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.

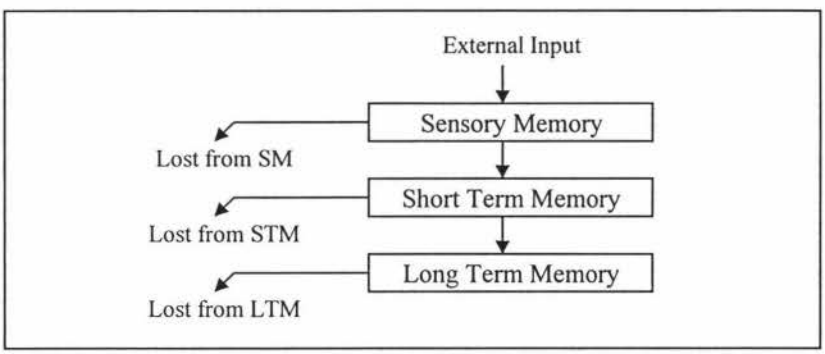


Figure 1 : The Atkinson-Schiffrin Model of Memory

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.

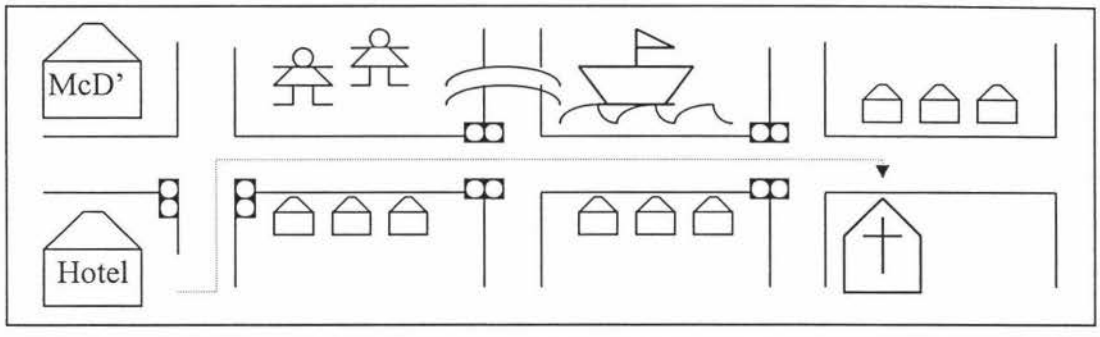


Figure 2 : A Tourist's Mental Model of Directions

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.

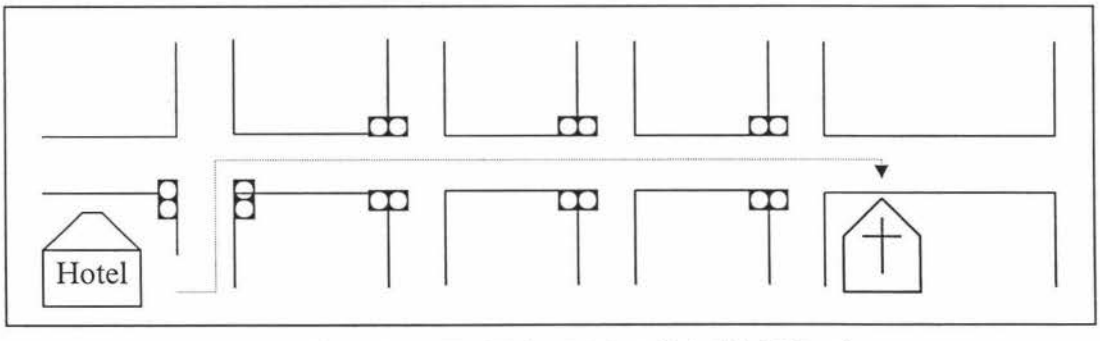


Figure 3 : A Taxi Driver's Mental Model of Directions

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

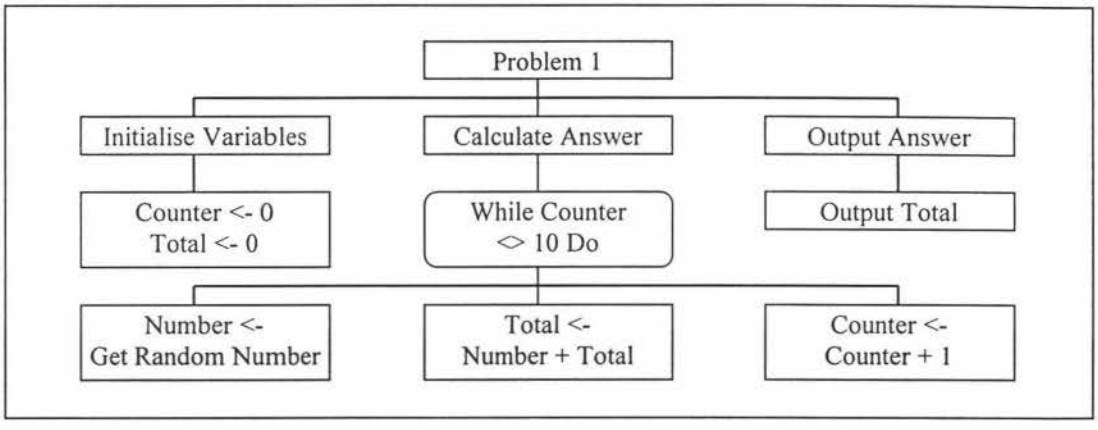


Figure 4 : Structure Diagram (Mental Model) of Problem 1

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.

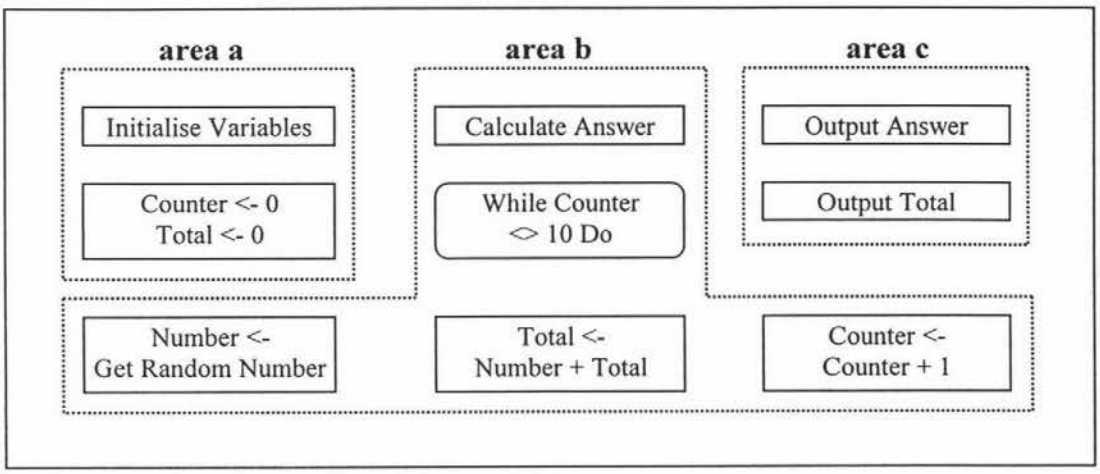


Figure 5 : Mapping Program Goals to the Mental Model

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

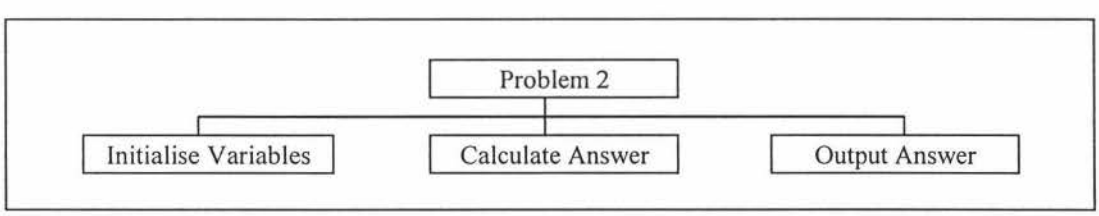


Figure 6 : Generic Strategic Plan for Solution to Problems 1 and 2

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).

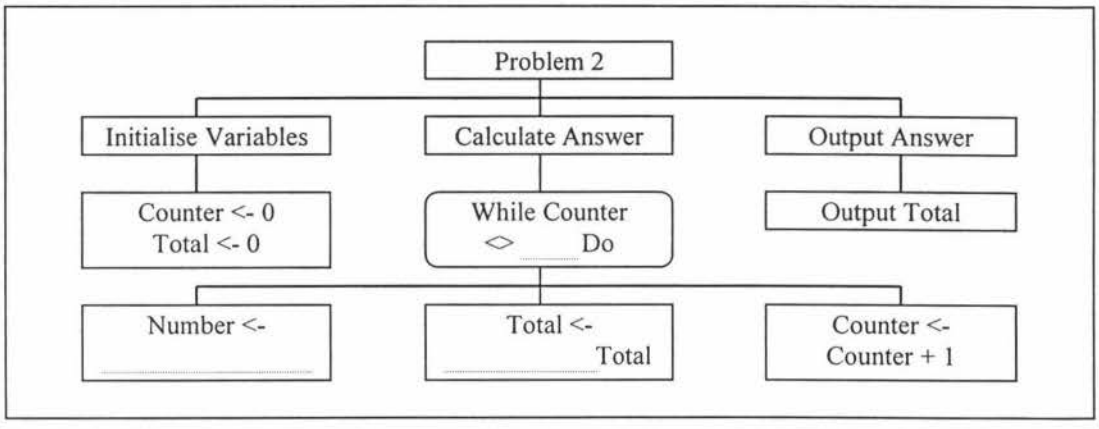


Figure 7 : Generic Tactical Plan for Solution to Problems 1 and 2

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.

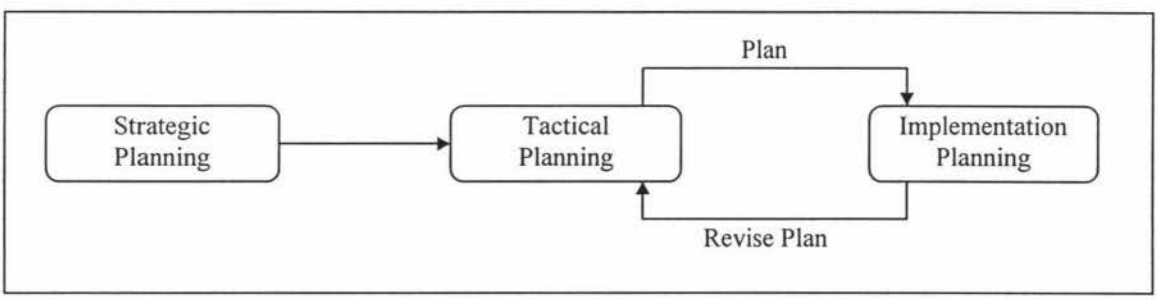


Figure 8 : Levels of Plan Knowledge Amongst Expert Programmers

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

2. PROGRAMMING CONCEPTS

This chapter describes how cognitive principles are used to create the ZL language and environment.

2.1 MENTAL MODELS AND THE ZL PROGRAMMING LANGUAGE

According to Böhm and Jacopini (1966), in its very simplest form, a programming language need only have two fundamental programming concepts in order for a complete program to be written. These concepts are composition and iteration.

As the ZL language is designed to aid novice programmers with the learning of computer programming languages in general, it must not only include the fundamental concepts of composition and iteration, but also basic programming concepts found in modern programming languages.

The ZL language exemplifies generic programming concepts common to some, if not all, programming languages. These concepts are:

- Value Naming
- Composition
- Iteration
- Conditionals
- Operator Application
- Pattern Matching
- Nesting
- Recursion

2.1.1 Value Naming

Value naming is the binding of a name to a value. Once bound the name can be used to represent its value in a program, i.e. the value can be manipulated through the use of the name.

In ZL qualified expressions are used to name values. For example:

```
let   one   <-   1
|     two   <-   1 + 1
|     three <-   1 + 2       in    one + two + three
```

2.1.1.1 Mental Model - Value Naming

Value naming is an important concept, as it can be likened to variable assignment, which is a generic concept common to many imperative languages. Variable assignment binds a variable to a value, just as value naming binds a name to a value.

It is not the exact details of the variable assignment concept that are important, but the concept itself. Thus, if a novice programmer can form a mental model of the concept variable assignment using a qualified expression then the objective has been achieved.

The above ZL example binds the values `1`, `2`, and `3` to the names `one`, `two`, and `three` respectively. The goal is to aid the novice programmer with the association of the names and values when they initially build their mental model. This is achieved by:

- Syntax - the symbol `<-` is a visual aid to the novice programmer, i.e. they are most likely to have seen and used this symbol previously, the symbol is visually stating to the user 'bind `one` to `1`', 'bind `two` to `2`', bind `three` to `3`'.

- Hierarchical Structure - a value is named before it can be used in a program. A ZL qualified expression ensures names are bound to values before they are referenced in an executed expression, this feature aids the building of breadth in a novice's mental model. Depth of a mental model is also aided by the consecutive binding of names to values.

- Explicit Mapping - in a ZL qualified expression a link is formed through the word '`in`' from the binding of the values to their names to the expression in which they are used.

Figure 9 illustrates how a novice user from the previous example of a ZL qualified expression can construct a mental model for value naming.

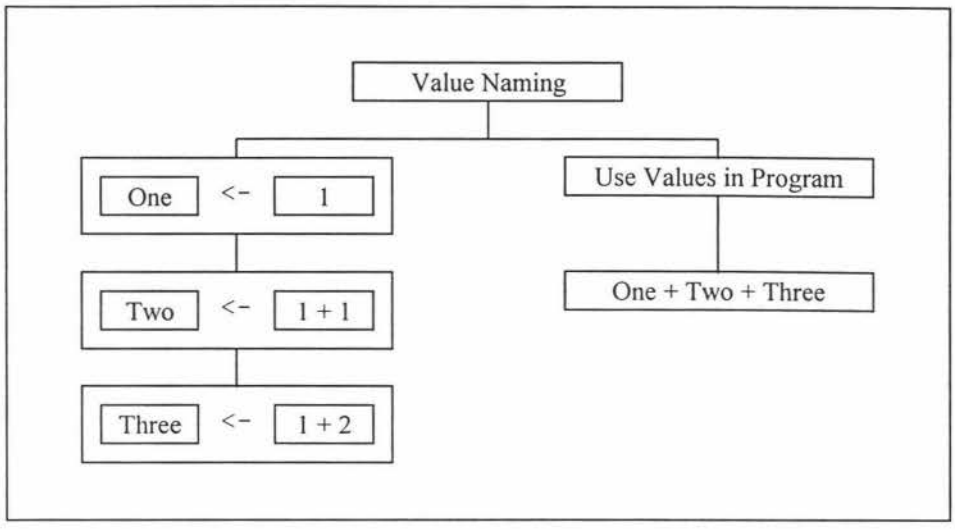


Figure 9 : Mental Model of a Value Naming

As shown in Figure 9, the mental model the novice user may create is generic, it will beneficially transfer from the ZL programming language to most other programming languages, i.e. the novice programmer will bind all names to values before the names are used in the program.

2.1.2 Operator Application

A basic programming concept found in high level programming languages is operator application. Operator application is where an operator is applied to one or more operands. The operator can either be relational, e.g. <, >, => or it can be mathematical, e.g. +, -, *. For example,

```
6 + (4 * 2)
```

2.1.2.1 Mental Model - Operator Application

The goal is not to build new mental models for operator applications, but to use existing models already constructed by the novice programmer. This is achieved by:

- Deep Levels of Processing - As the precedence of all mathematical operators in ZL follow conventional rules, evaluating an operator expression in ZL will not introduce anything new to the novice programmer. Therefore, it is not necessary for the novice programmer to create new mental models for operator applications, as these models should already exist within the programmer.

Figure 10 illustrates a mental model that may be constructed by the novice programmer from the previous example of an operator application.

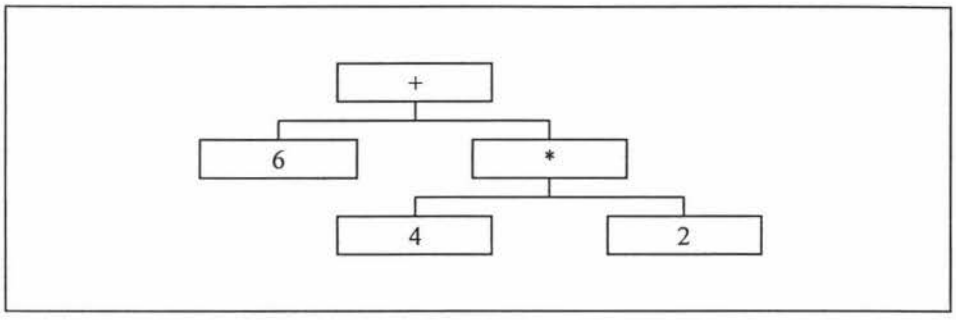


Figure 10: Mental Model of an Operator Application

2.1.3 Conditionals

A conditional is a program phrase, which selects one of a number of possible phrases, based on the value of an expression. For example:

```
if a > 0 then
      a * a
else
      a + 1
```

2.1.3.1 Mental Model - Conditionals

The above example evaluates the expression `a > 0`, the expression to next evaluate is dependant upon the resulting value. If the resulting value is `true` expression `a * a` is evaluated, if the resulting value is false expression `a + 1` is evaluated.

The goal is to aid the novice programmer with the construction of a mental model which is transferable and reflects the interaction between the three different expressions. This is achieved by:

- Basic Recurring Pattern - the simple 'if then else' syntax used, is the same syntax used in many other languages for conditionals. Using the same syntax in the ZL language, increases the conditionals level of processing, which also increases the transfer efficiency of the mental model from ZL to other languages.

- Well Connected - The if-then-else syntax is easily recognised from other languages. It is also very comprehensible, i.e. it is very easy to understand and follow - if this is

true, then do this, else if this is not true, do something else. This ensures that the mental model of the novice programmer is well connected, i.e. the interactions between the three different expressions are clearly understood.

Figure 11 illustrates how a novice user from the above example of a ZL conditional expression can construct a mental model for conditionals.

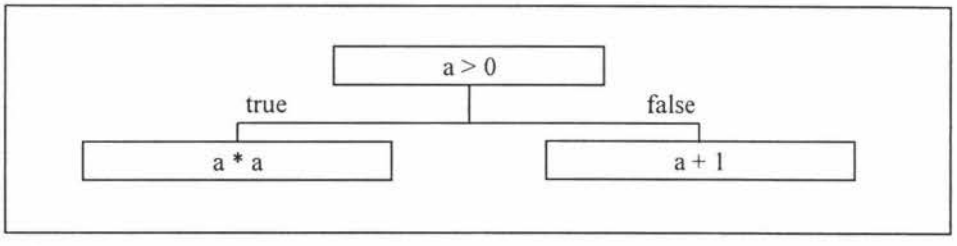


Figure 11 : Mental Model of a Conditional

As shown in Figure 11, the mental model a user may create in ZL should easily transfer from ZL to other languages.

2.1.4 Nesting

Expression nesting is when an expression of one type is placed inside the body of another expression of the same type. Function nesting is when one function is placed inside the body of another different function. Nesting is a very important concept for novice programmers to learn. This is because it is often the nesting of expressions and functions which give a programmer's mental model the hierarchical structure.

In ZL both expressions and functions can be nested. For expressions, this includes both tuple expressions and conditionals. For example:

```
fun compare (a, b) ->
   if a = b then
         1
   else
         if a > b
               2
         else
               3
```

```
fun swap (a, b) ->
   if compare (a, b) = 3 then
         (b, a)
   else
         (a, b)
```

2.1.4.1 Mental Model - Nesting

In the above example the `compare` function illustrates a nested conditional, and the `swap` function illustrates a nested function in ZL. The goal is to aid the novice programmer

with the construction of a mental model that reflects the concept of nesting. This is achieved by:

- Hierarchical structure - nested expressions and functions naturally have a hierarchical structure, they literally force the novice programmer to include depth in their mental model.

- Explicit Mappings - the links between the nested expressions and the nested functions are obvious to the novice programmer.

- Connection - nested functions are well connected. The simple interaction between the nested function and the outer function aids the understanding of connections within the novice programmer's mental model.

- Lazy Evaluation - ZL tuples use lazy evaluation, i.e. the elements within a tuple are not evaluated until their values are needed. A tuple with n elements forces the novice programmer to include breadth of at least size n. If the tuple's elements are not evaluated until necessary, then the novice programmer's mental model must also include depth for each tuple element. Thus, lazy evaluation of tuples aids the hierarchical structure of novice programmer's mental models.

Figure 12 illustrates how a mental model for nesting can be constructed by a novice user from the above examples of function nesting and conditional nesting.

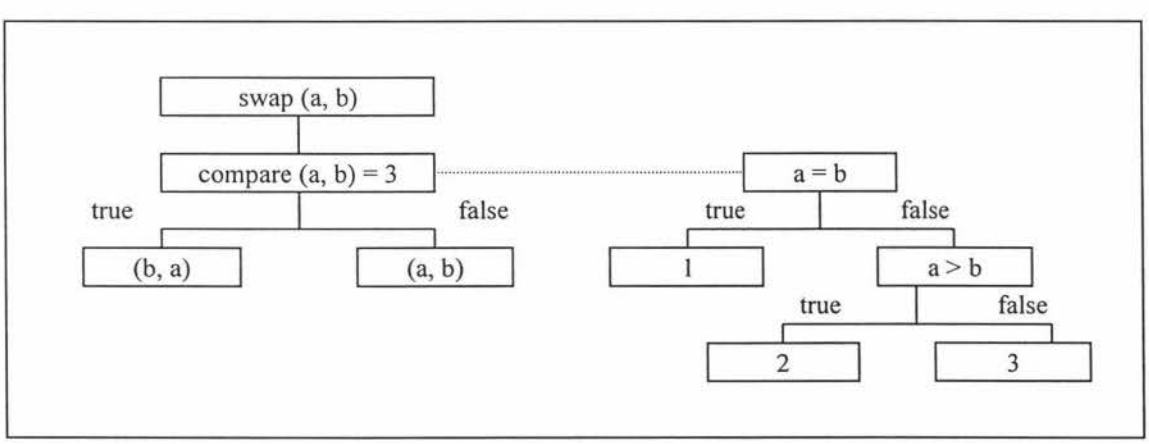


Figure 12 : Mental Model of Nesting

2.1.5 Iteration and Recursion

Recursion is where a function has the ability to call itself during the course of execution. Iteration is the repeated execution of a particular block of code, thus iteration can be defined as a simpler form of recursion.

As recursion is often the most natural way of expressing solutions to problems (Harrison, 1989) ZL directly supports recursion. As iteration is viewed as a less powerful form of recursion, it is only indirectly supported through the use of recursion. For example:

```
fun Fib (n) ->
      if n <= 2 then
            1
      else
            Fib (n-1) + Fib (n-2)
```

2.1.5.1 Mental Model - Iteration and Recursion

The above ZL example will return the fibonacci value of the number `n`, e.g. the fibonacci values of numbers 1 .. 9 are:

```
1, 1, 2, 3, 5, 8, 13, 21, 34
```

The fibonacci function stops after it returns one solution. However, to achieve a solution for a number > 2 it was necessary for the fibonacci function to call itself. This is the concept of iteration that it is necessary to convey to the novice programmer.

