

School of Computer Science and Software Engineering
Monash University

Bachelor of Computer Science Honours (4300)
Clayton Campus

Literature Review 2002

**AntiCompiler: an Educational Tool for First Year
Programming**

by

Kymerly Fergusson 11922176
Supervisors: Dr. Linda McIver and Mr. Martin Dick

Abstract

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

Many programming languages taught to novice programmers have obscure and terse syntaxes therefore a useful type of tool could convert code into a more descriptive, natural language. Semantic errors are extremely difficult for beginner programmers to locate and solve, a program that automatically detects and highlights common errors, could help, and may simplify the debugging cycle.

In order to discover if it would be feasible for a conversion program translating a more complex language to a more familiar language, to automatically detect common semantic errors, existing research in the areas of language design, educational techniques for first time programmers, common errors made by these programmers and various tools and languages currently available, are required to be reviewed.

Contents

1	Introduction	1
1.1	Outline of Review	2
2	How Novices Learn	3
2.1	Requirements For Learning	3
2.2	Linking New Concepts	4
2.2.1	Rephrasing	4
2.2.2	Instruction	5
2.2.3	Conflicting Information	5
2.3	Types of Learners	6
2.4	Learning Languages	6
2.5	Methods to Find a Solution	7
2.6	Summary	7
3	Mistakes Made By Novices	9
3.1	Categorisation of Errors	9
3.1.1	Syntax Errors	10
3.1.2	Semantic Errors	10
3.2	Common Mistakes - Experimental Findings	12
3.3	Summary	14
4	Tools and Techniques	15
4.1	Languages	15

<i>CONTENTS</i>	iv
4.1.1 Current Languages	15
4.1.2 Good Design	16
4.1.3 Bad Design	17
4.2 Tools	17
4.3 Teaching Methods	18
4.4 Error Resolution and Prevention	19
4.5 Summary	20
5 Why The AntiCompiler?	21
6 Concluding Remarks	23

Chapter 1

Introduction

Programming is a difficult task, especially for students who do not understand the workings of a computer, let alone have the background knowledge for linking programming concepts and learning a new language. Novice programmers frequently encounter obstacles that impede progress and cause much frustration including syntax errors caused by unfamiliarity with the language, logic errors, caused by unfamiliarity with programming concepts, and other problems which impede the learning process.

The difficulties that novices come across 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 aiding beginner programmers: the psychology of how people learn; language design; error detection and recovery methods, error reporting techniques, and various types of errors made.

Each of these aspects influence the way students learn. 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 difficult task as 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 understand confusing errors, especially when combined with a language with a terse non-natural syntax such as C (KR88). A language with a more natural syntax would 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 result of the error does not indicate where or why the error occurred. Some semantic errors that beginners produce are extremely common, and may be able to be detected automatically. If there existed a tool that could automatically identify and highlight such errors, it may simplify the debugging cycle, and result in a better understanding of the code and errors made.

The AntiCompiler aims to implement a translation from C to a more natural representation, and, if feasible, automatically detect a selection 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 semantic errors, many aspects need to be covered. Learning programming requires the understanding of how people learn, and more specifically, how novices learn programming. The psychology of learning is discussed in Chapter 2

Novices often make certain types of mistakes repeatedly and in similar locations. Categorisation schemes are often 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 teaching methods used in teaching novices programmers is provided in Chapter 4. Methods, languages and tools that may help with error resolutions are detailed more specifically as they are directly relevant to the development of the AntiCompiler. Common debugging methods used by novice programmers is an important area of study, as improved methods could be developed and taught in order to aid the learning of programming and a new language.

The AntiCompiler combines useful features of existing research, and is discussed in Chapter 5. Further research into the various aspects of learning, and more specifically learning 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 and unfamiliar technology are both involved in studying a new programming language. Learning requires a supportive environment, and some prior knowledge to connect new ideas to, otherwise the beginner either learns by rote, or encounters serious and sometimes unsurmountable obstacles. When teaching any new or difficult concept, the various types of learners need to be considered, in order to deliver guidance that can help the majority overcome obstacles.

