The Cognitive Construction of Programs by Novice Programmers

A thesis presented in partial
fulfillment of the requirements

for the degree of
Master of Science
in Computer Science at
Massey University

Zoë Joanne Rose
1998

This page intentionally left blank

ABSTRACT

Human memory and cognition are studied to aid novice programmers with the cognitive construction and the acquisition of program plans. Particular emphasis is placed on the storage and retrieval of program knowledge, the cognitive structure of stored program knowledge, the effects of transferring cognitive structures from one programming language to another, and the learning activities involved with learning a new programming language. Cognitive principles are applied to the design of a programming language and environment. The design of both the programming language and environment are discussed together with an introduction of how they are used. The hypothetical results of two experiments are argued to demonstrate that the programming language and environment are well suited in supporting the development of program plans.

This page intentionally left blank

ACKNOWLEDGMENTS

To Grant for having the patience of a saint.

For being great parents and giving unlimited support a special thankyou to Paul and Julie Rose.

Happy smiley faces also go to my supervisors ☺ Nigel and ☺ Doug for patiently waiting for a masterpiece to be completed - hope this thesis will suffice instead.

Giant cuddles and big meaty bones also go to my dog Akela for all the daily walks I missed.

This page intentionally left blank

TABLE OF CONTENTS

This page intentionally left blank

TABLE OF FIGURES

1. INTRODUCTION

Computer programming is a form of problem solving, and like all forms of problem solving it takes time to reach an expert level of proficiency. According to Campell, Brown and DiBello (1992), this time period is approximately five or more years.

In order to optimise this time period and understand why it takes so long to become a proficient programmer it is necessary to study human memory and cognition (Holt, Boehm-Davis and Schultz, 1987). Specifically, it is necessary to study:

- How knowledge is cognitively stored and retrieved in human memory, in particular how programming knowledge is cognitively stored and retrieved.

- The differences between an expert programmer's knowledge store and a novice programmer's knowledge store.

- What type of knowledge is transferred from one programming language to another, and does this transfer of knowledge help or hinder the programmer?

- What stages of learning does a programmer progress through in order to learn a new programming language?

1.1 THEORIES OF HUMAN MEMORY STORAGE

Memory involves retaining information over time; this can include memories retained for less than one second or memories that are retained over a lifetime. At present there are three main theories on human memory storage:

- The Atkinson-Schiffrin Model (1968);
- The Levels of Processing Approach (Craik and Lockhart, 1972);
- The Episodic and Semantic Model of Memory (Tulving 1972).

1.1.1 The Atkinson-Schiffrin Model of Human Memory

Figure 1 presents a flow chart of the Atkinson and Schiffrin model of memory, the flow indicates that information is transferred from one storage area to another.

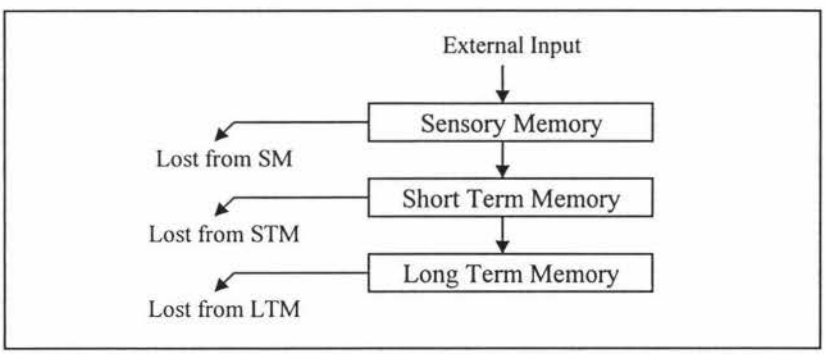

Figure 1 : The Atkinson-Schiffrin Model of Memory

Information from the environment that is raw and unprocessed will first enter sensory memory. Sensory memory is the large capacity storage system that records information from the senses.

Relevant information then passes from sensory memory into short term memory, while all other irrelevant information is discarded (as shown in Figure 1). For example, a student studying in a library will often hear the whispers of other library patrons talking. The whispering noises are irrelevant to the studying student, thus these noises will be discarded and only the relevant studied facts will be transferred to the student's short-term memory.

