the Journal of Educational Computing Research, Volume 2(4).

Raggett, Dave (1998) *Raggett on HTML 4.* Addison-Wesley, Harlow, England.

Reason, James (1990) *Human Error.* Cambridge University Press, Cambridge.

Reid Report (1999) http://irenaeus.ccsu.ctstateu.edu/~www/reference/language-lists/lang-list-20.html

Ritchie, Dennis (1996) *The Development of the C Language.* In History of Programming Languages – II, Thomas J. Bergin Jr and Richard G. Gibson Jr (Eds), ACM Press, New York.

Rumelhart, D. & Norman, D. (1978) *Accretion, tuning and restructuring: Three modes of learning,* In. W. Cotton, J.and Klatzky, R. (eds.), "Semantic Factors in Cognition", Erlbaum.

Schneiderman, Ben (1987) *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley.

Shafer, Dan, (1991) *The complete book of Hypertalk* 2. Addison-Wesley.

Sime, M. E., Green, T.R.G., & Guest, D.J. (1973) *Psychological Evaluation of Two Conditional Constructions Used in Computer Languages.* International Journal of Man-Machine Studies, 5, pp 105-113.

Soloway, Eliiot, Bonar, Jeffrey & Ehrlich, Kate (1989) *Cognitive Strategies and Looping Constructs: An Empirical Study.* In E. Soloway and J.C. Spohrer, editors, Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale.

Spohrer, James C., and Soloway, Elliot. (1989) *Novice Mistakes: Are the Folk Wisdoms Correct?* In E. Soloway and J.C. Spohrer, editors, Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale.

Spohrer, James C., Soloway, Elliot, and Pope, Edgar (1985) *Where The Bugs Are.* In Proceedings of the CHI'85 Conference on Human Factors in Computing Systems, San Francisco, April.

Springer, G. & Friedman, D.P. (1989) *Scheme and the Art of Programming*, The Massachusetts Institute of Technology.

Stroustrup, B. (1991) *The C++ Programming Language, 2nd edition*, Addison Wesley.

Stroustrup, B. (1994) *The Design and Evolution of C++*, Addison-Wesley.

Thorburn, W. M. (1915) "Occam's razor," Mind, 24, pp. 287-288.

Thorndike, E. (1932) *The Fundamentals of Learning*, New York: Teachers College Press.

Turner, D.A. (1985) *Miranda: A non-strict functional language with polymorphic types,* in J.-P. Jouannaud (ed.) Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201, Springer Verlag.

US Department of Defence, (1981) *The Programming Language Ada Reference Manual*, Lecture Notes in Computer Science, Volume 106, Springer-Verlag.

Van Someren, Maarten W. (1990) *What's Wrong? Understanding beginners' problems with Prolog.* in Instructional Science 19, pp 257-282.

Wall, L., Christianson, T., & Schwartz, R.L. (1996) *Programming Perl, 2nd Edition*, O'Reilly & Associates.

Wiedenbeck, S. (1985) *Novice/expert differences in programming skills.* International Journal of Man-Machine Studies, 23, 383-390.

Wilensky, R. (1984) *LISPcraft*, Norton.

Wirth, N. & Jensen, K. (1975) *Pascal User Manual and Report*, Springer-Verlag.

Wirth, Niklaus (1988) *Programming in Modula-2*, 4th edition, Springer Verlag.

Appendix A - GRAIL Grammar

```
program:        typeDecl(s)opt
                function(s)opt
                program ID
                        decl(s)
                        stmnt(s)
                end program
                function(s)opt

stmnt:          variable ← expr
                | ifStmnt
                | loopStmnt
                | IOStmnt
                | functionCall
decl:

ifStmnt:        if expr then
                        stmnt(s)
                        elseIfStmntopt
                        elseStmntopt
                end if
elseIfStmnt:    else if expr then
                        stmnt(s)
elseStmnt:      otherwise
                        stmnt(s)

loopStmnt:      loop
                        stmnt(s) opt
                        if (expr) then
                                exit loop
                        end if
                        stmnt(s) opt
                end loop

orExp:          andExp | orExp or logExp
andExp:         notExp | andExp and logExp
notExp:         logExp | not logExp
logExp:         (orExp) | boolean | boolExp | eqExp

boolExp:        gtExp1 | ltExp1 | boolean | (boolExp)
```

eqExp: *expr* = *expr* | *expr* ≠ *expr* | *boolExp* = *boolExp* | *boolExp* ≠ *boolExp*

gtExp1: *gtExp1* > *expr* | *gtExp1* ≥ *expr* | *gtExp*

gtExp: *expr* > *expr* | *expr* ≥ *expr*

ltExp1: *ltExp1* < *expr* | *ltExp1* ≤ *expr* | *ltExp*

ltExp: *expr* < *expr* | *expr* ≤ *expr*

expr: *expr* + *term* | *expr* - *term* | *term*

term: *term* + *factor* | *term* × *factor* | *factor*

factor: *value*
| (*expr*)
| - *expr*
| + *expr*
| *functionCall*

simpleVal: *NUM*
| *TEXT*
| *BOOLEAN*

value : *simpleVal*
| *variable*

variable : *ID*
| *variable* 's *ID*
| *variable* ' *ID*
| *variable* [*value*]

valueList : *value*
| *value* , *valueList*

simpleDecl: **item** ID **is** *type_name* $simpleVal_{opt}$

decl : *simpleDecl*
| **constant** ID **is** *simpleVal*

typeDecl: **type** ID **is** *type_name*
| **type** ID **has**
fieldDecl(s)
end type

fieldDecl : **field** ID **is** *type_name*

type_name : **number**
| **text**
| **boolean**
| ID
| **array of** *NUM type_name*
| **array of** *NUM* × *NUM type_name*