Learning languages poses many problems, especially one where there is no 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 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 experience and enough positive continuing experience to continue trying to solve problems and learn new concepts. McIver (McI02), 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 (May86). If this mapping cannot occur, the student may be able to learn by rote, and will not fully understand the concept or its relations to other concepts. 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 a concept to link to. 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 an array, the student needs to map this knowledge 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 to deal with multi-dimensional arrays, requiring nested loops (loops within loops).

To aid the process of learning a new concept, several techniques can be employed to help link the new knowledge to existing knowledge. Two helpful techniques to learn a concept and deal with obstacles include rephrasing in their own words and receiving guidance or instruction from a tutor. However, both of these techniques may not help when there is conflict between the new concept and existing knowledge or beliefs.

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 (May86). Rephrasing their knowledge of a concept often can highlight misconceptions, showing conflicts between the new and existing knowledge, allowing tutors to correct understanding.

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 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. This will fail if the knowledge is not available to be linked to, 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 instructor may need to rephrase the concept, or the student may be needed to explain their understanding so misconceptions can be corrected or missing knowledge supplied (PHH⁺89; HS98). Perkins et.al. (PHH⁺89) suggests that the need for guidance may indicate that the beginners feel unconfident, due to the lack of existing knowledge which would normally provide them with avenues for testing solutions. This may result in them giving up, or continually tinkering.

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, 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* (Cor94), and can cause much difficulty in learning conflicting concepts. One example of this is where in computing, multiplication is 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, not being able to deal with the way the computer or language works.

A further example, more specific to the C programming language conflicting with the existing knowledge of a novice, is where counting is normally done from 0. The 3rd item in an array actually has a position of 2. Even though the counting mechanism is the same, counting in C conflicts with their existing knowledge, where novices would typically expect the first element to be at a position of 1. This requires a separation of

concepts in memory according to the context in which they apply, which is more difficult to assimilate.

2.3 Types of Learners

Different types of learners need different intervention and techniques to help them learn. According to Perkins, Hancock, Hobbs, Martin and Simmons (PHH⁺89) 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.

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. (PHH⁺89) 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 (Pan00) found supporting evidence such 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 the student’s code in a more familiar language provides a more familiar setting for the program, avoiding the need for a good understanding of the programming language. This would illustrate how to represent programming concepts in a precise but natural way, and 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 that maps to the programming representation. Stoppers may find that this level of instruction makes trying a solution less daunting, and 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 (DM85) compared the understanding of problems and representation of solutions of relatively experienced BASIC

programmers to natural language representations of the problems by non-programmers. They found no notable differences between the two groups which strongly suggests there are no differences in cognitive processes at different levels of experience. 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 compared two languages and found commonalities in the errors and error rates (EL92). 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 fewer errors are made (BS85; May87; SBE89; Mur93; MC96; AE99; McI00; Pan00; PRM01). Some of these studies also found that when a novice reached an impasse they frequently used a natural language representation to solve the problem (BS85; AE99).

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. (BS85).

Another study found that even experienced programmers are more comfortable representing problems and solutions in a natural language than in their programming language, with three categories of solution finding behaviour being observed:

- 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 (GO86; McI02). This is a problem especially where languages force the programmer to represent their solutions unnaturally, using complex constructs and confusing syntax. (Pan00)

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.

Programming languages use 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 (GO86) 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.

The mistakes that novices make are very common, and are typically similar across language paradigms. In order to discuss common errors, categorisations of error types need to be analysed.

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 (End75; McI00). Both of these areas have been sub-divided further, but as most compilers find and report 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,

thus several error categorisation schemes have been developed for common errors made by novices.

Pennington (Pen87) 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 (Pen87) 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 Syntax Errors

Endres (End75) found that only 15% of the errors made by experienced programmers in the DOS system (VS release 28), were syntax errors. The majority of the errors made were logic or understanding errors (semantic errors). This was supported by the empirical study looking at specific types of errors when novices were using LOGO and SOLO (EL92). 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, these errors are usually dealt with by the compiler, leaving 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 (Mur93). As they are not generally 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 teacheris is also made easier with the knowledge of common causes for groups of semantic errors.

Spohrer, Soloway and Pope (SSP85) classified semantic errors into two types:

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

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

This classification scheme is useful as it illustrates the cognitive processes behind the errors, why the novices made them. Both errors occur when novices merge goals together, which results in a more complex structure, resulting in steps being done out of order, or being forgotten.

