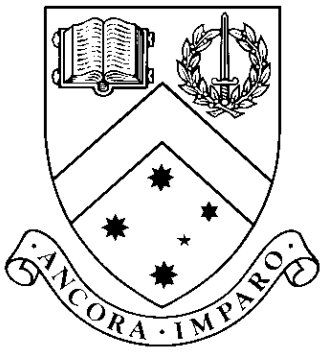
AntiCompiler: an Educational Tool for First Year Programming

by

Kymberly Fergusson, BCompSci
kef@mail.csse.monash.edu.au



Literature Review

Bachelor of Computer Science with Honours (4300)

Supervisor: Dr. Linda McIver and Mr. Martin Dick
{lindap,mdick}@mail.csse.monash.edu.au

School of Computer Science and Software Engineering

Monash University

August, 2003

AntiCompiler: an Educational Tool for First Year Programming

Kymberly Fergusson, BCompSc(Hons)
Monash University, 2003

Supervisor: Dr. Linda McIver and Mr. Martin Dick

Abstract

Learning a programming language is a difficult challenge, especially for those who have never programmed before. Various obstacles can impede progress and cause frustration amongst novices. Common problems include learning various new programming concepts, dealing with logic errors in programs (semantic errors) that are not easily identifiable and difficulties understanding the syntax of the programming language. It is therefore important in teaching a first programming language to minimise the problems that novices have with learning new concepts coupled with obscure syntax and unfamiliar problem solving techniques. Some teaching aids include using a well designed language, using tools to help novices develop an understanding of the language. It is also useful in minimising problems to teach novices various techniques to identify and correct common pitfalls.

Many programming languages taught to novice programmers have obscure and terse syntaxes which complicate the process of semantic error detection. Therefore it would be useful for a tool to convert code into a more descriptive, natural language, hopefully allowing for a better understanding of programming concepts and errors. Semantic errors are extremely difficult for beginner programmers to locate and correct, thus it may be useful for novices to use a program that automatically detects and highlights common errors.

The goal of this project is to develop a tool with the primary function of translating C language code into a more natural and accessible pseudo-language. If feasible, the tool should also be able to detect and highlight some common semantic errors. For this purpose existing research into the areas of language design, educational techniques for first time programmers and common errors made by these programmers were reviewed. Further attention was paid to various tools and languages currently available.

Acknowledgments

I would like to thank Bernd Meyer, Caroline McGregor, Martin Dick and Linda McIver for providing feedback, proofreading and support during the writing of this report.

Kymberly Fergusson

Monash University
August 2003

Contents

Chapter 1

Introduction

Programming is a difficult task, especially for students who do not understand the workings of a computer, let alone have sufficient background knowledge to link new programming concepts to while learning a new language. Novice programmers frequently encounter obstacles that impede progress and cause much frustration. These include new programming concepts, logic errors and syntax errors caused by unfamiliarity with the programming language.

The difficulties that novices experience illustrate the need for programming educators to research strategies, tools and languages to help the learning process. Extensive research has shown that there are many important aspects to consider when trying to aid beginner programmers: the psychology of how people learn; language design; error detection and recovery methods; error reporting techniques; and determining various types of errors made. Each of these aspects influences the rate at which students learn and the level of frustration they experience.

Many introductory programming courses teach students languages that were specifically designed by experts with little or no regard for the accessibility of the language, let alone the psychology of how novices learn new concepts. Beginner programmers often find programming an extremely challenging task because they are required to learn several new concepts at once. Concentrating on learning programming constructs, which are often completely different to prior knowledge, is difficult when trying to learn debugging techniques and to understand confusing errors at the same time. This is especially difficult when combined with a language which uses a terse unnatural syntax such as C [Kernighan and Ritchie, 1988]. A language with a more natural and familiar syntax may be more accessible to beginners, even if used to represent a translation of their program.

The two main types of errors in programming are syntax or language specific errors and semantic or logic errors. Syntax errors are often discovered by compilers, but semantic errors most often are not. Students frequently get bogged down in the language syntax while trying to analyse their code for semantic errors and understand misleading error messages. This

results in frustration and can lead to students giving up, regarding programming as an impossible achievement.

Typically, semantic errors are not indicated by the compiler, and do not produce error messages. In many cases the observed symptoms of the error do not indicate where or why the error occurred. Some semantic errors that beginners produce are extremely common, and may possibly be detected automatically. If there was a tool that could automatically identify, highlight and explain such errors, it may simplify the debugging cycle. This could result in the novice gaining a better understanding of the code and errors made. Many programming tools successfully report syntax problems, but do not deal with semantic errors. Providing further instruction and error detection for semantic problems should allow the novice to concentrate on learning unfamiliar programming concepts.

The AntiCompiler aims to implement a translation from a subset of C to a more natural representation, and, if feasible, to automatically detect some of the more common semantic errors that beginners make. Helping beginners overcome obstacles will free up tutor resources normally dedicated to those problems, and instead allow them to focus on the trickier conceptual problems of programming.

1.1 Outline of Review

In order to study novice programmers and their semantic errors, many aspects of learning and programming need to be studied. Teaching beginner programming requires an understanding of how people, and more specifically novices, learn programming. The psychology of learning is discussed in chapter 2

Novices often make certain types of mistakes repeatedly and in similar locations. Several categorisation schemes have been developed in order to analyse error creation patterns, some schemes lending themselves to be useful in automatic detection programs. Some empirical and observational studies conducted on both novice and experienced programmers, and existing categorisation schemes are analysed in chapter 3.

An overview of various current and past languages, tools and methods used in teaching novice programmers is provided in chapter 4. Methods, languages and tools that may help with error resolution are detailed more specifically as they are directly relevant to the development of the AntiCompiler. Common debugging methods used by novice programmers are an important area of study, to improve teaching methods in order to better facilitate the learning of programming and a new language.

The AntiCompiler combines useful recommendations from existing research, and is discussed in chapter 5. Further research into the various aspects of learning, and more specifically learning of a programming language for the first time is needed, as novices continue to find learning programming a daunting process. Some concluding remarks are provided in chapter 6.

