Analysis and research into errors produced while compiling a Miranda program and examining how these errors can be resolved.

Patrick Kielty
email : P.Kielty@cs.ucl.ac.uk

This report is submitted as part requirement for the B.Sc. (Hons) Degree in Computer Science at University College London. It is substantially the result of my own work except where explicitly indicated in the text.

Date : 25th April, 1997

Contents

1. Introduction

Having gone through A-level computer science, I had already been exposed to a number of styles of programming before I started my Computer Science degree at UCL. However, 'awareness' of styles was as far as it went. A lot of emphasis was placed on the imperative style of programming, which manifested itself through a great deal of Pascal programming. Tie this with teaching myself the Basic programming language when I was younger, and I couldn't imagine any other way of solving problems using computers.

This was until I started learning a functional programming language called Miranda[1]. The difference between imperative and functional programming styles could not have been greater - describing the solution to a problem as opposed to giving many instructions to a computer, which is what I was used to. I very quickly adjusted to the functional style of programming even though Miranda was my only experience with the functional paradigm.

Throughout the first and second years of my degree, I chose to tackle many coursework assignments using Miranda, abandoning C++ which was adopted by many of my colleagues, simply because of the style that functional programming employed. In the third and final year of my degree, it seemed that there would not be many opportunities to use Miranda, so I was quite enthusiastic when the opportunity arose to investigate an area of the Miranda programming language for my third year project.

This project is concerned with the error messages that can be produced when a Miranda program is being compiled, and how these messages can be interrogated in order to make them easier to understand for the new programmer, possibly offering hints and suggestions as to how an error can be resolved. The report can be read by someone with no knowledge of the Miranda language, as every effort will be made to explain everything in detail, although some familiarisation with programming in general would be an advantage.

It is not my primary objective by the end of this project to have completely implemented a new Miranda compiler which outputs more meaningful messages to a programmer - the nature of the work I am undertaking is not of that sort. Rather, I wish to research ways of making an error message more understandable, and verify some of these ideas using prototyping.

There are three main sections that form the majority of this report. After chapter 2, which details the essential features of the Miranda language and my main motivations for tackling this project, the core part of the report begins. Chapter 3 provides a detailed description of errors that can occur at compile time of a Miranda program, along with their causes and the overall effects on the compilation process. Chapter 4 considers ideas which may fulfil the goal of providing a more meaningful error message, while chapter 5 sees the specification for a prototype system which does exactly that, albeit for a subset of all the features that Miranda has to offer.

[1] Miranda is a trademark of Research Software Limited.

2. Background information

This section gives a very brief introduction to the Miranda programming language (I'm not trying to teach Miranda through this project - a detailed Miranda manual would be wasteful) and details my main motivations for undertaking this project. Also included are some applications of functional programming languages being used around the world, which helps to dispel some of the myths that people hold over functional programming languages.

2.1 Miranda in a nutshell

Miranda is a purely functional programming language, which was developed by David Turner in 1985. The main features of Miranda, which are presented shortly, are inherited from its predecessor languages SASL and KRC, also developed by David Turner at St. Andrew's University in the late seventies and at the University of Kent in the early eighties. The Miranda system was designed to run under the UNIX[2] operating system, but is not constrained to run on any particular computer. Incidentally, the name Miranda is Latin for "to be admired", and is thought to have been first used as a woman's name in Shakespeare's *The Tempest*.

A Miranda program is known as a '*script*', within which a number of *functions* can be written. A function takes *arguments* and has a *definition*, which defines how the function acts on the arguments that have been provided to it. Each argument to a function has an underlying type which is either built in (a *primitive* type) or *user-defined*. It is also possible that a function can be *polymorphic*. A polymorphic function is one which is unconcerned with the type of the argument that it is given. An example of a simple polymorphic function can be seen in the built in `id` function, that given any argument, returns it unchanged.

It is not mandatory that type information be provided by the programmer concerning data types to a function, i.e. a *function specification* or *signature*, although Miranda is *strongly typed*.

'Strongly typed' means that the system uses information about types of data to make sure that they are being used properly, for example, making sure that a function expecting a number argument does not get a boolean one. If this is the case, a *type error* is reported to the programmer.

Miranda has three basic data types built into its language : numbers (*num*), characters (*char*) and truth values (*bool*). There are also two built-in data structures : *lists* and *tuples*. Along with the functionality provided through the predefined functions and reserved names, this forms the building blocks of the Miranda Language.

A Miranda function has an '*equation*' which is the declaration of the function's arguments, and a definition which dictates the way it behaves on its arguments. If there is more than one definition per function equation, the function definition is said to be '*guarded*' for that equation, where a guard is a boolean expression preceded by a comma written after the equation itself. For example, a function which tells you if a number is odd or even can be written :

```
odd_or_even x    = "Odd", if x mod 2 = 1
                 = "Even", if x mod 2 = 0
```

[2] UNIX is a trademark of AT&T Bell Laboratories.

This function has two equations, `"Odd"` and `"Even"`, and two guards, `if x mod 2 = 1` and `if x mod 2 = 0`.

A keyword, `otherwise` exists, which can be used as the final guard and is evaluated if all other guards fail. In the function `odd_or_even`, if the guard `x mod 2 = 1` fails, then for all other numbers, `x mod 2 = 0` must be true. The second line of the function could therefore have been written `= "Even", otherwise`. The equation that precedes the `otherwise` is known as the *common* - it is common everything that is not included in one of the guards.

Miranda functions are allowed to have several alternative equations, which distinguish between possible different patterns in arguments to a function. This is known as *pattern matching*. In essence, pattern matching is another way of writing a guard, though the former is considered to be more elegant than the latter. An example of pattern matching can be seen in the function `add`, which adds all numbers in a list. The pattern match is used to detect when the list is empty, the sum of which is 0!

```
add [] = 0
add (x : xs) = x + add xs
```

Miranda is a *higher-order* language and functions are considered to be *first-class citizens*, which means that functions themselves can be passed as arguments to another function or returned as results. Functions can also be *partially applied*, that is, a function can be given less than its full complement of arguments without causing an error.

Expressions in Miranda are *'lazily evaluated'*. Lazy evaluation means that no expression is evaluated until its result is required to fulfil some need. This is a direct contrast to the *'strict'* style of evaluation, which is employed in the programming language Standard ML [12]. An important implication of this is that the programmer has exposure to infinite data structures, which provide the means for *"handling problems of interactive input/output and communicating processes within a functional framework"* [16].

2.2 *Teaching Miranda*

There were several reasons behind my choosing to analyse the problems that occur during the compilation of a Miranda script. Most notable of these was the fact that at Computer Science in UCL, the first programming language that undergraduates are exposed to is Miranda.

While it is likely that a first year undergraduate will have had some exposure to an imperative programming language before, like C, Pascal, Basic etc., the chances of anyone having used a functional language are small. It is important therefore that the starting level is right, and as much help is provided throughout the duration of the course as possible, not only to ease the new programmer into programming as a whole but to 'reorient' the imperative programmer as well.

During the twelve weeks in which undergraduates are exposed to Miranda, the most frustrating thing is not the amount of work that has to be done, but according to students, lecturers and demonstrators alike, the problem lies in the quality of the error messages that are presented to the programmer after a failed compilation of a Miranda script file. Another aspect of Functional Programming that often confronts new programmers is the idea of *Typing*.[6]

New users of Miranda are unlikely to have encountered concepts such as unification and application of types, yet are presented with errors which report that something cannot be 'unified' or 'applied' to something else. Presenting such an error in an alternative, easier to understand way would clearly be of benefit here and help flatten the steep learning curve usually associated with familiarising yourself with a new style of programming.