As shown in Figure 1 information will then pass from short-term memory to long-term memory. Atkinson and Schiffrin state that if information is rehearsed frequently and kept for a long period of time in short term memory, the information will be more likely to transfer to long term memory. Information that is contained in long term memory is relatively permanent and not easily lost (Matlin, 1989).

In more recent years there has been an abundance of research into human memory that suggests memory may not be stored in fixed structures, as with the Atkinson-Schiffrin model. This research has lead to the popularity of the levels of processing approach.

1.1.2 The Levels of Processing Approach

In contrast to the Atkinson-Schiffrin model, which places the emphasis on fixed structures, the levels of processing theory stresses the flexibility humans use when processing information (Matlin, 1989; Reed, 1988).

The levels of processing model suggests humans analyse new information in many different ways, from shallow levels of processing to deep levels of processing. Shallow levels of processing may involve judgements about letters, e.g. the height, colour, or shape of a letter, whereas deeper levels of processing may involve judgements about words, e.g. whether a word is appropriate for a particular sentence (Matlin, 1989).

According to Craik and Lockhart (1972), deeper processing of information leads to better retention and recall of information, because deeper levels of encoding will extract more from a stimulus. When an individual analyses new information they may think of other associations, images, and past experiences related to the new information. A deeper association is thus made to the new information, and it is less likely to be forgotten.

Craik and Lockhart believe that the levels of processing approach uses distinctiveness and elaboration to help promote information recall. Distinctiveness describes the extent to which a stimuli's memory trace is different from other memory traces in an individual's memory system. Elaboration involves processing in terms of meaning (Matlin, 1989, Reed, 1988).

1.1.3 The Episodic and Semantic Model of Memory

Like Atkinson and Schiffrin's model of memory Tulving (1972) also categorises memory into two types: Episodic and Semantic.

Episodic memory contains information about time-date events, e.g. "I have a dentist appointment at 3:30 p.m. tomorrow". Semantic memory holds fairly constant, organised knowledge about the world, e.g. "I remember that the chemical formula for water is H_20".

Just as there are many different theories on how memory as a whole is stored, there are many different theories on how Tulving's semantic memory is stored. One of the most popular theories is Anderson's Propositional Network Model (1983). According to Anderson the propositional network model proposes that there is a netlike organisation of propositions in memory, where a proposition is the smallest unit of knowledge that can be judged true or false.

Each proposition is represented as a node in one of many networks contained in human memory. According to Anderson the links between the nodes vary in strength. The more the links are used the stronger they become. When one node is activated, the activation spreads to other linked nodes, which in turn are also activated.

Both the levels of processing approach and the propositional network approach agree that the more associations a given piece of information receives, then the easier that information is to recall and retain. However, if an incorrect association occurs the wrong information can mistakenly be recalled.

1.2 Cognitive Representation of Computer Programs

In order to aid programmers it is not only necessary to study theories of human memory storage, it is also necessary to study the way in which a program is stored cognitively, and how the programmer uses this cognitive representation (Holt, Boehm-Davis & Schultz 1987).

Letovsky (1986) suggests that mental models are used to help form the basis of a programmer's cognitive model. Holt et.al. (1987) suggests that programmers form this mental model from a program's structure and function.

1.2.1 Mental Models

A mental model is an internal representation that an individual has of a problem (Matlin, 1989), i.e. a picture in one's mind. For example, if a tourist needs directions from their hotel to a famous church, they might be given these instructions:

```
Turn left when you leave the hotel. Turn right at the first
set of lights.  On the same intersection is a McDonalds
```

```
restaurant and a children's park.  Carry on down this road
until you reach the third set of traffic lights.  Just
before you reach the traffic lights there is a lake for
sailing boats.  The church is in the opposite corner of the
intersection.
```

Figure 2 shows the mental model the visitor might construct from the above directions.