Chapter 2

How Novices Learn

The field of learning theory, and its applications in Computer Science and Engineering is important as learning new concepts and dealing with unfamiliar technology are both involved in studying a new programming language. Learning requires a supportive environment as well as some prior knowledge is needed with which to connect new ideas. Otherwise the beginner either learns by rote, or encounters serious and sometimes insurmountable obstacles. When teaching any new concept, different types of learners need to be considered in order to deliver guidance that can help the majority of students.

Learning languages poses many problems, especially if there is little similarity to the beginner's own language. This is especially true when learning a programming language, as it is often paired with learning constructs that are completely new. Many programmers represent programs in their natural language as a method of finding a solution. Novices have trouble with this problem solving method as they often lack the knowledge or confidence to try various solutions. Instructors often find that rephrasing programming questions or programs using languages more familiar to the beginner helps the student to learn.

2.1 Requirements For Learning

To learn effectively, students require a safe physical environment, prior knowledge and enough continuing positive experiences to continue trying to solve problems and learn new concepts. McIver [2002, page 57], notes that if novices have had serious problems in the past, say with a computer crashing consistently, the fear that the computer will crash again will pose a serious barrier to learning new concepts. Safety is an important consideration, as it will block those students from trying out solutions, fearing their attempts will break the computer.

2.2 Linking New Concepts

To learn a new concept or idea, a person needs to link or map the new knowledge to existing concepts already in long term memory [Mayer, 1986]. This process is commonly called assimilation and is graphically represented in Figure 2.1. This diagram shows that when a new idea is received in short term memory, long term memory is queried to see if there is an existing concept with which to form a link. If this linking of new knowledge to existing knowledge cannot occur, the student may be able to learn by rote, but will not fully understand the concept or its relation to other concepts. Only if there is existing knowledge will meaningful learning occur, as opposed to rote learning.

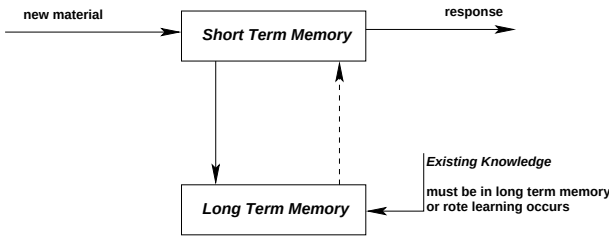


Figure 2.1: *The Linking of a New Concept.*

For example, in Computer Science, to learn the concept of array access, the student needs to map this information to their existing knowledge of loop constructs in order to understand how to step through each element of the array. If the knowledge of loops was not in long term memory, preventing assimilation, the student may learn how to step through an array by rote, but may find it impossible to extend this knowledge, for example to deal with multi-dimensional arrays that require nested loops.

To aid the process of learning a new concept, several techniques can be employed to help link the new knowledge with existing knowledge. Encouraging the novice to rephrase concepts in their own words, and providing some guidance or instruction to the student, are two such helpful techniques. However, both of these techniques may not help when there is conflict between the new concept and existing knowledge or beliefs (see section 2.2.3).

2.2.1 Rephrasing

One technique that has been helpful in teaching in all fields is encouraging students to rephrase the question, or a paragraph, in their own words. Elaboration often cements their understanding of the concepts involved, linking them more closely with existing related knowledge in long term memory [Mayer, 1986]. Rephrasing their knowledge of a concept often can highlight misconceptions, showing conflicts between the new and existing knowledge and allowing tutors to correct misunderstanding.

Once the technique of rephrasing has been learned by a student, it can be applied to learning any new concept, and is useful to check that their understanding is correct. Elaborating by including links to existing knowledge helps to retain the new concept in long term memory, and to build a more complete concept map. To help students learn this technique, instructors often rephrase concepts and questions, trying to find a closer link to existing knowledge. If the knowledge is not available to be linked to, this will fail and the student will often end up learning by rote.

2.2.2 Instruction

Students need some form of instruction and guidance in order to assimilate new information effectively. This is especially important when an impasse is reached. To overcome an obstacle, the concept may need to be rephrased by an instructor, or the student may be needed to explain their understanding so misconceptions can be corrected or missing knowledge supplied [Perkins, Hancock, Hobbs, Martin and Simmons, 1989; Hsi and Soloway, 1998]. Perkins et.al. [1989] suggest that the need for guidance could potentially indicate that the beginners lack confidence. This may be due to the lack of existing knowledge which would normally provide them with avenues for testing solutions. This can result in them giving up, or continually tinkering and thereby obtaining a solution more by chance than by reason.

Using the array and loop example, if the student did not know what loops were, and it was explained that they needed to look at each element in an array, they may write a program that looks at each element in turn, instead of using a repeated set of steps (a loop). This would result in a very long program that could not be used to generalise the concept to deal with different sized arrays or multi-dimensional arrays.

2.2.3 Conflicting Information

Information that conflicts with existing knowledge often gets disregarded or altered to fit existing conceptual knowledge. This process is called *cognitive dissonance* [Corsini, 1994], and can cause much difficulty in learning conflicting concepts. One example of this is that in computing, multiplication is often represented with an asterisk instead of a small '×'. Another difficulty arises when two concepts are only subtly different, encouraging the learner to believe they are the same. Both of these situations can cause novice programmers distress and lead to a feeling of helplessness, or of not being able to deal with the way the computer or language works.

In C and many other programming languages, assignment of a value to a variable is done using the operator '='. This can be quite confusing as 'equals' is most often used as a comparison in natural language. Comparisons in C are done with the operator '==', which may compound the confusion of novices. The '=' and '==' operators conflict with the way natural language is typically used to represent the concepts of assignment and comparison.

A further example, that highlights the way the C programming language conflicts with the existing knowledge of the novice, is where counting is normally done from `zero`. The third item in an array actually has a position of `two`. Even though the counting mechanism is the same, counting in C conflicts with the novices' existing knowledge, where they would typically expect the first element to be at a position of `one`. This requires a separation of concepts in memory according to the context in which they apply, which makes them more difficult to assimilate.

2.3 Types of Learners

Different types of learners need different intervention and techniques to help them learn. There are many different schemes classifying types of learners. According to Perkins, Hancock, Hobbs, Martin and Simmons [1989] there are two main types of learners: Stoppers and Movers. Both types of learners benefit from some kind of instruction and guidance, helping them get past obstacles. This scheme is used to describe the novice programmers as they commonly exhibit two different behavioural patterns when faced with a problem.