The goal is to the aid the novice programmer with the construction of a mental model which reflects the 'repetition' concept and can be easily transferred by the novice programmer from ZL to any other language. As stated earlier, it is the concept of 'repetition' that is important, not how it is achieved (through recursion in this case).

In ZL, the concept of 'recursion' is achieved by:

- Grounding - (refer to 1.3.1.5 Well Grounded) recursive functions are naturally well grounded, i.e. recursive code is local to the recursive function - there isn't any need to hunt for it.

- Strategic Planning - recursive functions require more thought and preparation than simple iteration. The novice programmer therefore has to prepare a strategic plan before they can code a solution (implementation plan). The strategic plan is naturally language independent, thus promoting beneficial transfer to other languages.

- Hierarchical Structure - physically placing the name of the function in the actual code aids the hierarchical structure of the novice programmer's mental model. This is because the novice programmer will place the mental model they have created for the function inside or below the mental model they are presently creating - their models will end up with a layered effect, i.e. one function on top of another.

- Explicit Mappings - physically placing the name of the function in the actual code not only aids the hierarchical structure of a mental model, but it also aids the links between the layers of the mental model, i.e. directly naming the function, directly names the links or 'loops' in this case.

Figure 13 illustrates how a user might construct a mental model for recursion using the ZL function Fib.

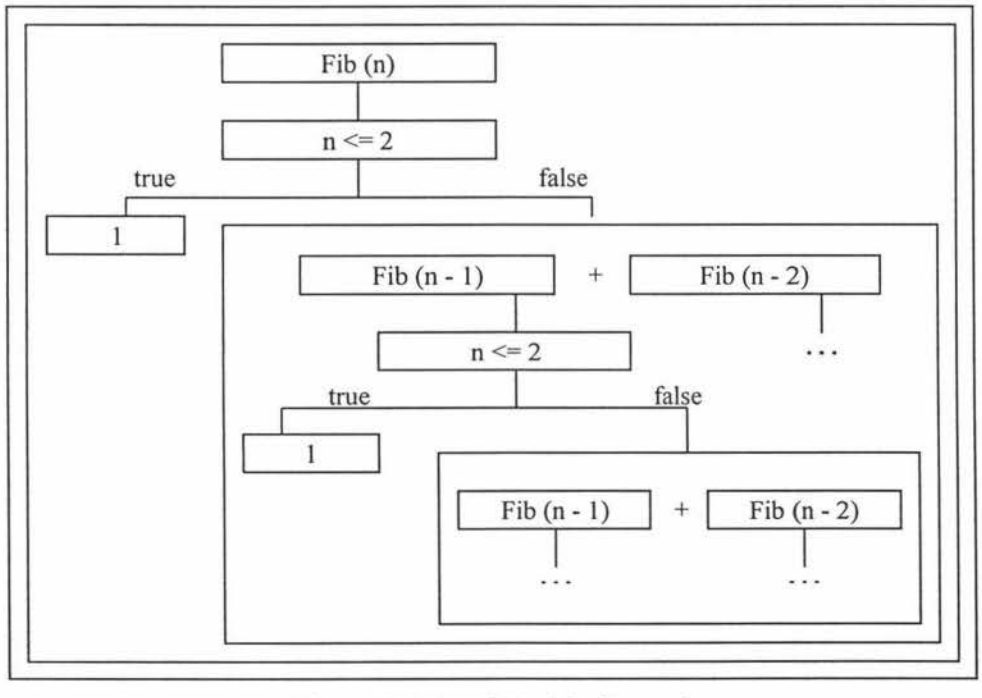


Figure 13 : Mental Model of Recursion

Note: the above illustration is only an assumption on the type of mental model created by a novice user. Each user is different, and, it must be stressed that the model used may differ from one user to another.

2.1.6 Pattern Matching

Pattern matching matches together the types and values of two patterns. A pattern match can either be successful or unsuccessful. If a pattern match is successful then some form of operation occurs, if it is unsuccessful then an error may result. ZL uses pattern matching specifically in qualified expressions and functions.

For example, the fibonacci function from section 2.1.5 Iteration and Recursion, can be rewritten using pattern matching:

```
fun Fib (n) ->
  if n <= 2 then
      1
  else
      Fib(n-1) + Fib(n-2)
```

```
fun Fib
    (1) -> 1
|   (2) -> 1
|   (n) -> Fib(n-1) + Fib(n-2)
```

2.1.6.1 Mental Model - Pattern Matching

The above ZL example shows the fibonacci function rewritten using pattern matching. Both functions take exactly the same arguments and both output exactly the same result. The goal is to aid the novice programmer with the construction of a mental model which reflects the concept of pattern matching. This is achieved by:

- Planning - as shown in the above example, the fibonacci function can be coded in ZL by using pattern matching or by using a conditional expression. In this case, the novice programmer is forced to making a decision on which is the best solution. This promotes both the strategic and implementation stages of planning.

- Syntax - the symbol `->` is a visual aid to the novice programmer, i.e. it visually states 'from `0` return `1`', 'from `1` return `1`', etc ...

- Semantics - pattern matching is clear and simple to understand. Novice programmers can easily identify with the concept of 'matching', which aids the novice programmer in forming a mental model of the pattern matching concept.

- Strict Evaluation - a pattern type can only contain elements of a basic or pattern type (refer to section 3.2 Types). Therefore, if the value of an expression is to be bound to a pattern element, then the expression must first be evaluated. Ensuring that the expression is evaluated before pattern matching begins, helps focus the novice programmer on the concept of pattern matching and not on the evaluation of expressions.

Figure 14 illustrates how a mental model for pattern matching can be constructed by a novice user from the above example of a ZL fibonacci function with pattern matching.

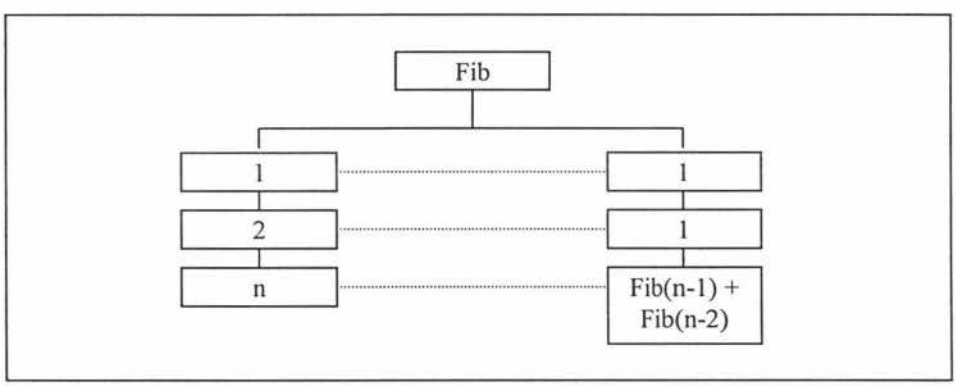


Figure 14 : Mental Model of Pattern Matching

2.2 MENTAL MODELS AND THE ZL ENVIRONMENT

Figure 15 shows a screen shot of the ZL environment.

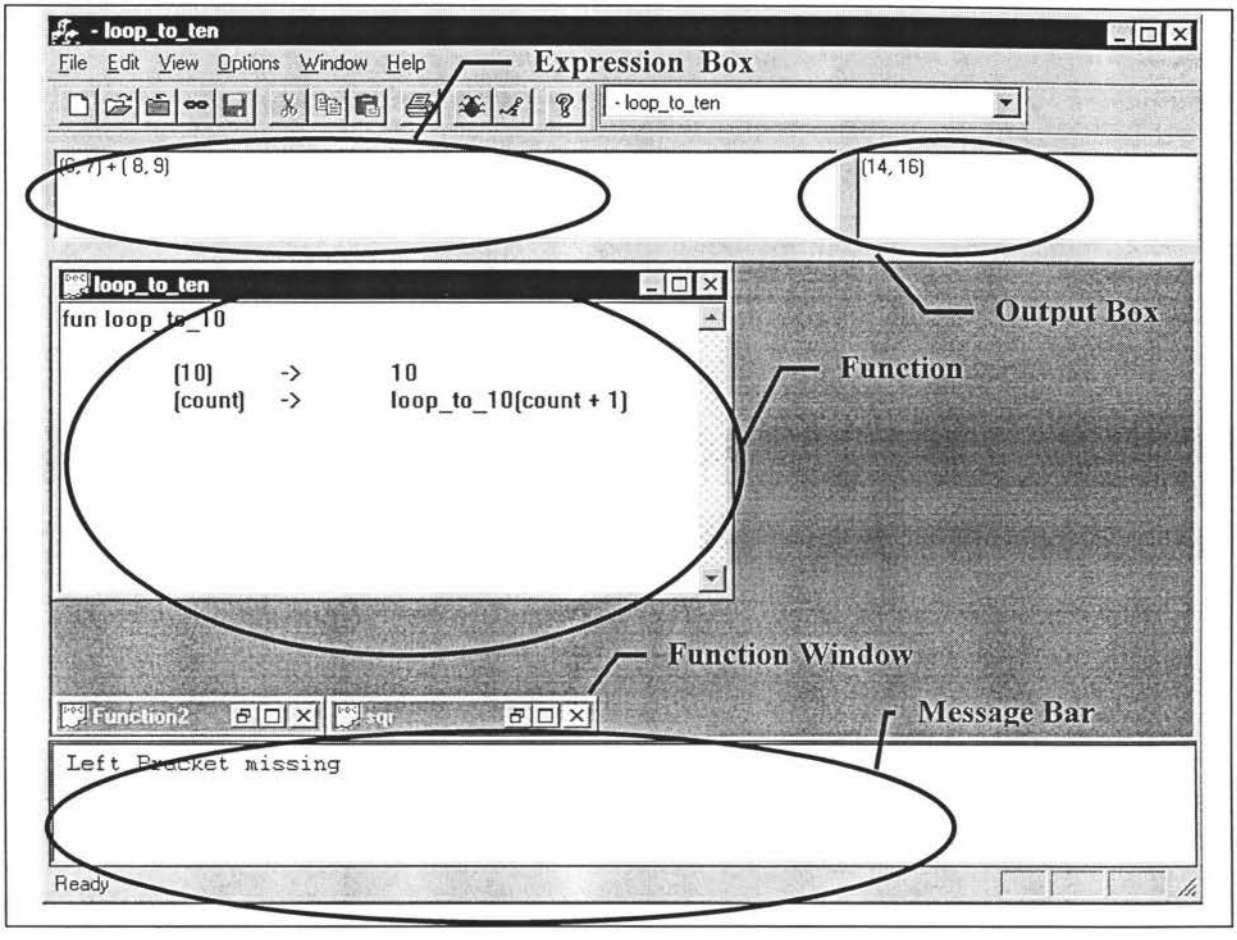


Figure 15 : The ZL Environment

The ZL environment is designed to be functional, yet easy to use. This was achieved by:

- Stationary Windows
- Microsoft Standard
- Relating Iconic Pictures to Tool Bar Button Functions
- Separate Window for each ZL Function

2.2.1 Stationary Windows

There are only four different windows used in the ZL environment:

- The Expression Box
- The Output Box
- The Function Window
- Message Bar

Three of the above four windows remain stationary and cannot be moved around the screen by the user. Keeping the windows stationary has three benefits:

- Avoids confusion about the function of each window, i.e. the window at the bottom of the screen will always be the message window, the novice user cannot confuse it with any other window.

- Places emphasis on the only moveable window - the function window, thus emphasizing the perceived importance of that window, i.e. 'Open', 'Save', and 'Print', from the file menu, refer to the function window.

- Avoids unnecessary clutter. Many programming environments tend to over clutter a computer screen with too much information, this can lead to confusion for novice users.

2.2.2 Microsoft Standard

The ZL environment is designed for use on an IBM compatible Personal Computer (PC). Most PC's use a common software brand, Microsoft, which has its own standard, or rules, for designing software environments. The ZL environment conforms to the Microsoft standard in the areas of:

- Tool bar buttons
- Menu Titles and Menu Items
- Window minimisation icons

There are other areas of the ZL environment in which the Microsoft standard is followed, however only the above are the most obvious to novice programmers.

The Microsoft standard is followed in the ZL environment because as previously stated it is commonly known among IBM compatible PC users. This means that the Microsoft icons and menus should already be deeply encoded within the programmer's memory (from using other products such as Microsoft Word, Microsoft Excel, etc). Using the

same icons and menus in the ZL environment is, therefore, an advantage as they promote recall about the behaviour of the buttons and menus, and they also give the ZL environment a nice friendly feeling.

2.2.3 Relating Iconic Pictures to Toolbar Button Functions

All but three of the tool bar buttons on the main tool bar use standard Microsoft icons. These three buttons are the hide button, the debug button, and the run button.

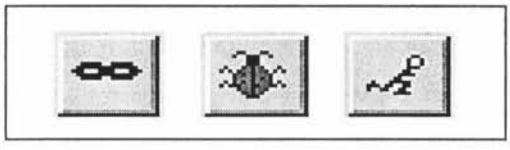


Figure 16 : The Hide, Debug, and Run Buttons

As shown in Figure 16, the three buttons have iconic pictures which match their names to their tasks. Matching iconic pictures to button tasks promotes recall about button behaviour.

- Hide Button: icon - a mask. Used for hiding function windows from the user's view.

- Debug Button: icon - a bug. Used for 'debugging' functions. The button is actually responsible for doing much more than simply removing errors from functions. However, from the user's perspective, the only action performed is the identification of 'bugs' in their functions – hence the name 'debug'.

- Run Button: icon - running stick man. This button is used for executing or 'running' an expression.

2.2.4 Separate Window for each ZL Function

Although the ZL environment was designed specifically to be simple and easy to use, it also indirectly aids the novice programmer with the construction of their mental models. This is because a unique feature of the ZL environment is that each function has its own window, i.e. one function one window. Placing functions in separate windows encourages the novice programmer to form breadth in their mental model. It also ensures that the user maintains functions as separate entities, which in turn aids the identification of basic recurring patterns in mental models.

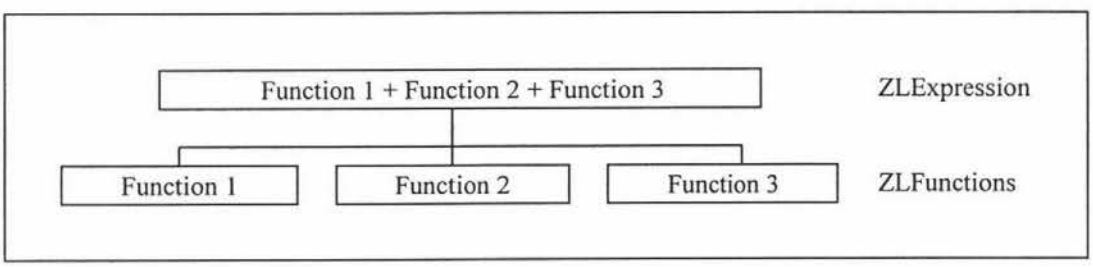


Figure 17 : ZL Expressions and Functions in a Novice Programmer's Mental Model

3. THE ZL LANGUAGE

3.1 OVERVIEW

An algorithm describes how a task is accomplished. With respect to a computer, an algorithm describes the operations a computer performs to complete a task.

The functional programming paradigm views an algorithm as a mathematical function from its input to its output (Goldschlager & Lister, 1988). For example, an algorithm to add two numbers (n_1 and n_2) can be regarded as a mathematical function 'sum' with input and output.

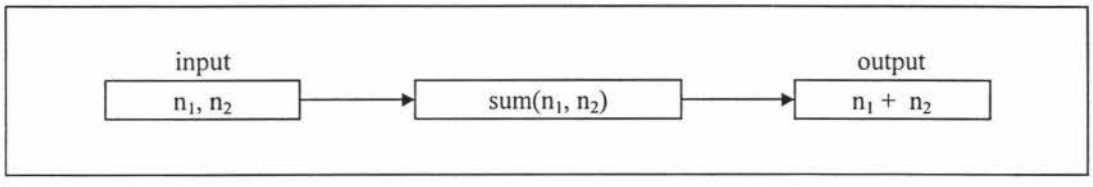


Figure 18 : Functional Perspective of the Sum Algorithm

The ZL language is a functional language, which similar to the above example, maps input values to output values through the use of functions and expressions.

3.2 TYPES

Every value in the ZL language has a type. Types are not declared in ZL, they are instead derived from assumptions made about ZL values.

3.2.1 Basic Types

ZL has two basic types; number and boolean. The domains of the basic types are:

```
number      the integers,
boolean     the boolean values true and false.
```

In ZL any value which falls within either of the above domains is given the type of that domain, i.e. if a value is within the domain of integers, then that value will have the type 'number'. Conversely, if a value is within the boolean domain, then that value will have the type 'boolean'.

3.2.2 Tuple Types

A tuple type is the product of n ≥ 2 types. The number of elements contained within a tuple is referred to as the tuple's size. It is possible to create tuples within tuples. For example, these are all tuple types:

```
(number x number)                - an integer tuple of size 2.
(number x boolean x number)      - a mixed type tuple of size 3.
(number x (boolean x number x number))
                                 - a mixed type tuple of size
                                 2, where the second type is a
                                 mixed type tuple of size 3.
```

3.2.3 Pattern Types

A pattern type is a product of n ≥ 1 basic or pattern types, i.e. a pattern can contain basic types and other pattern types. For example, these are all pattern types:

```
 number
(boolean)
(number x number)
(number x (boolean x number))
```

The pattern type differs to the tuple type because the pattern type's elements must either be of basic or pattern type. The pattern type can also contain only one element in comparison to the two elements necessary for a tuple type.

3.2.4 Function Types

Function types have both a domain and a range:

```
<function type>   ::=   <type1> -> <type2>
```

Where $type_1$ is the domain and $type_2$ is the range. The domain of a function type is a pattern type of size n. The range of the function type can be any ZL type. For example:

```
 number -> boolean
(number x number) -> number
(number x (number x number)) -> number
```

3.3 EXPRESSIONS

There are three classes of expression in the ZL language: simple expressions, qualified expressions, and application expressions:

```
<expression>      ::=   <simple>
                  |     <qualified>
                  |     <application>
                  |     '(' <expression> ')'
```

3.3.1 Simple Expressions

Simple expressions denote numbers, booleans, and tuples:

```
<expression>      ::=   <simple>
<simple>          ::=   <literal>
                  |     <tuple>
```

3.3.1.1 Literal Constants

Literal constants denote numbers and booleans:

```
<literal>         ::=   <integer literal>
                  |     <boolean literal>
<integer literal> ::=   {0|1|..|9}+
<boolean literal> ::=   true | false
```

ZL does not contain a character type, therefore character literals cannot be included as simple expressions. Integer literals denote values of type number. If x is an integer literal then x has the type number:

$$x : number$$

Boolean literals denote values of type boolean:

$$true : boolean$$
$$false : boolean$$

Examples of Literal Constants:

```
9, 467
true, false
```

3.3.1.2 Tuple Expressions

Tuples contain two or more elements. They are constructed using parentheses, where each element within the tuple is separated by a comma.

$$\langle tuple \rangle ::= \text{`('} \langle expression_1 \rangle \{\text{`,'}\ \langle expression_n \rangle\}^+ \text{`)'}$$

The type of the tuple expression is the product of the types of the argument expressions. The rule has n a premises.

$$\frac{A \vdash expression_1 : \sigma_1 \ \ldots\ A \vdash expression_n : \sigma_n}{A \vdash (expression_1, \ldots, expression_n) : \sigma_1 \times \ldots \times \sigma_n}$$

ZL tuples use lazy evaluation, where the elements of a tuple are not evaluated until it is necessary to use their values.

Examples of Tuple Expressions:

```
(true, false)
(1 + 2, 4)
(sqr(7), (false, true, (6, 9)))
(let a = sqr(7) in  a*a, let b = sqr(b) in  b*b)
```

3.3.2 Qualified Expressions

Qualified expressions allow values to be given a name. A qualified expression is constructed in ZL using a 'let' expression:

```
<qualified>       ::=   <let exp>
<let exp>         ::=   'let' <letelement1>
                              { '&' <letelementn> }*
                              'in' <expression1>
<letelement>      ::=   <pattern> '<-' <expression2>
```

Both the pattern (refer to section 3.4 Patterns) and the expression of a let element must be of the same type, i.e. `expression2` must have the same number and type of elements as `pattern`:

$$\frac{A \vdash \texttt{expression}_2 : \sigma \qquad A \vdash \texttt{pattern} : \sigma}{A \vdash (\texttt{pattern} \texttt{ <- } \texttt{expression}_2) : \sigma}$$

Qualified expressions are strict, i.e. `letelement1` and `letelementn` are evaluated before `expression1`, e.g. `expression2` is evaluated and the resulting value is bound to `pattern` before `expression1` can be evaluated. When `expression1` is evaluated each pattern variable (identifier) within `expression1` is replaced with its associated bound value.

The type of the `pattern`, and thus the types of the elements within the pattern are deduced from the type of `expression2`, refer to section 3.4, for a more detailed description of `pattern`. The type of the let expression is given by the type of the `expression1`:

$$\frac{A \vdash \texttt{expression}_2 : \sigma \qquad A.\texttt{pattern} : \sigma \vdash \texttt{expression}_1 : \tau}{A \vdash (\texttt{let pattern <- expression}_2 \texttt{ in expression}_1) : \tau}$$

Examples of Qualified Expressions:

```
let a       <- 5
                  in sqr(a)

let sqra    <- let a <- 5
                        in sqr(a)
                  in sqr(sqra)

let (a, b)  <- (sqr(2), sqr(4))
&   (c, d)  <- (sqr(4), sqr(2))
                  in isEql((a, b), (b, c))
```

3.3.3 Application Expressions

There are five different classes of application expression:

```
<application>     ::=   <conditional>
                  |     <funapplication>
                  |     '-' <expression1>
                  |     'not' <expression1>
                  |     <opapplication>
```

The type of argument n to '-' and its result must be number:

$$\frac{A \vdash n : number}{A \vdash (-n) : number}$$

The type of argument n to 'not' and its result must be boolean:

$$\frac{A \vdash n : boolean}{A \vdash (not\ n) : boolean}$$

The value of the resulting expression for the above operators '-' and 'not' is obtained by evaluating argument n and applying this result to the operator.

Examples:

```
not true
- 37
```

3.3.3.1 Conditional Expressions

```
<conditional>     := 'if' <expression1>
                        'then' <expression2>
                        'else' <expression3>
```

The type of $expression_1$ must be boolean. The types of $expression_2$ and $expression_3$ must be the same. The type of the conditional expression is that of expressions 2 and 3:

$$\frac{A \vdash exp_1 : boolean \quad A \vdash exp_2 : \tau \quad A \vdash exp_3 : \tau}{A \vdash (if\ exp_1\ then\ exp_2\ else\ exp_3) : \tau}$$

A conditional expression will first evaluate the boolean expression and the either $expression_2$ or $expression_3$. If the boolean expression ($expression_1$) evaluates true, then $expression_2$ is evaluated. If the boolean expression evaluates false, then $expression_3$ is evaluated.