2.3 Functional programming in practice

The comparison between imperative and functional styles of programming over the years has tended favourably towards the imperative style, due mainly to speed and efficiency considerations. However, more recent implementations of functional programming languages have comfortably competed with the speed of compiled imperative languages, such as Pascal and C. This has undoubtedly made functional languages more attractive. [8, 14]

In his very interesting paper, "Conception, Evolution, and Application of Functional Programming Languages" [9] Paul Hudak dedicates a whole section of it to *dispelling myths* about functional programming, the one he seems most concerned about is that functional programming languages are toys!

Some very interesting, efficient and worthwhile implementations of functional languages are presented, which show that they are not just teaching aids or whatever people may have been led to believe. Some of the more prominent implementations and real application developments that he mentions are :

- The Alfl compiler at Yale, which generates code that rivals code produced by conventional language compilers, i.e. non-functional ones.

- Los Alamos and Lawrence Livermore laboratories, along with the dataflow groups at the Michigan Institute of Technology and the functional programming groups at Yale have written many functional programs for scientific computation.

- Quite a large expert system (EMYCIN) has been written in SASL, a Miranda predecessor.

- A lazy functional language is being used by IBM for graphics and animation.

- Another lazy functional language has been used by Oxford researchers to write a program for GEC Hirst Research Laboratory to design VLSI chips.

In industry today, there is a great requirement to have programs fully specified before they are implemented. For example, given some task that has to be completed, specify it in Object Z, and then implement it in C++. David Turner has presented the idea of *executable specifications* [13], where instead of specifying something in one language and having it implemented in another, the language of specification hence the specification itself, is executable.

Miranda is such a language, but functional languages are 'traditionally' associated with lack of input/output, GUI support and speed of execution. However, there are certain benefits to an executable specification, most notable of which are the savings in costs involved in a two-fold specification/implementation development cycle for non specification languages.

3. Description of errors produced by Miranda

Errors can arise from Miranda programs at both compile-time and at run-time[5]For the purposes of this project, I am concerned with errors that have been produced at compile-time, as I am working on the assumption that if a Miranda script compiles successfully, i.e. without error, then run-time errors will not occur unless the program is given erroneous input by a user, e.g. A boolean argument is given to a function instead of a number.

It is worth noting however, that although a script file may compile without error the script may contain semantic errors. A semantic error may occur when a function is defined to work on its arguments in some way that is not what is required by the programmer. As an example, consider the function `mult`, which multiplies two numbers.

```
mult x y = x + y
```

The definition of `mult` is syntactically correct and compiles without error. However, given two number arguments, `mult` returns their sum instead of their product. This is an error in the semantics of the function. Whether a semantic error has occurred in a script file or not, it will not be detected by the compiler and as long as the Miranda type system is obeyed and the syntax of the script is correct, semantic errors are of no consequence to this project.

This chapter of my project report is very important in the sense that it provides an *understanding* of the common errors that occur during compilation of a Miranda script and presents some *reasons why* certain errors occur.

The first thing that I do in this chapter is to categorise the various errors that can occur into groups, which are representative of the severity of the error. This is usually determined by the effect that the error has on the compilation process. As the chapter progresses, I consider each of the new categories of error and depending on which category it is, discuss specific errors that occur within each group and likely causes of the error.

3.1 Categorisation of Miranda Errors

After research, I have concluded that a compile-time error can be produced for any one of the four different reasons below. These are :

- A syntax error, undeclared typename or badly formed type has been detected in the source code.
- A function definition error is present.
- A type error has been found.
- The programmer specifies or defines something which isn't used, in which case a warning is produced.

Depending on which of the above are detected by the Miranda compiler, compilation will either be abandoned (an error is detected which causes compilation to halt) OR compilation will terminate naturally. Note that a completed compilation does not necessarily mean a successful one!! In general, the effects of error detection on the compilation process are :

1. Compilation will be abandoned immediately, if at any point the compiler detects a syntax error, undeclared typename or badly formed type in the source code. It does not matter where in the script the error has occurred, none of the functions in that script file will be operational after compilation has been halted. Similarly, no functions will be operational until this error has been resolved. For this reason, these errors will be known as **FATAL**.

2. Compilation will continue if a type error and/or a function definition error occurs in a script. Unlike when a syntax error occurs, only the function in which the error is found will be unusable at run-time or from within the rest of the script. This will be the case until the error has been resolved by the user and the script recompiled. Although the compiled script will be operational if a type or function definition error occurs, if there is a strong dependency by the rest of the program on the erroneous function then it is unlikely that the program will operate properly. These errors will be referred to as **SERIOUS**.

3. Compilation will also continue if the Miranda compiler detects unused local definitions, or a function which is specified but not defined. In this case, a 'warning' is written to the user notifying them of what has happened. Provided that no Fatal or Serious errors have occurred, program operation will be unaffected. Representative of the actual error, these errors will be known as **WARNINGS**.

3.2 Order of compilation

The Miranda compiler will check for any Fatal errors that may be in a script file before it looks for Serious errors or Warnings. As compilation is immediately halted on detection of a Fatal error, only one of these will ever be reported at one time. If a script is full of Fatal errors that go unnoticed by the programmer, a lot of time may be spent on editing and recompilation of the script file!!

Failing the presence of a Fatal error, the compiler checks the script file noting all Serious errors and Warnings that occur. As the presence of these errors does not cause compilation to be abandoned, they are all reported to the programmer at the end of the compilation session.

Having categorised the errors that can occur during compilation of a Miranda script, I will now present some specific causes of Fatal errors and Warnings by way of erroneous functions and their subsequent compilation.

During my research of various errors and their causes, I contacted David Turner, the author of Miranda, in an attempt to obtain a list of all Fatal errors and Warnings that could occur during compilation of a script. Unfortunately, he was unable to provide this list. Therefore, the list below covers as many Fatal errors and Warnings as I could generate, but is by no means exhaustive. There are many areas of the Miranda language that I am unfamiliar with, and I can't be certain that a Fatal error or Warning that I am not aware of would not occur in one if these areas.

As for Serious errors, the number of these that can arise during compilation is potentially infinite and is really determined by the complexity of the code being compiled. For this reason, I intend to present some common reasons why Serious errors may occur and highlight these with specific examples.

3.3 Fatal errors

Although these errors are the most fatal to a script, they are often the easiest to resolve. The format of a fatal error message can be any of :

syntax error - *<what has actually caused the error>*
error found near line *<line number>* of file "*<file in which the error occurred>*"
compilation abandoned

undeclared typename "*<...>*" (line *<line number>* of "*<file in which the error occurred>*")
typecheck cannot proceed - compilation abandoned

badly formed type "*<...>*" in specification for "*<...>*"
<some characteristics of the badly formed type>
(line *<line number>* of "*<file in which the error occurred>*")
typecheck cannot proceed - compilation abandoned

Fatal errors, by their very nature, imply that the programmer has misunderstood the way a script should be written syntactically. Most errors that do occur can be resolved by inspection, e.g. if the error reports an "unexpected end of file", you can be sure that the programmer has either omitted a closing bracket in the definition of the last function in the script, or completely omitted the body of the function altogether.

For an inexperienced programmer, simply reporting an "unexpected end of file" may lead to a lot of wasted time looking at a screen full of code that you can't see anything wrong with. What would be more appropriate here, was if some diagnosis of the problem was also given.