Stoppers are the more common of the two types. They generally 'give up' when they reach an impasse, and stop looking for ways to solve the problem.

Movers are the opposite of Stoppers in that they continually try new solutions, often introducing more errors and repeating solutions, never thinking clearly about which solution seemed to work better.

Perkins et.al. [1989] discovered that with some instruction and guidance, both groups often were encouraged to find a solution. Based on their research they proposed that the students lacked a sufficient 'mental model' of the language required to break the problem down. Pane and Myers [2000] found supporting evidence suggesting that a representation of a solution to a programming problem in a programming language often did not match the 'mental model' or natural language representation of that solution.

Representing a student's code in a more familiar language provides a more familiar setting for a program, hopefully facilitating more comprehensive understanding of the programming language. This would illustrate how to represent programming concepts in a precise but natural way, and may help students to convert that representation to the programming language. Movers are less likely just to tinker with some parts of the program and instead may see more clearly how concepts are represented in the more natural language, and how they map to the programming representation. Stoppers may find that this level of instruction makes trying a solution less daunting, and may be encouraged to continue past an impasse.

2.4 Learning Languages

Many empirical studies have been done to compare whether there are any cognitive differences between understanding programming concepts in a natural language versus a more obscure programming language.

Dyck and Mayer [1985] compared the understanding of problems and representation of solutions by relatively experienced BASIC programmers to the understanding of natural language representations of the problems by non-programmers. They observed that the pattern of response times was the same for both English and BASIC statements. This indicates that many of the problems that novice programmers have may be due to the unfamiliarity and complexity of the programming language they are using.

This evidence has been supported by a large number of studies, one of which [Eisenstadt and Lewis, 1992] compared two languages specifically designed for novices, LOGO and SOLO. Eisenstadt and Lewis found commonalities in the errors and error rates within both languages. Other studies have shown that the closer a language resembles the programmer's natural language and existing knowledge, the easier it is to learn and the fewer errors are made [Bonar and Soloway, 1985; Mayer, 1987; Soloway, Bonar and Ehrlich, 1989; Murnane, 1993; Bruckman and Edwards, 1999; McIver, 2000; Pane, 2000; Pane, Ratanamahatana and Myers, 2001]. Some of these studies also found that when novices reached an impasse they frequently used a natural language representation to solve the problem[Bonar and Soloway, 1985; Bruckman and Edwards, 1999].

2.5 Methods to Find a Solution

As was mentioned in the previous section, many novices will represent solutions in their own natural language more easily than in a programming language and will fall back on those representations when their programming knowledge is inadequate [Bonar and Soloway, 1985].

Experienced programmers are also often more comfortable representing problems and solutions in a natural language than in their programming language, as it often highlights semantic errors. Three categories of solution finding behaviour using natural language representations have been observed by Brooks [1977]:

- General statements of problems and solutions, in non-programming terms and constructs and language
- Statements noting similarities to previous problems and solutions
- Solutions to problems written in a meta-language

Those programmers who reliably reached a correct solution, following the methods outlined above, were typically experienced. Novices find problem solving to be much harder,

as they do not have the required knowledge to instinctively try a possible solution [Gugerty and Olson, 1986; McIver, 2002]. This is a problem especially where languages force the programmer to represent their solutions unnaturally, using complex constructs and confusing syntax [Pane, 2000].

Wiedenbeck [1986] noted that experienced programmers look for 'beacons', sections of code that indicate the presence of a particular structure or operation. Novices, on the other hand, tend to search through a program from the start, not looking for patterns or typical programming constructs, as they do not have the knowledge required. Often novices will make arbitrary changes to code to fix errors [Perkins et al., 1989]. Perkins and Martin [1986] suggest that strategic prompting in this situation will often lead to the novices applying more structured error finding techniques, including self-prompting and stepping through their code. Representing code in a more natural language may help novices recognise common patterns and also encourage them to step through their code rather than randomly guess where errors occur. This more structured problem solving behaviour can then be applied when debugging code.

2.6 Summary

Learning new concepts is a multi-faceted challenge. Novices must feel secure in their environment and have some existing knowledge that new knowledge can be linked to. Learning programming languages uses the same cognitive processes as normal learning, thus techniques that work for natural language problem solving should work for learning programming concepts and language syntax. Rephrasing concepts and receiving guidance may encourage those that are stuck to find new ways to link information with knowledge stored in long-term memory, rather than learning by rote or giving up. Experienced programmers rely on problem solving techniques that novices find difficult. Inexperienced programmers must deal with new concepts and syntax, making the problem solving process much more laborious.

The next chapter will discuss mistakes made by novices, by contrasting findings of various studies, and comparing these with studies of mistakes made by experienced programmers. Additionally, some existing categorisation methods for errors will be discussed and compared.

Chapter 3

Mistakes Made By Novices

Novices make a great number of errors, however, errors are the building blocks of learning. For example, an error that causes a program to loop indefinitely (an infinite loop), helps the novice learn how loops work. Gugerty and Olson [1986] observed that experienced programmers discovered existing errors in programs much faster and more reliably than novices, and did not introduce as many errors as did novices in the course of the debugging task set.

There are two main types of errors, language specific errors are known as syntax errors, and typically cause compilation to fail, as the error is detected by the compiler. Semantic errors are usually found by the user when the program runs and has unexpected outcomes. These are also known as logic errors, and are often not detectable by the compiler.

Many semantic mistakes that novices make are very common, and are typically similar across language paradigms. Analysis of various error categorisation schemes is required to allow the discussion of common error types.

3.1 Categorisation of Errors

Several schemes for the categorisation of errors have been developed, the most common being the simple division between syntactic or language specific errors and semantic or logic errors [Endres, 1975; McIver, 2000]. Both of these areas have been sub-divided further, but as most compilers find and report most syntax errors, semantic errors are more interesting and difficult to detect.

Semantic errors are typically discovered when the program is run, often resulting in unexpected behaviour that is confusing to the novice programmer. These errors are more difficult to detect automatically as they are typically caused by incorrect assumptions about how to structure the flow of control or beliefs that the compiler 'knows' what the programmer wants. In order to detect control flow errors, all possible paths of execution need to be

tested. Normal compilers do not perform such analyses. Detecting some other errors is nearly impossible without knowing what the novice was trying to do. Several error categorisation schemes have been developed for common errors made by novices.