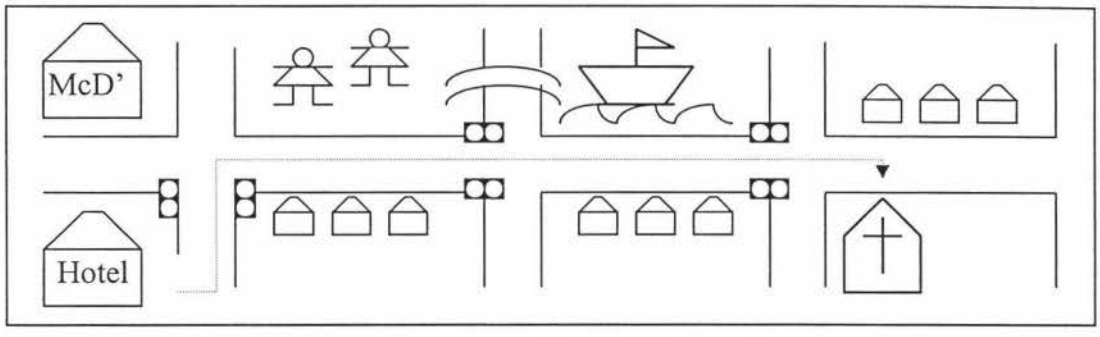

Figure 2 : A Tourist's Mental Model of Directions

As shown in Figure 2 the tourist has included unnecessary information in his model. The tourist has even included assumed information that was not stated. This unnecessary attention, by the tourist, has caused inaccuracies in the model to occur. The tourist now believes the church is one traffic light closer than it actually is.

If the same directions were given to a taxi driver, then Figure 3 could represent the outcome of the driver's mental model.

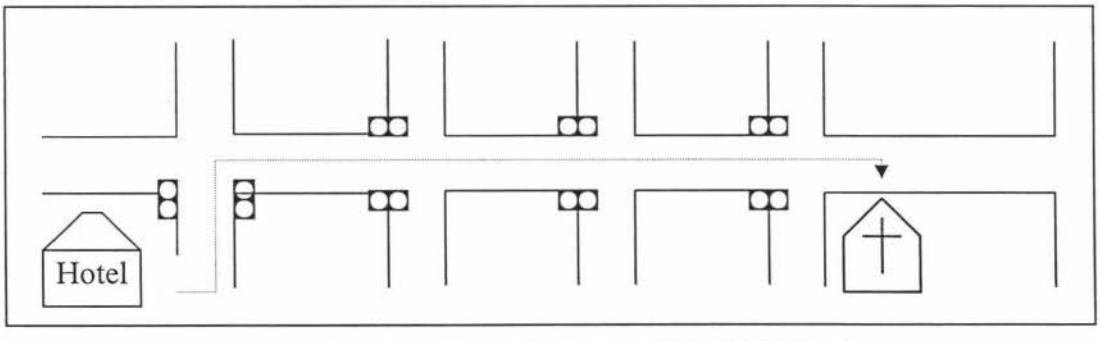

Figure 3 : A Taxi Driver's Mental Model of Directions

As the taxi driver is assumed to be an expert in the field of deciphering directions his mental model is simple and uncomplicated. He has managed to eliminate all unnecessary information, and is left with a simple model that will guide him directly to the church.

From the above example, it can be seen that if the tourist doesn't learn to build mental models similar to the expert's, then he is going to have to buy a map.

Just as the tourist needs to learn how to construct a taxi driver's expert mental model, a novice programmer needs to learn how to construct an expert programmer's mental model.

Letovsky (1986) suggests that an expert programmer's mental model is created through the combination of information from:

- reading the program documentation and code;
- knowledge from a knowledge base of expertise.

Reading a program's documentation and code is easy, but how does a programmer build a knowledge base of expertise?

1.3 Expert Programmers Versus Novice Programmers

According to Fix, Wiedenbeck & Scholtz (1993) a general research finding is that expert programmers have a better functional understanding about what a program does rather than how it does it. Novice programmers fail to extract the necessary information to form a complete mental representation. Fix et.al. (1993) go further to suggest that there are five abstract characteristics which exist in an expert's mental representation of a program, which do not appear in a novice's mental representation.