In general, this is the case for most Fatal error messages. The fundamental problem with the way these errors are reported is that not enough information is given as to the cause of the problem. Rather than just presenting the error itself, a cure could also be offered, possibly based on some rule which might say *"If this is the error, then we know that this is what has caused it..."*

Over the page are 10 of the most common Fatal errors that can be detected during the compilation of a script and examples of code from which the error may have arisen. After the examples, an explanation of the reason for the occurrence of the error is given, with relation to the specific code example. This explanation could be used to facilitate the above mentioned 'cure' to an error, whereby the programmer is advised how to resolve a problem.

3.3.1 *Error : "formal text not delimited by blank line"*

Code causing the error and the result of compilation :

```
> || Literate Script.

> fool x = x
Comment : this comment hasn't got a blank line between it and
the code above
```

```
syntax error: formal text not delimited by blank line
error found near line 5 of file "errors.m"
compilation abandoned
```

By convention, there are two ways in which you can code a Miranda script - either as a 'Literate Script' or not, the former being more popular than the latter. In Literate Script, lines of code are prefixed by '>', and comment lines, although not prefixed by any unique symbol, must have at least one blank line between themselves and any previous or subsequent lines of code.

The compiler uses these rules to differentiate between what should and shouldn't be compiled. If Literate Script is not used, comments must be prefixed by two vertical bars : '||'. This is obviously not the case in the function `fool`, above.

Concerning this error, the reason given above for its cause is the only one that leads to this error.

3.3.2 *Error : "non-escaped newline encountered inside string quotes"*

Code causing the error and the result of compilation :

```
> message
>     = "This is a message that I want to put to the screen, but
>       I haven't got enough room in my text editor, so I'll go
>       to the next few lines."
```

```
syntax error: non-escaped newline encountered inside string quotes
error found near line 5 of file "errors.m"
compilation abandoned
```

If at any point you are required to type within "quotes", i.e. you are entering data that is text, the Miranda compiler will not let you continue onto the next line without closing the quotes, going to the next line and then reopening them. Each of the "quoted" lines must be joined together with the append '++' operator. Similarly, the same error will be reported if only one line is intended to be within quotes, but the quotes are not closed.

As was the case for the error, "formal text not delimited by blank line", this error will only detected when code similar to that shown above is detected within a script, i.e. quotes are not closed.

3.3.3 *Error : "unexpected end of file"*

Code causing the error and the result of compilation :

```
> foo3 x = x, if (x < 3

syntax error - unexpected end of file
error found near line 4 of file "errors.m"
compilation abandoned
```

A minimal requirement for successful compilation of a Miranda script is that all functions are defined, i.e. there is a body to the function, and that all brackets that have been opened are closed.

In some cases when the above criteria are not met, the compiler will parse all of the source code looking for the rest of an expression or the function body, right up until no more code exists, at which point the error is generated. It can be seen above that `foo3` does have a function body, but an open bracket goes unclosed before the end of the code.

Two reasons have been presented here that lead to the unexpected end of file error : brackets aren't closed or a function doesn't have a body. Although these are the only two explanations for why such an error would occur, it is not contained within the error message which function is actually in error. Some manipulation of the original script would be required here if this function were to be provided to the programmer.

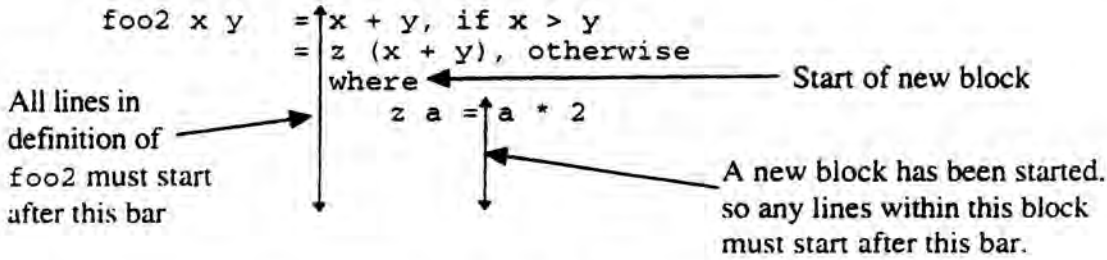
3.3.4 *Error : "OFFSIDE"*

Code causing the error and the result of compilation :

```
> foo2 x = x, if
>    x < 3

syntax error - unexpected token "OFFSIDE"
error found near line 5 of file "errors.m"
compilation abandoned
```

Although the above code may seem correct, the way it is written defies the Miranda Offside rule (actually derived from Landin's 'offside rule' [10]) which governs the way in which the body of a function must be written. Each time a programmer types an '=', unless it is to compare arguments or results, it can be considered that a new 'block' is being implemented. The first and all subsequent lines of the current 'block', must be indented at least one space right of the preceding '='. Put more eloquently, *"Indentation of inner blocks is compulsory, as layout information is required by the compiler to determine the correct parse"* [13]. The below diagram gives a pictorial description of the offside rule :



An offside error will only occur when the offside rule has been disobeyed.

3.3.5 Error : "unreachable case in defn (definition) of .. .. .."

Code causing the error and the result of compilation :

```
> foo4     = foo4 x
> foo4 x   = x
```

```
syntax error: unreachable case in defn of "foo4"
error found near line 4 of file "errors.m"
compilation abandoned
```

On its own, `foo4 x = x` simulates the identity function. On its own this is fine, until you consider that the function `foo4` has been assigned to an identifier called `foo4`. This causes obvious problems in that `foo4` can never actually evaluate itself completely - a final case is unreachable.

This is not the only reason why this error may occur, as it may do so whenever an identifier is given the same name as the function that it calls. Already included in the original error message is the name of the identifier / function that is in error and this can be presented to the programmer with some details of what might cause an unreachable case.

3.3.6 Error : "nameclash, ... already defined"

Code causing the error and the result of compilation :

```
> foo5 = "Functional programming is great…"
> foo5 = "…but I'd rather be surfing!"
```

```
syntax error: nameclash, "foo5" already defined
error found near line 5 of file "errors.m"
compilation abandoned
```

Miranda does not allow identifiers or functions to be assigned more than one value in a script file, unless they are in the scope of a where block, in which case a redefined identifier or function overrides the definition of the corresponding identifier or function outside the where block.

If more than one value is assigned and not done so within a where block, instead of picking a value randomly at run-time if called upon, The Miranda compiler reports an error. This problem is illustrated in the definition of `foo5`. This explanation for the nameclash that has occurred above is fundamentally the same for all nameclashes that occur - if anything is defined twice, this error is produced.

3.3.7 Error : "unexpected token .. .. .. "

Code causing the error and the result of compilation :

```
> get_rest [] = []
> get_rest (x : xs) = x : get_rest xs, if (x ~= '')
>                   ++ get_fn xs, otherwise
```

```
syntax error - unexpected token "++"
error found near line 69 of file "errors.m"
compilation abandoned
```

The syntax of a Miranda script file is very specific. When a script is parsed, the compiler breaks down the code into 'understandable' chunks, known as tokens. If the compiler detects a token where it shouldn't be, it will report this fact to the programmer by way of the above error.

In the common of the function `get_rest`, the compiler sees a '++' when in fact, there should probably be an '=' there. However, there are times when an unexpected token is more obscure than the trivial one above, so it is hard to do anything but report the unexpected token to the programmer. This error is very general and is easily caused by many different unexpected tokens.

3.3.8 *Error : "unbound type variable ..... "*

Code causing the error and the result of compilation :

```
> polylist == [*]
> a :: polylist
> a = ['x']
```

```
syntax error: unbound type variable *
error found near line 3 of file "errors.m"
compilation abandoned
```

The polymorphic type variable `*`, is considered 'not bound' to the left hand side of the definition of the type synonym `polylist` on line 3 above. This Fatal error is similar to the Serious type error "undefined name ....." (see later), which is used when names are out of scope in functions.