Examples of Conditional Expressions:

```
if a = 0 then a else a - 1
if a = 0 then true else a - 1        - illegal, as the then and
                                     else expressions are not of
                                     the same type
```

3.3.3.2 Function Applications

A function application is constructed using an identifier (the name of the function) and an expression (the argument for the function):

```
<funapplication>  :=    <identifier> <expression>
```

The `identifier` has a function type, where the domain type is the type of the `expression` and the range type is the type of the function application:

$$\frac{A \vdash \texttt{identifier} : \sigma \rightarrow \tau \qquad A \vdash \texttt{expression} : \sigma}{A \vdash (\texttt{identifier expression}) : \tau}$$

The `expression` is evaluated first, before the function named by `identifier`. The value of `expression` is then used as an argument to the function (`identifier`). The function is then evaluated and the resulting value is given to the function application.

Examples of Function Applications:

```
sqr(2)
sum(sqr(2), 2)
```

3.3.3.3 Operator Applications

An operator application applies an operator to two expressions.

```
<opapplication>   :=    <expression1> <operator> <expression2>
```

The `operator` is associated with two types; the domain type and the range type. The types of $\texttt{expression}_1$ and $\texttt{expression}_2$ are determined from the operator domain type. The type of the operator application expression is determined by the operator range type. For example, the operator expression '`1 + 2`' has both a domain and range type of number.

$$\frac{A \vdash \texttt{operator} : \sigma_1 \times \sigma_2 \rightarrow \tau \qquad A \vdash \texttt{exp}_1 : \sigma_1 \qquad A \vdash \texttt{exp}_2 : \sigma_2}{A \vdash (\texttt{exp}_1\ \texttt{operator}\ \texttt{exp}_2) : \tau}$$

Table 1 lists the domain and range types for operators within ZL:

Table 1 : Operator Domain and Range Types

Operator	Domain	Range
+	number x number	number
*	number x number	number
/	number x number	number
-	number x number	number
<	number x number	boolean
>	number x number	boolean
=	number x number	boolean
<>	number x number	boolean
<=	number x number	boolean
>=	number x number	boolean
and	boolean x boolean	boolean
or	boolean x boolean	boolean

The value of the operator application is the resulting value of applying the operator to `expression1` and `expression2`. The usual precedence rules apply to operators within ZL.

In ZL, operators which have a domain type and range type of number can not only be applied to simple expressions, but also to tuple expressions. However, both tuple expressions applied to the operator must have the same tuple type. The resultant tuple type is that of the expressions, for example:

<opapplication> ::= <tuple$_1$> <operator> <tuple$_2$>

$(e^1_1, \ldots, e^1_n)$ op $(e^2_1, \ldots, e^2_n) \Rightarrow (e^1_1$ op $e^2_1, \ldots, e^1_n$ op $e^2_n)$

The value of the operator application in the above case, is a tuple of evaluated tuple elements. The expressions (or elements) within $\texttt{tuple}_2$ are evaluated first, then the expressions within $\texttt{tuple}_1$. Each evaluated expression within $\texttt{tuple}_2$ is then applied together with the operator to the corresponding evaluated expression within $\texttt{tuple}_1$ to obtain the resulting tuple. For example:

```
Expression to be evaluated:   (sqr(2), 6)  +  (3, 4 + 5)
Evaluate the tuple elements:  (4, 6) + (3, 9)
Evaluate resulting tuple:     (4 + 3, 6 + 9)
Result:                       (7, 15)
```

Examples of Operator Applications:

```
1 + 1 - 3
(3, 4) + (4, 3)
(((7 * 9), (6 * 7)) - (2, 3))
```

3.4 PATTERNS

Similar to the syntax of a tuple, a ZL pattern is constructed using parentheses, where each element within the pattern is separated by a comma:

```
<pattern>       ::=   [(] <patelement1> {','<patelementn>}* [)]
<patelement>    ::=   <literal>
                |     <identifier>
                |     <pattern>
```

A ZL pattern can contain multiple literals of the same value, however, multiple identifiers of the same name are not allowed, i.e. each identifier within a pattern must be unique.

Patterns are used in qualified expressions and function declarations. The scope of an identifier within a qualified expression is the expression on the right hand side of the qualified expression, i.e. <expression$_1$> - the expression succeeding 'in'. Thus, if there are multiple letelements in a qualified expression, then each identifier contained within each letelement pattern must be unique. For the scope of identifiers in function declarations refer to 3.5 Functions.

The type of the pattern is the product of the types of the pattern elements. The rule has n premises.

$$\frac{A \vdash patelement_1 : \sigma_1 \quad \ldots \quad A \vdash patelement_n : \sigma_n}{A \vdash (patelement_1, \ldots, patelement_n) : \sigma_1 \times \ldots \times \sigma_n}$$

3.4.1 Pattern Matching

Pattern matching is used to match a pattern to an expression. A match between the pattern and expression occurs if the expression type is a specialisation of the pattern type. Pattern matching occurs in ZL qualified expressions and ZL functions.

Pattern matching is used in qualified expressions to match the elements of `pattern` to $expression_2$:

```
<let exp>          ::=   'let' <letelement1>
                               { '&' <letelementn> }*
                               'in' <expression1>
<letelement>       ::=   <pattern> '<-' <expression2>
```

Pattern matching is used in functions to match the elements of an expression in a function application to a pattern in a function:

```
<funapplication>   :=    <identifier> <expression>
<function>         ::=   'fun' <identifier> <funelement1>
                               { '|' <funelementn> }*
<funelement>       ::=   <pattern> '->' <expression>
```

For more information on functions refer to section 3.5 Functions.

To pattern match a pattern to an expression it is necessary to:

1. Find the pattern type, this includes finding the type of each element and subpattern contained within the pattern. If the pattern contains an identifier element, then the identifier receives an unknown type, i.e. its type is set to unknown.

2. Find the type of the expression.

3. Unify the pattern with the expression. Unifying the pattern to the expression involves matching of the pattern elements to the type of the expression:

   ```
   <patelement>       ::=   <literal>
   ```

   If the pattern element is a literal then a match will succeed against the expression if the value of the expression is the same as the value of the literal.

   ```
   <patelement>       ::=   <identifier>
   ```

   If the pattern element is an identifier then a match will succeed against the expression for any value. Once matched the type and value of the identifier is bound to the type and value of the expression.

```
<patelement>          ::=   <pattern>
```

If the pattern element is another pattern then a match will succeed against the expression if each subpattern element of the pattern matches that of the expression.

3.5 FUNCTIONS

A ZL function definition has an identifier (the function name) and one or more pattern based definitions known as function elements. Each function element (funelement) consists of a pattern and an expression.

```
<function>        ::=   'fun' <identifier> <funelement1>
                              { '|' <funelementn> }*
<funelement>      ::=   <pattern> '->' <expression>
```

A function element has a function type, where `pattern` type is the domain and the type of `expression` is the range:

$$\frac{A \vdash \texttt{pattern} : \sigma \quad A \vdash \texttt{expression} : \tau}{A \vdash \texttt{(pattern -> expression)} : \sigma \texttt{ -> } \tau}$$

The scope of an identifier contained in a funelement `pattern` is the funelement `expression`.

Each function element within a ZL function may contain different values but they must all have the same function type. Therefore, all function elements within a function are unified before the function is evaluated. The type of `identifier` and the actual function is the same as the function elements:

$$\frac{A \vdash \texttt{funelement}_i : \sigma \texttt{ -> } \tau \quad A \vdash \texttt{funelement}_n : \sigma \texttt{ -> } \tau}{A \vdash \texttt{(fun identifier funelement}_i \ldots \texttt{funelement}_n\texttt{)} : \sigma \texttt{ -> } \tau}$$

Although a ZL function may have many function elements, each with its own expression, only one of these expressions will ever be evaluated through a function application.

The decision on which expression to evaluate is made by pattern matching the function application's expression to each function element's pattern (starting from the topmost funelement):

```
<funapplication>  :=    <identifier> <expression>
<function>        ::=   'fun' <identifier>
                              <pattern> '->' <expression>
                        |     <pattern> '->' <expression>
                        |     <pattern> '->' <expression>
                        |     <pattern> '->' <expression>
```

The first function element found which possesses a pattern that matches the function application's expression will have its associated expression evaluated. The result of the evaluated expression is the final value of the function.

Examples of Function Definitions:

```
fun sqr
      a       ->    a*a

fun loop_to_10
      10            ->    10
      count         ->    loop_to_10(count + 1)

fun fac
      0       ->    1
      n       ->    n * fac(n-1)

fun fac2
      (0, n)        ->    n
      (n, m)        ->    fac2(n-1, n*m)
```

This page intentionally left blank

4. THE ZL ENVIRONMENT

4.1 OVERVIEW

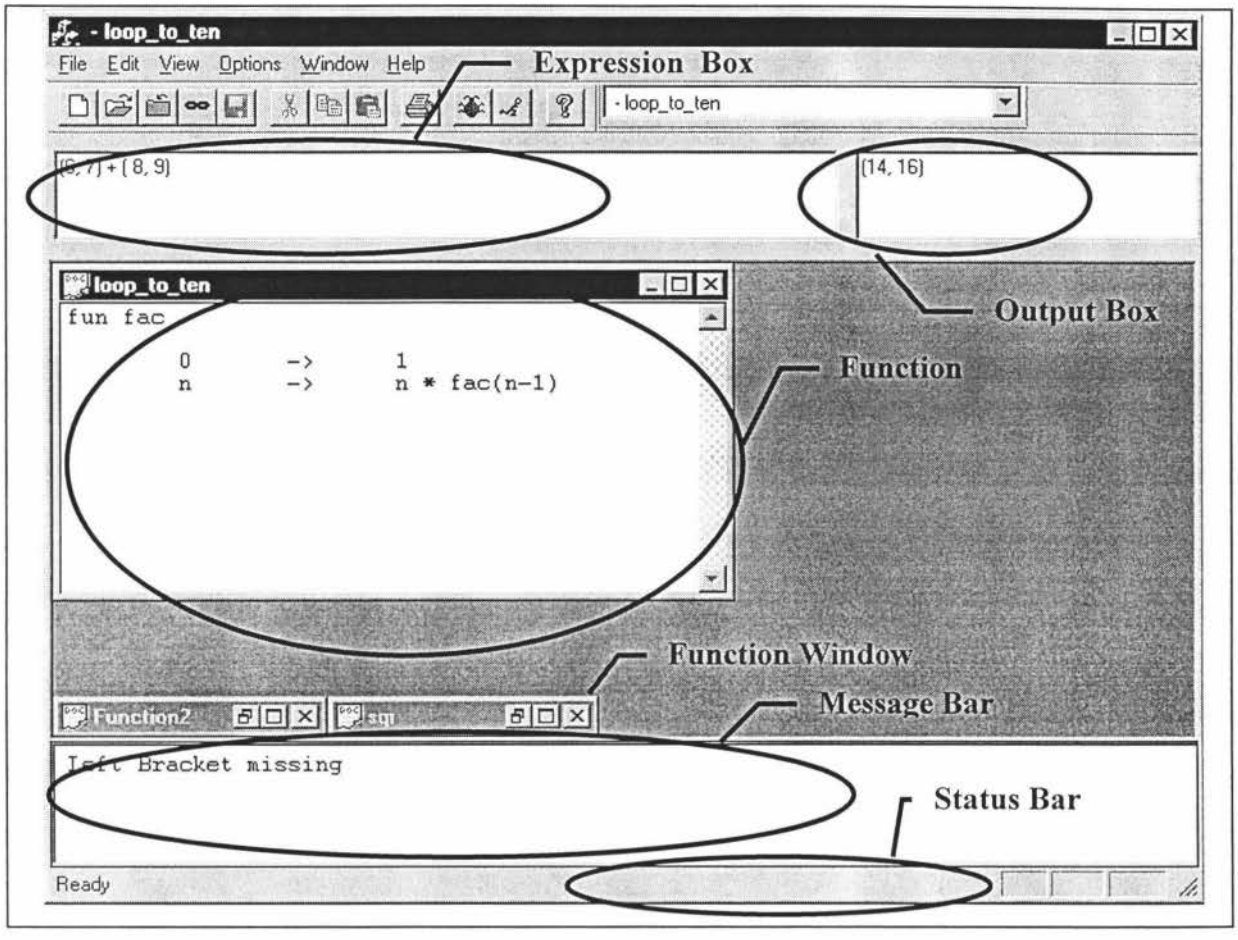


Figure 19 : The ZL Environment

Figure 19 shows a typical screen shot of the ZL environment, describing a state in which several functions have been loaded, some of which have been iconised. The current function is being edited, and the output window shows the results of a previous evaluation.

Some of the features of the ZL environment include:

- Menubar
- Toolbar
- Status Bar
- Function Toolbar
- Expression Bar
- Message Bar

4.1.1 The Menubar

The menubar for the ZL Application contains six pull-down menus, Figure 20.

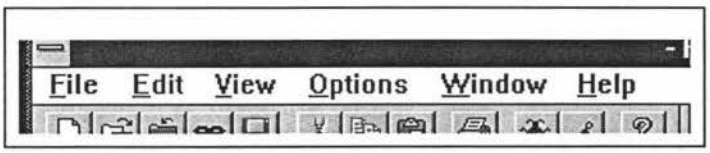


Figure 20 : The Menubar

The 'File' menu contains the following options:

- New: creates a new function window.
- Open: opens an existing function.
- Close: closes the active function.
- Save: saves the active function.
- Save As: saves the active function with a new name.
- Print: prints the active function.
- Print Setup: printer settings.
- Exit: exits the ZL Application

The 'Edit' menu contains the following options, these options apply to the function window and the expression bar:

- Undo: undo the previous edit command.
- Cut: cut the selected text and place it in the clipboard.
- Copy: copy the selected text to the clipboard.
- Paste: paste the text from the clipboard at the correct insertion point.

The 'View' menu contains options to display and hide the tool, status, function, expression, and message control bars.

The 'Options' menu contains the following options:

- Debug: debug the active function, i.e. check for errors.
- Run: evaluate the current expression in the expression bar.

The 'Window' menu contains the following options:

- Hide: hides the currently active function window from the user's view.
- Cascade: arranges all open function windows in an overlapping format.
- Tile: arranges all open function windows in a non-overlapping format.
- Arrange Icons: arranges all iconic function windows.

The 'Help' menu contains only one item - 'About ZL'. This item displays an 'About box', showing the current version of the ZL Application.

4.1.2 The Toolbar

As shown in Figure 21 the main toolbar from the ZL environment uses standard icons.

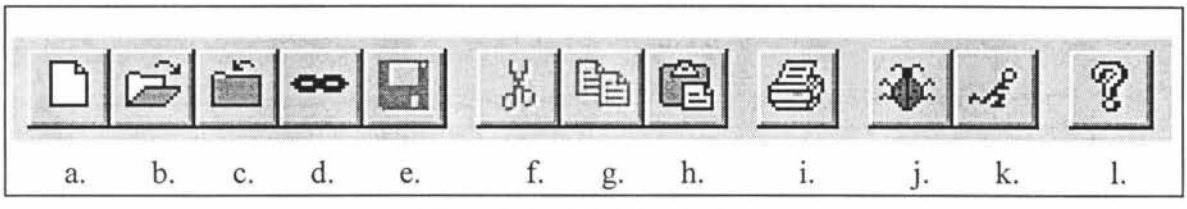


Figure 21 : The Main Toolbar

Each toolbar button provides quick access to an item in one of the pull-down menus:

a,b,c The New, Open, and Close buttons, correspond to the New, Open and Close options in the File menu.

d. The Hide button, corresponds to the Hide option in the Window menu.

e. The Save button, corresponds to the Save option in the File menu.

f,g,h. The Cut, Copy, and Paste buttons, correspond to the Cut, Copy and Paste options in the Edit menu.

i. The Print button, corresponds to the Print option in the File menu.

j,k. The Debug and Run buttons, correspond to the Debug and Run options in the Options menu.

l. The Help button, corresponds to the Help button in the help menu.

Each button within the main tool bar contains a tooltip. A tooltip is a small piece of text that describes the function of the button. The tooltip is displayed to the user when the mouse moves over the button.

4.1.3 Status Bar

The ZL environment contains a Status Bar, located at the very bottom of the main window, Figure 22.

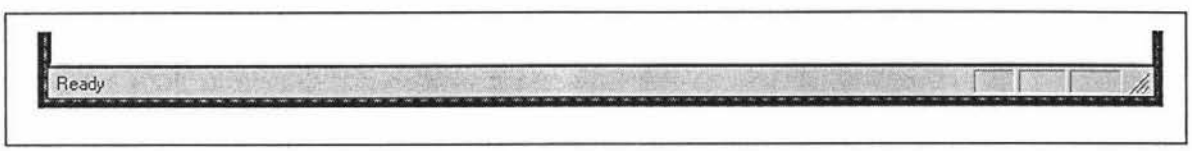


Figure 22 : The Status Bar

When a toolbar button, or a menu item is selected, the status bar will display simple messages that briefly explain the function of this selection. The status bar also displays functions which the application may be carrying out in the background, e.g. 'Opening ...' or 'Autosaving ...'.

The status bar also contains three small boxes (Figure 22). These boxes indicate whether the 'Caps Lock', 'Num Lock', or 'Scroll Lock' buttons have been selected on the keyboard.

4.1.4 Function Toolbar

The task of the Function Toolbar is to maintain and display a list of the currently active functions. These are the user-defined ZL functions available for use.

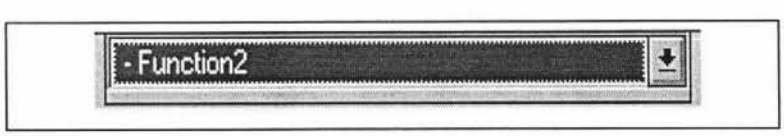


Figure 23 : The Function Toolbar

When a function is created, the name of the function is placed in the function list. The function name remains in the list, until the window containing the function is closed.

If a function is selected from the toolbar, the window containing the function is made the current window, i.e. given the 'input focus'. If the window is either hidden or iconised, then it is redisplayed, activated, and made the current window.

4.1.5 Expression Bar

The Expression Bar contains two text boxes, Figure 24.

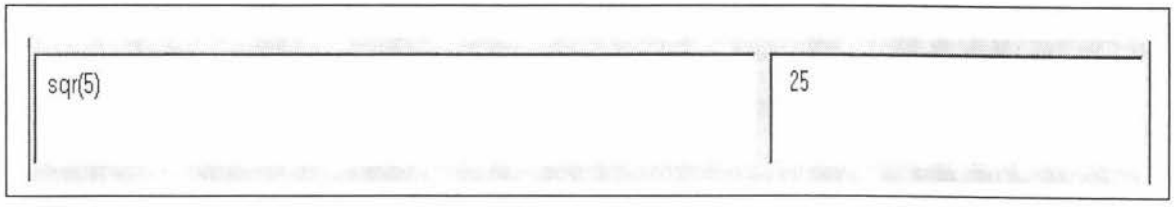


Figure 24 : The Expression Bar

The text box on the left-hand side is the expression box. This is used to enter ZL expressions. The text box on the right-hand side is the output box. This is used to display the result of an evaluated expression.

The output box is cleared when the 'run button' from the toolbar is selected. It will not show any output unless an expression has been correctly evaluated.

4.1.6 Message Bar

The Message Bar is used to convey messages to the user. These include error messages from the lexer, parser, typechecker and evaluator.

The message bar is permanently anchored to the bottom of the main window, Figure 25

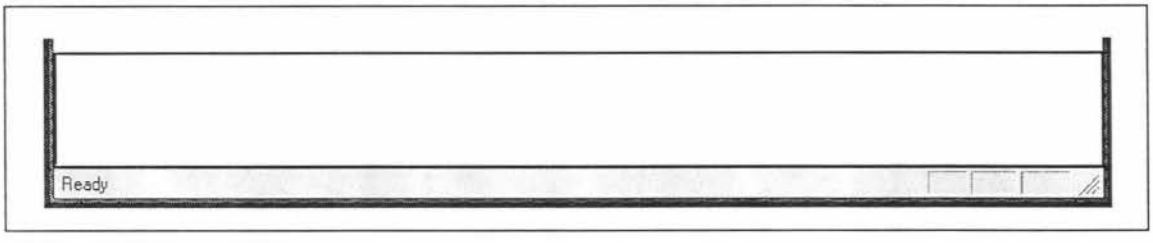


Figure 25 : The Message Bar

The message bar is cleared when either the 'run' or 'debug' buttons from the toolbar are selected.

4.1.7 Function Window

A Function Window is used to enter and display a single ZL function.

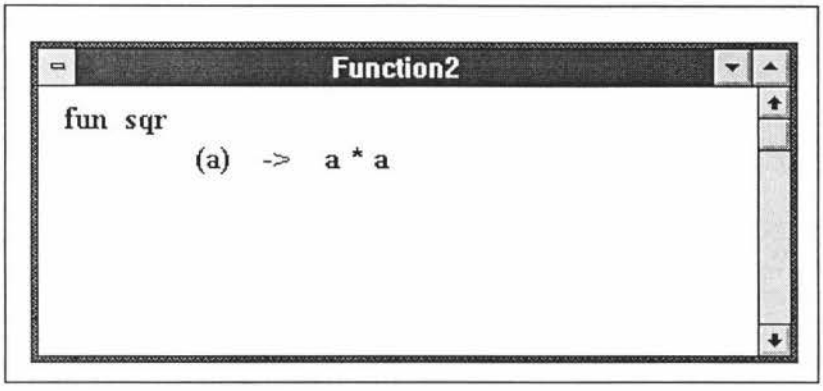


Figure 26 : The Function Window

To ensure that a function is error free, it must be debugged before it is used. This can be achieved by either selecting the 'debug' button on the toolbar or by selecting 'debug' from the 'Options' menu.

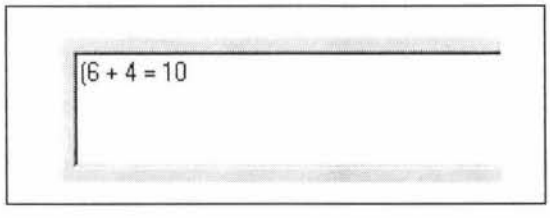
4.2 Using The ZL Environment

The next section describes how the user interacts with the system in evaluating an expression.

4.2.1 Entering and Evaluating an Expression

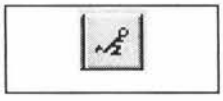
4.2.1.1 Entering an Expression:

- Select the expression box in the expression bar.
- Enter an expression into the expression box.