Pennington [1987] provided a categorisation of errors that spanned both semantic and syntactic mistakes:

Operations - reading/assigning.

Control Flow - order of actions.

Data Flow - transformations of data objects throughout the program .

State - the state of the rest of the program when a specific place is reached.

Function - main goals and subgoals of a program and the role of procedures.

Pennington [1987] shows that the majority of the errors made were in the categories **State** and **Function**, the next most common errors being **Data Flow**, and the least common categories **Control Flow** and **Operations**.

3.1.1 Syntactic Errors

Endres [1975] found that only 15% of the errors made by experienced programmers when programming system modules for the DOS/VS operating system released in 1973, were syntax errors. The majority of the errors made were logic or understanding errors (semantic errors) and were detected during internal testing.

This finding was supported by an empirical study looking at specific types of errors when novices were using LOGO and SOLO [Eisenstadt and Lewis, 1992]. It was found that 43.3% of the errors made were syntax errors and could be pre-empted by the development environment. This study also showed that novices do make more syntax errors than experienced programmers, however, this left the semantic errors to be detected by hand surrounded by unfamiliar and confusing concepts and language syntax.

In both cases, semantic errors comprise the majority of 'bugs' in programs, which is also supported by Murnane [1993]. As they are rarely detected by a compiler, these are the errors that can cause novices the most grief.

3.1.2 Semantic Errors

Categorisation of semantic errors is important as it allows teachers to identify the most common conceptual errors made by novices. Detection of semantic errors by both novices and teachers is also made easier with the knowledge of common causes for groups of semantic errors.

Spohrer, Soloway and Pope [1985] used programming goals as a guide to developing a classification scheme for semantic errors. These goals represent a set of steps needed to acheive a certain result, such as those steps involved in traversing an array. The classification scheme contains two types of semantic errors:

Goal Drop Out is where a goal is forgotten when a merging of goals occurs.

Goal Fragmentation is where a goal is broken into sub-goals and these are evaluated out of order.

This classification scheme is useful as it illustrates the cognitive processes behind the errors, why the novices made them. Both types of error result in an incorrect and often more complex program structure, causing steps to be done out of order, or be forgotten.

An alternative and more focused classification scheme (discussed below) was developed by Pea [1986], again providing insights into why the novices made the errors. This shows more clearly how novices use their existing knowledge of natural language constructs and try to apply it to a programming problem, which creates a number of problems.

Parallelism
In natural language, a loop condition is continually tested at every step taken through the repeating set of actions. This means it is being done in parallel with the steps inside the loop. Another form of parallelism is where multiple steps are completed at the same time - again, this is often how repeated actions are represented in natural language. Consider the following example:

```
while (watching TV)
{
     eat dinner;
     drink coffee;
     read paper;
}
```

The person completing these actions would normally be doing the three steps inside the loop at the same time (in parallel), and if at any time the `TV` is turned off, they would stop all other tasks, no matter where they are in the loop. This is an example of a very common mistake that novices make.

Intentionality This is where the novice attributes forward looking-capabilities to the computer. For example, if there were a condition that changed the state of `watching TV` further along in the program, outside the loop, the novice is most likely to assume that the program could look ahead and check that condition while inside the loop.

Egocentrism The final category attributes intelligence to the computer, where the novice assumes that the computer knew what they meant, which is not necessarily the same

as what they programmed. An example is where the novice had meant to initialise a variable, but believed that the computer could correctly initialise the variable, possibly due to them giving it a descriptive name. Sometimes this may occur with procedures - where the novice calls a procedure `SUM` and thinks the computer can work out what to do from the name alone.

Pea's scheme is useful as it illustrates what the novice may have been thinking and their intentions when they caused the error [Pea, 1986]. Other categorisation techniques which highlight the misapplication of existing natural language knowledge in a similar way have been discussed in various papers. Du Boulay [1989] proposes three different categories:

Misapplication of analogy - an example would be believing a variable is like a box, and is able to store multiple items of either the same or different types.

Over-generalisation - what works for one thing, works for another, an example: not requiring semi-colons at the end of comments in C means all statements do not require them.

Inexpert handling of complexity in general, and interaction in particular - subsections are incorrectly interleaved resulting in the wrong ordering of steps in the program.

These various categorisation schemes serve as an illustration of the misconceptions novices and even experienced programmers have of programming and language constructs. Many papers proposing categorisation techniques have highlighted the ambiguity in using natural language constructs for a highly specific and ordered task such as programming. This will be important to consider when designing a natural translation language to help novices understand the more cryptic C programming language.

However, these classification schemes do not lend themselves to aiding the automatic detection of typical semantic errors in programs. It is therefore necessary to consider more specific examples of common semantic errors.

3.2 Common Mistakes - Experimental Findings

Many studies of various programming languages have concentrated on problem areas for novices, focusing on common mistakes. Several important programming constructs have been found to be notoriously difficult for novices to understand and represent correctly. Many of these constructs are ambiguous in natural language, resulting in novices misapplying their existing knowledge while using intuition or guesswork to search for the correct solution.

Loops

Natural languages have a completely different interpretation of repeating

tasks (loops) than strictly sequential programming languages such as C [Spohrer, Soloway and Pope, 1985; Kernighan and Ritchie, 1988]. This was also mentioned in the analysis of the categorisation scheme developed by Pea [1986] above as parallelism. Many other studies have verified these findings, noting that novices typically assume continuous testing of loop conditions [Bonar and Soloway, 1985; Du Boulay, 1989; Pane, 2000].

Soloway, Bonar and Ehrlich completed an empirical study of two types of looping strategies: `READ then PROCESS` and `PROCESS then READ`. The first construct employwed a 'loop ... leave ... again' strategy, which is quite different to the C `while` construct. They found that the first case was correctly implemented by novice, intermediate and advanced programming students on average nearly twice as often as the second method. The first method is a more natural representation of the looping construct found in PASCAL [Wirth and Jensen, 1975], and the second representation, the actual looping construct of that language.