To resolve the problem above, the type variable on the right hand side of the definition of `polylist`, must also be present on the left hand side, i.e.

```
> polylist * == [*]
```

Whenever a type is unbound to a type synonym this error will be produced, and I have never experienced any previous case when this error is produced for a different reason.

3.3.9 *Error : "badly formed type ..... "*

Code causing the error and the result of compilation :

```
> polylist * == [*]
> a :: polylist
> a = ['x']
```

```
badly formed type "polylist" in specification for "a"
(typename polylist has arity 1)
(line 13 of "errors.m")
typecheck cannot proceed - compilation abandoned
```

This error is directly related to the previous unbound type variable error. Once the definition of `polylist` has been altered so that the polymorphic type variable `*`, on the right hand side, also features on the left, an error is presented complaining about the line `a :: polylist`. This error is all to do with the arity of `polylist`. What is arity? Something is said to have arity 1, if it takes one argument, arity 2 if it takes two arguments and so on.

In the line `polylist * = [*]`, `polylist` is given arity 1, yet in the following line, `polylist` is used with arity 0. The compiler expects to see here a type which can be substituted

for * in the definition of `polylist`. To make the definition of `a` correct above, the previous line would be written

```
a :: polylist char
```

This is one example of a cause of a badly formed type error, but in general these will arise whenever an identifier is constructed incorrectly using a type synonym or otherwise that is not used properly.

3.3.10 Error : "undeclared typename ..... "

Code causing the error and the result of compilation :

```
> foo :: nu → num

undeclared typename "nu" (line  47 of "errors.m")
typecheck cannot proceed - compilation abandoned
```

When a function specification is provided, the types that are used must themselves be completely declared, that is, they must be user defined (algebraic types / abstypes) or built-in Miranda types. In the above case, the typename `nu` is not recognised as either a user defined or a built-in type by the compiler, so an appropriate error is reported.

There are a potentially infinite number of misspellings of typenames or incorrectly defined types, but the principle behind the cause of this error is always the same. The type system is the fundamental building block of the Miranda language and must be adhered to when composing scripts. If the compiler can't understand the types with which it is to work, it simply stops compilation.

3.4 Serious errors

3.4.1 The Function Definition error

A function definition error has the general format :

incorrect declaration (line *<line number>* of "*<file in which the error occurred>*")
specified, *<function name>* :: *<the programmers idea of the function specification>*
inferred, *<function name>* ::*<what Miranda made of the function from its definition>*

Function definition errors are typically caused when a programmer has implemented a function to behave exactly as they wish, yet specified the signature of the function incorrectly. If this is the case, then all that is needed to rectify the problem is to rewrite the function specification just as the Miranda compiler inferred it.

However, sometimes when this error occurs, the programmer has specified the function as intended and it is the implementation which is incorrect. It is not now so easy to rectify the problem as before.

The problem that I am faced with here is one of indeterminism. There is no way of telling what was actually meant to be implemented by the programmer in the first place. Although there is a strong chance that it is the specification which is wrong, there is also a possibility that it is the implementation which is incorrect.

Therefore, as there is no software that I know of that can read a programmer's mind, there is not one generic rule that can be applied to provide a sure fire solution to a function definition error! I am limited in my capacity to advise here, in that I could only offer a suggestion to the programmer as to what they may *like* to do. The only real solution here is for the programmer to plan carefully what they want to implement before hand. An example of a Function definition error can be seen below :

```
> foo :: num → num
> foo x    = True, if x = 0
>          = False, otherwise
```

```
incorrect declaration (line 4 of "errors.m")
specified, foo::num→num
inferred,  foo::num→bool
```

From the definition of `foo`, it can be seen the function is consistently defined to return a type `bool` in each case for the function, yet the specification provided by the programmer says that this return type should be a `num`.

3.4.2 The Type error

There are many different reasons why a type error may be detected during the compilation of a script. The format of the error message can be either of :

type error in definition of *<function name>*
(line *<line number>* of "*<file in which the error occurred>*") *<description of error>*

(line *<line number>* of "*<file in which the error occurred>*") undefined name "*<...>*"

The nature of any Serious error that is detected, depends entirely on the complexity of the portion of code in which the error occurs - the simpler the program, the less likely you are to get a confusing type error. Type errors can range from the quite easy to comprehend, such as

```
cannot unify bool with num
```

, to the slightly more difficult to decipher, one of which may look like

```
cannot unify [char]→[char]→num→[num] with [char]→[[char]]→num→num
```

So what does all of this mean? In short, a type error will occur if certain patterns in an equation of a function definition conflict with other patterns, where a pattern can be a return type, or part of an ongoing operation which will eventually lead to a return type. If a conflict does exist, it is said that one particular equation does not **unify** with the other.

In a complex function with many equations and guards, making one slight change to an argument type in order to unify two equations, may have side effects on the compilation of the rest of the function causing further and potentially harder to resolve errors to occur. What I think would make life easier for the programmer, would be if the compilation process could attach appropriate argument names to the types that are in error.

By doing this, you may be able to report in the error message the arguments to which types are attached, as opposed to the types themselves. This alone would save programming time, as well as the sanity of the programmer! Below are four reasons why type errors may have occurred in a script.

3.4.2.1 A function is defined to return differing types for more than one evaluation of one of its definition patterns.

Consider the following two Miranda functions and their subsequent compilation :

```
> wrong1 x = x, if x < 3
>          = True, otherwise

> wrong2 []     = []
> wrong2 x      = x * 2
```

```
type error in definition of wrong1
(line 4 of "errors.m") cannot unify bool with num
type error in definition of wrong2
(line 7 of "errors.m") cannot unify num→num with [*]→[**]
```

A Miranda function must return a value of the same type for every evaluation of that function. This is clearly not the case here, as the function `wrong1` can return a 'num' in the first line of the function definition (the guard `'if x < 3'` determines x to be a num here, as `'< 3'` is a test on numbers) but the common evaluates to the boolean (bool) value True. Hence, there is a conflict in types and the appropriate error is reported.

However, the reason for `wrong2` giving a type error may not be so obvious. Things become more clear when you consider that the function can be also written like this :

```
> wrong2 x = [], if x = []
>          = x * 2, otherwise
```

Now, just like in the definition of `wrong1`, it can be seen that the guard on the first line of `wrong2` infers a different return type for the function than the common does in line 2. However, note the original error message :

```
...cannot unify num→num with [*]→[**]
```

The above reasoning explains the conflict in types for the those highlighted above, but there is also a conflict reported with argument types. The reason for this is because, unlike in the definition of `wrong1`, `wrong2` makes a specific check on its' argument type in each of the possible cases, i.e. if x is the empty list or if x is a number (not actually a check, but it is inferred by the fact that multiplication is an operation on numbers). In `wrong1`, the common is unconcerned with the type of x.

3.4.2.2 *An operator or other function (built-in or user defined) is provided arguments of inappropriate types.*

The next two functions illustrate this.

```
> list_of_bools :: [char] → char → [char] → [bool]
> list_of_bools x y z = [(member x y)] ++ transform z

> transform :: * → bool
> transform z = True
```

Attempted compilation of these functions gives the error message :

```
type error in definition of list_of_bools
(line 2 of "errors.m") cannot unify bool with [bool]
```

The reason that this error occurs is because the type signature of the append operator is ++ :: [*] → [*] → [*]. The built in function `member` is used so that it returns a [bool], but the function `transform` returns only a bool. So in this case, the arguments to ++ are incorrect and a type error is reported.