1.3.1 Characteristics of an Expert Programmers Mental Representation

1.3.1.1 Hierarchic Structure

A hierarchically structured mental model is a representation that contains depth and breadth proportional to a program's structure. Nanja and Cook (1987) also observed this characteristic and noted that expert programmers read a program in the order it is executed. Nanja and Cook believe this feature leads to the hierarchical structure of a programmer's mental model. For example, Figure 4 illustrates 'Problem 1' with a structure diagram that has both depth and breadth.

```
Output the sum of ten random numbers
```

Problem 1

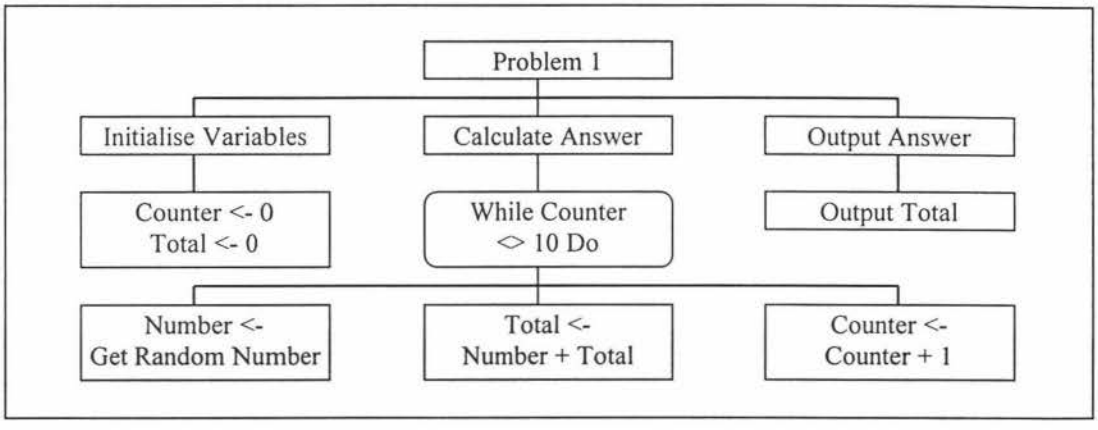

Figure 4 : Structure Diagram (Mental Model) of Problem 1

1.3.1.2 Explicit Mappings

Explicit mappings are the links between the layers of a hierarchically structured mental model. Letovsky (1986) argues that programs can be quite clear to novices through the use of documentation, variable names, data structures etc. However, overall program comprehension cannot be achieved unless there are links between the different entities, i.e. a mapping between high-level program goals and their code representation. Pennington (1987) further verified that there existed differences in the ability of expert and novice programmers to link specific segments of code to program goals.

As a simplified example of this case, problem 1 has three simple goals:

1. Initialise variables
2. Calculate answer
3. Output answer

As illustrated in Figure 5, expert programmers relate goals 1-3 to areas a-c respectively.

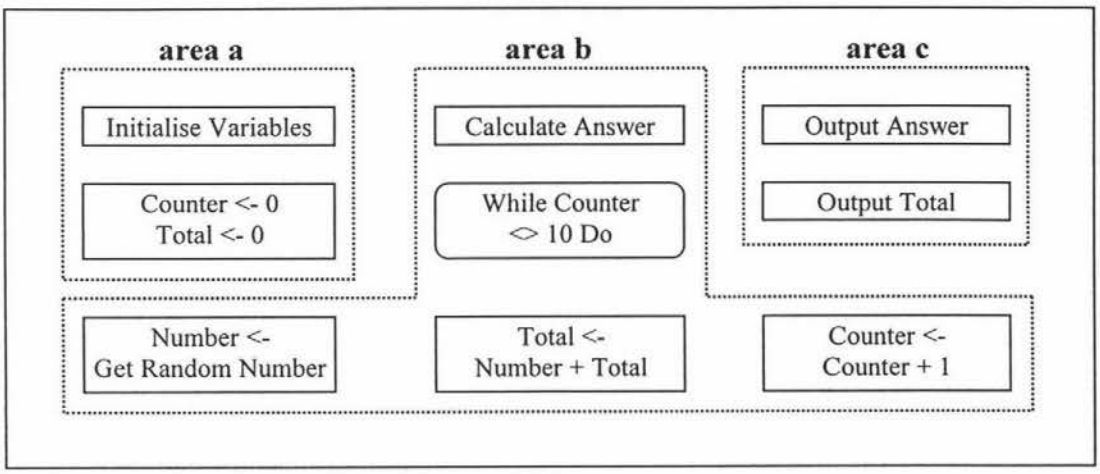

Figure 5 : Mapping Program Goals to the Mental Model

1.3.1.3 Basic Recurring Patterns

Soloway and Ehrlich (1984) suggest that expert programmers store programmer knowledge as 'plans' (mental models) for managing frequently recurring situations. They showed that if programs were not structured in a 'plan' format then experienced programmers comprehension was disrupted.