Conditions

The study by McQuire and Eastman [1998] concentrated on database queries using negation, and found there were a multitude of misconceptions on the part of students about precedence and scope of the `NOT` construct. They found that a more complex statement containing disjunctions, conjunctions and prepositions, resulted in more ambiguity and less understanding of the statement by the students.

The `AND` construct is equally ambiguous in natural language, mathematics and programming languages. For example, a search for *cats AND dogs* on the Internet will yield resources that contain references to **both** cats and dogs, instead of returning references to cats as well as references to dogs. Several studies have found that the `AND` conditional causes many problems [Bonar and Soloway, 1985; Pane, 2000].

Assignment

A more specific problem is found in languages that use `=` for assignment of a value to a variable and `==` as a comparison checking for equality [Du Boulay, 1989]. Sometimes this error may simply be a typing mistake, but it is often not detected by the compiler, and is therefore counted as a semantic error.

Using Incorrectly Initialised Variables

Many novices re-initialise variables because the computer may have 'forgotten' the value, or assume that the computer knows what values are needed to initialise variables. As a result the novice initialises them incorrectly or multiple times [Joni and Soloway, 1986; Pea, 1986]. This can result in an illegal memory address causing the program to crash or produce the wrong output.

Wrong Ordering or Merged Goals

An extremely common error made by novices is the programming of steps in the wrong order [Spohrer et al., 1985; Du Boulay, 1989]. For example putting a loop

counter outside of a loop, which would mean the loop would never finish. This often occurs when programmers 'merge goals', creating a combined, complicated and often wrong process of stepping through the operations in a program [Spohrer et al., 1985; Joni and Soloway, 1986]. An incorrectly ordered execution path can result in the program crashing unexpectedly or not terminating at all (infinite loop). These are quite difficult to debug as the flow of control in the program may not be the same each time the program is run.

3.3 Summary

There has been much research into various types of common errors that novices make, with several different categorisation schemes used to describe the mistakes. Studies of common errors specific to C seem to be rare, as none could be found at this time. However, as semantic errors are basically language independent, these classification schemes and generic common error findings are still useful to consider. Most of the categorisation methods illustrate why novices made the various types of semantic errors, but these schemes would be difficult to use as methods to automatically detect errors. It is more useful for an automated tool to deal with more specific common error types as outlined above, rather than groups of similar errors.

The next chapter will describe some of the existing tools, techniques and languages that have been developed to help novice programmers, and various educational methods to aid the understanding of difficult constructs.

Chapter 4

Tools and Techniques

In order to create a tool to help novices learn a difficult language (C) and unfamiliar concepts, it is necessary to analyse existing tools and techniques created for this purpose. This chapter will discuss both desirable and undesirable features for a first programming language, and some of the C programming language's shortcomings. Following that, an analysis of existing tools and teaching methods will show how they can be used to help error resolution in introductory programming.

4.1 Languages

Languages that have obscure and terse syntaxes are much harder to learn than those which have more descriptive and natural representations. This has been illustrated by many studies [Murnane, 1993; McIver and Conway, 1999; Pane et al., 2001; Bonar and Soloway, 1985]. Even though natural language errors are made, they are usually easier to recover from than errors made in a programming language [Bruckman and Edwards, 1999]. Despite the advantages of descriptive and natural languages, many current programming languages use obscure and terse syntaxes.

4.1.1 Current Languages

Most programming languages were developed by experts, focusing on speed, or properties specific to their domain, hardly ever considering the beginner programmer [Murnane, 1993; Pane, 2000]. Most of the languages used in first year programming courses are popular languages used in industry: C, Scheme, Ada, C++, and more recently Java [Levy, 1995; Conway, 1994]. These languages were not designed to teach programming concepts to new students, but to allow experienced programmers to implement these concepts efficiently.

4.1.2 Good Design

A good language design is necessary to help novices link their existing natural language knowledge to the more specific requirements of a programming language. A first language should feel familiar to the students and not contradict their existing knowledge. The syntax of a beginner language should be simple and not overly terse.

Conway [1994] developed some criteria for selecting a beginner programming language:

- The syntax should be concise and semantics straightforward.
- The language should be consistent with the natural language of the novice and not confuse beginners.
- It should be powerful and flexible, to encourage experimentation and allow for multiple solutions.
- It should be robust and graceful in failure, having descriptive error management.
- It should assist good programming styles and allow fast development.

These goals, though laudable, are typically not supported in languages available today. However, many new languages have been specifically developed to aid novice programmers, GRAIL[McIver, 2002], SOLO[Eisenstadt and Lewis, 1992], LOGO [Friendly, 1992], BLUE and BLUE J [Kölling and Rosenberg, 1996] and MOOSE [Bruckman and Edwards, 1999] are some recent examples. These attempt to simplify the language and semantics to allow for novices to understand programming concepts more easily.

MOOSE was designed to be as close to the users' natural language as possible and was used inside a MUD (Multi User Dungeon) environment by children. Bruckman and Edwards [1999] concluded that a language that was close to the natural language was highly recommended, as error recovery by novices was a lot quicker than when using a more obscure language and syntax.

GRAIL was designed with a general goals of facilitating learning and maximising readability [McIver, 2002]. More specifically:

- Start where the novice is - use language and concepts they understand
- Maximise prior knowledge of the user
- Make features self-explanatory
- Avoid unexpected results - minimise default, unexplained behaviour
- Use different syntax to identify different semantics
- Use a readable and consistent syntax

- Be careful with Input/Output
- Provide better feedback - compilation and runtime
- Prevent errors

These goals are eminently more achievable when creating a language from scratch and were also supported by Hsi et.al. [1998] and Pane et.al. [2000].

4.1.3 Bad Design

Many languages are designed with no thought to usability and learnability, although a language may be very difficult to learn but highly usable. Badly designed programming languages typically contain the following aspects [Moylan, 1992; Mody, 1993; Murnane, 1993; Pane, 2000]:

- Terse and obscure syntax.
- Contradictory syntax or constructs.
- Bad or no error reporting.
- Unnatural representation of concepts.
- Pointers.
- Inconsistencies or serious errors in the compiler.
- Compiler does not enforce good programming practice.