An alternative and more focused classification scheme was developed by Pea (Pea86), 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 that 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, 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 most likely to assume that the program could look ahead and check that condition when inside the loop.

Egocentrism The final category attributes intelligence to the computer, where the novice assumes that computer knew what they meant. 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.

Other categorisation techniques that highlight the misapplication of existing natural language knowledge have been discussed in various papers. Du Boulay (DB89) proposes a different three categories:

Misapplication of analogy - an example would be believing a variable is like a box, and is able to store multiple items.

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 don't require them.

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

These various categorisation schemes serve as a varied illustration of the misconceptions novices and even experienced programmers have of programming concepts and the language constructs. Many of the 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 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 to search for the correct solution.

Loops

Natural language has a completely different interpretation of repeating tasks (loops) than does a step-by-step of imperative programming language such as

C (SSP85; KR88). This was also mentioned in the analysis of the categorisation scheme developed by Pea (Pea86) above as parallelism. Many other studies have verified these findings, noting that novices typically assume continuous testing of loop conditions (BS85; DB89; Pan00).

Soloway, Bonar and Ehrlich completed an empirical study of two types of looping constructs: `READ then PROCESS` and `PROCESS then READ`. 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 (WJ75), and the second representation, the actual looping construct of that language.

Conditions

The study by McQuire and Eastman (ME98) 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 (BS85; Pan00).

Assignment

A more specific problem for languages that use `=` for assignment of a value to a variable and `==` as a comparison checking for equality (DB89). Sometime 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 assume that the computer knows what values are needed to initialise variables. As a result the novice initialises them incorrectly or multiple times (JS86; Pea86). 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 (SSP85; DB89). 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 (SSP85; JS86). A 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. 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 and languages that are taught.

Chapter 4

Tools and Techniques

In order to create a tool to help novices learn a difficult language (C) and unfamiliar constructs, it is necessary to analyse existing tools and techniques created for this purpose. This Chapter will discuss both desirable and undesirable features for a beginner programming language, and discuss 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 (Mur93; MC99; PRM01; BS85). Even though natural language errors are made, they are usually easier to recover from (AE99). Despite the advantages of descriptive and natural languages, many current programming languages use obscure and terse syntaxes.

4.1.1 Current Languages

Most languages were developed by experts, focusing on speed, or properties specific to their domain, hardly ever considering the beginner programmer (Mur93; Pan00). Most of the languages used in first year programming courses are popular languages used in industry: C, Scheme, Ada, C++ (Lev95; Con94). These languages were not designed to teach programming concepts to new students, but for 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 be familiar to the students and not contradict their existing knowledge. The syntax of a beginner language should be simple and not overly terse.

Conway (Con94) 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 and not confuse the 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(McI02), SOLO(EL92), LOGO and MOOSE (AE99) are some recent examples. These languages attempt to simplify the language 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. This study 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 the general goals to facilitate learning, and maximise readability (McI02). 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 can be applied to a meta-language. These goals were also supported by Hsi et.al. (HS98) and Pane et.al. (Pan00).

4.1.3 Bad Design

Many languages are designed with no thought to usability and learnability and typically contain the following aspects (Moy92; Mod93; Mur93; Pan00):

- 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 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) (EL92). Du Boulay and Matthew (DBM84) built compilers that check for both syntactic and semantic errors, reporting problems with verbose comments. KyMir (BKMT94), a Russian program that uses marginal notes to debug programs without I/O was developed to reduce wasted time by teachers and students.

Compilers have gradually improved to report most syntax errors, but still are lacking the capability to analyse programs for semantic errors. Debugging tools have been created to make static verifications of code 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 (SO00; Joh78). `LCLint` is an improved version which uses annotations made by the user throughout the code to analyse for a larger number of semantic errors (Eva96; Eva03). Although these tools were targetted 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 to novices as they are required to annotate their programs with comments to aid automatic detection of errors. This may confuse students more, or enable them to understand program goals more clearly. Unfortunately, studies of novice programmers using these tools are rare.

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 (DLNS98). 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 bru:93 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.

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 (Bru93) comment that teaching support for the programming language should contain the following elements:

- 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 the language with useful features)
- Incremental development (learning a subset fully before moving on)

Reduced languages can address many shortcomings in complex languages. Safe-C, developed by Dix and Lien for an introductory programming course, provided a ‘safer’ subset of the C language (DL93). 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 altered to further aid understanding.