3.4.3.3 *A function passes too few or too many arguments to another function.*

The two functions below, although trivial ones, demonstrate this.

```
> display :: [*] → **
> display x = myshow (min x, max x)

> myshow :: (*, **, ***) → *
> myshow (x, y, z) = x
```

```
type error in definition of display
(line 3 of "errors.m") cannot unify (*, *) with (*, *, **)
```

`display` is a function that takes one argument, and passes the results of certain operations on this argument to the function `myshow` (the operations are unimportant here). The problem lies with the fact that `myshow` is expecting a three tupled argument, but only gets a two tuple. It is this fact that is reported in the error message.

3.4.2.4 A function calls or tries to use something which is undefined.

For example :

```
> silly [] =[]
> silly (x : xs) = (shuff x) ++ silly xs

> add x y = x + y + z

(line 4 of "errors.m") undefined name "shuff"
(line 6 of "errors.m") undefined name "z"
```

In the case of the `silly` function, the argument `x` is used as an argument to an undefined function, whereas the function `add` tries to use something called `z`, which is neither in the scope of the function as an argument, nor is it a function.

3.4.3 A special case for Type errors : 'show'

The 'show' function in Miranda, provides a 'front-end' to functions such as shownum, showbool, showchar etc., which will take an arbitrary Miranda value and convert it to a printable representation.

From the above, one would expect 'show' to be polymorphic, but in fact, *"show is not a true polymorphic function of type * → [char], but rather a family of monomorphic functions with the types T → [char] for each possible monotype T"* [15].

It is often the case that programmers do not realise this, and may attempt to write a function such as

```
> to_screen x = "x : " ++ show x
```

and promptly be presented with the type error :

```
type error in definition of to_screen
(line 3 of "errors.m")  use of "show" at polymorphic type *
```

The compiler has to know the actual type of `x`, so that it knows which of the family of monomorphic functions to use. In the absence of this type information, the compiler considers the function `to_screen` to be illegal, as can be seen.

The way to resolve this problem is by providing adequate type information in the script file to be compiled. In the case of user defined types, where a suitable monomorphic function may not exist, the programmer can write a function, say `my_show` (or any other name), and specify it precisely to take whatever the user defined type is and return a `[char]`. For example, to show a user defined `tree` type, the programmer may write :

```
> my_show :: tree → [char]
> my_show x = "This is it : " ++ show x
```

Providing that `tree` is correctly defined, the compiler will not complain about this function.

The reason that this subsection is dubbed a special case, is because the error above is a common type error and is probably the only one that can be resolved on inspection, by which I mean, whatever the context of the error, the remedy will always be the same.

3.5 Warnings

As with all other errors, there isn't one specific format for a Warning, which can be either of :

SPECIFIED BUT NOT DEFINED : *<what it is!>*;

warning, script contains *<description of what is being reported>*
(line *<line number>* of "*<file in which warning occurred>*") "<...>"

Similar to the way a Fatal error can usually be resolved by inspection, so can a Warning. In reality, Warnings make no difference whatsoever to the operation of a program, unless of course you are calling something which you have specified but not defined!

Warning messages are very straightforward and are the only ones that I consider to give enough information on the problem that has occurred. Even though, some diagnosis could still be provided, if only for the benefit of an inexperienced Miranda programmer. If anything this would promote good programming practices! There are two specific warnings that can be detected during compilation.

3.5.1 Error : "SPECIFIED BUT NOT DEFINED : .. .. .. "

Code causing the error and the result of compilation :

```
> foo :: num → num

SPECIFIED BUT NOT DEFINED: foo;
```

The specification of the function `foo` is perfectly valid, but the only problem is that there isn't a function definition to go with it. The compiler simply warns the programmer that this is the case. This error will always be produced when something is specified but not defined.

3.5.2 Error : "warning, script contains unused local definitions:-"

Code causing the error and the result of compilation :

```
> name = first_name
>          where
>          first_name = "Patrick"
>          surname = "Kielty"

warning, script contains unused local definitions:-
(line 6 of "errors.m")                surname
```

The identifier `name` should hold the string `"Patrick"`, which is assigned to the identifier `first_name` within a 'where' block. However, also within the 'where' block, an identifier `surname` is assigned a string, even though this is never used anywhere within the local definition. The compiler reports this to the programmer, but as is the case with any other warning, program operation is not affected.

4. Analysis of methods to resolve errors

The Miranda functional programming language already has its own fully operational compiler which performs the necessary tasks in order to tell if a program is error free or not. The problem that I am trying to tackle is not to write a new compiler but to incorporate more information into the content of an error message in order to speed up the process of locating, resolving and successfully recompiling a script file.

Bearing the above in mind, I have two things that I can work with to try and fulfil my goal : the source code from which the error occurred and the error that was produced by the compiler in the first place. The error message gives me an idea of where the problem has occurred and its nature, and if necessary the erroneous code can be manipulated in some way to provide more information on the error, which can then be presented to the programmer.

The amount of effort required to be in a position to offer more information on an error depends very much on the error that has occurred. No error that presents itself through the compilation process is trivial, although it is the case that with experience, certain errors become more easy to resolve than others. The bulk of this section of my report looks at the methods involved in eliciting that extra information on an error, in order to try and make most errors that occur reasonably easy to resolve, regardless of previous programming experience.

4.1 Isolating an error

Whenever Miranda is started, if the file (default 'script.m' or otherwise) with which it is started contains text, this file will be compiled and any error messages be output to the screen. For example, starting up Miranda with the file `script.m` may take this course :

```
UNIX prompt : mira script.m


                T h e   M i r a n d a   S y s t e m

              version 2.014 last revised 24 May 1990

              Copyright Research Software Ltd, 1990


(1000000 cells)
compiling script.m
syntax error: nameclash, "foo" already defined
error found near line 9 of file "script.m"
compilation abandoned
for help type /help
Miranda
```

Much of the information that the programmer is presented with is not related to the actual error that has occurred in the script file. The programmer will only be concerned with the nature of the error, where it has happened in the code and the overall effect that the occurrence of the error has on the compilation process. More importantly, any system that could be written to provide more information on an error would also only be interested in the above, namely :

```
syntax error: nameclash, "foo" already defined
error found near line 9 of file "script.m"
compilation abandoned
```

Moving away from the specifics of the above error, it is noticeable that each individual error message that occurs has a certain number of lines over which information about the error is spread. It is important that this is understood : *one* error message is considered to be the number of lines over which information about *one* of the errors that has occurred is spread. To isolate a particular error based on the error message itself, it is important to know how many lines are contained within a message for each of the errors that can occur.

The table below shows the number of lines that are relevant to one specific error depending on what the error has started with.

ERROR STARTS WITH :	LINES OF RELEVANT INFORMATION FROM START OF ERROR
syntax error...	3
badly formed type...	4
undeclared typename...	2
incorrect declaration...	3
SPECIFIED BUT...	1
warning, script contains...	2

It is debatable whether there should be an entry for 'incorrect declaration...' in the table above, as the number of lines over which the complete error is spread depends on the incorrect signature or inference of a function. However, it can be assumed that any line which continues onto the next and it is not desirable for this to happen, can be joined back together as required for investigation. By this reasoning, there could actually be an entry in the above table for type errors and undefined names, the relevant lines of code from the start of an error being 2 and 1 respectively.

In summary, it is reasonable to assume that it is possible to isolate each error that occurs during the compilation of a script, regardless of how many errors there are. This can be achieved by 'recognising' the *main* feature of the error that is present (the left hand column of the above table), and taking the appropriate number of lines (right hand column) and grouping them together in some way. These groups of errors could then be interrogated to try and offer some ideas on how they could be solved.