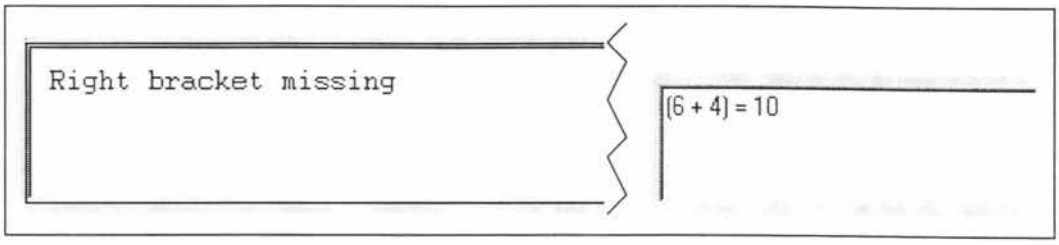


4.2.1.2 Evaluating an Expression

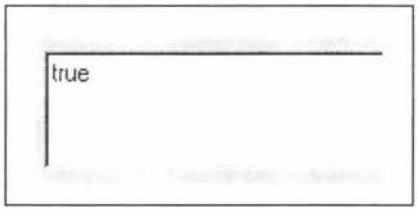
- Select the 'Run' button on the toolbar, or select the 'Run' option from the 'Options' menu.



- Fix any error messages displayed in the message bar and re-evaluate the expression.



- View the result in the output box.

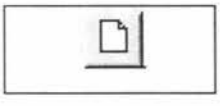


4.2.2 Entering and Evaluating a Function

Each function requires its own function window. Thus, in order to create a new function it is necessary to create a new function window.

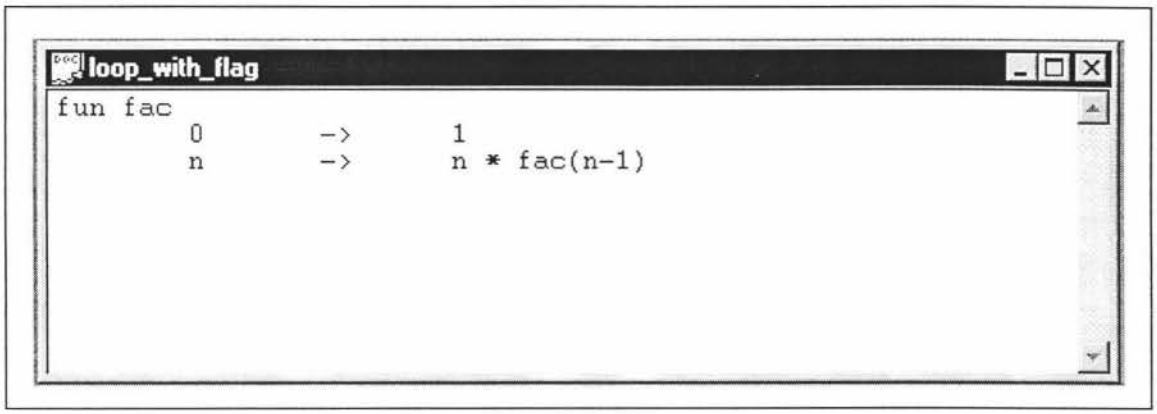
4.2.2.1 Creating a New Function

- Select the ' New' button on the toolbar, or select the 'New' option from the 'File' menu.



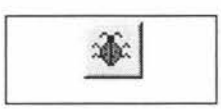
Note: when the ZL Application is started, an empty function window is automatically created.

- Enter the new function into the new function window



4.2.2.2 Debugging a Function

- Select the 'Debug' button on the toolbar, or select the 'Debug' option from the 'File' menu.



- Fix any error messages displayed in the message bar and again debug the function

Note: A function cannot be evaluated until it has been correctly debugged. This ensures that it has no static errors.

4.2.2.3 Using a Predefined Function

- Select the 'Open' button on the toolbar, or select the 'Open' option from the 'File' menu.

- Using the 'Open' dialog box provided, select the desired function.

- Select the 'Debug' button on the toolbar, or select the 'Debug' option from the 'File' menu to debug the function.

Note: Although an existing function may have been previously debugged, it is necessary to debug that function again before it can be used in the present ZL session. This is to ensure that the function has not changed in any form since its previous use and to guarantee it is error free.

This page intentionally left blank

5. IMPLEMENTATION

5.1 INTRODUCTION TO DESIGN METHODOLOGY

The Unified Method (Booch and Rumbaugh, 1995) is the design methodology used to design the implementation of the ZL programming language and the ZL environment. As many of the diagrams and illustrations for ZL contain symbols and elements which are unique to the Unified Methodology a brief overview of the methodology follows.

5.1.1 The Unified Method

The Unified Method is a method for specifying, visualising, and documenting the artifacts of an object-oriented system under development (Booch & Rumbaugh 1995). The Unified Method was chosen over other object oriented methodologies, as it unifies the two most popular object oriented design methods - OMT (Rumbaugh, 1991) and Booch (Booch, 1991).

Below is a brief description of some of the basic concepts from the Object Oriented Methodology and the Unified Method (Version 0.8).

5.1.1.1 Classes

Booch and Rumbaugh (1995) describe a class as being 'a definitional entity that has instances with identity'. For example, a car and a boat are both instances of the class 'transport vehicle'.

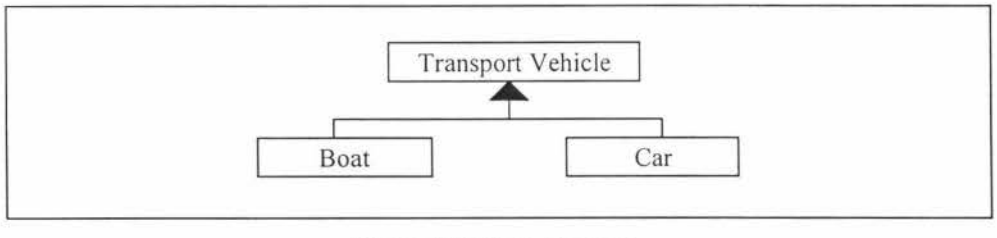


Figure 27 : Class Diagram

Figure 27 illustrates a class diagram for the class transport vehicle. The class diagram is used to show a generic description of a system.

Classes are represented in the diagram by solid-outline rectangles, which show an inherited relationship between the subclasses (boat and car) and their superclass

(transport vehicle). This indicates that the subclasses inherit the attributes and behaviour of the superclass.

5.1.1.2 Objects

An object, also known as an instance, is a particular occurrence of a class. It encapsulates both state and behaviour. For example, a Honda City is a car and a Suzuki Alto is a Car.

Messages are used to convey control and information between objects. Objects and messages are represented together in the Unified Method as Object Message Diagrams (Figure 28).

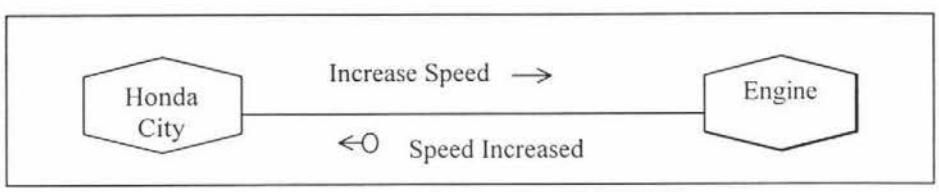


Figure 28 : Object Message Diagram

An object message diagram shows a sequence of messages that implement an operation or transaction. Objects are represented in the diagram by a hexagon, messages are represented by arrows, where a small circle at the end of an arrow indicates a returning value. Thus, in Figure 28, the object 'car' sends a message to the object 'engine' to 'increase speed'. The engine object then returns a message to the car confirming that speed has increased.

5.1.1.3 Aggregation

Aggregation is the 'whole-part' or 'has a' relationship between classes, it is where one object has 'ownership' of another object. For example, a car 'has a' door, and a boat 'has a' propeller. Aggregation is represented in the Unified Method by a hollow diamond.

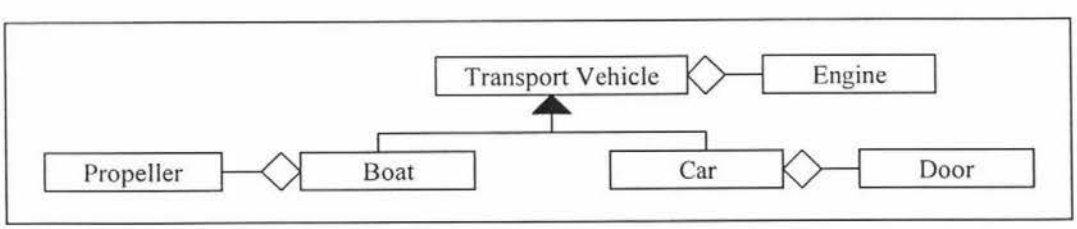


Figure 29 : Aggregation

Figure 29 states that every boat 'has a' propeller, while every car 'has a' door. Both cars and boats (assume motor boat) have an engine, thus the superclass transport vehicle has the engine association. The association is inherited by the two subclasses, boat and car, implying that cars and boats both have engines.

5.2 Overview

There are five major components which interact together to evaluate an expression in the ZL environment:

- Interface
- Lexical Analyser
- Parser
- Typechecker
- Evaluator

Each of these components is represented in the implementation by an object. To evaluate an expression it is necessary for these objects to pass messages between each other. If an expression contains a reference to a function, then that function must have a parse tree contained in the global function table. Figure 30 shows an object message diagram of the messages passed between the interface, lexer, parser, and typechecker objects to create a parse tree for a ZL function.

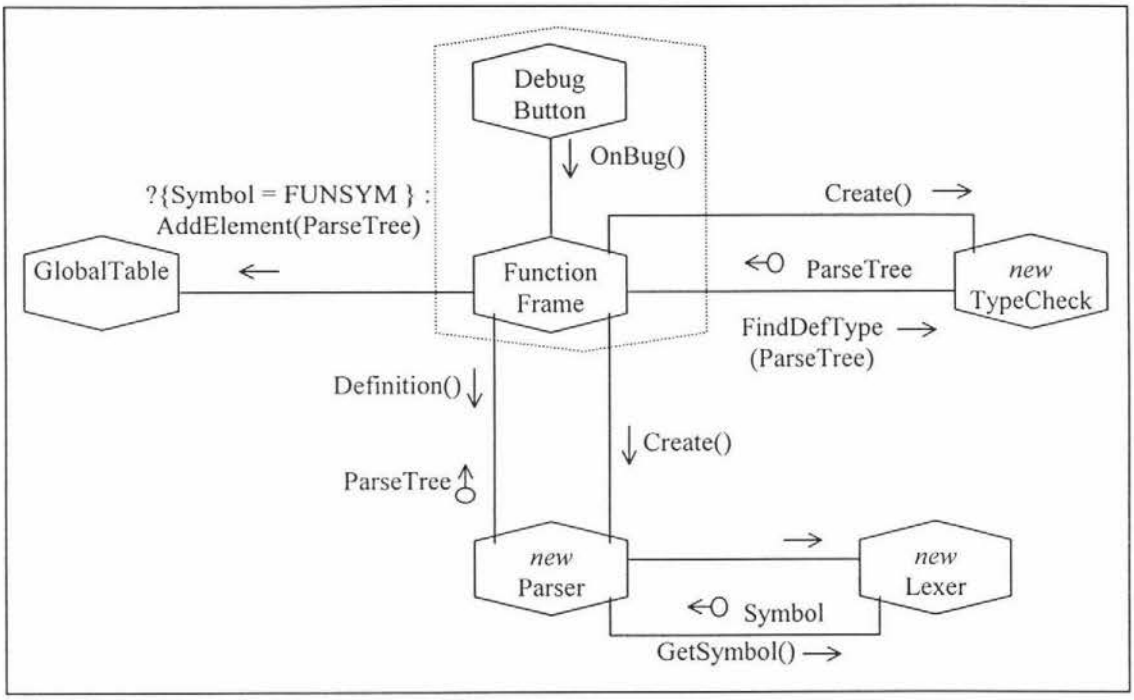


Figure 30 : Creating a Parse Tree for a ZL Function

The interface object (illustrated as a dotted hexagon in Figure 30) is a composite object, i.e. it is comprised of other objects. Figure 30 shows two such objects, the debug button object and the function frame object. To create a parse tree the user selects the debug button from the toolbar. This causes the debug button object to send the message 'OnBug' to the function frame object. The function frame object reacts to the 'OnBug' message by creating a parser object, which then creates a lexer object. When both the lexer and parser objects are created, the function frame object sends the message 'Definition' to the parser object to create a parse tree.

When the parser object returns the parse tree to the function frame object, the function frame object creates a typechecker object, and sends it the message 'FindDefType' which causes the typechecker object to typecheck the parse tree.

Once the parse tree has been typechecked and returned to the function frame object, the function frame object will store it in the global function table for any future references to the function.

Figure 31 shows an object message diagram for evaluating an expression.

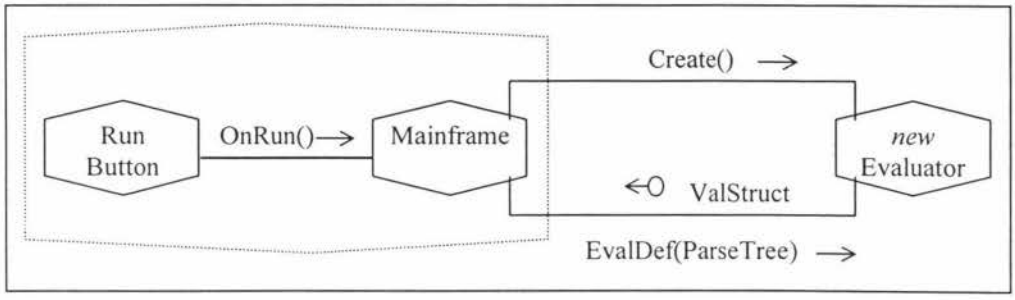


Figure 31 : Evaluating an Expression

Figure 31 again shows the interface object as a composite. To evaluate an expression the user selects the run button from the toolbar. This causes the run button object to send a message 'OnRun' to the mainframe object. The mainframe object will create and typecheck a parse tree for the expression using the same method as the function frame object. When the typecheck object returns the typed parse tree of the expression, the mainframe object will create an evaluation object. This object will then be sent the message 'EvalDef' by the mainframe object. The evaluator object will react to the message by evaluating the parse tree. Once the evaluator object has evaluated the parse tree, it will return the result (ValStruct) to the mainframe object. This result will then be displayed to the user.

5.3 INTERFACE

The ZL environment uses ten primary classes to create and maintain screen objects for the interface:

- Application
- Mainframe
- Toolbar
- Function Toolbar
- Status Bar
- Expression Bar
- Message Bar
- Function Document
- Function View
- Function Frame

5.3.1 Application Class

The application class is a subclass of the Microsoft Foundation Class (MFC) 'CWinApp'. It is responsible for initialising, executing, and terminating the application object.

Initialising the application includes:

- creating the mainframe object
- initialising the function window

The application class is also responsible for the creation of new function windows and maintaining a link to the mainframe class. Figure 5.2, illustrates a class diagram of the application class.

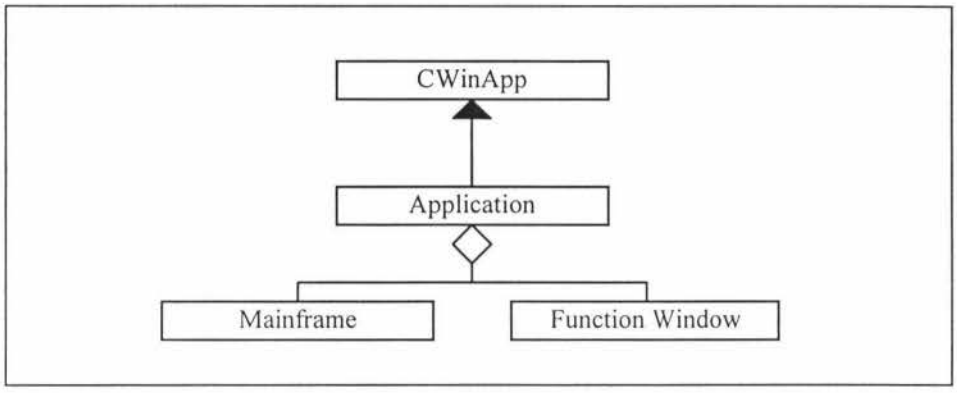


Figure 32 : The Application Class

5.3.2 Mainframe Class

The Mainframe Class is a subclass of the MFC class CMDIFrameWnd. There is only one instance of this class created in the ZL environment.

The mainframe class represents the bounding application frame window. It includes the menu bar and all control bars (Figure 33).

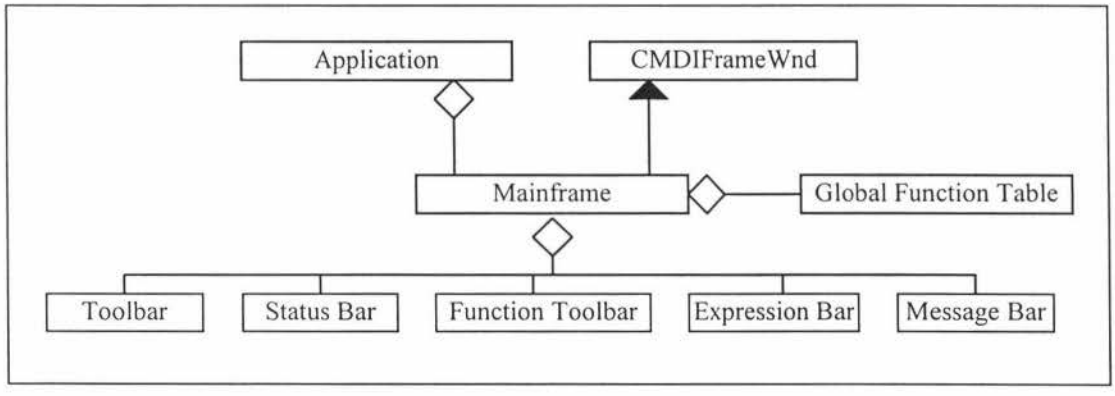


Figure 33 : The Mainframe Class

The Mainframe class is responsible for:

- sizing and positioning of the application window
- initialising and managing the control bars
- managing messages between the function window and the function toolbar
- initialising the global function table

5.3.3 Control Bars

CControlBar is the MFC superclass for all toolbars, status bars, and dialog bars in the ZL environment. Excluding the main toolbar and the status bar there are three other control bars used in the ZL environment, the function toolbar, the expression bar, and the message bar. The mainframe class, Figure 34, manages all five of these control bars.

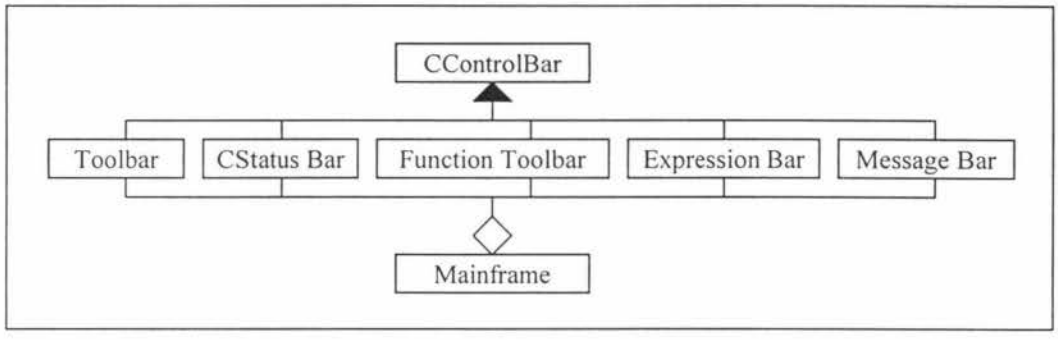


Figure 34 : Control Bars in the ZL Environment

Of the five control bars, only one is a direct instance of a MFC class - CStatusBar. All functions used by this class are standard Microsoft functions, thus this class is discussed no further.

5.3.3.1 Toolbar

Each button and menu item has an associated message handler and command target class. If a menu item and a toolbar button perform the same command then they will share the same message handler and command target class. The message handler contains the function to be performed when a button is selected. The command target class is the class that contains the message handler. Table 2 lists the command target classes and the responsibilities of the message handlers for the various toolbar buttons.

Table 2: Command Target Classes and Handlers for Toolbar Buttons

Toolbar Button	Command Target Class	Responsibility of Message Handler
New	Application	Opens a new function window, and sends a message to the function toolbar to display the name of the new function.
Open	Application	Opens an existing function in a new function window, and sends a message to the function toolbar to display the name of the new function.
Close	Application	Closes the currently active function window, and sends a message to the function toolbar to remove the name of the closed function.
Hide	Application	Hides the currently active function window from the user's view.
Save	Function Document	Saves the function in the currently active function window.
Cut	Function View	Removes the selected text, and copies it to the Microsoft clipboard.
Copy	Function View	Copies the selected text to the Microsoft clipboard.
Paste	Function View	Pastes the contents of the Microsoft clipboard to the current cursor position.
Print	Function View	Prints the currently active function.
Debug	Function Frame	Sends a message to the function frame class to create a parse tree from the currently active function.
Run	Mainframe	Sends a message to the mainframe class to create a parse tree and evaluate it.
Help	Application	Displays an 'About' box, showing the current version of the ZL Application.

5.3.3.2 Function Toolbar

The prime task of the function toolbar is to display and maintain a list of functions. This list is known as the 'function list'. The function toolbar contains routines that update the function list, however, these routines are only activated through messages sent by toolbar buttons and menu items, e.g. the Open toolbar button.

5.3.3.3 Expression Bar

The expression bar class contains two MFC edit control classes. An edit control is used for entering and editing small amounts of text. The edit controls contained in expression bar are named expression box and output box respectively (Figure 35).

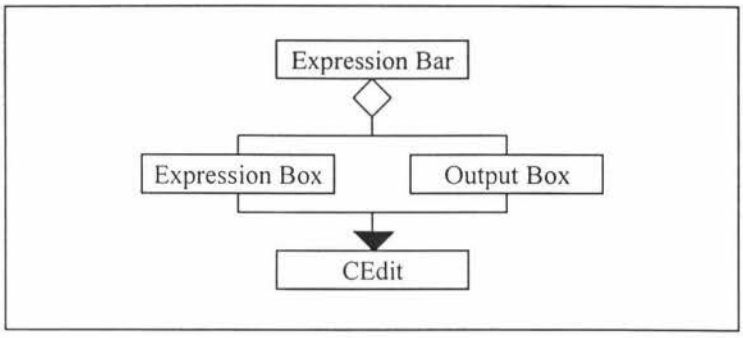


Figure 35 : The Expression Bar

The expression box is used for entering ZL expressions. The output box is used for displaying the results of evaluated expressions. Each edit control is responsible for updating and maintaining its own display.

5.3.3.4 Message Bar

The message bar class contains one MFC CListBox class (Figure 36).

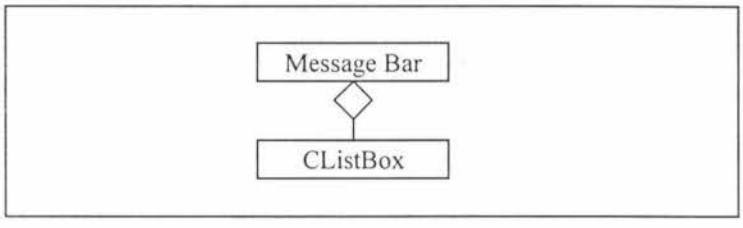


Figure 36 : Message Bar

The list box is used to display error messages sent by the parser and typechecker to the user.

5.3.4 Function Window

The function window is constructed by the combining together three classes: Function Document, Function View, and Function Frame.