These negative aspects leave novices floundering and they are often unable to concentrate on concepts as they are attempting to cope with a badly designed, difficult to learn language. As the languages typically used for professional programming are taught as a first language, tools have been developed to lessen the impact of the negative aspects, especially for the debugging cycle.

4.2 Tools

Programming in a text only environment is challenging to novices, especially as current operating systems typically have a Graphical User Interface (GUI). GUI programming environments have always been popular, often incorporating syntax highlighting and easy access to help about various functions. Some developers have taken this further and provided tools that have built-in syntax checkers (where the program does not require compilation in

order to find syntax errors) [Eisenstadt and Lewis, 1992]. Du Boulay and Matthew [1984] built compilers that check for both syntactic and semantic errors, reporting problems with verbose comments. KuMir [Brusilovsky, Kouchnirenko, Miller and Tomek, 1994], a Russian program that uses marginal notes to debug programs without I/O, was developed to reduce wasted time by teachers and students. This tool has been successfully used in Russian schools to aid the teaching of programming to 12-16 year old students. The Moscow State University has also used KuMir in the Department of Mathematics to help students in their first term to solve numerous programming, calculus and algebra problems quickly.

Compilers have gradually improved to report all syntax errors as these types of errors prevent compilation, although the quality of reporting errors can be improved further. Compilers are still lacking the capability to thoroughly analyse programs for semantic errors. Debugging tools have been created to perform static verifications of code in order to find semantic errors. One example is `lint` created in the early 1970's to check for possible run-time errors without executing the code [Santo Orcero, 2000; Johnson, 1978]. `LCLint` is an improved version which uses annotations made by the user throughout the code to analyse for a larger number of semantic errors [Evans, 1996; Evans, 2003]. Although these tools were targeted at experienced programmers in order to reduce the amount of time and effort expended while debugging semantic errors, the output is verbose and phrased simply. These tools still may pose some additional difficulty for novices as they are required to annotate their programs with comments to aid automatic detection of errors. This may confuse students more, or it may enable them to understand program goals more clearly. Unfortunately, studies of novice programmers using these tools seem to be rare as none have been found at this stage.

An Extended Static Checking (ESC) system was developed for Modula-3, which analyses the semantics of conditional statements, loops, procedure and method calls and exceptions without running the code [Detlefs, Leino, Nelsn and Saxe, 1998]. This tool was successful in discovering a number of common errors, reporting specific details about the errors, but not in an easily accessible form for novices.

Brusilovsky [1993] developed an intelligent tutoring system which integrated a visual tool to step through programs, an interactive development platform and an intelligent programming environment. He commented that this approach can be used for other languages, as it is simply built around the language. In addition to a program development and debugging environment, the intelligent tutor tracked students' progress, basing suggestions for further lessons and programming tasks on their performance.

Many of the tools developed to aid the programming process rely on well structured programs, commonly used looping heuristics and sometimes require additional annotations in the code to allow the tool to automatically detect errors. This requires a strict adherence to a good and consistent programming style, which is often neglected by novice programmers. When teaching beginners, it is important to emphasise the benefits of using a clear and consistent style, even if not using additional tools to aid debugging.

4.3 Teaching Methods

There are many teaching methods specifically designed for novices. Brusilovsky [1993] comments that to support the teaching of a programming language the following elements are important:

- Program design and coding (development environment and support).
- Pre-stored examples as templates, and previous solutions to build new programs.
- Program debugging and investigation.
- Online help system for querying language use.

Some methods for teaching a language have focused on:

- Cut down languages (mini-languages)
- Languages designed for beginners
- Sub-Languages (extend or limit the language to increase learnability)
- Incremental development (learning a subset fully before moving on)

Reduced languages can address many shortcomings in complex languages [Brusilovsky, Calabrese, Hvorecky, Kouchnirenko and Miller, 1997]. Safe-C, developed by Dix and Lien [1993] for an introductory programming course, provided a 'safer' subset of the C language. Safe-C simplified the syntax for some of the more difficult aspects, and prohibited many obscure and easily misused language constructs. A static checker was incorporated to enhance the automatic detection of errors, but the syntax of C was not majorly altered. Unfortunately there is no study of whether this helped novice programmers and the language was not used for teaching purposes.

The choice of reliable and comprehensive tools to support the novices' language development is of high importance. The main aim of a beginner programming course is not to learn a difficult language, but to learn the concepts of programming [Levy, 1995].

4.4 Error Correction and Prevention

It is important to help programmers find and prevent errors as the majority of the time spent programming a solution to a problem is spent debugging. Using a natural and verbose language helps to reduce the number of errors made as programmers can understand their program without having to wade through confusing syntax [Bruckman and Edwards, 1999]. It is therefore important for a programming language designed to be used by novices to be consistent with natural language constructs.

Obviously accurate and descriptive error reporting techniques are required by beginner programmers, as these would cut down the amount of time and frustration spent on debugging programs. Currently most compilers and run-time environments do not check for semantic errors. Tools such as LCLint, described above, can be used to generate more comprehensive and verbose error messages for errors not commonly found by compilers. However, these tools may report spurious errors in sections of code that contain no errors, if those sections do not follow common heuristics [Evans and Larochelle, 2002; Larochelle and Evans, 2001].

Visual environments that can highlight errors as the novice types, and even auto-correct some errors, can dramatically reduce the number of errors made [Brusilovsky et al., 1994; Bruckman and Edwards, 1999]. However, this may not aid the novice in learning about their errors, especially with auto-correction. The tutor must be aware of common errors and explain them to novices before they get frustrated and give up. This allows the novice to look out for those errors, and to develop strategies for dealing with them.

4.5 Summary

Current languages are typically obscure and hard to learn, not made any easier by the varied tool-sets that support them. Good teaching languages should to be based on the beginner's existing knowledge of natural language. First programming languages should be simple and consistent. The tools used for program development should have excellent error detection and reporting capabilities, commenting in simple, verbose language about both syntax and semantic errors made. Some of the tools that are currently available are of an excellent standard, but they do not support the most commonly taught and used languages. The languages that are used are typically poorly supported by teaching tools, and have many negative aspects which hinder the learning of the novice. Although restricted subsets of more difficult languages simplify the learning process, they do not provide for more solid linking between existing knowledge of natural language and the implementation of programming constructs in a terse language. Many tools currently available for debugging and program analysis are intended for use by experienced programmers, neglecting novices.