IOStmnt : **write** *valueList*
| **read** *valueList*

functionCall : ID **(** $value_{opt}$ (,$value_{opt}$)**)**

function: **function** ID (*params*$_{opt}$ **)**
decl(s)
stmnt(s)
end function
| **function** ID (*params*$_{opt}$ **) returns** *type_name*
decl(s)
stmnt(s)
return *value*
end function

new function syntax:

function ID (*params*$_{opt}$ **) returns** *type_name*
decl(s)
stmnt(s)
end function returning *value*

params: *simpleDecl* (, *params*$_{opt}$)

Appendix B

Jump Start Programme Notes (GRAIL)

Algorithms

An algorithm is a description of how to complete a task - for example, a recipe, instructions for using a video recorder, or directions on how to reach your house. Algorithms do not have to be written in a programming language. When planning a program, a good approach is to write a description in English of what needs to be done, and then translate that into a programming language. Algorithms need to be very clear and orderly. They are often written in point form, eg, an algorithm for getting to Monash from Huntingdale Station might be:

1. Go to the Monash bus stop.
2. wait for the Monash bus.
3. when the bus arrives, get on it.
4. Get off the bus at the Monash bus loop.

Input/Output (I/O)

Input and Output are the ways computers communicate with users. Input is usually something the user types in (ie from the keyboard) and output is something the computer writes on the screen. There are other types of I/O (eg. input using the mouse, output to a printer etc).

The GRAIL I/O commands are "read" and "write". "read" reads from the keyboard, and "write" writes to the screen.
eg

```
read name
write name
```

Items

Items (also called "variables") can be thought of as places to store things. If you need to remember someone's name, a simple way is to write it down until you need it. Items in GRAIL can be of different types, for example you would store someone's ID number in a "number" item, and you would store their name in a "text" item. GRAIL needs to know in advance what sort of items you plan to use, so each item needs to be "declared" like this:

```
item name is text
item id is number
```

To put a value in an item, you use an "assignment" statement (so called because you are "assigning" a value to an item) like this:

```
name ← "Linda McIver"
id  ← 12345678
```

Selection

Most algorithms will require you to make choices at some point - eg "if the bus is a Monash bus, get on it, otherwise wait for the next one." In programming terms, this is referred to as "selection", because you are asking the computer to make a selection between two or more possible actions.
In GRAIL, this is done with an "if" statement, like this:

```
if  id >= 20000000 then
     write "That's not a valid ID number."
     id ← backup
else
     write "The ID in question is ",id
end if
```

Iteration

Iteration involves doing things repeatedly. For example, you might want to read in a list of numbers. If you work your way through the list, reading in each number progressively, that is known as iterating through the list.
In GRAIL, iteration is generally done with a loop, like this:

```
loop
     write "Please enter a positive number"
     read n
     if (n<0) then
          exit loop
     end if
     sum ← sum + n
end loop
```

Structures/Records

Structures allow you to keep different items together - for example, you might want to keep a list of student names and id numbers together, or CD titles, artists and lengths etc. In GRAIL, you can declare an "aggregate type" to hold whatever items you wish to keep together, like this:

```
type student has
     field name is text
     field id is number
end student

type cd has
     field title is text
     field artist is text
     field length is number
end type
```

You can then declare an item of that type, like this:

```
item myStudent is student
```

and access the separate parts of that type like this:

```
myStudent's name  ← "Linda"
```

```
myStudent's id  ← 12345678

write "Name: ",myStudent's name,  "ID: ",myStudent's id
```

Arrays

Arrays allow you to store lots of items of the same type, for example a list of names, or a list of numbers, or even a list of structures. To access a specific element of an array, you use an "index". Array indices start at 1, and go to the size of the array, like this:

```
item numList is array of 5 number

numList[1] ← 4
numList[2] ← 56

item class is array of 10 student

class[1]'s name  ← "Fred Nerk"
class[1]'s id  ← 23456789
```

Of course, it gets rather clumsy accessing each element separately like this, so it makes sense to combine loops and arrays, like this:

```
item index is number

index ← 1
loop
     if (index>10) then
          exit loop
     end if
     write "Please enter the next name."
     read class[index]'s name
     write "Please enter this student's id."
     read class[index]'s id
     index←index+1
end loop
```

Functions/Procedures

For large programs, it can be helpful to break a problem down into smaller chunks. These chunks are often called "functions" or "procedures". A function usually performs a small, well-defined task, such as finding the square root of a number, or printing out a list of names. Functions often have values given to them, upon which they operate - for example, if you wanted to find the square of 4, you would "pass" the value 4 to the function "square", like this:

```
result  ← square(4)
```

In this case, the function also returns a value, which we have placed into "result". Functions may return values of any type (including structures and arrays), but they do not always return anything.