For example, the programmer's plan in Figure 4 is to use a 'while statement' for summing 10 random numbers. The programmer in this example could just have easily used a 'for statement'. However, it might be that the programmer has used 'while statements' several times previously, and so already has a planned mental model of 'while statements' formed.

According to the Levels of Processing Approach and Anderson's Propositional Network Model, this is because the more the 'while statement' is used by the programmer the deeper the statement will be processed in memory and the more associations it will have. Thus, when recalling information about a statement that can be used for iteration, the 'while statement' immediately occurs to the programmer.

1.3.1.4 Well Connected

Fix et.al. (1993) state that a model is well connected if a programmer understands how parts of a program interact together. They go further to state that an experienced programmer will concentrate more on areas of their code which require interaction, i.e.

module interfaces, whereas a novice is unlikely to pay attention to this type of information.

Using Figure 5 as an example, the experienced programmer is able to understand how the different areas interact together, i.e. the programmer knows that 'area a' is always executed first, followed by 'areas b and c'.

1.3.1.5 Well Grounded

A mental model is well grounded if it includes information on the physical locality of structures and operations in the program code. Fix et.al. (1993) state that in general experts will have a mental model which is well grounded in the program text, whereas a novice's model is only well grounded for fixed information. For example, an expert programmer will know the locality of different programs and functions throughout their code, whereas a novice programmer may only know where to locate declared variables.

1.4 TRANSFER

Once a novice programmer learns the basic steps of creating a 'programmer's' mental model, they can use those steps as building blocks to help create future programs. However, what if it is necessary for a programmer to write a program in a target language that is different from the language in which they initially learned to program?

Learning second and subsequent programming languages involves transferring previously learned skills and concepts (Scholtz and Wiedenbeck, 1990). Although the transfer remains in the domain of programming it is still difficult, even for experienced programmers. This is because having an understanding of the new language is not enough, the programmer must also build a foundation of mental models with the new language.

When learning a second or subsequent programming language, two types of transfer can occur: negative transfer and beneficial transfer. Negative transfer occurs when the learning of a new skill is in direct conflict with a skill already well known (Anderson, 1985). An example of negative transfer is mental set. Mental set is a characteristic of problem solvers to use a solution they have previously used to solve a new problem,

even when there maybe better methods. Mental set can therefore be described as blocking the problem solver from effective problem solving (Matlin, 1989).

Luchins (1942) demonstrated the problem of mental set very effectively. He used a 'water jar' puzzle to test the mental set of subjects. The problem has seven sub-problems. All seven problems could be solved using the same solution method. The last two, however, have a much simpler and easier solution. Luchins discovered that if a subject solved problems 1 - 5 first, then they were more likely to use the same method for solving problems 6 and 7. However, if a subject solved problems 6 and 7 first, then the subject was more likely to use the simpler method.

Beneficial transfer occurs when an old skill aids the learning of a new skill (Anderson, 1985). Singley and Anderson (1985) showed that if two text editors shared common elements, then knowledge learned from one text editor beneficially transfers to the other.

Learning a text editor can be quite different from learning a programming language. Does the Singley theory hold for learning new programming languages? In other words, do common elements between programming languages transfer, and if so, what are these common elements, and in which areas do programmers have the most difficulty transferring knowledge and skills.

1.5 Central Learning Activities

Scholtz and Wiedenbeck (1990) showed that when a programmer is given a problem to solve in a new programming language, the programmer:

- first forms a mental model of the solution in a language they already know,
- then tries to find ways to implement this solution in the new language.

Often the concepts or constructs a programmer needs are not found in the new language. This is a clear case of negative transfer, where the programmer has made incorrect mental model associations.

Scholtz and Wiedenbeck (1990) also discovered three learning activities associated with the learning of a new programming language.