More formally, given any file that has to be compiled, an overview of the complete procedure of what has to be done to provide a 'meaningful' error message is as such :

1. Start up Miranda with that file
2. Redirect results of compilation to another file, outfile
3. Remove all Miranda headers and non-error information from outfile
4. If errors have occurred during compilation
  4.1 Recognise and group all errors so they can be investigated (isolate errors)
  4.2 Perform some kind of analysis on the errors
  4.3 Present meaningful error messages to the programmer
5 Else, report a 'clean' (error free) compilation to the programmer

4.2 Resolving by inspection

As mentioned in chapter 3, Description of Errors produced by Miranda, a Fatal error or Warning can usually be resolved by inspection, that is, some rule can be devised that says "*If this is the problem, then this is how it is solved*". There was also the special case for type errors concerning the show function (3.4.3), which could also be resolved by inspection.

The first thing that springs to mind when you think that for every Fatal error or Warning that occurs, for which there is a common cause and subsequently a common remedy, is to build some kind of expert system that can recognise what error has been detected and offer a suggestion to the programmer as to what they might want to do to get rid of it. However, the system would still need to maintain the useful information that the original error message contained in the first place, such as the function in which the error occurred and its line number in the source code, as well as the actual specifics of the error.

The rules that would govern the operation of the expert system would be determined by the content of the error message. From the description of Fatal errors and Warnings presented in 3.3 and 3.5 respectively, you can see that each error has a unique part to it, for example, "unexpected end of file" or "unreachable case in definition of....." etc.

Fatal errors or warnings will occur in a script file because of possibly two or three reasons. More often than not, an error has one specific cause - nothing else could cause the same error to appear.

In section 4.1, a table was presented which showed that depending on what an error began with, there were a certain number of lines *from the start of the error message* that contained all of the information about the error that the compiler produced. Within this

information, more specifics are given as to the error's nature, e.g. unexpected end of file, nameclash etc. If you can recognise certain patterns in an error message, e.g. an unexpected end of file has occurred, then a predefined solution could be on hand to be presented to the programmer.

The afore mentioned unique parts of an error message would form the rules of the expert system. An example of one of these rules might be :

```
if "unexpected end of file" is part of the error message, then offer
appropriate solution for unexpected end of file.....
```

The solution that would be presented to the programmer would only form part of the output. They would also be presented with most of the information that the Miranda compiler would have originally output, although in a different form. The general format of the output would be :

```
A <error type> has been detected on line <line number>
Problem : The error itself
Diagnosis :
          <Solution or hints on  how to solve the error>
```

4.2.1 Function definition errors

Section 3.4.1 highlighted the problem that exists with Function definition errors, in that you can never be sure that the programmer has implemented a function correctly and specified it incorrectly or vice versa. All that you can do here is offer some hint as to what the programmer may wish to do to try and get rid of the error, but never a firm solution. For this reason, Function definition errors can be considered to be resolvable by inspection.

4.3 Formal parameters and return types

When a Type error occurs in a function, what would constitute a more meaningful error message than already exists? Let us return to the simple type error :

```
type error in definition of wrong1
(line 4 of "errors.m") cannot unify bool with num
```

which is generated when trying to compile the code

```
> wrong1 x = x, if x < 3
>          = True, otherwise
```

`cannot unify num with bool` may seem easy enough to understand, and looking at the code, you can see that the return types are different and have to be altered.

However, the 'new' functional programmer is unlikely to be familiar with the intricacies of unification and may need some more information on the error that has occurred. General information about how functions have to return the same types etc. can be provided regardless of the content of the error message, but to make the information that is provided comprehensive, some specific information about the function that is in error would need to be given.

Each Miranda function has formal parameters and return types which are calculated by applying some equation within a function definition to the formal parameters. A type error message can report conflicts in the formal parameters or the return types or both, yet it is never stated which of these is actually the case. In the example function `wrong1`, the error that is reported leaves the programmer to work out where the conflict in types has occurred - it just reports what cannot be unified. The first thing that it might be useful to do then, is work out for the programmer whether the error reports conflicts in formal parameters or return types or both.

At this point, it is worth noting that each equation in the definition of a function can itself be written as a separate function. To understand what I mean here, consider that two separate functions can be built from the single function `wrong1` :

```
> wrong1&1 x = x, if x < 3
> wrong1&2 x = True
```

Note that the guard remains but the common is removed. The reason why is explained below.

Each new function that has been written takes its definition from one of the equations of the original functions. A unique identifier has been appended to the end of each of the new function names, so that a nameclash isn't detected by the compiler.

Each of these new functions are written in a new file, along with any other code that was present in the script apart from the erroneous function. This is so that if one of these new functions relies upon another user-defined function, it can be detected by the compiler and another error (undefined name) isn't detected. This new script file is now given a name other than that of the original script - it is not the intention here to alter the programmer's code - and Miranda called with this new file.

In the definition of `wrong1&1`, it is important that the guard remains as it did in the original function - type inference determines the type of each expression at compile-time when type specifications aren't provided. If the guard wasn't present in the definition of `wrong1&1`, the type of the function would be determined as * → *, which isn't consistent with the actual type of the corresponding equation in wrong1, namely * → num. In the definition of `wrong1&2`, the common is removed, as it not required by the compiler to determine any type of x - True is the return type for all other instances of x in the original function that are not less than three.

At the Miranda prompt, it is possible to invoke the Miranda type checker on a function by typing two colons (::). If the type checker was now called on the two new functions that were written, the response would be :

```
Miranda wrong1&1 ::
num → num
Miranda wrong1&2 ::
* → bool
```

Once the type checker had been invoked, it could be argued here that the guard used in `wrong1&1` should have also been used as a guard in the definition of `wrong1&2`, simply because there seems to be a conflict in both the formal parameters and the return types above. This would not have been necessary though, as the compiler inferred the formal parameter of `wrong1&2` to be polymorphic (as it is unconcerned with its type) which does match with the num type formal parameter of `wrong1&1`. It is therefore the return types that are in conflict. This can be noted and reported to the programmer in the more meaningful error message.

4.3.1 Checking operators

Following along the same lines as the method for detecting whether a conflict has occurred in the formal parameters or return types above, if a function has been written in such a way that it supplies the wrong arguments to an operator, then it is the case that the new function built from this erroneous equation will itself contain a type error.

The reason for this is that there will only be one equation to this new function. No conflicts can occur in arguments or return types when there is only one equation and the function does not rely on any other erroneous functions, so the only reason that an error could occur is if an operator is supplied the wrong arguments.

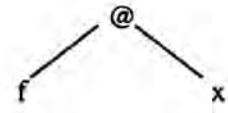
When the Miranda type checker is invoked on a function with a type error, the result is invariably

```
Miranda any_wrong_func ::
WRONG
```

In the same way that it could be returned to the programmer whether arguments or return types are in conflict, it could also be detected when the whole function is incorrect by recognising `WRONG` when the type checker is invoked, and reporting this to the programmer.

4.4 Alternative representation of a function

Each function that forms a Miranda script file can be represented in the form of a *parse tree*. Take any function and an argument, say `f` and `x`. Written as code, `f` applied to `x` is written `f x`, and its parse tree representation is



The symbol '@' (known as a *tag*) is used in the tree for a specific purpose, and denotes application of one thing to another. Any node in a tree with this symbol can be considered an application node. This tag is very important in the sense of application, as its existence allows a representation that supports a curried application style and can subsequently be used to represent partially applied and higher order functions, as well as regular ones.