A function may be defined in GRAIL like this:

```
function  square (item n is number) returns number
     return n×n
end function
```

or

```
function printList (item names is array of text)
     item i is number

     i ←1

     loop
          if (i>10) then
               exit loop
          end if
          write "name ",i," is ", names[i]
          i ←i+1
     end loop

end function
```

Appendix C

Jump Start Programme - Notes (LOGO)

Algorithms

An algorithm is a description of how to complete a task - for example, a recipe, instructions for using a video recorder, or directions on how to reach your house. Algorithms do not have to be written in a programming language. When planning a program, a good approach is to write a description in English of what needs to be done, and then translate that into a programming language. Algorithms need to be very clear and orderly. They are often written in point form, eg, an algorithm for getting to Monash from Huntingdale Station might be:

1. Go to the Monash bus stop.
2. wait for the Monash bus.
3. when the bus arrives, get on it.
4. Get off the bus at the Monash bus loop.

Input/Output (I/O)

Input and Output are the ways computers communicate with users. Input is usually something the user types in (ie from the keyboard) and output is something the computer writes on the screen. There are other types of I/O (eg. input using the mouse, output to a printer etc).

The LOGO I/O commands are "print" (abbreviated to "pr") for printing to the screen, and "readword" for reading from the keyboard.
eg

```
print [hello]
make "mynum readword
```

Items

Items (also called "variables") can be thought of as places to store things. If you need to remember someone's name, a simple way is to write it down until you need it. In LOGO, to put a value in an item, you use an "assignment" statement (so called because you are "assigning" a value to an item) like this:

```
make "name [Linda McIver]
make "id 12345678
```

The first statement gives **name** the value "Linda McIver" (a string of text). The second gives **id** the value 12345678 (a number).

Selection

Most algorithms will require you to make choices at some point - eg "if the bus is a Monash bus, get on it, otherwise wait for the next one." In programming terms, this is referred to as "selection", because you are asking the computer to make a selection between two or more possible actions.
In LOGO, this is done with an "if" statement, like this:

```
if  :id > 20000000 [
     (print [That's not a valid ID number.] )
     make "id 0
]
```

The "if" statement only allows you to execute an action if the condition is true. If you'd also like to do something when the condition is false, you need an "ifelse" statement, like this:

```
ifelse :id > 20000000 [
     (print [That's not a valid ID number.] )
     make "id 0
][
     (print [The ID in question is] :id)
]
```

Iteration

Iteration involves doing things repeatedly. For example, you might want to draw a square. If you execute one command, or group of commands, multiple times, that is known as iteration.
In LOGO, iteration is generally done with a "repeat" statement, like this:

```
repeat 4 [fd 30 rt 90]
```

Arrays

Arrays allow you to store lots of items of the same type, for example a list of names, or a list of numbers, or even a list of structures. To declare an array in LOGO, you'd use a statement like this:

```
make "mylist array 10
```

where **mylist** is the name of the variable, and **10** is the size, or length, of the array.

To access a specific element of an array, you use an "index". Array indices start at 1, and go to the size of the array, like this:

```
print (item 1 :mylist)
```

and to set the value of an element of an array, you use "setitem", like this:

```
setitem 1 :mylist [12345]
setitem 2 :mylist [6]
```

We can also use structures called **lists** to combine related items, like this:

```
make "newlist (list [fred] [1234] )

(setitem 1 :mylist (list [fred] [1234]) )
(setitem 2 :mylist (list [ralph] [1235]) )
```

Using this technique we can keep things like names and id numbers together.

Of course, it gets rather clumsy accessing each element separately like this, so it makes sense to combine loops and arrays, like this:

```
make "mylist array 10

make "index 1

repeat 4 [
     print [Please enter the number]
     make "num readword
     print [Please enter a name]
     make "name readword
     (setitem :index :mylist (list :name :num) )
     make "index :index+1
]
```

And then we can print out the resulting array like this:

```
make "index 1
repeat 4 [
     (print [item] :index [is] (item :index :mylist))
     make "index :index+1
]
```

Functions/Procedures

For large programs, it can be helpful to break a problem down into smaller chunks. These chunks are often called "functions" or "procedures". A function usually performs a small, well-defined task, such as finding the square root of a number, or printing out a list of names. Functions often have values given to them, upon which they operate - for example, if you wanted to find the square of 4, you would "pass" the value 4 to the function "square", like this:

```
make "result square 4
```

In this case, the function also returns a value, which we have placed into "result". Functions may return values of any type (including structures and arrays), but they do not always return anything.

A function may be defined in LOGO like this:

```
to square :num
     output :num * :num
end

or

to draw.squares :side
     repeat 4 [
          repeat 4 [fd :side rt 90]
          pu lt 45 fd :side pd
     ]
end
```