1.5.1 Language Syntax

"Syntax describes the form of the sentences in a language" (Terry, 1986). For example, the English sentence 'The boy ran away' is syntactically correct. Whereas the sentence 'Boy away ran the' isn't. According to Scholtz and Wiedenbeck, learning the syntax is the easiest part of learning a new programming language.

If an experienced programmer is already familiar with a language, they will spend very little time thinking about syntax. If however, an experienced programmer is learning a new language, they will devote more time to studying the new syntax.

1.5.2 Language Semantics

"Semantics describe the meaning of a syntactically correct sentence in a language" (Terry, 1986). For example the sentence "The frog ate the fly" is syntactically and semantically correct. Whereas the sentence 'The fly ate the frog' although syntactically correct, is not semantically correct.

Scholtz and Wiedenbeck discovered that an experienced programmer will concentrate more on the semantics of a language than on the syntax of a language. Experienced programmers are very careful to understand the semantics of constructs they use. If constructs in a new language have a similar syntax to constructs in a known language then assumptions are made as to their function. This is where the programmer can make incorrect mental model associations. Negative transfer of this type is very persistent amongst programmers.

According to Scholtz and Wiedenbeck when learning new programming languages, experienced programmers will effectively apply and transfer their past knowledge of syntactic and semantic information.

1.5.3 Planning Activities

Planning is associated with the structure of a solution. It is associated with how the programmer's mental model is put together. Soloway and Ehrlich (1984) identified three levels of plan knowledge among expert programmers.

1.5.3.1 Strategic Plans

Strategic plans are language independent and are used by the programmer to form an overall strategy for solving a problem. According to Scholtz et al. (1990) strategic planning occurs at the very beginning of the problem solving process. An experienced programmer is inclined to spend very little time forming strategic plans. This is because experienced programmers will beneficially transfer a solution they have previously used to help solve a similar problem.

For example, the same programmer who solved problem 1 is asked to solve problem 2, However this time the target language is different to the target language of problem 1. As the structure of the two problems is similar, the programmer may use the same strategic plan (Figure 6).

```
Output the product of the first three number entered by a user
```

Problem 2

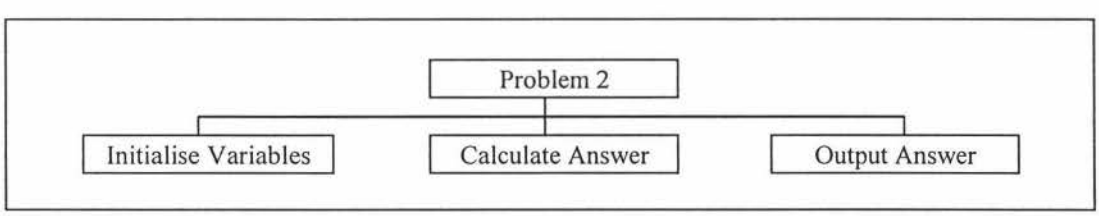

Figure 6 : Generic Strategic Plan for Solution to Problems 1 and 2

As can be seen from Figure 6, the programmer has beneficially transferred his knowledge from a previous problem to help solve a new problem.

1.5.3.2 Tactical Plans

Tactical plans are language independent and are used by the programmer to form a local strategy for solving a problem. Using problem 2 again as an example, the programmer can beneficially transfer his tactical plan from problem 1 to help solve problem 2. All

that is necessary is for the programmer to 'fill in the gaps' to find the new solution (Figure 7).