The next chapter will briefly discuss how the AntiCompiler will attempt to remedy some of the problems outlined above, followed by some concluding remarks.

Chapter 5

Why The AntiCompiler?

The AntiCompiler aims to take C programs produced by novice programmers and translate them into a natural language representation that may be more verbose and clearer for beginners. One of the most important factors influencing the ability of a novice to learn a new language is the requirement that the language be as close to natural language as possible. It is therefore fairly surprising that one of the most commonly taught languages is C, with obscure and terse syntax and poor error reporting. C compilers have improved over the years and can detect and warn of many syntax errors, albeit those warnings may be confusing to novices at times.

As C is a commonly taught first programming language, it would be useful to be able to translate C programs into a more natural representation, which may allow the novice to look for logic errors much more easily. It may also help the novice learn programming concepts more thoroughly and allow a better mapping between the solution they develop in natural language and the program they write to implement that solution.

The AntiCompiler will also attempt to highlight some common semantic errors that are not detected by the GNU C compiler `gcc` if found feasible. A semantic error checker would be a boon for many programmers, novices and advanced programmers alike, as it may cut down the time spent searching for steps that are done out of order, or uninitialised or wrongly initialised variables, the annoying logic errors that are always so hard to find, especially for the beginner. Some semantic errors are impossible to automatically detect with certainty as they are not computable, such as whether a loop terminates. However, many novice errors in loop termination follow patterns, like incrementing loop counters outside the loop. These may be more amenable to automatic detection.

Although there are existing tools that detect some semantic errors, they do not translate the program into a more natural representation. Thus they require the user to try to understand where the error is amongst confusing and unfamiliar syntax.

The reports generated by the AntiCompiler may be displayed as HTML or plain text, with options to create a separate file for the translation or intersperse the translation throughout

the C code. The indication of errors will be optional and may be shown in either the C code or the natural language version of the program, provided the discovery of semantic errors is found to be feasible. This translation may provide the instruction needed by both of the two types of learners discussed in section 2.3. The error reports may help both Stoppers and Movers to work through an impasse created by an error.

The AntiCompiler may help tutors in practical classes concentrate on helping the students learn programming constructs rather than spend their time helping the students debug their programs, thus encouraging a less stressful learning environment.

It remains to be seen if a reliable semantic error detection method could be designed based on a natural language version of a C program. That aside, simply translating a confusing C program to a more natural representation may enable the novice to find their errors quicker and with less frustration.

Chapter 6

Conclusion

A huge amount of research has been conducted, investigating what types of errors novices make, the frequency of such errors, as well as aspects of accessible language design. However, more research is needed into how much difficult languages impede the learning process, especially as the main teaching languages are typically chosen from those most popular in industry, rather than those designed to aid the novice in learning the concepts of programming without having to fight with an obscure and frustrating language. More research is also needed, specifically in the area of novice programming in C, to discover ways of helping the novice learn the confusing syntax and unfamiliar programming constructs easily.

C is commonly taught as an introductory programming language as it is one of the more popular languages currently used in the programming industry. This language has the ability to represent both relatively high level and low level concepts, many of which are not representable in an object-oriented language such as Java. An imperative language is closer to novice programmers natural language knowledge of procedures, whereas object-orientation would need to be taught in addition to programming concepts if a language such as Java were to be taught as a beginner language.

In the School of Computer Science and Software Engineering at Monash University, many subjects throughout the Computer Science and Software Engineering degrees rely on both high and low level C knowledge, thus it is taught as the first programming language.

Common errors made by novices have been categorised in various ways, but not many of these classification schemes assist the automatic detection of semantic errors. Specialised tools that provide detection of some common semantic errors aid the development of programs at all levels, by reducing the amount of time and stress spent in debugging. This is especially important for novice programmers, who typically are confused by the language syntax and basic programming constructs.

Novices require an integrated set of tools that provide a 'safe' environment to learn programming. The ability to describe a program written in a terse and obscure language in a language that is similar to natural language would foster more complete and correct linking

between existing knowledge and new ideas. Tools that help the novice detect and learn about common errors will enable them to develop a better understanding of programming techniques and procedures. Ideally the language used to introduce novices to programming should be as close to natural language as possible, but must not contain contradictions to existing knowledge.

It may in fact not be necessary or even possible to change which language is taught to novices, providing a good set of tools support the difficult language. These should provide excellent error reporting, enforce good program design, and provide a supportive and interactive development environment. The achievability of these goals and the impact they may make on the learning process are important areas to research further.

Maybe novice programmers would learn more effectively, and be less frightened if they felt comfortable enough with the language, the learning environment and the supporting tools.

References

Bonar, J. and Soloway, E. [1985]. Preprogramming knowledge: A major source of misconception in novice programmers, *Human Computer Interaction* **1**(2): 133–161.

Brooks, R. [1977]. Towards a theory of the cognitive processes in computer programming, *Man-Machine Studies* **9**: 737–751.

Bruckman, A. and Edwards, E. [1999]. Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language, *CHI'99 Papers*.

Brusilovsky, P. [1993]. Towards an intelligent environment for learning introductory programming, *in* E. Lemut, B. Du Boulay and G. Dettori (eds), *Cognitive Models and Intelligent Environments for Learning Programming*, Springer-Verlag, pp. 114–124.

Brusilovsky, P., Calabrese, E., Hvorecky, E., Kouchnirenko, A. and Miller, P. [1997]. Mini-languages: A way to learn programming principles, *Education and Information Technologies* **2**(1): 65–83.

Brusilovsky, P., Kouchnirenko, A., Miller, P. and Tomek, I. [1994]. Teaching programming to novices: a review of approaches and tools, *in* T. Ottman and I. Tomek (eds), *ED-MEDIA'94 - World Conference on Educational Multimedia and Hypermedia*, pp. 103–110.

Conway, D. [1994]. Criteria and considerations in the selection of a first programming language, Technical Report no 93/192, Monash University.

Corsini, R. [1994]. *Encyclopedia of Psychology*, Wiley.

Detlefs, D., Leino, K., Nelsn, G. and Saxe, J. [1998]. Extended static checking, *Technical report*, COMPAQ Research Center, California.