It is important to choose reliable and comprehensive tools to support the novices’ language development. The main aim of a beginner programming course, is not to learn a difficult language, but to learn the concepts of programming (Lev95).

4.4 Error Resolution 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 in 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. (AE99). It is therefore important for a programming language to be used by novices, to be well designed and 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 (EL02; LE01).

Visual environments that can highlight errors as the novice types, and even auto-correct some errors. This can dramatically reduce the number of errors made. (BKMT94; AE99) However, this may not aid the novice in learning about their errors, especially with autocorrection. 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 learn strategies ready for dealing with them.

4.5 Summary

Current languages are typically obscure and hard to learn, not made any easier by the varied tool-set that supports them. Good languages need to be based on the beginner's existing knowledge of natural language. Programming languages must be simple and consistent. The tools used for program development must 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 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 where 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 will be more verbose and clearer for beginners. As one of the most important factors in 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 fairly surprising that one of the most commonly taught language is C, with its obscure and terse syntax and poor error reporting. C compilers have improved over the years and can detect and warn of syntax errors, albeit confusing at times.

As C is a commonly taught introductory programming language, it would be useful to be able to translate C programs into a more natural representation. This would allow the novice to look for logic errors much more easily. It may help the novice learn the concepts more thoroughly and allow a better mapping between the solution they develop in natural language the 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. A semantic error checker would be a boon for many programmers, novices and advanced programmers alike, as it could cut down the time spent searching for steps that are done out of order, or initialised variables, the annoying logic errors that are always so hard to find, especially for the beginner.

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

The reports generated by the AntiCompiler will be able to 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 will be indicated in either the C code or the natural language version of the program.

Understanding common errors made by novices will help improve teaching methods of both lecturers and tutors. 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 by translating a confusing C program to a more natural representation, the novice may be able to find their errors quicker and with less frustration.

Chapter 6

Concluding Remarks

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.

More research is needed into how much difficult languages impede the learning process. Especially as the main teaching languages are decided by what is 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.

Common errors made by novices have been categorised in various ways, but not many of these classification schemes enable the automatic detection of semantic errors. 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 not be necessary or in fact possible to change which language is taught to novices, if a good set of supporting tools are provided with 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 make on the learning process is an important area to further research.

If instead of frightening novice programmers, they felt comfortable enough with the language, the learning environment and supporting tools, maybe better programmers would result.

Bibliography

- Bruckman A. and E. Edwards. Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. In *CHI'99 Papers*, 15–20 May 1999.
- P. Brusilovsky, A. Kouchnirenko, P. Miller, and I. Tomek. Teaching programming to novices: a review of approaches and tools. In T. Ottman and I. Tomek, editors, *ED-MEDIA'94 - World Conference on Educational Multimedia and Hypermedia*, pages 103–110, 1994.
- P. Brusilovsky. Towards and intelligent environment for learning introductory programming. In E. Lemut, B Du Boulay, and G. Dettori, editors, *Cognitive Models and Intelligent Environments for Learning Programming*, pages 114–124. Springer-Verlag, 1993.
- J. Bonar and E. Soloway. Preprogramming knowledge: A major source of misconception in novice programmers. *Human Computer Interaction*, 1(2):133–161, 1985.
- D. Conway. Criteria and consideration in the selection of a first programming language. Technical Report no 93/192, Monash University, December 1994.
- R.J. Corsini. *Encyclopedia of Psychology*. Wiley, 1994.
- B. Du Boulay. *Some Difficulties of Learning to Program*, pages 283–299. Lawrence Erlbaum Associates, 1989.
- B. Du Boulay and I. Matthew. Fatal error in pass zero: how not to confuse the novices. *Behaviour and Information Technology*, 3(2):109–118, 1984.
- T. Dix and T Lien. Safe-c for introductory undergraduate programming. In *Proceedings of the sixteenth Australian computer science conference*, pages 371–378, 1993.
- D. Detlefs, K. Leino, G. Nelsn, and J. Saxe. Extended static checking. Technical report, COMPAQ Research Center, California, 1998.