5.3.4.1 Function Document Class

The function document class is a subclass of the MFC CDocument class. It holds the actual function data that is entered by the user.

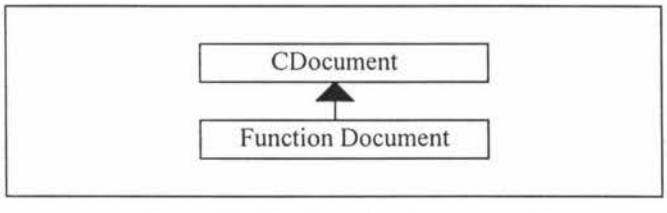


Figure 37 : The Function Document Class

The function document class is responsible for:

- opening a function file
- saving a function to a file

5.3.4.2 Function View Class

The function view class is a subclass of the MFC CEditView class. It is the blank area inside the function window frame which acts as an intermediary between the user and the function document class. Any keyboard or mouse events are interpreted by the function view and passed to the function document.

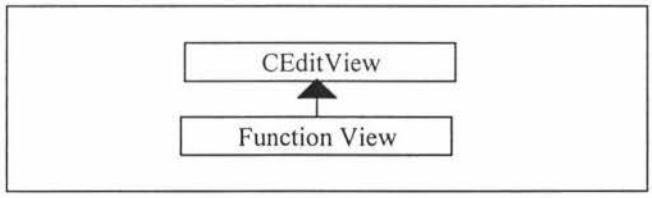


Figure 38 : The Function View Class

The function view class is responsible for:

- interpreting keyboard and mouse events for the function document class
- displaying data for a function document class on the screen
- printing data from a function document class

5.3.4.3 Function Frame Class

The function frame class is a subclass of the MFC class CMDIChildWnd. The function frame is the actual 'frame' that bounds the function window. It does not have its own menu, toolbar or status bar like the mainframe class, instead it shares those components from the mainframe.

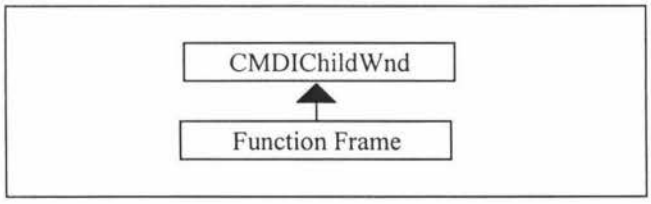


Figure 39 : The Function Frame Class

The function frame class is the command target class for the 'OnBug' message handler, and as such it is responsible for managing the parsing and typechecking of the function held in the function document class.

The function frame class is responsible for:

- positioning the function window in relation to the mainframe class
- managing the "OnBug" message handler

5.4 LEXICAL ANALYSER

The lexical analyser (or lexer) is responsible for breaking down a ZL expression into tokens and their values. An instance of the lexer class is created when a parser object sends a message.

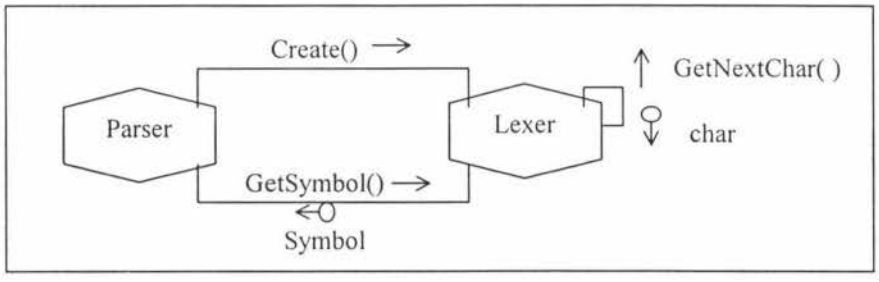


Figure 40 : Creating the Lexer Object

When initially created, the lexer object reads the text contained in the expression box of the expression bar. However, it does not return a token until requested by the parser object.

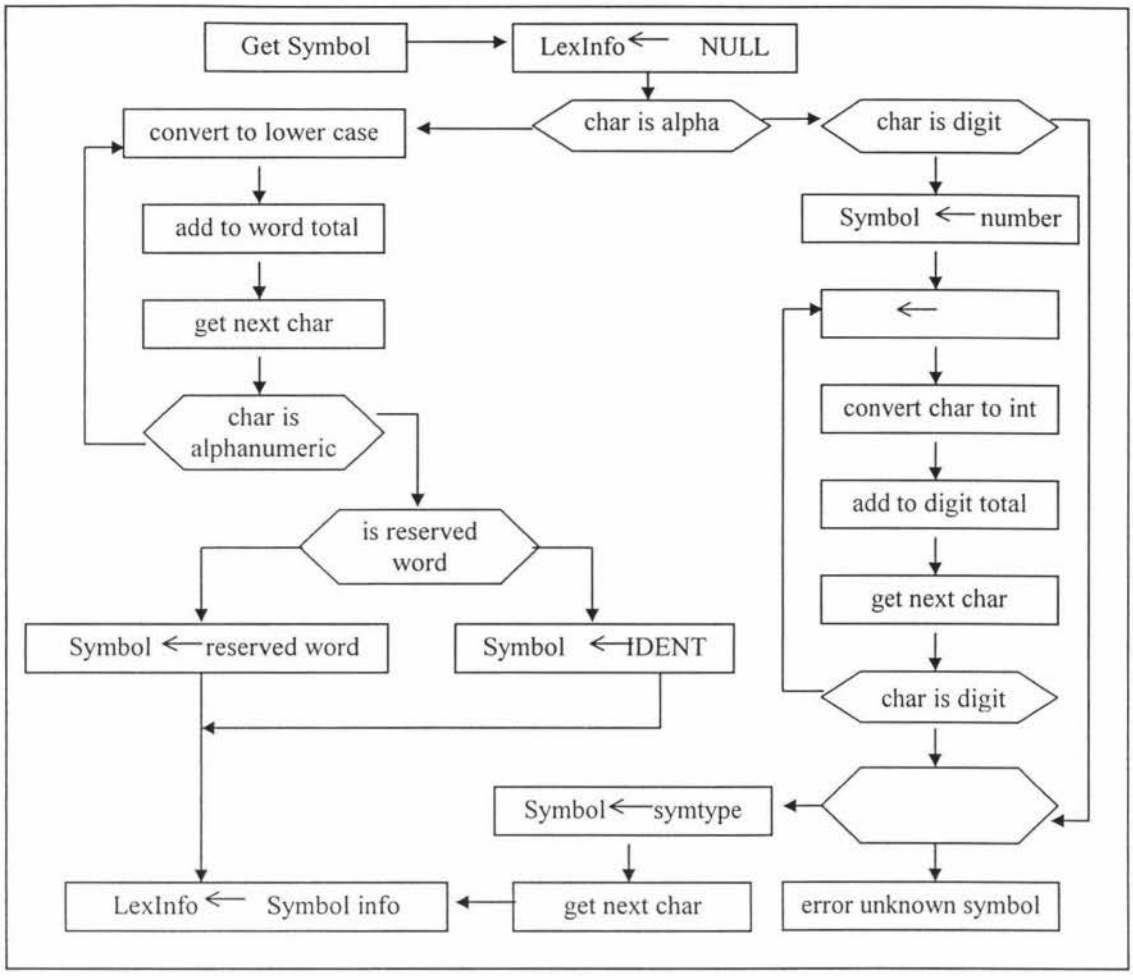


Figure 41 : Retrieving a Symbol from the lexer - activated by message Get Symbol()

5.4.1 Tokens

Each token object contains a symbol and some data, i.e. it is more than just a token. For this reason, and to avoid confusion, the token object is known as the lexinfo object. Figure 42 shows the structure of the lexinfo object.

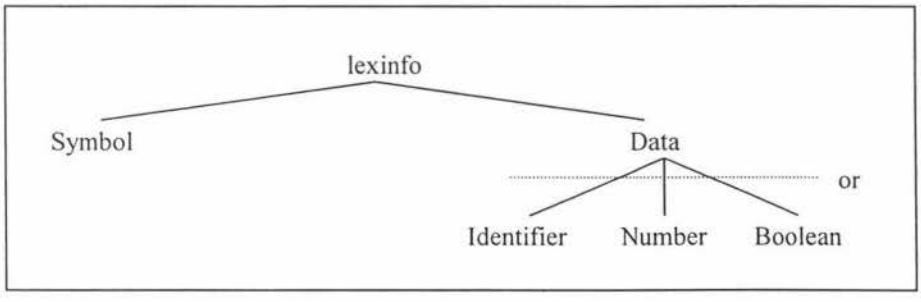


Figure 42 : Structure of lexinfo (token)

'Symbol' contains the type of the token created by the lexer, and 'data' contains the value of the symbol. If the created token is a reserved symbol or operator, data is set to empty.

For example:

```
fun sqr
      (a) ->       a * a
```

The above example is lexically analysed into the following symbols:

```
FUNSYM IDENT LBRACKET IDENT RBRACKET FUNPTR IDENT MULTSYM IDENT
```

For each symbol a lexinfo object is created, where lexinfo.symbol would contain the type of the symbol, and lexinfo.data would either contain an identifier name or remain blank.

5.5 PARSER

A parser object is created by the 'OnBug' message for parsing a function, or the 'OnRun' message for parsing an expression. The creation of this object automatically triggers the creation of a lexer object (Figure 43).

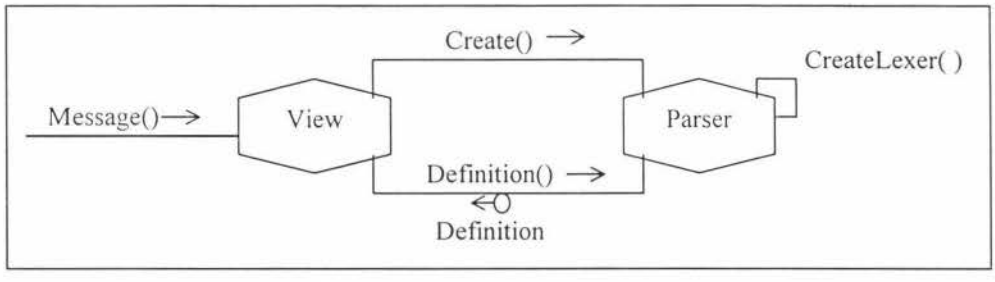


Figure 43 : Creating the Parser Object

Once both the parser and lexer objects are created, a message is sent to the parser to create the parse tree.

5.5.1 Parse Tree

The parse tree is a combination of objects from different classes formed together in an aggregate hierarchy.

There are eleven classes which can used to form the parse tree:

```
Definition          Expression          Lexinfo
Function            Tuple               Apply
Funelement          Letexp              Patelement
Pattern             Letelement
```

5.5.1.1 Definition Class

The definition class is the topmost class of a ZL parse tree. It contains the parse tree of an expression or function, i.e. a definition object either 'has a' function or it 'has an' expression.

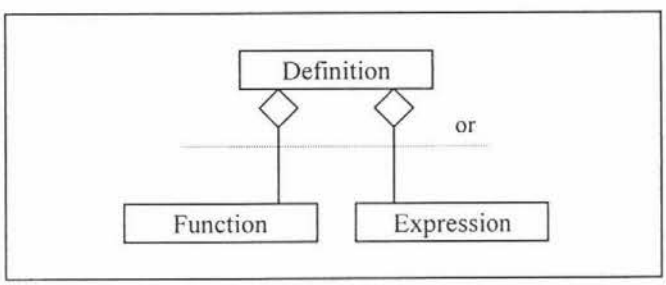


Figure 44 : The Definition Class

5.5.1.2 Function Class

A function object contains a funelement object, which inturn contains a pattern object and an expression object.

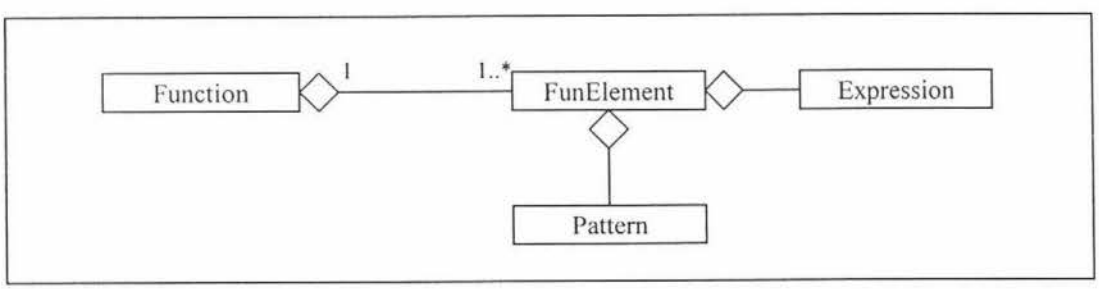


Figure 45 : The Function Class

5.5.1.3 Pattern Class

A pattern object contains one or more patelement objects, which inturn contains either a pattern object or a lexinfo object.

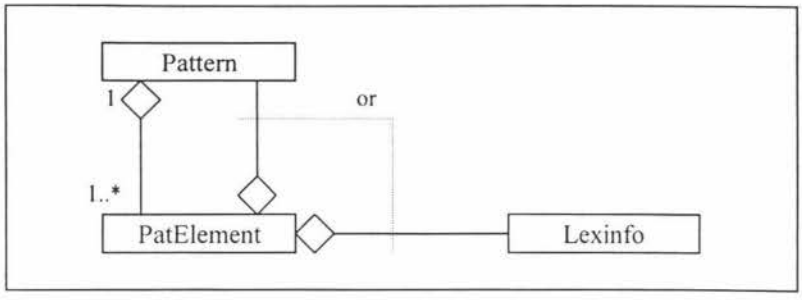


Figure 46 : The Pattern Class

5.5.1.4 Expression Class

An expression object, contains either a tuple object, an application object, a lexinfo object, or a qualified expression object.

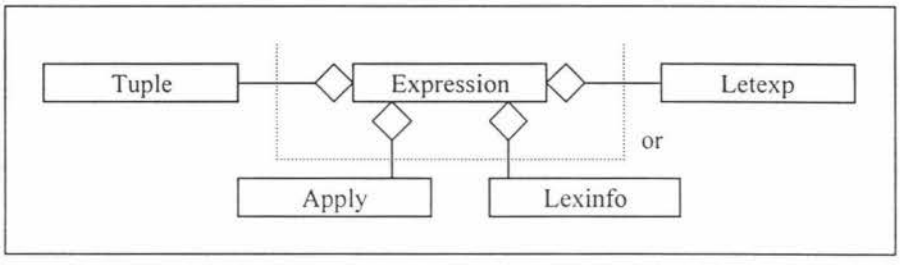


Figure 47 : The Expression Class

5.5.1.5 Qualified Expression Class

A qualified expression object, or 'let' expression object, contains one or more letelement objects, which each inturn contain a pattern object and an expression object.

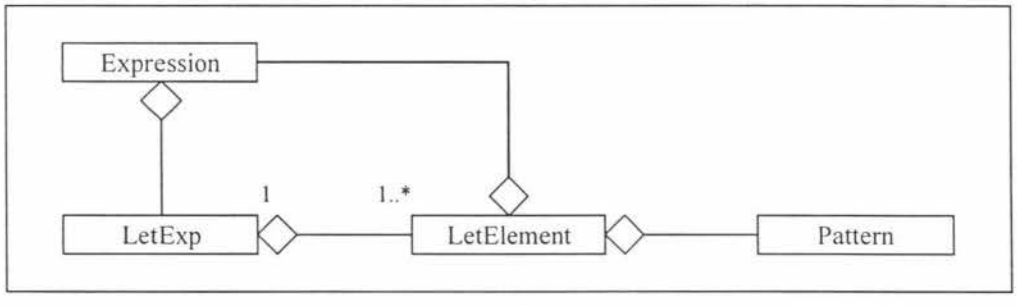


Figure 48 : The Qualified Expression Class

5.5.1.6 Tuple Expression Class

A tuple expression object contains two or more expression objects.

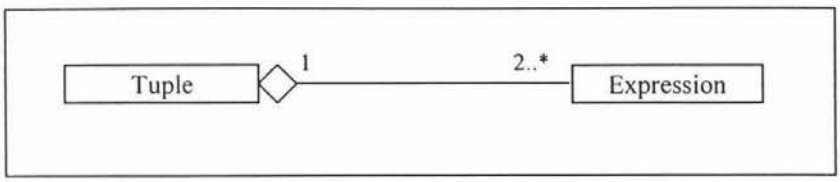


Figure 49 : The Tuple Expression Class

5.5.1.7 Application Class

An application object contains an attribute 'symbol' and a tuple object. The symbol holds the name of the function to be applied to the expressions contained in the tuple object.

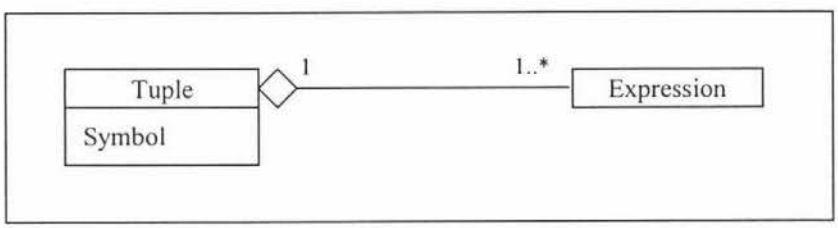


Figure 50 : The Application Class

Figure 51 shows a class diagram of the full ZL parse tree.

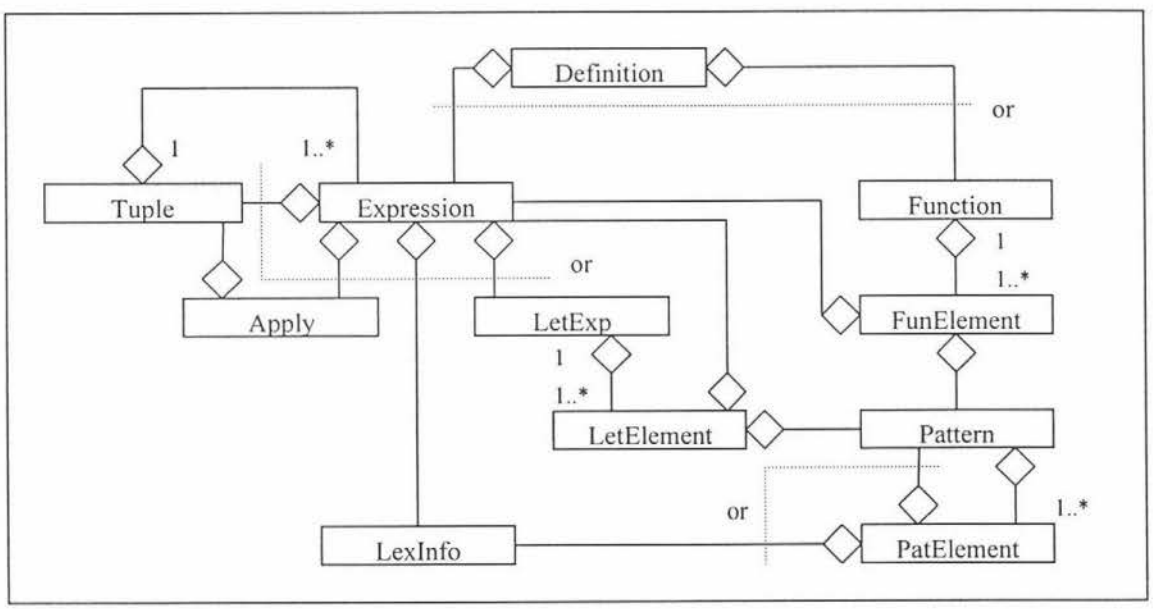


Figure 51 : The ZL Parse Tree

Figure 52 shows the chain of messages used to create a full ZL parse tree.

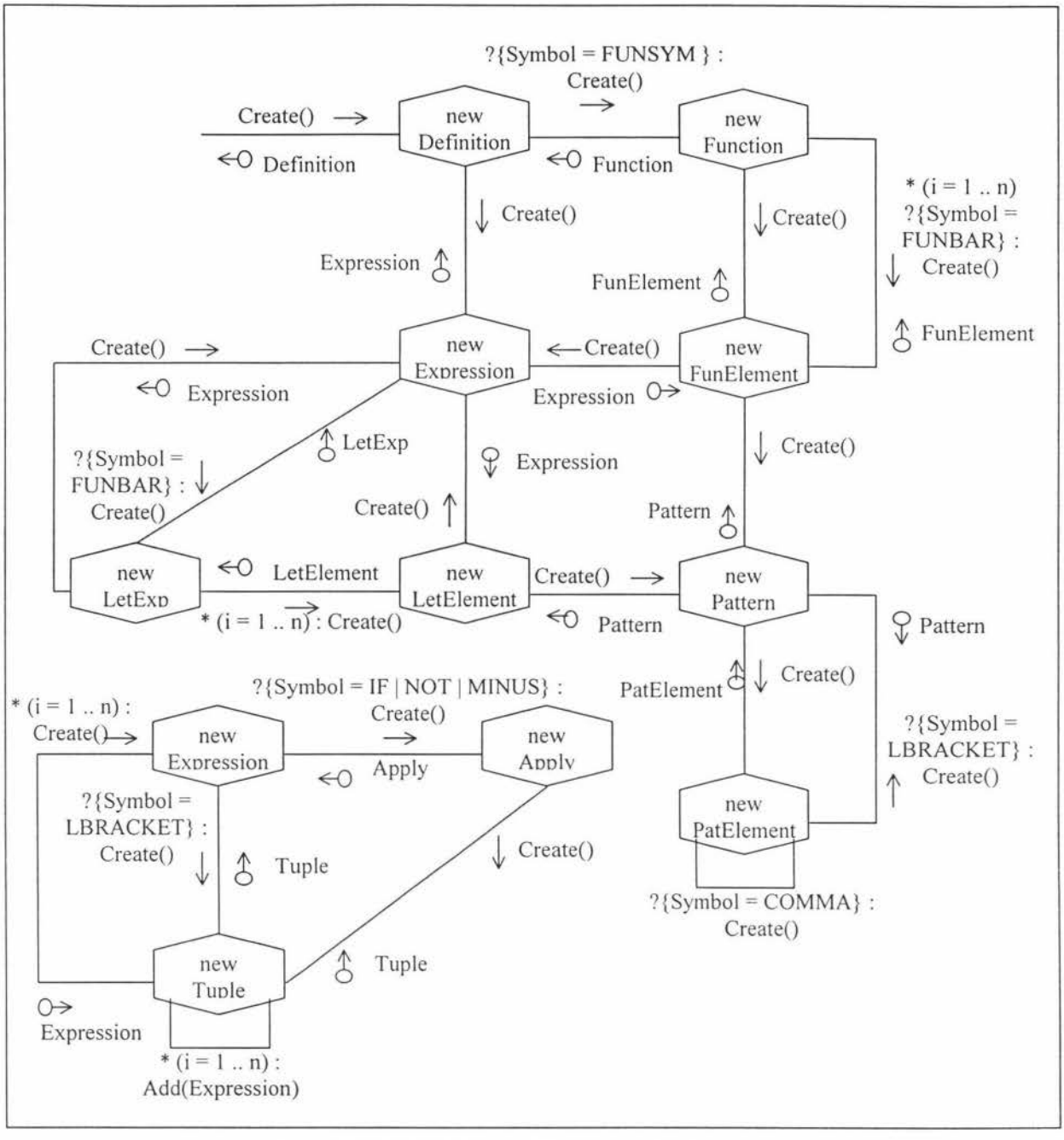


Figure 52 : Object Message Diagram for the ZL Parser

Figure 52 shows the object message diagram used to create the full ZL parse tree, however, in most cases a full parse tree would not be created. For example:

```
fun sqr
      (a) ->       a * a
```

The chain of messages passed through the ZL objects to create a parse tree from the above function is shown in Figure 53.