Dix, T. and Lien, T. [1993]. Safe-C for introductory undergraduate programming, *Proceedings of the sixteenth Australian computer science conference*, pp. 371–378.

Du Boulay, B. [1989]. Some difficulties of learning to program, *in* E. Soloway and J. Spohrer (eds), *Studying the Novice Programmer*, Lawrence Erlbaum Associates, pp. 283–299.

Du Boulay, B. and Matthew, I. [1984]. Fatal error in pass zero: how not to confuse the novices, *Behaviour and Information Technology* **3**(2): 109–118.

Dyck, J. and Mayer, R. [1985]. BASIC versus natural language: Is there one underlying comprehension process?, *CHI'85 Proceedings, Comference on Human Factors in Computing Systems.*

Eisenstadt, M. and Lewis, M. [1992]. Errors in an interactive programming environment: Causes and cures, *in* M. Eisenstadt, M. Keane and T. Rajan (eds), *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence*, Lawrence Erlbaum Associates, chapter 5.

Endres, A. [1975]. An analysis of errors and their causes in system programs, *IEEE Transactions on Software Engineering* **SE-1**(2): 140–149.

Evans, D. [1996]. Static detection of dynamic memory errors, *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96).*

Evans, D. [2003]. *Splint Manual*, 3.1.1-1 edn, www.splint.org/manual/manual.pdf.

Evans, D. and Larochelle, D. [2002]. Improving security using extensible lightweight static analysis, *IEEE Software* **19**(1): 42–51.

Friendly, M. [1992]. *Advanced LOGO: A Language for Learning*, Prentice-Hall.

Gugerty, L. and Olson, G. [1986]. Debugging by skilled and novice programmers, *CHI'86*, pp. 171–174.

Hsi, S. and Soloway, E. [1998]. Learner centered design, *SigCHI Bulletin* **30**(4): 53–55.

Johnson, S. [1978]. Lint, a C program checker, *Technical Report 65*, Bell Laboratories.

Joni, S. and Soloway, E. [1986]. But my program runs! Discourse rules for novice programmers, *Educational Computing Research* **2**(1): 95–128.

Kernighan, B. and Ritchie, D. [1988]. *The C Programming Language, 2nd Ed.*, Prentice Hall.

Kölling, M. and Rosenberg, J. [1996]. Blue - a language for teaching object-oriented programming, *27th SIGCSE Technical Symposium on Computer Science Education*, ACM, pp. 190–194.

Larochelle, D. and Evans, D. [2001]. Statically detecting likely buffer overflow vulnerabilities, USENIX Security Symposium.

Levy, S. [1995]. Computer language usage in CS1: Survey results, *SIGCSE Bulletin* **27**(3): 21–26.

Mayer, R. [1986]. *The Psychology of How Novices Learn Computer Programming*, Baywood Publishing Co. Inc., pp. 129–159.

Mayer, R. [1987]. Cognitive aspects of learning and using a programming language, *in* J. Carroll (ed.), *Interfacing Thought*, MIT Press, pp. 61–79.

McIver, L. [2000]. The effects of programming language on error rates of novice programmers, *12th Annual Meeting of the Psychology of Programming Interest Group.*

McIver, L. [2002]. *Syntactic and Semantic Issues in Introductory Programming Education*, PhD thesis, Computer Science and Software Engineering.

McIver, L. and Conway, D. [1999]. Grail: A zero'th programming language, *in* G. Cummings, T. Okamoto and L. Gomex (eds), *7th International Conference on Computers in Education ICCE'99*, Vol. 2, IOS Press, pp. 43–50.

McQuire, A. and Eastman, C. [1998]. The ambiguity of negation in natural language queries to information retrieval systems, *Journal of the American Society for Information Science* **49**(8): 686–692.

Mody, R. [1993]. C in education and software engineering, *SIGSCE Bulletin* **24**(3): 45–56.

Moylan, P. [1992]. The case against C, Technical Report EE9240, Centre for Industrial Control Science, Department of Electrical and Computer Engineering, The University of Newcastle, NSW 2308 Australia.

Murnane, J. [1993]. The psychology of computer languages for introductory programming courses, *New Ideas in Psychology* **11**(2): 213–228.

Pane, J. Myers, B. [2000]. The influence of the psychology of programming on a language design: Project status report, 12th Annual Meeting of the Psychology of Programming Interest Group. PPIG 2000.

Pane, J., Ratanamahatana, C. and Myers, B. [2001]. Studying the language and structure in non-programmers' solutions to programming problems, *International Journal of Human-Computer Studies* **54**: 237–264.

Pea, R. [1986]. Language-independent conceptual "bugs" in novice programming, *Educational Computing Research* **2**(1): 25–36.

Pennington, N. [1987]. *Comprehension Strategies in Programming*, Ablex Publishing Corporation, New Jersey, pp. 100–112.

Perkins, D., Hancock, C., Hobbs, R., Martin, F. and Simmons, R. [1989]. Conditions of learning in novice programmers, *in* E. Soloway and J. Spohrer (eds), *Studying the Novice Programmer*, Lawrence Erlbaum Associates.

Perkins, D. and Martin, F. [1986]. Fragile knowledge and neglected strategies in novice programmers, *in* E. Soloway and S. Sitharama Iyengar (eds), *Empirical Studies of Programmers*, Ablex Publishing Corporation, New Jersey, pp. 213–229.

Santo Orcero, D. [2000]. The code analyser LCLint, *Linux Journal* **73**.

Soloway, E., Bonar, J. and Ehrlich, K. [1989]. Cognitive strategies and looping constructs: An empirical study, *in* E. Soloway and J. Spohrer (eds), *Studying the Novice Programmer*, Lawrence Erlbaum Associates, pp. 191–207.

Spohrer, J., Soloway, E. and Pope, E. [1985]. Where the bugs are, *CHI'85 Conference on Human Factors in Computing Systems*, pp. 261–279.

Wiedenbeck, S. [1986]. Processes in computer program comprehension, *in* E. Soloway and S. Sitharama Iyengar (eds), *Empirical Studies of Programmers*, Ablex Publishing Corporation, New Jersey, pp. 48–57.

Wirth, N. and Jensen, K. [1975]. *Pascal User Manual and Report*, Springer-Verlag.