- J. Dyck and R. Mayer. Basic versus natural language: Is there one underlying comprehension process? In *CHI'85 Proceedings, Conference on Human Factors in Computing Systems*, April 1985.
- M. Eisenstadt and M. Lewis. Errors in an interactive programming environment: Causes and cures. In M. Eisenstadt, M. Keane, and T. Rajan, editors, *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence*, chapter 5. Lawrence Erlbaum Associates, 1992.
- D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.
- A. Endres. An analysis of errors and their causes in system programs. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.
- D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.
- D. Evans. *Splint Manual*. www.splint.org/manual/manual.pdf, 3.1.1-1 edition, 2003.
- L. Gugerty and G. Olson. Debugging by skilled and novice programmers. In *CHI'86*, pages 171–174, April 1986.
- S. Hsi and E. Soloway. Learner centered design. *SigCHI Bulletin*, 30(4):53–55, 1998.
- S.C. Johnson. Lint, a c program checker. Technical Report 65, Bell Laboratories, 1978.
- S.A. Joni and E. Soloway. But my program runs! discourse rules for novice programmers. *Educational Computing Research*, 2(1):95–128, 1986.
- B. Kernighan and D. Ritchie. *The C Programming Language, 2nd Ed.* Prentice Hall, 1988.
- D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. To appear in 2001 USENIX Security Symposium, August 2001.
- S.P. Levy. Computer language usage in cs1: Survey results. *SIGCSE Bulletin*, 27(3):21–26, 1995.
- R. Mayer. *The Psychology of How Novices Learn Computer Programming*, pages 129–159. Baywood Publishing Co. Inc., 1986.
- R. Mayer. Cognitive aspects of learning and using a programming language. In J. Carroll, editor, *Interfacing Thought*, pages 61–79. MIT Press, 1987.

- L. McIver and D. Conway. Seven deadly sins of introductory programming language design. In *Software Engineering: Education and Practice*. IEEE Computer Society Press, 1996.
- L. McIver and D. Conway. Grail: A zero'th programming language. In G Cummings, T. Okamoto, and L. Gomex, editors, *7th International Conference on Computers in Education ICCE'99*, volume 2, pages 43–50. IOS Press, November 1999.
- L. McIver. The effects of programming language on error rates of novice programmers. In *12th Annual Meeting of the Psychology of Programming Interest Group*, April 2000.
- L. McIver. *Syntactic and Semantic Issues in Introductory Programming Education*. PhD thesis, Computer Science and Software Engineering, January 2002.
- A. McQuire and C. Eastman. The ambiguity of negation in natural language queries to information retrieval systems. *Journal of the American Society for Information Science*, 49(8):686–692, 1998.
- R. P. Mody. C in education and software engineering. *SIGSCE Bulletin*, 24(3):45–56, September 1993.
- P.J. Moylan. 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, July 1992.
- J. Murnane. The psychology of computer languages for introductory programming courses. *New Ideas in Psychology*, 11(2):213–228, 1993.
- B. Pane, J. Myers. The influence of the psychology of programming on a language design: Project status report. 12th Annual Meeting of the Psychology of Programming Interest Group, 10–13 April 2000. PPIG 2000.
- R. D. Pea. Language-independent conceptual “bugs” in novice programming. *Educational Computing Research*, 2(1):25–36, 1986.
- N. Pennington. *Comprehension Strategies in Programming*, pages 100–112. Ablex Publishing Corporation, New Jersey, 1987.
- D.N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R Simmons. Conditions of learning in novice programmers. In E. Soloway and J.C Spohrer, editors, *Studying the Novice Programmer*. Lawrence Erlbaum Associates, 1989.
- J. F. Pane, C. Ratanamahatana, and B. A. Myers. Studying the language and structure in non-programmers’ solutions to programming problems. *International Journal of Human-Computer Studies*, 54:237–264, 2001.

- E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. In E. Soloway and J.C. Spohrer, editors, *Studying the Novice Programmer*, pages 191–207. Lawrence Erlbaum Associates, 1989.
- D. Santo Orcero. The code analyser lclint. *Linux Journal*, 73, May 2000.
- J. G. Spohrer, E. Soloway, and E. Pope. Where the bugs are. In *CHI'85 Conference on Human Factors in Computing Systems*, pages 261–279, April 1985.
- N. Wirth and K. Jensen. *Pascal User Manual and Report*. Springer-Verlag, 1975.