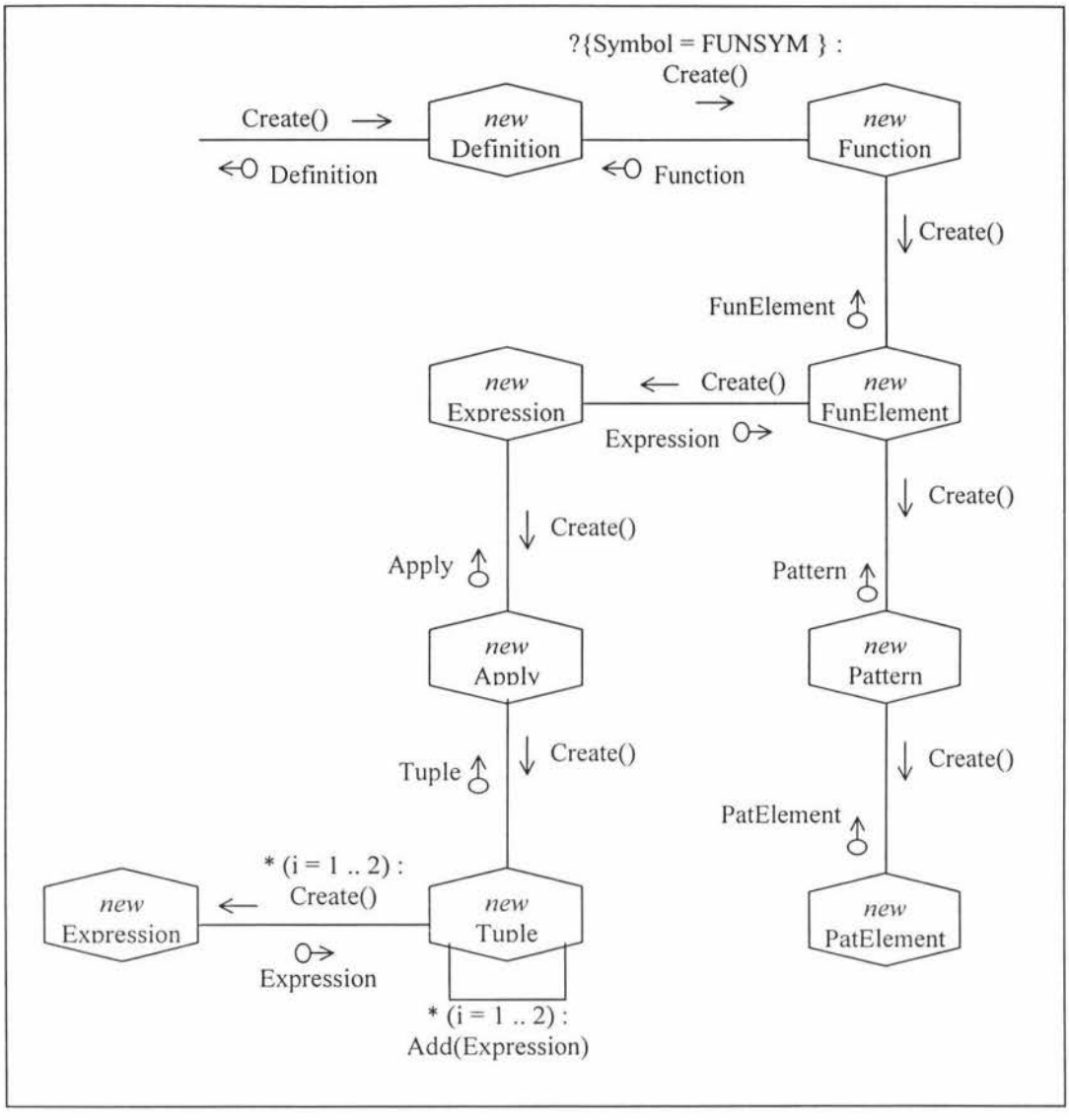


Figure 53 : Creation of a Parse Tree for ZL Function 'sqr'

A definition object is created by the parser to store the parse tree of function 'sqr'. FUNSYM indicates that the object is a function. Each function object contains one or more funelement objects, therefore a funelement object is automatically created. In the current example the funelement class consists of:

```
(a) ->      a * a
```

Each funelement object has both a pattern object and an expression object. In the above example '(a)' is the pattern and 'a * a' is the expression. The pattern in the example contains only one element, so only one instance of the patelement class is created. This patelement object also holds the lexinfo information for the pattern element.

The operator '*' indicates an application expression, therefore an application object is created. This inturn causes the creation of a tuple instance. The tuple contains two expressions, 'a' and 'a'. These objects will contain the lexinfo information for the expressions.

Figure 54 shows the resulting parse tree created from the object message diagram in Figure 53.

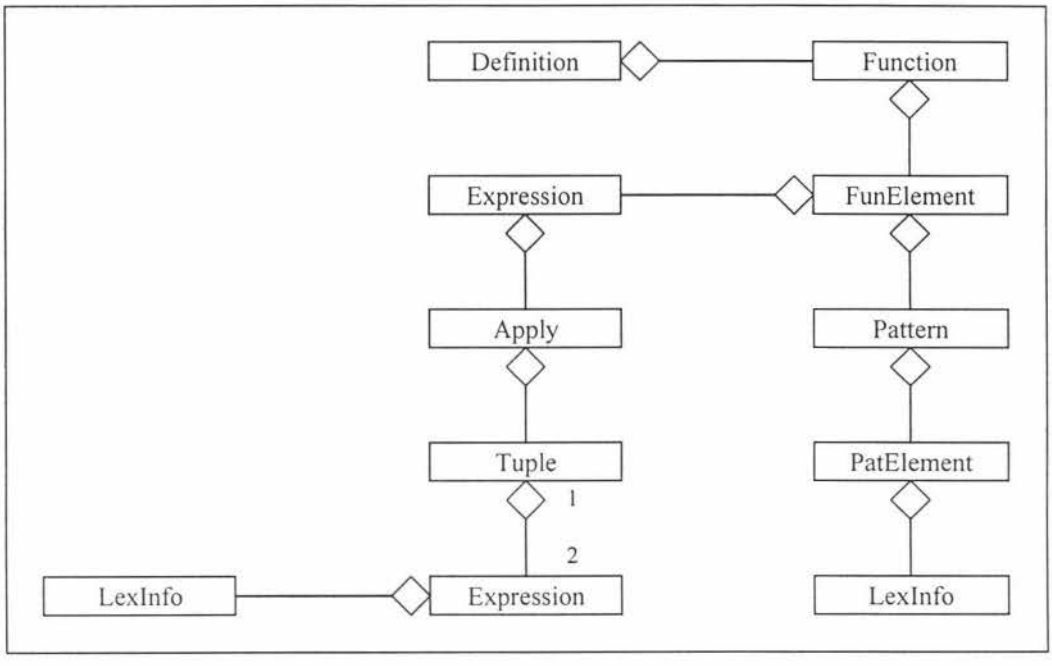


Figure 54 : Parse Tree for ZL Function 'sqr'

As seen from Figure 53 and Figure 54, the structure of the parse tree reflects that of the object message diagram.

5.6 TYPE CHECKER

A typechecker object is created by either a mainframe object or a function frame object, using the 'OnRun' and 'OnBug' messages respectively, Figure 55.

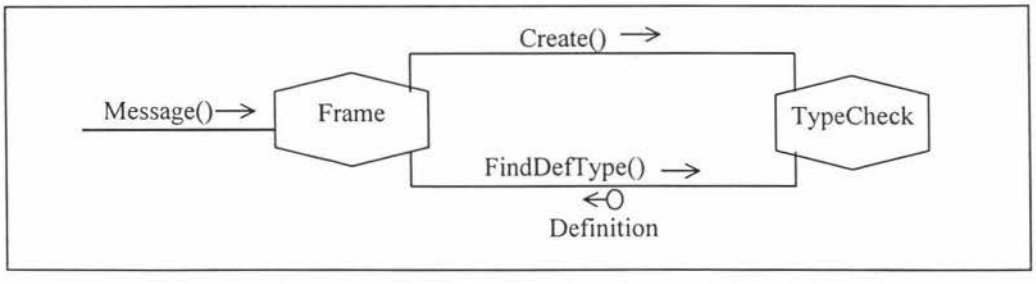


Figure 55 : Creating a Typechecker Object

A typechecker object is only used to typecheck the current parse tree. Thus, once a parse tree is evaluated its corresponding typechecker object is destroyed.

The typechecker is used to add a type to each object in the parse tree. A type is the 'type' of the object. As stated in section 3.2 Types, there are just two primitive types: number and boolean; and three structured types: tuple type, pattern type and function type.

5.6.1 Tables

In order to typecheck a parse tree, the typechecker object utilises three tables: the operator table, the identifier table, and the global function table.

5.6.1.1 The Operator Table

The operator table is 'owned' by the mainframe object (refer to section 5.3.2 Mainframe Class) and lasts for the life of the interaction, i.e. it is created when a user starts the ZL application, and it is destroyed when the user exits the ZL application.

The operator table is used to store the types of operators. Figure 3.13 shows the structure of an operator table element.

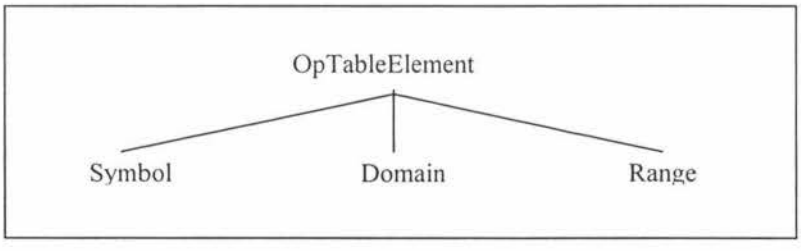


Figure 56 : Structure of an Operator Table Element

Each operator table element contains:

- Operator: the operator name,
- Domain: the type of the domain,
- Range: the type of the Range.

For example:

Operator	Domain	Range
PLUS	number x number	number
EQUAL	number x number	boolean
AND	boolean x boolean	boolean

5.6.1.2 The Identifier Table

The identifier table is owned by the parse tree and lasts for the life of the parse tree. The identifier table is used to store identifiers, their types and their values. Figure 57 shows the structure of an identifier table element.

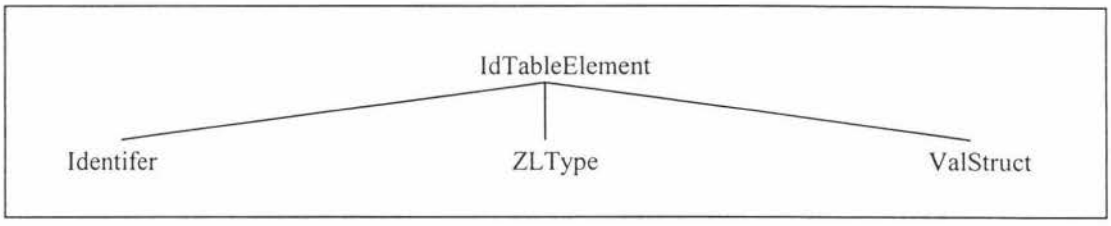


Figure 57 : Structure of an Identifier Table Element

Each identifier table element contains:

- Identifier: the identifier name,
- ZLType: the identifier type,
- ValStruct: the identifier value (refer to section 5.7 Evaluator).

When the typechecker locates an identifier, it will check that the identifier is not already contained in the table. If it isn't, then a new table element is created (containing the identifier information) and added to the table.

5.6.1.3 The Global Function Table

The global function table is 'owned' by the mainframe object (refer to section 5.3.2 Mainframe Class) and lasts for the life of the interaction, i.e. it is created when a user starts the ZL application, and it is destroyed when the user exits the ZL application.

The global function table is used to store the parse trees of functions available to the user. When a function is parsed and typechecked, its parse tree is placed in the global function table. This parse tree is then available for use by other typechecker and evaluator objects.

5.6.2 Typechecking a Parse Tree

Once a typechecker object has successfully created, it is sent a message to typecheck a parse tree.

Figure 58 illustrates an object message diagram of the typechecking process.

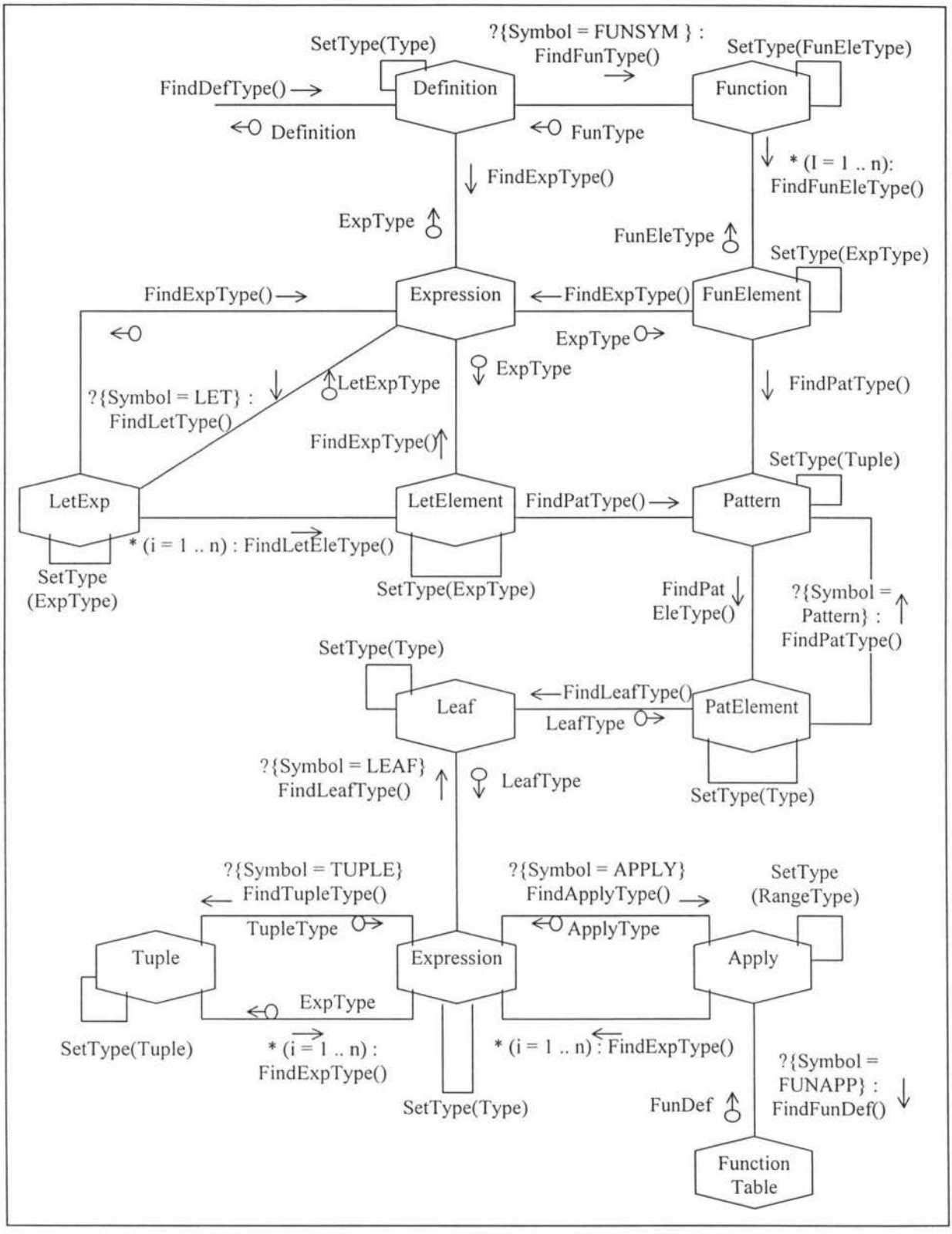


Figure 58 : Object Message Diagram of the ZL Typechecker

The objects LetExp and FunElement (Figure 58) both send multiple messages to other objects. The order in which these messages are sent is irrelevant. For more information

on how the returning types of these messages are unified, refer to sections 3.4.1 Pattern Matching and 3.5 Functions.

Figure 58 illustrates how the typechecker object uses the Function Table. When an application expression of type function application is received, the function definition used in the application object is located in the function table. The parameters of the application object are unified with those of the function definition to ensure that the types are compatible.

Figure 58 shows the object message diagram used to typecheck a full ZL parse tree, however, as stated in the previous section, it is unlikely that a full parse tree would ever be created. For example:

```
fun sqr
      (a) ->       a * a
```

Section 5.5.1 showed how a parse tree for the ZL function 'sqr' is created. The function of the typechecker object is to assign a type to the expression that the parse tree represents, Figure 59.

According to the type rules for the ZL grammar, the typechecker recursively moves through the parse tree assigning a type to each object in the ZL function 'sqr', Figure 59.

Once the parse tree for a function definition has successfully been typed it is placed into the global function table to be later used for typechecking and evaluating other expressions.

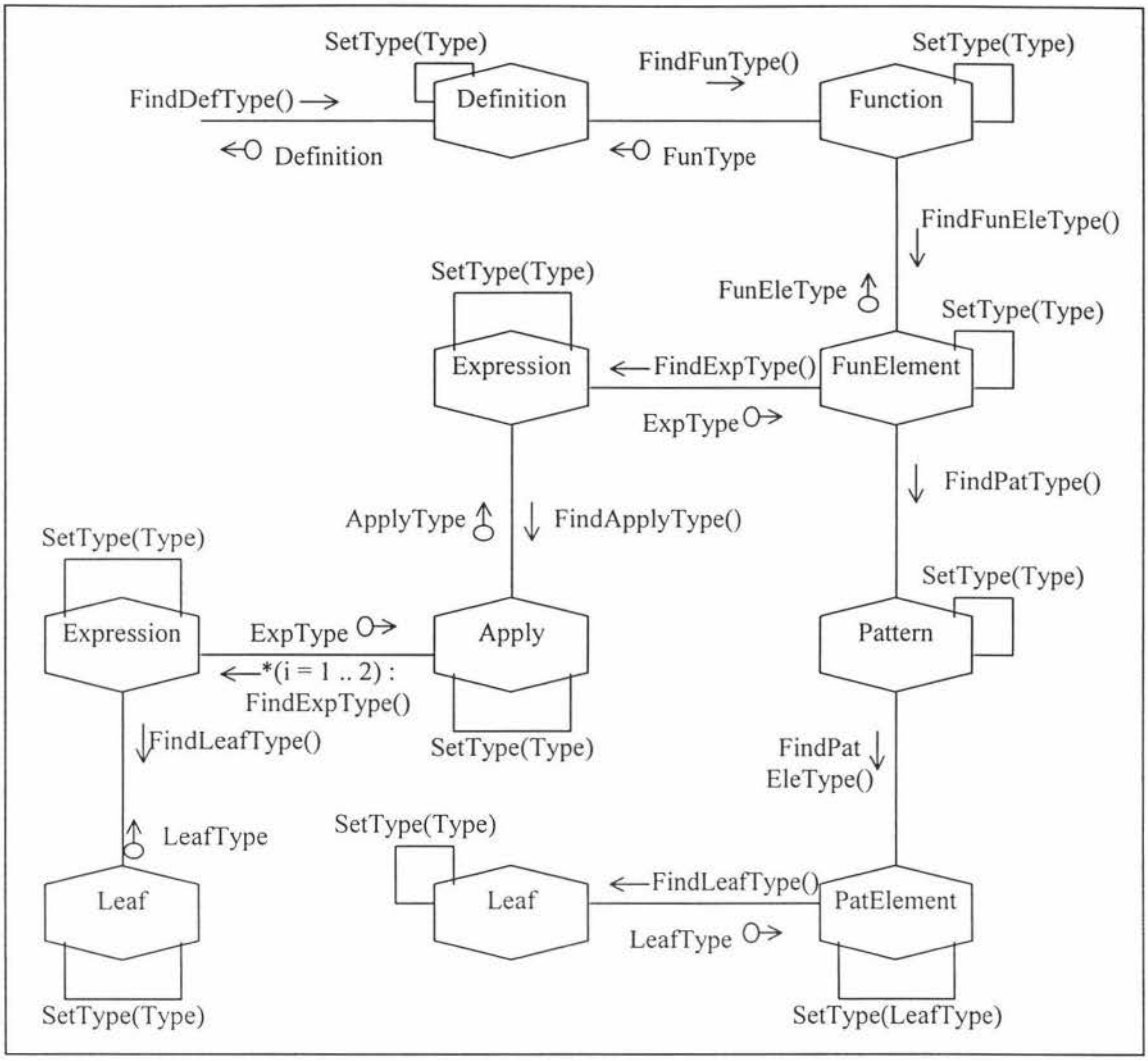


Figure 59 : Typechecking the ZL Function 'sqr'

5.7 EVALUATOR

Although the function frame can create a parser and typechecker object, it cannot create an evaluator object. An instance of the evaluator class can only be created by the mainframe by an 'OnRun' message. This is because functions are only evaluated through the use of expressions, therefore it is not necessary for function frame objects to have the ability to create evaluator objects.

An evaluator object is specifically created to evaluate only one expression. Once the expression is evaluated, the evaluator object and the expression parse tree object are destroyed. Any function parse trees used by the evaluator to evaluate expressions, remain complete in the global function table.

5.7.1 Value Structure

A value structure is used to store the value of an object (Figure 60).

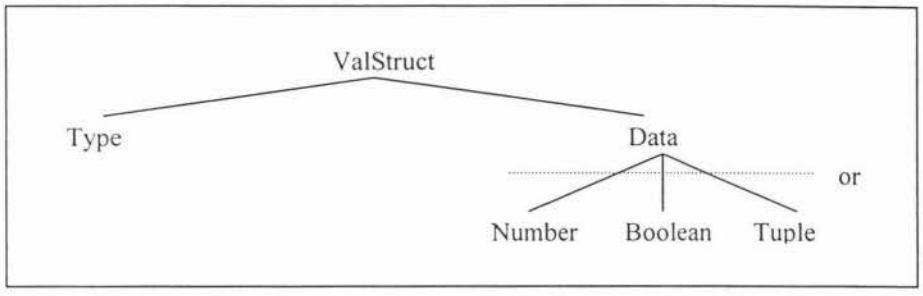


Figure 60 : Value Structure

Each value structure contains a value (the actual data) and the type of that value.

5.7.2 Evaluating the Parse Tree

When the evaluator object is created, a message is sent to it from the mainframe object to evaluate a parse tree. Figure 61 presents an object message diagram of the evaluation process.