4.5 Type-checking and inference of types

The notion of Miranda being a strongly typed language was mentioned in section 2.1, but there are more serious implications to this feature which should be considered. When a function is defined, it is always good programming practice for the programmer to give each function that is written a type signature beforehand. However, the programmer is not obliged to provide type information for the compiler to use. The compiler should be able to infer these types from the contexts in which they are used. The polymorphic type discipline, first set forth in detail by R. Milner in 1978 [11] is all about how this can be done.

When a Miranda program is being compiled, the programmer will see a message output to the screen during compilation, which will look something like :

```
compiling <filename.m>
checking types in <filename.m>
```

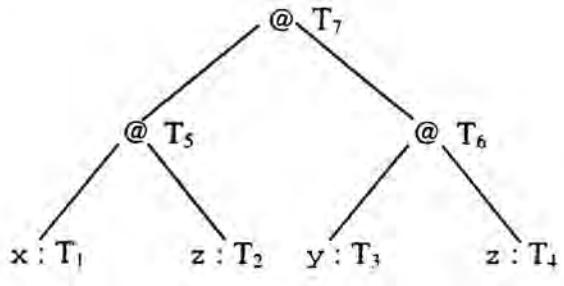
It is the second of the two lines that I am interested in here. When types are checked during the compilation process (compile-time type-checking is also known as *static* type-checking), the main objective is to make sure that the program is well typed, in other words, if a type error is present in the code make sure it is detected.

It is possible to find types that a function deals with, by representing the body of the function as a tree, and attaching types to each of the leaves and nodes of the tree. In this case, each leaf in a tree will represent an argument. As an example, consider the function

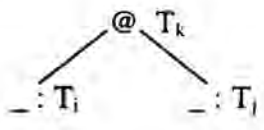
```
foo x y z = x z (y z)
```

Given three formal parameters, the operational semantics of `foo` - how the definition of the function defines the formal parameters to behave - are the application of `x` to `z`, applied to the application of `y` to `z`. The corresponding parse tree for the body of the function `foo`, with arbitrary types $T_1 \ldots T_n$ attached to the formal parameters at each leaf and node of the tree can be seen here.

```
foo x y z =
```



When a function is provided with a type signature in Miranda, it may look something like `some_func :: num → num → bool`. Similarly, type specifications for each application that occurs in the parse tree can be written down. If you ignore the arguments to a function and create a general parse tree which signifies application of one thing to another, only labelling *types* on the nodes and leaves of the tree, you would get :



From the above parse tree, you can reason that if a function of type T_i is applied to an argument of type T_j, then the return type is T_k, which can be written, $T_i = T_j \rightarrow T_k$.

Returning to the parse tree representation of the function `foo`, the following type specifications can be deduced based on the above :

$T_1 = T_2 \rightarrow T_5$
$T_3 = T_4 \rightarrow T_6$

$T_5 = T_6 \rightarrow T_7$

However, it may or may not have been noticed that the formal parameter z in the function `foo` has two different type variables associated with it, namely T_2 and T_4. As all occurrences of a formal parameter in a function body must have the same type[3], it must be the case that T_2 is the same type as T_4, so any occurrence of T_4 in the derived type equations can be replaced with T_2. Also replacing T_5 with $T_6 \rightarrow T_7$, this gives the revised type equations :

$T_1 = T_2 \rightarrow (T_6 \rightarrow T_7)$
$T_3 = T_2 \rightarrow T_6$

So, the overall types of the three formal parameters of the function `foo` are x :: $T_2 \rightarrow (T_6 \rightarrow T_7)$, y :: $T_2 \rightarrow T_6$ and z :: T_2, and finally the return type of the function is T_7.

`foo` :: $\underbrace{T_2 \rightarrow (T_6 \rightarrow T_7)}_{x} \rightarrow \underbrace{(T_2 \rightarrow T_6)}_{y} \rightarrow \underbrace{T_2}_{z} \rightarrow \underbrace{T_7}_{\text{Return type}}$

Looking at the complete function signature for `foo`, it looks similar to something which may be output as the description of a type error when a failed compilation has taken place, for example

```
cannot unify num → [num] -> bool with [num] → [num] → bool
```

When it is reported that the compiler cannot unify *a* with *b*, where a and b are some type expressions, *a* is what the compiler is unable to unify consistently with some previously defined equation with the type expression, *b*. So in the example above, `num → [num] → bool` is not consistent with what the compiler is expecting to see based on an equation that has occurred before - `[num] → [num] → bool`.

4.5.1 Moving from arbitrary to concrete types

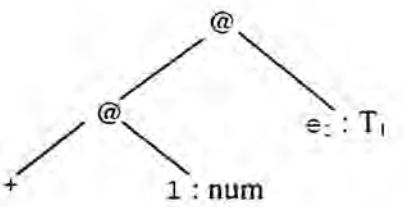
Having inferred an *arbitrary* type equation using parse trees, it is still undetermined which arguments have which types. Inferring a type equation for a particular function is only really of any use if there is some chance that specific or *concrete* types can be attributed to the formal parameters.

Once the parse tree has been created, the actual types of the arguments can be deduced by looking at the operators or connectives which link them together. When the function `foo` was considered earlier, no operators were shown, merely abstract applications to demonstrate how type equations could be inferred in the absence of a type signature. To show how concrete types can be assigned, consider the simple function one_to_list, which adds 1 to each element in a numbered list.

```
one_to_list x = map (+1) x
```

[3] This restriction on the types of formal parameters within the body of a function is often referred to as the *monomorphism limitation* - each formal parameter must be used monomorphically.

The built-in function, `map`, is applied to a function and a list, and returns a list containing the results of the given function having been applied to every element of the original list. In reality then, the function `one_to_list` is the application of `+1` many times to an element in the list $e_1 \ldots e_N$. The parse tree with attached types for the first application is such :



Now, the type signature for the `+` operator used in a prefix manner is `+ :: num → num → num`. `1` is obviously of type num, and for the definition to be correct, the compiler determines that e_1 is also of type `num`. This correctly fulfils the signature of `+`, which must return a type `num`. As e_1 is one element from a list, it follows that all elements must be of the same type, as this is a restriction of list usage in Miranda. Therefore, `x` must have the overall type `[num]` and the return type is also a `[num]`. At last, the complete function signature can be inferred, which was done so without any help from the programmer!

```
one_to_list :: [num] → [num]
```

Despite the success in finding the concrete types for the function above, it is not always the case that this can be done. Sometimes the context in which an expression is written is not enough to deduce specific types. When a specific type cannot be found, as long as the function is not in error, the type is considered to be polymorphic.

Having explained about how types can be inferred, it remains to be seen how this can be linked with the problem that I am dealing with : creating more meaningful and informative error messages. Given any function which contains a type error, it has been shown that this function has a corresponding representation in the form of a parse tree. In the introduction and the very first paragraph of this chapter, I mentioned that it was not my intention to build a new Miranda compiler and type checker from scratch. However, to go forward and produce more meaningful error messages about type errors, it is possible that part of the type checker that already exists internally can be reproduced to solve this problem, which is of interest outside the compilation process.

4.6 Producing meaningful type error messages

A simple way of thinking of how the compiler checks for type errors, is that for *every* equation that exists within a function definition, the compiler creates a parse tree representation. For the function to be *well typed*, which means that no type errors occur, every parse tree for each equation must be identical in terms of the types that are used and the derived type equations for each application in the tree. The compiler reports conflicts between differing parse trees.

Considering error messages in this fashion, it is reasonable to start thinking that a more meaningful type error message could be presented to the programmer by giving some feedback on *why* the error has occurred, with which ***arguments*** the error has occurred and by recommending some possible remedies.