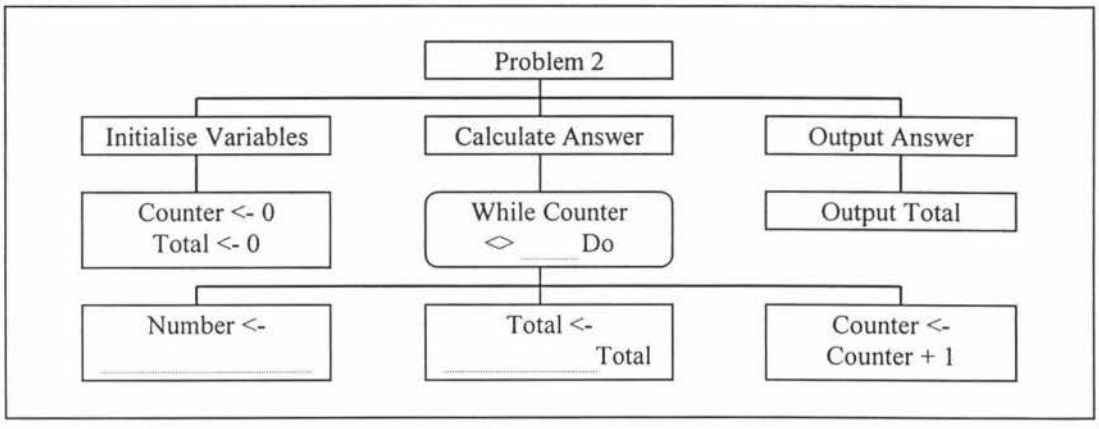

Figure 7 : Generic Tactical Plan for Solution to Problems 1 and 2

In assuming the use of variables and iteration the programmer is also assuming a similarity of target languages.

When creating tactical plans programmers make assumptions about the language in which the solution will eventually be written. Experienced programmers will have, over time, built up a reasonable number of tactical plans, so that, if one plan fails, another can be selected.

This method, however, does have drawbacks. As the programmer is relying on tactical plans for a given language, they maybe inappropriate in some situations.

Unlike strategic plans a programmer will often come back to re-evaluate their tactical plans during implementation.

1.5.3.3 Implementation Plans

Implementation plans are language dependent and are used by the programmer to determine how they will achieve their strategic and tactical plans in the target language.

Programmers spend the bulk of their planning time 'planning' the implementation stage. The programmer will usually start their implementation plan by finding constructs with a resemblance to constructs they have previously used. Implementation plans will go smoothly if this is the case, but often programmers cannot find such constructs, and a

revision of their tactical plan is necessary. This is because even though tactical plans should be language independent they often make language commitments. For this reason there is a strong interaction between tactical and implementation planning.

When learning a new language, implementation planning and the interaction is has with tactical planning is often the most difficult stage of the planning process for a programmer. Figure 8 shows the interactions between the planning stages.

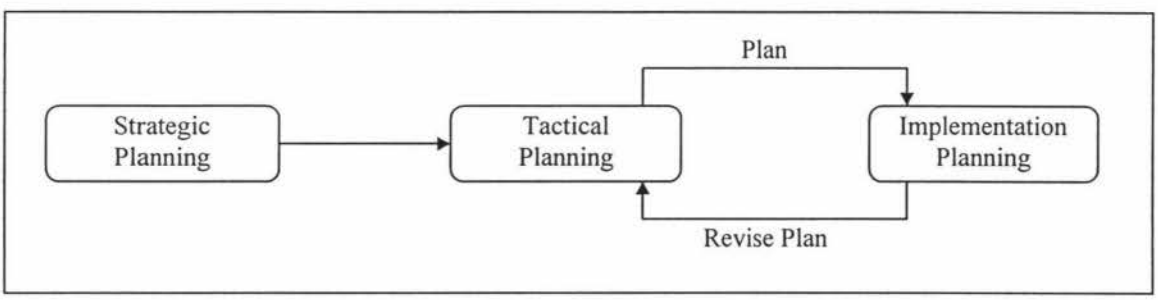

Figure 8 : Levels of Plan Knowledge Amongst Expert Programmers

As shown in Figure 8 , the programmer may move several times between the two stages, until a solution is eventually reached.

This by-play between the two stages of planning eventually leads to a store of mental models for the programmer.

1.6 OBJECTIVE

The objective of the present work is:

> "to aid novice programmers in the cognitive construction of programs and the acquisition of program plans".

This goal will be achieved by applying cognitive principles:

1. To the development of a simple programming language designed to enable novice programmers to develop simple mental models.

2. To the development of an easy to use programming environment for this language.

Creating simple mental models are beneficial to the novice programmer because they:

- can be used as building blocks, to build more complicated models.

- are easy to positively transfer to other languages.

Creating an easy to use environment will:

- Encourage the novice programmer to explore the programming language and build up a store of strategic, tactical and implementation plans.

This page intentionally left blank