As shown in Figure 61, the evaluator descends through the parse tree, until a terminal (lexinfo) is located. The terminal is evaluated and its value passed back up the tree.

For example:

```
fun sqr
      (a) ->       a * a

sqr(5)
```

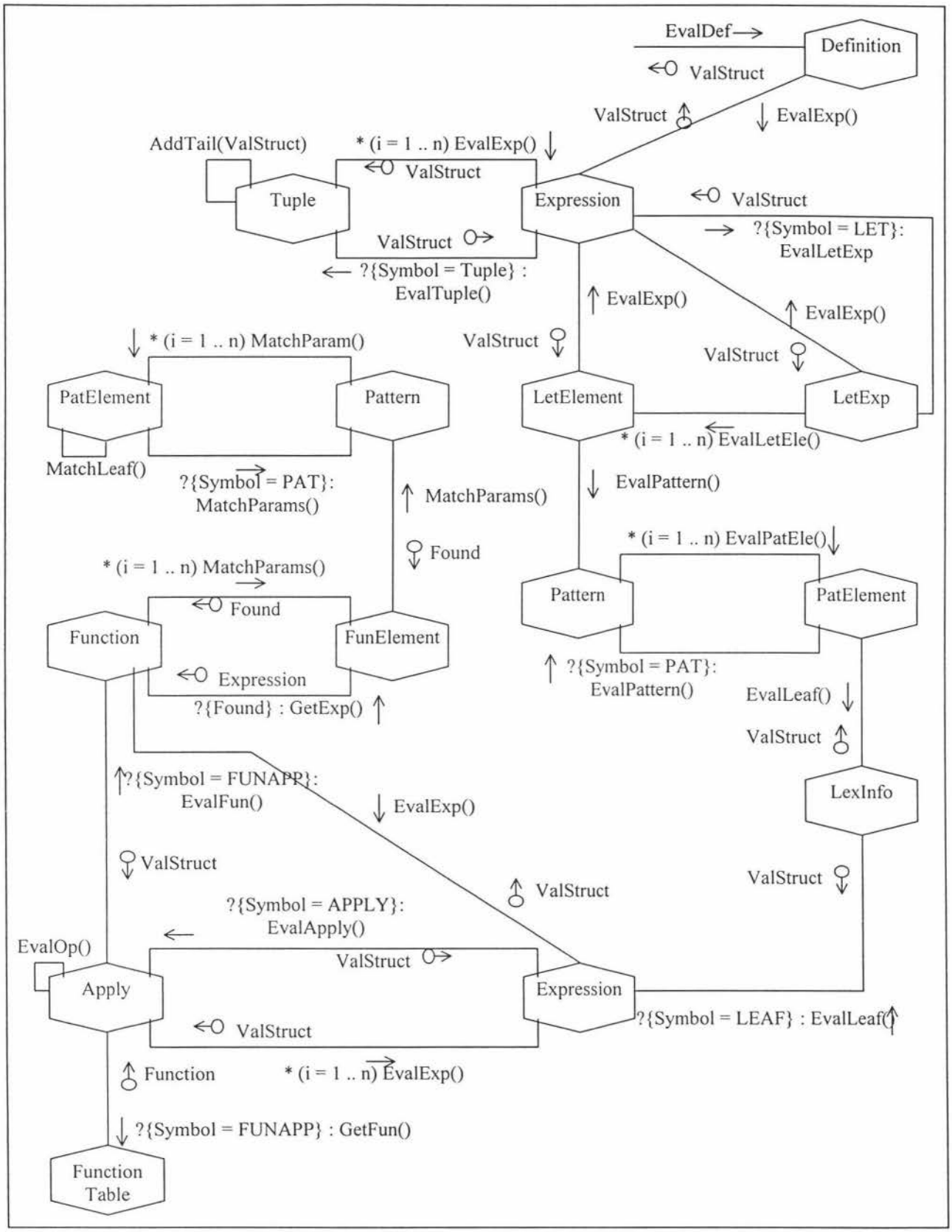


Figure 61 : Object Message Diagram for Evaluating a ZL Parse Tree

The parse tree for the previous expression, `sqr(5)`, is shown in Figure 62.

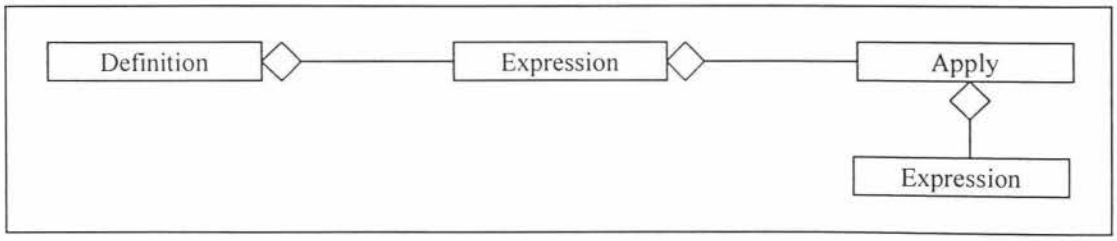


Figure 62 : Parse Tree for the ZL Expression 'sqr (5)'

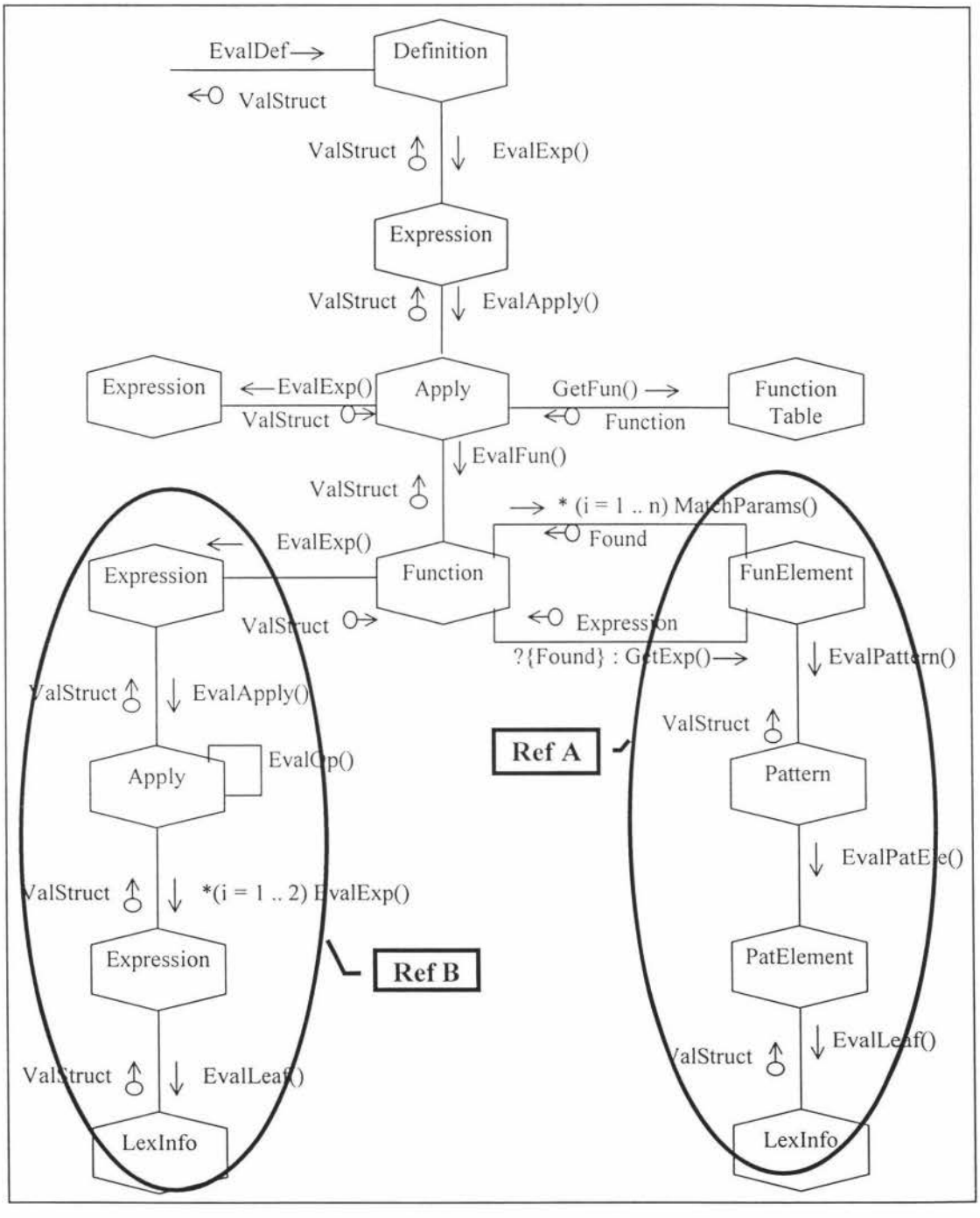


Figure 63 : Evaluating the ZL Expression 'sqr(5)'

Figure 63 shows the links of messages created by the evaluator to evaluate the parse tree from Figure 62.

The evaluator descends through the parse tree until it encounters the function application object, where upon it evaluates the expression '5' and searches the global function table for the parse tree of function 'sqr'. The evaluator then proceeds to evaluate the parse tree by matching the parameters of expression 'sqr(5)' to the pattern object of the first funelement for function 'sqr' (Ref A. in Figure 63). As the first funelement object of the function 'sqr' is a match for the expression 'sqr(5)', the associated expression object for this funelement is evaluated (Ref B. in Figure 63).

The right hand side of figure 3.20, shows the evaluator matching the user entered parameters to the function's funelements. If there is a successful match, the expression of the successful funelement is evaluated - illustrated on the left-hand side of figure 3.2.

In this instance the successful match will be the expression 'a*a'. This expression is evaluated using the user-entered parameter of '5'. The result is then passed back up the tree and displayed to the user.

6. TESTING THE OBJECTIVE

Chapter 1 described the types of cognitive models that are thought to characterise expert programmer skills. Chapter 2 linked these models to specific programming concepts. Subsequent chapters described the ZL system and its implementation. It remains then to argue that the ZL system is well suited to supporting the development of the appropriate mental models in the novice. This is the goal of the current chapter.

6.1 IDENTIFYING OBJECTIVES

The most general objective of the ZL programming language and environment is to aid the novice programmer in the cognitive construction of computer programs. We hypothesize that the ZL language and environment can achieve this general objective by:

- Improving the ability of novice programmers to acquire the mental model characteristics of experts.
- Increasing the novice programmer's ability to beneficially transfer programming skills from ZL to another language.
- Focusing on the importance and order of the three planning stages involved in program design.
- Highlighting simple and generic programming concepts that are common to most, if not all, programming languages.
- Having a functional and easy to use programming environment.

The current chapter hypothesizes the ways in which the features of the ZL language and environment achieve some of these objectives. It does this by outlining two experiments to test the hypothesis. Both experiments are described in the following way, by outlining:

- The objective of the experiment
- The subjects required for the experiment
- An experimental procedure suitable for testing the objective
- The positive results expected from running the experiment

6.2 Testing Objectives

6.2.1 The Construction of Mental Models for Generic Programming Concepts

The objective is to test whether ZL improves the construction of mental models for generic programming concepts. Specifically, we test whether ZL improves the novice programmer's ability to acquire the five expert mental model characteristics that were identified in section 1.3.1. These are:

- Hierarchic Structure: the depth and breadth of a mental model.
- Explicit Mappings: the links between a model's hierarchically structured layers.
- Basic Recurring Patterns: mental models for frequently recurring situations.
- Well Connected: how parts of a mental model interact together.
- Well Grounded: information on the physical location of program structures and operations.

6.2.1.1 Subjects

Four groups of subjects are required for the experiment. Two of novices and two of experts. One group of novices and one group of experts will form the ZL sample. The remainder will form the control group.

6.2.1.2 Procedure

Both groups are given the same problem to solve. The ZL sample will use the ZL system. The control group will use some other programming system, for example, Pascal or C.

Fix et.al. (1993) identified eleven comprehension questions designed to show differences between the characteristics of expert programmers' mental models and novice programmers' mental models. Table 3 outlines these comprehension questions together with the corresponding model characteristic they are designed to test.

Table 3 Comprehension Questions

Q No.	Question	Model Characteristics
1	Match function calls to function definitions	Hierarchical Structure
2	List function names	Hierarchical Structure
3	Write description of goals of selected functions	Explicit Mappings
4	Write description of principal goals of program	Explicit Mappings
5	Label complex code segments with plan labels	Recurring Patterns
6	Label simple code segments with plan labels	Recurring Patterns
7	List names used for same data objects in different functions	Well Connected
8	List important value names	Well Connected
9	Fill in names of functions in a skeleton outline of the program	Well Grounded
10	Match argument values in a call to parameter names in a definition	Well Grounded
11	Indicate the location of keywords in the program	Well Grounded

After performing the task subjects are questioned along the lines developed by Fix et.al. The plan is to use a version of Fix's generic questionnaire to assess the degree to which subjects are able to comprehend the important hierarchy, mappings, patterns and so on that are present in the given problem.

Hierarchical Structure

Hierarchical structure is tested with questions of type 1. Questions of type 2 are designed to test a subject's memory, e.g. their ability to list function names. Thus, if there is no significant difference between expert and novice responses to questions of type 2, then this suggests that differences in results from questions of type 1 are explained by factors which are more than just good memory.

Explicit Mappings

The ability to make explicit mappings is tested by questions of type 3. The ability to write a description of a selected goal (e.g. a function call) within the sample program is a plausible measure of a subject's ability to map between program code and program goals. If a subject further describes the actual calculated steps taken to achieve the goal,

then this also shows that their mental model includes methods on how goals are achieved. In contrast, questions of type 4 demonstrate the ability to write a description of the principal goals of a sample program. Questions of this type might show that a subject comprehends the goal of a program as a whole, but their mental model may still lack the explicit links between code segments and sub-goals.

Recurring Patterns

The ability to identify recurring patterns can be tested using questions 5 and 6. The ability to label code segments with their semantic role, e.g. "Function Call", is evidence of an ability to identify patterns in a subject's mental model. Questions 5 and 6 differ only on the complexity of the code segments given to the subjects.

Well Connected

Questions of type 7 can be used to test the 'connectedness' of a programmer's mental model. Providing different names shared by the same value through a sample program is a way of testing that a subject understands how that value is used in the program. Simply listing value names, as with questions of type 8, shows knowledge of program components. However, questions of this type do not demonstrate knowledge of component interconnectedness.

Well Grounded

Questions 9, 10 and 11, can be used to test the degree to which a mental model is well-grounded. Subjects are given a skeleton outline of a sample program and asked to fill in the gaps (questions of type 9 and 10). This is a way of assessing how well grounded a subject's mental model is. It indicates whether or not a subject has an overall idea of where program structures occur in the program code. Questions of type 11 identify the location of keywords (e.g. where a 'fun' or 'begin' statement is located), questions of this type indicate that a subject knows the basic structure of a program. However, questions of type 11 alone, do not indicate that the subject's mental model is well grounded.

6.2.2 Hypothetical Test Results

Fix et.al. identified six questions from their question template questionnaire which showed a significant difference between novice and expert mental model characteristics. If ZL does aid novice programmers by improving their ability to gain an expert model, we hypothesize that the significant difference between expert and novice users in the ZL sample should be less than the significant difference between expert and novice users in the control group.

Hierarchical Structure

It is claimed that the mental models of expert programmers display a higher degree of hierarchical structure than do those of novices. Thus, the results of question 1, for both the control group and the ZL sample should reflect this statement. However, a difference in the ZL sample that is significantly smaller than that of the control group might reasonably be taken as evidence of a positive benefit. As question 2 is merely a memory test no significant difference between experts and novices is expected in either group.

Explicit Mappings

Question 4 requires subjects to describe the principal goal of an example program. As subjects should be able to extract this information from value and function names, there should be little difference between experts and novices in either sample. For question 3, however, a significant difference between expert and novice programmers from the control group is expected, as this question is intended as a test of a subject's ability to model explicit mappings. For the ZL sample, a difference between experts and novices is also expected, but this difference should be smaller than the difference for the control group.

Recurring Patterns

Expert programmers from both the ZL and control groups should obtain higher results than novice programmers for question 5. However, it is also expected that the gap between novices and experts in the ZL sample will be smaller than the gap in the control group. As question 6 uses simpler code segments the differences in the gaps for this question should be correspondingly smaller.

Well Connected

It is expected that experts from both the ZL and control samples would obtain higher results than novices for question 7. However, a smaller difference between novices and experts in the ZL sample, as compared to the control group, might be taken as evidence of a positive learning effect. No significant difference between novices and experts in both groups is expected for question 8.

Well Grounded

There should be no significant difference between experts and novices in either group for question 11. It is expected that experts from both the ZL and control groups would obtain higher results than novices for questions 9 and 10. However, similar to question 8, a smaller difference between novices and experts in the ZL sample, as compared to the control group, might indicate that ZL has a positive learning effect.

6.2.3 Planning and the Transfer of Mental Models

Chapter 2, Programming Concepts, described generic programming concepts exemplified by ZL. A brief description of each concept was given together with some assumptions about the type of mental model used by the novice in comprehending the concept. It was argued that the mental models created by the novice when using ZL are generic and easily transferred to other programming languages.

The objective of this experiment is to test whether ZL increases the novice programmer's ability to beneficially transfer programming skills across programming paradigms. To be more precise, transferring skills learnt in the functional programming family to another programming family, e.g. imperative or logical. Transfer is an elusive concept and can be very difficult to measure. Therefore it follows that the present experiment, which aims to measure transfer, is more complicated than the first experiment described in this chapter.

The three levels of plan knowledge identified by Soloway and Ehrlich (1984) can be used as a base for testing the transfer of programming knowledge (Scholtz and Wiedenbeck, 1990). The three levels of plan knowledge are:

- Strategic Plans: language independent plans used by the programmer to form an overall strategy for solving a problem.
- Tactical Plans: language independent plans used by the programmer to form a local strategy for solving a problem.
- Implementation Plans: language dependent plans used by the programmer to determine how to achieve their strategic and tactical plans in the target language.

Scholtz and Wiedenbeck (1990) studied the problem of learning second programming languages and the problems associated with transfer. The following experiment follows their guidelines by using planning levels to study the effects ZL has on transfer in novice programmers.

6.2.3.1 Subjects

Two groups of novice programmers are required; a ZL sample and a control group.

6.2.3.2 Procedure

Both the ZL sample and the control group are given a problem (problem1) to solve. This problem:

- is the same for both groups.
- requires generic programming concepts common to the two languages.

To solve problem1:

- the ZL sample will use the ZL system.
- the control group will use some other programming system which belongs to the same programming paradigm as ZL.

Both groups are asked to verbalise their thoughts as they are working on the problem. This process is known as thinking-aloud. While subjects are solving the problem their thoughts are recorded.

When a subject has achieved a solution to problem1 (or a specified time period has elapsed) they are asked to work on a second problem (problem2) using a new programming language. The new programming language will be:

- nonfunctional, i.e., not in the functional paradigm.
- the same for both the ZL and control groups.

Problem2 should:

- be different to problem1, but
- require the same generic concepts as required by problem1.

It must be stressed that the two problems used in the experiment should be of a similar level of complexity, otherwise any results obtained will be inconclusive. For example:

- Problem 1: Calculate the greatest common divisor of `a` and `b`.
- Problem 2: Calculate the least common multiple of `a` and `b`.

or

- Problem 1: Write a function to convert a temperature from Celsius to Fahrenheit.
- Problem 2: Write a function to convert a measurement from inches to centimetres.

Analysing the Data

To find the time taken for each subject to complete each planning stage it will be necessary to decompose each subject's 'thoughts' into different episodes. Where each episode represents a distinct behaviour. A distinct behaviour can be defined as either:

- a change in physical activity, e.g. switching from entering text at the keyboard to drawing diagrams on a piece of paper, or
- a change in attention focus, e.g. switching between programming concepts (Scholtz & Wiedenbeck 1992).

The time taken to complete each episode can then be recorded. Each episode can then be categorised into one of the three planning stages. The total time for each subject to complete each solution will also be recorded.

We hypothesize that if ZL increases beneficial transfer then the ZL sample will spend less time at the tactical and implementation planning stages than the control group when solving problem2.

6.2.4 Hypothetical Test Results

Strategic Planning - Problem1

It is expected that both the ZL sample and the control group will spend an equal amount of time at the strategic planning stage. This is because both sample groups contain novice programmers of an approximately equal skill level. Scholtz and Wiedenbeck used experienced programmers in a similar study and classified only 3.5% of total planning episodes in their study as strategic. However, as this experiment will use novice programmers it is expected that subjects will spend considerably more than 3.5% of their total planning episodes at this stage.

Strategic Planning - Problem2

It is again expected that both the ZL sample and the control group will spend an equal amount of time at the strategic planning stage for solving problem2. However, the total time for both groups at the strategic planning stage for the second problem, will be less than the total time spent at the strategic planning stage for problem1. This is because subjects from both groups will already have been exposed through problem1 to the strategies required to solve problem2.

Problems 1 and 2 are different problems. However, for a solution to problem2, the same concepts used to solve problem1 are required. For example, if one of the generic concepts used to solve problem1 is a conditional choice, then the subject needs to:

a. discover the existence of conditionals,
b. construct a mental model (basic recurring pattern) for conditionals, and
c. tailor the mental model so it is applicable for solving problem1.

The above steps will take time to complete. To achieve a strategic plan for a conditional choice to solve problem2, a subject has to complete neither step 'a' nor step 'b'. Therefore, if the existing model for a conditional choice is reused, then it is plausible

that the total time on strategic planning taken by subjects for solving problem2 will be less than the total time taken for solving problem1.

Tactical Planning - Problem1

It is expected that both the ZL sample and the control group will spend an equal amount of time at the tactical planning stage for problem1. As both groups contain novice programmers, subjects will have previously had little practice using the generic concepts involved in solving problem1. Thus, subjects from both groups will need to build new tactical plans for each use of the concepts. The total time spent by subjects from both groups at this stage in planning will be greater than the time spent by subjects at the strategic planning stage.

Tactical Planning - Problem2

It is expected that subjects from the ZL sample will spend less time at the tactical planning stage than subjects from the control group, for the following reasons:

- The tactical plans the ZL sample formed when solving problem1 should be beneficially transferred to help solve problem2.
- Even though tactical plans should be language independent, subjects from the control group will use tactical plans that are more appropriate to the language in which they solved problem1 (section 1.5.3.2 Tactical Plans).
- It is expected that subjects in the control group should find it necessary to revise their tactical plans once they discover they do not work at the implementation planning stage.

In a similar study conducted by Scholtz and Wiedenbeck (1990) subjects often returned to revise and modify their tactical plans after discovering at the implementation stage, that their plans were too dependent upon previously used languages.

Implementation Planning - Problem1

It is expected that both the ZL sample and the control group will spend an equal amount of time at the implementation planning level. Again, this is because both groups contain novice programmers. As both groups of subjects should be unfamiliar with the

languages they are using it should take them a considerable amount of time to find constructs which can be used to carry out their tactical plans. In Scholtz and Wiedenbeck's (1990) study 79% of the time subjects spent planning was at the implementation planning stage. A similar result would be expected in this experiment.

Implementation Planning - Problem2

It is expected that subjects from the ZL sample will spend less time at the implementation planning stage than subjects from the control group. This is because tactical plans should beneficially transfer from ZL to the new language. Therefore, subjects from the ZL sample should successfully be able to create implementation plans from their tactical plans. Subjects from the control group may spend more time at the implementation planning stage for the following reasons:

- Tactical plans created by subjects in the control group will most likely make language commitments to the language used in solving problem1 and therefore will not work at the implementation stage (section 1.5.3.2 Tactical Plans).
- When implementation plans of incorrect tactical plans do not work the implementation plans themselves will also have to be revised.

Figure 64 illustrates the time spent by subjects from both groups at each of the three planning stages for solving problem1.

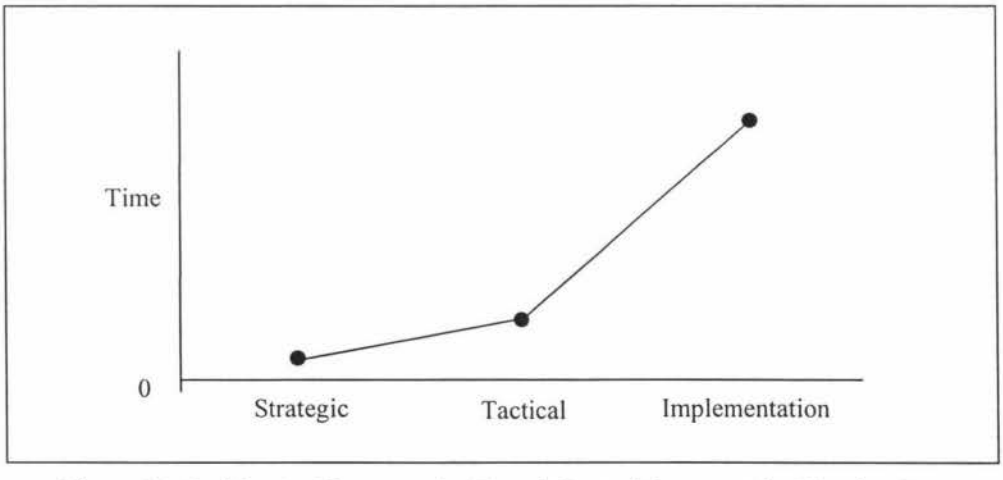


Figure 64 : Problem1 - Time spent by ZL and Control Groups at the Planning Stages

As illustrated in Figure 64, there should be no significant time difference between the ZL sample and the control group at each of the planning stages

Figure 65 illustrates the time spent by subjects from both groups at each of the three planning stages for solving problem2.

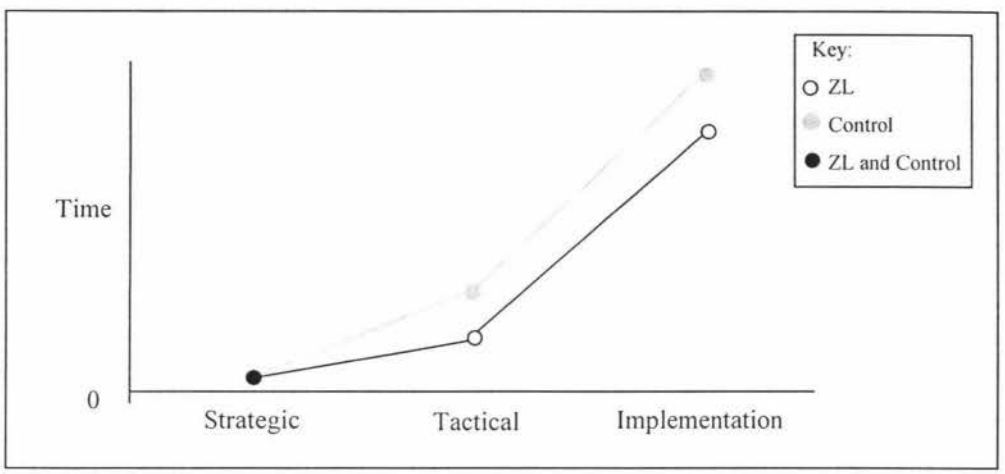


Figure 65 : Problem2 - Time spent by ZL and Control Groups at the Planning Stages

Figure 65 illustrates the differences between the time taken for each group to solve problem2. As shown, it is expected that the control group will spend more time at both the tactical and implementation stages of planning than the ZL sample. If this is the case, then it is evidence that the mental models created by the ZL sample in solving problem1 transferred to the solution of problem2 in the new language.

6.2.5 Difficulties of Measuring Transfer

Transfer is a fundamental concept in the psychology of learning information technology. We have already mentioned the difficulties of measuring it. In this section, we look in more detail at some of the potential difficulties raised by this design.

The Think-Aloud Method

The think-aloud method is a widely used technique for accessing concept formation. However, it has previously drawn criticism, particularly in the following areas:

- Verbalising thoughts can affect the performance of a subject. This can be both a positive and negative factor. In the negative, it can lead a subject to focus on a particular mental model or planning stage more than they normally would. In the

positive it can make subjects work more carefully and concentrate more on their work.

- Verbalising thoughts can affect the accuracy of the subject's account of the cognitive processes they are using. It has been suggested that some cognitive processes are unconscious and therefore, by definition, a subject cannot give an accurate account of them.

In spite of these drawbacks the think-aloud method has been used successfully many times previously for accessing concept formation. Scholtz and Wiedenbeck also used the method in their 1990 study of transfer and obtained promising results. The think-aloud method also allows access to thought processes which aren't available to anyone other than the individual (Matlin 1989). For these reasons, the think-aloud method has been used in the transfer experiment.

Construct Confusion

Careful consideration must be given to the choice of language used to solve problem2. The language must contain constructs which are sufficiently different from both the languages used to solve problem1 so as the transfer problem is made a significant part of the novice's task.

If constructs contained in the programming language used to solve problem2 too closely resemble constructs used previously by subjects to solve problem1, then the implementation plans could be affected. Subjects could confuse the use of constructs, i.e. subjects will assume a construct performs a certain task, when in reality, it performs a totally different, unrelated task. Therefore, it must be stressed that before the transfer test is implemented, careful consideration would be necessary in choosing the languages involved.

ZL Versus Imperative Languages

This particular transfer experiment does not test the effectiveness of ZL against more commonly used learning languages, which in most cases are imperative. Alternative experimental designs may be explored for this purpose. For example, the control group

could use an imperative language (i.e. a common learning language like Pascal) to solve the first problem. Thereby, we could test whether ZL or the common learning language was more beneficial to novices in the transfer of programming skills across paradigms.

7. CONCLUSION

The objective of the present work was to:

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

This goal was achieved by:

- Applying cognitive principles to the development of a simple programming language that enables novice programmers to develop simple uncomplicated mental models.

- Applying cognitive principles to the development of an easy to use programming environment for the programming language.

To gain an insight into the way programming knowledge is storage and retrieved, three theories of human memory storage were discussed.

- The Atkinson-Schiffrin Model: Atkinson and Schiffrin believed that human memory was comprised of three different storage areas: Sensory memory, which is a large capacity storage system for recording information from all senses. Short term memory which contains only the relevant information processes in sensory memory. Long term memory which is the long term storage of human memory.

- Levels of Processing: Craik and Lockhart proposed that deep processing of knowledge leads to better recall and retention of information, as deeper levels of encoding extract more of a stimulus.

- The Episodic and Semantic Model: Tulving categorised memory into two areas: episodic and semantic. Where episodic memory contains information about time-date events and semantic memory contains information about

organised knowledge of the world. Anderson proposed that this type of memory was stored as a netlike organisation of propositions in memory.

The cognitive structure of the storage of programming knowledge was also discussed. Letovsky suggested that mental models were used to help form the basis of a programmer's cognitive model. Holt et.al. suggested that programmers formed this mental model from a program's structure and function.

Mental models of expert programmers were discussed to gain an insight into the type of knowledge necessary for a novice programmer to form an expert programmer's mental model. Fix et.al. suggested five abstract characteristics which exist in an expert's mental representation that do not appear in a novice's mental representation:

- Hierarchic Structure: the depth and breadth of a mental model.
- Explicit Mappings: the links between a mental model's hierarchically structured layers.
- Basic Recurring Patterns: mental models for frequently recurring situations.
- Well Connected: how parts of a mental model interact together.
- Well Grounded: information on the physical location of program structures and operations.

The difficulty of transferring mental models from one programming language to another was also discussed. It was discovered that 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. Beneficial transfer occurs when an old skill aids the learning of a new skill.

To aid novice programmers with the learning of a new programming language the learning activities of expert programmers was discussed. Scholtz and Wiedenbeck discovered three learning activities associated with the learning of a new programming language.

1. Language Syntax: very little time is spent by experienced programmer to learn the syntax of a new language.

2. Language Semantics: as negative transfer often occurs when assumptions are made about language semantics, experienced programmers will concentrate more on learning the semantics of a new language than on the syntax.

3. Planning: planning is associated with how the levels of a mental model are put together. Soloway and Ehrlich identified three levels of plan knowledge among expert programmers:

    - Strategic Plans: language independent plans used by the programmer to form an overall strategy for solving a problem.
    - Tactical Plans: language independent plans used by the programmer to form a local strategy for solving a problem.
    - Implementation Plans: language dependent plans used by the programmer to determine how to achieve their strategic and tactical plans in the target language.

To aid the novice programmer with the construction of mental models, generic concepts common to some, if not all, programming languages were implemented in the programming language ZL.

The generic concepts and the cognitive principles they use are outlined in Table 4.

Table 4: Cognitive Principles used to aid the Novice Programmer form Mental Models of Programming Concepts

Concept	Cognitive Principles
Operator Application	• Deep Levels of Processing
Value Naming	• Syntax • Hierarchical Structure • Explicit Mapping
Conditionals	• Well Connected
Nesting	• Hierarchical Structure • Explicit Mappings • Well Connected • Lazy Evaluation
Iteration/Recursion	• Grounding • Strategic Planning • Hierarchical Structure • Explicit Mappings
Pattern Matching	• Strategic Planning • Implementation Planning • Syntax • Semantics • Strict Evaluation

Encouraging novices to use and learn the ZL programming language was achieved by applying cognitive principles to the development of the ZL environment.

The ZL environment was designed to be functional, yet easy to use. This was achieved by:

- Stationary Windows: Keeping three of the four window types stationary avoids confusion about the function of each window, places emphasis on the only moveable window, and avoids screen clutter.

- Using the Microsoft Standard: The Microsoft standard is deeply encoded within IBM PC compatible users, therefore using standard icons and names promotes button and menu behaviour recall.

- Relating Iconic Pictures to Toolbar Button Functions: Toolbar buttons unique to ZL have iconic pictures which match their names to their tasks. Matching iconic pictures to button tasks promotes recall about button behaviour.

- Separate Window for each ZL Function: The ZL environment also indirectly aids the novice programmer with the construction of their mental model by placing functions in separate windows. As the separation of functions encourages novice programmers to form breadth in their mental model, which aids the identification of basic recurring patterns.

Two experiments and their hypothetical results were discussed to demonstrate that the ZL system is well suited to supporting the development of mental models in novice programmers. The objective of the experiments were to test whether:

- ZL improves the construction of mental models for generic programming concepts. Specifically, to test whether ZL improves the novice programmer's ability to acquire the five expert mental model characteristics identified by Fix, Wiedenbeck, and Scholtz (1993).

- ZL increases the novice programmer's ability to beneficially transfer programming skills across programming paradigms. To be more precise, transferring skills learnt in the functional programming family to another programming family.

The main objective of the present project was to aid novice programmers with the cognitive construction of programs and the acquisition of program plans. This goal was achieved by researching the cognitive principles involved in:

- The storage and retrieval of programming knowledge.
- The cognitive structure of stored programming knowledge.
- The effects of transferring cognitive structures from one programming language to another.
- The learning activities involved with learning a new programming language.

These principles were then applied to the development and implementation of the programming language ZL and the ZL programming environment.

This page intentionally left blank

8. REFERENCES

Ackermann, D., Stelovsky, J., & Greutmann, T. (1990). Action regulation and the mental operational mapping process in human-computer interaction. In *Cognitive ergonomics: Understanding, learning and designing human-computer interaction.*, Falzon, P. Ed. London: Academic Press, 107-132.

Anderson, J. (1983). The architecture of cognition. In *Cognition*, M.W. Matlin, Ed. New York: Holt, Rinehart and Winston, 196-200.

Anderson, J. (1985). *Cognitive Psychology and Its Implications* (2nd ed.) New York: W.H. Freeman and Company.

Atkinson, R.C, & Schiffrin, R.M. (1968). Human memory: A proposed system and its control processes. In *The psychology of learning and motivation: Advances in research and theory* (2nd ed.). New York: Academic Press.

Böhm, C. and Jacopini, G. (1966). Computational Linguistics: Flow Diagrams, Turing Machines and Languages with only two formation rules. *Communications of the ACM*, Vol 9(5), 366-371.

Booch, G. (1991). *Object Oriented Design with Applications*. California: Benjamin-Cummings.

Booch, G., & Rumbaugh, J. (1995). *Unified method: Overview version 0.8*. Rational Software Corporation.

Campbell, R.L., Brown, N.R. & DiBello, L.A. (1992). The programmers burden: Developing Expertise in Programming. *The psychology of expertise: cognitive research and empirical A,* Hoffman, R.R Ed. NewYork: Springer-Verlag, 269-294.

Cardelli, L. (1984). Basic polymorphic type checking. *Science of computer programming,* Vol 8(2), 147-172.

Craik, F.I.M, & Lockhart, R.S. (1972). Levels of processing: A framework for memory research. *Journal of verbal learning and verbal behaviour*, Vol 11, 671-684.

DeTienne, F. (1990). Program understanding and knowledge organisation: The influence of acquired schemata. In *Cognitive ergonomics: Understanding, learning and designing human-computer interaction.*, Falzon, P. Ed. London: Academic Press, 245-256.

Fix, V., Wiedenbeck, S. & Scholtz, J. (1993). Mental representations of programs by novices and experts. *Interchi '93*, 74-79.

Goldschlager, L., & Lister, A. (1988). *Computer Science A Modern Introduction* (2nd ed.). London: Prentice Hall.

Harrison, R. (1989). *Abstract Data Types in Modula-2*. Chichester, England: John Wiley & Sons.

Holt, R.W., Boehm-Davis, D.A., & Schultz, A.C. (1987). An analysis of the online debugging process. *In Emprical Studies of Programmers: Second Workshop*, G.M. Olson, S. Sheppard, and E. Soloway, Eds., Norwood, NJ: Ablex, 33-46.

Kieras, D.E., & Bovair, S. (1990). The role of a mental model in learning to operate a device. In *Human-computer interaction,* Preece, J. & Keller, L. Eds., Cambridge, Prentice Hall.

Letovsky, S. (1986). Cognitive processes in program comprehension. In *Emprical Studies of Programmers*, E. Soloway & S. Iyengar, Eds., Norwood, NJ: Ablex, 58-79.

Luchins, A.S. (1942). Mechanization in problem solving. *Psychological Monographs*, 54(Whole No. 248).

MacLennan, B. J. (1989). *Functional Programming: Practice and Theory*. Reading, Massachusetts: Addison-Wesley.

Matlin, M.W. (1989). *Cognition* (2nd ed.). New York: Holt, Rinehart and Winston.

Nanja, M. & Cook, C.R. (1987). An analysis of the online debugging process. In *Emprical Studies of Programmers: Second Workshop,* G.M. Olson, S. Sheppard, and E. Soloway, Eds., Norwood, NJ: Ablex, 295-341.

Pennington, N. (1987). Comprehension strategies in programming. In *Emprical Studies of Programmers: Second Workshop,* G.M. Olson, S. Sheppard, and E. Soloway, Eds., Norwood, NJ: Ablex, 100-113.

Perkins, D.N., & Martin, F. (1986). Fragile Knowledge and Neglected Strategies in Novice Programmers. In *Emprical Studies of Programmers*, E. Soloway & S. Iyengar, Eds., Norwood, NJ: Ablex, 213-229.

Perry, N. (1992). *Massey Hope*$^{+}$*C*, Version 1.0. Department of Computer Science, Massey University.

Reed, K.R. (1988). *Cognition: Theory and applications* (2nd ed.). Pacific Grove, California: Brooks/Cole.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenson, W. (1991). Object-Oriented Modelling and Design. New Jersey: Prentice Hall.

Samurcay, R., (1990). Understanding the cognitive difficulties of novice programmers: a didactic approach. In *Cognitive ergonomics: Understanding, learning and designing human-computer interaction*, Falzon, P. Ed. London: Academic Press.

Scholtz, J. & Wiedenbeck, S. (1990). Learning second and subsequent programming languages: a problem of transfer. *International Journal of Human-Computer Interaction,* 2(1), 51-72.

Singley, M.K., & Anderson, J.R. (1985). The transfer of text-editing skill. *International Journal of Main-Machine Studies,* 22, 403-423.

Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10, 595-609.

Terry, P.D. (1986). *Programming Language Translation; A Practical Approach.* Wokingham, England: Addison-Wesley.

Tulving, E. (1972). Episodic and semantic memory. In *Organisation of memory,* E. Tulving and W. Donaldson, Eds. New York: Academic Press.

Van der Veer, G.C., Wijk, R., & Felt, M.A.M., (1990). Metaphors and metacommunication in the development of mental models. In *Cognitive ergonomics: Understanding, learning and designing human-computer interaction*, Falzon, P. Ed. London: Academic Press.

Waern, Y. (1990). Human learning of human-computer interaction: An introduction. In *Cognitive ergonomics: Understanding, learning and designing human-computer interaction.*, Falzon, P. Ed. London: Academic Press, 69-84.

This page intentionally left blank

9. Appendix A: The ZL Grammar

The standard BNF notation used to describe the ZL language is as follows:

::=	is defined as
'symbol'	terminal symbol
<item>	non-terminal symbol
{...}	used for grouping
+	one or more of the previous item
*	zero or more of the previous item
[...]	optional item
|	or

```
<definition>      ::=   <function>
                  |     <expression>

<function>        ::=   'fun' <identifier> <funelement>
                                  { '|' <funelement> }*
<funelement>      ::=   <pattern> '->' <expression>

<pattern>         ::=   <patelement>
                  |     '(' <patelement> {',' <patelement>}* ')'
<patelement>      ::=   <literal>
                  |     <identifier>
                  |     <pattern>

<expression>      ::=   <simple>
                  |     <qualified>
                  |     <application>
                  |     '(' <expression> ')'

<simple>          ::=   <literal>
                  |     <tuple>

<tuple>           ::=   '(' <expression> {',' <expression>}* ')'

<qualified>       ::=   <let exp>
<let exp>         ::=   'let' <letelement> { '&' <letelement> }*
                              'in' <expression>
<letelement>      ::=   <pattern> '<-' <expression>

<application>     ::=   <conditional>
                  |     <funapplication>
                  |     '-' <expression>
                  |     'not' <expression>
                  |     <opapplication>

<conditional>     :=    'if' <expression>
                              'then' <expression>
                              'else' <expression>

<funapplication>  :=    <identifier> <expression>

<opapplication>   :=    <expression> <operator> <expression>
                  |     <tuple> <operator> <tuple>
```

```
<literal>          ::=   <integer literal>
                   |     <boolean literal>
<integer literal>  ::=   {0|1|..|9}*
<boolean literal>  ::=   true
                   |     false

<operator>         ::=   <arith_op>
                   |     <rel_op>
                   |     <logic_op>

<arith_op>         ::=   '+' | '-' | '*' | '/'
<rel_op>           ::=   '<' | '>' | '=' | '<>' | '<=' | ' >='
<logic_op>         ::=   'and'
                   |     'or'

<identifier>       ::=   <character literal> {<integer literal>
                                             | <character literal>}*
<char literal>     ::=   a | .. | z | A | .. | Z
```

10. Appendix B: The ZL Type Rules

The ZL type calculation rules are as follows:

Integer Literal:

$$\texttt{x : number}$$

Boolean Literal:

$$\texttt{true : boolean}$$
$$\texttt{false : boolean}$$

Tuple Expressions:

$$\frac{\texttt{A} \vdash \texttt{expression}_1 : \sigma_1 \ \ldots \ \texttt{A} \vdash \texttt{expression}_n : \sigma_n}{\texttt{A} \vdash \texttt{(expression}_1\texttt{, ..., expression}_n\texttt{)} : \sigma_1 \times \ldots \times \sigma_n}$$

Qualified Expression:

$$\frac{\texttt{A} \vdash \texttt{expression}_2 : \sigma \qquad \texttt{A} \vdash \texttt{pattern} : \sigma}{\texttt{A} \vdash \texttt{(pattern <- expression}_2\texttt{)} : \sigma}$$

$$\frac{\texttt{A} \vdash \texttt{expression}_2 : \sigma \qquad \texttt{A.pattern} : \sigma \vdash \texttt{expression}_1 : \tau}{\texttt{A} \vdash \texttt{(let pattern <- expression}_2\texttt{ in expression}_1\texttt{)} : \tau}$$

Unary Expression:

$$\frac{\texttt{A} \vdash \texttt{x : number}}{\texttt{A} \vdash \texttt{(-x) : number}}$$

'Not' Expressions:

$$\frac{\texttt{A} \vdash \texttt{x : boolean}}{\texttt{A} \vdash \texttt{(not x) : boolean}}$$

Conditional Expressions:

$$\frac{\texttt{A} \vdash \texttt{exp}_1 : \texttt{boolean} \qquad \texttt{A} \vdash \texttt{exp}_2 : \tau \qquad \texttt{A} \vdash \texttt{exp}_3 : \tau}{\texttt{A} \vdash \texttt{(if exp}_1\texttt{ then exp}_2\texttt{ else exp}_3\texttt{)} : \tau}$$

Function Applications:

$$\frac{\texttt{A} \vdash \texttt{identifier} : \sigma \rightarrow \tau \qquad \texttt{A} \vdash \texttt{expression} : \sigma}{\texttt{A} \vdash \texttt{(identifier expression)} : \tau}$$

Operator Applications:

$$\frac{A \vdash \text{operator} : \sigma \to \tau \qquad A \vdash \text{exp}_1 : \sigma \qquad A \vdash \text{exp}_2 : \sigma}{A \vdash (\text{exp}_1\ \text{operator}\ \text{exp}_2) : \tau}$$

Patterns:

$$\frac{A \vdash \text{patelement}_1 : \sigma_1 \quad \ldots \quad A \vdash \text{patelement}_n : \sigma_n}{A \vdash (\text{patelement}_1, \ldots, \text{patelement}_n) : \sigma_1 \times \ldots \times \sigma_n}$$

Functions:

$$\frac{A \vdash \text{pattern} : \sigma \qquad A \vdash \text{expression} : \tau}{A \vdash (\text{pattern} \to \text{expression}) : \sigma \to \tau}$$

$$\frac{A \vdash \text{funelement}_i : \sigma \to \tau \qquad A \vdash \text{funelement}_n : \sigma \to \tau}{A \vdash (\text{fun identifier funelement}_i \ldots \text{funelement}_n) : \sigma \to \tau}$$