It has often crossed my mind that if the type checker has to go through the process of inferring types of equations anyway, why was it not written in such a way that the checker

retained the information about formal parameters and their types, and included this information with the error message?! Somewhere within the type checker, valuable information about the error - some of which could help in easily locating and possibly resolving it - is lost.

In order to regain this information, or to create it for the first time in my case, I believe that the way to do this is by creating parse trees for every equation that exists within a function definition and comparing them, just as the type checker does in the first place. The only difference would be that more emphasis would be stressed upon aspects of the comparison that are shadowed in the Miranda type checker - things like *formal parameters* that are used incorrectly and the types that are associated with them, as well as applications of functions to arguments which cause incongruous return types for different equations, hence different parse trees.

During the comparison, any conflicts that are detected can be noted, along with the formal parameters that are used incorrectly. When the comparison is complete, output could be prepared for the programmer detailing the exact findings.

5. Prototyping a static Miranda debugger

I have decided to prototype a system that deals with aspects of meaningful error messages that do not so much depend on the code that has been written by the programmer, *but on the error messages that have occurred during compilation*. Examples of such errors would be ones for which some kind of remedy is already prepared and ready to be presented to the programmer when the error is detected, i.e. they can be resolved by inspection. A static Miranda debugger, as named in the above chapter title, simply means a system that help locate and diagnose errors (debugger) that have occurred at compile time (static).

The main reason for choosing to deal with these errors is because of the varying styles that Miranda programmers employ when typing scripts. It is possible when creating a script to use an infix or prefix manner. For example, the following two expressions mean exactly the same thing and are both perfectly valid in Miranda :

```
> y = 2 + 3      || infix
> y = (+) 2 3    || prefix
```

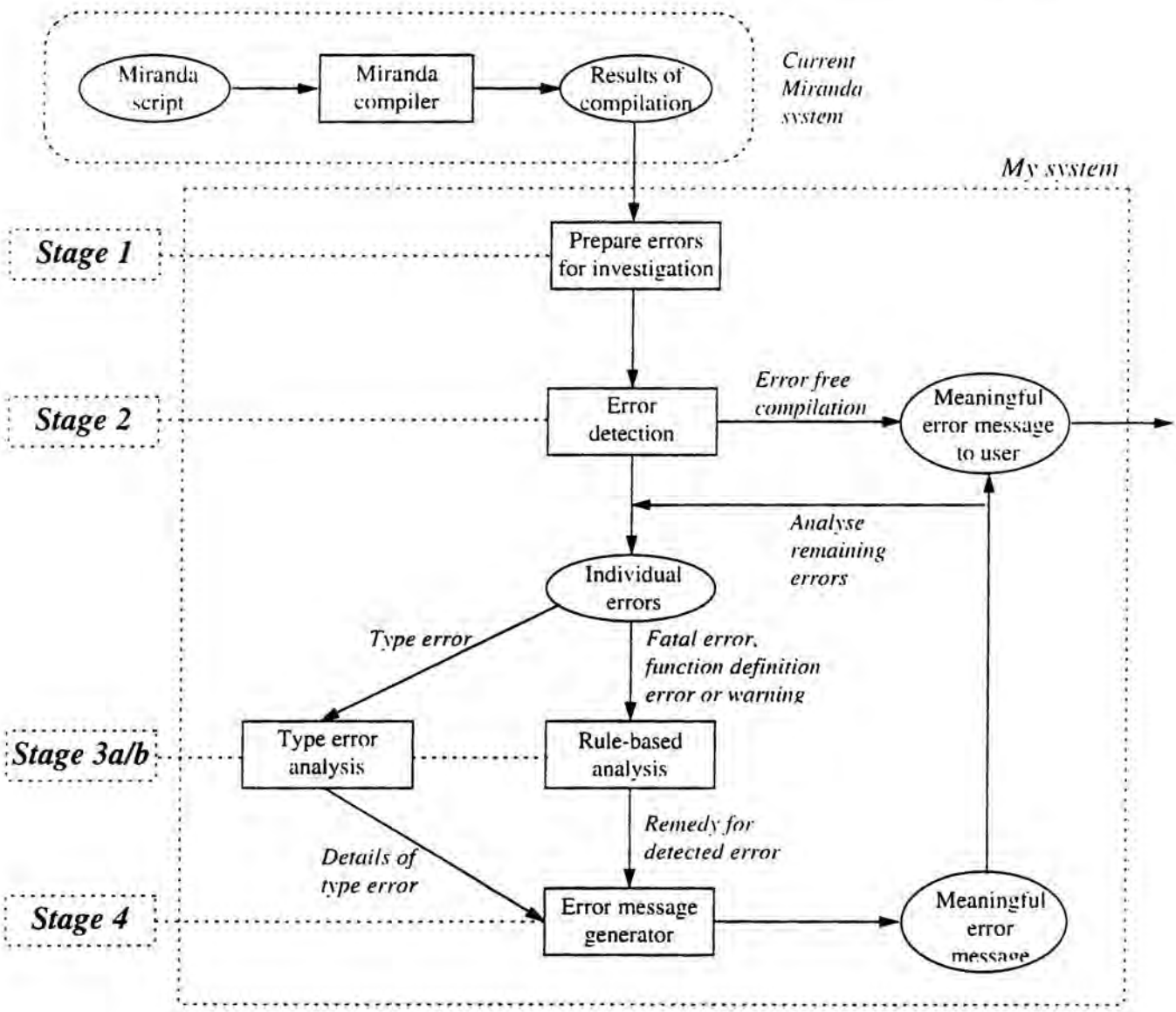
A prefix style of expression looks quite unorthodox, but is often used unknowingly. Function applications are prefix as a function is merely *"an abstraction of values over some common behaviour"* [9], and application of a function to its arguments takes the form `function arg1 arg2.....argN`, i.e. the function is named first, just as `+` is in the second definition of `y` above.

For most applications, infix is the preferred manner over prefix for expressing an application, although there is invariably some degree of mixture between the two. In order to be able to create full parse trees to represent program code, the preferred style in which the code should be written is prefix, and so a conversion of the program from a mixture of infix and prefix styles to *just prefix* has to take place before anything else can be done. (This was discovered during the background reading which ran concurrently with the duration of this project, mainly in [9,13,16] .)

The time that it would have taken to perform this conversion would have delayed the progress of the actual system that I have decided to prototype. The first part of this chapter details my prospective system's design, followed by the specification of the integral components that make the system work.

5.1 Overall system design

The system that is specified further on in this section fits into the overall system design represented below. The system that I am prototyping can be best viewed as an extension to the current Miranda system, in that the operation of the Miranda system is not altered in any way.



Each of the main parts of system have been grouped into various stages in the design diagram. Stage 1 involves capturing the errors that have occurred during compilation and preparing them for investigation. This preparation of the errors involves removing all non-error information from the compilation, e.g. the Miranda header, Miranda prompt etc.

Stage 2 groups each detected error into a list containing all information on that particular error, while stages 3 a and b deal with analysing type errors and fatal/function definition errors and warnings respectively. Finally, stage 4 deals with compiling a meaningful error message depending on the results of the various analyses that have taken place in stage 3. The final part of my system involves the meaningful error message being output to the user.

Prior to the specification of the functions that fit directly into the clear cut stages above, some functionality is required that is utilised by most or all stages. These functions cannot be said to belong to one particular stage and are specified first in this chapter.

5.2 *Specification of Functions*

As referred to in section 2.3, the idea of executable specifications [17] has been presented by David Turner. Miranda is itself a specification language and can be used to outline the functional strategy used throughout the prototyping undertaken in this project, just like pseudocode may be used if C++ was the chosen language for implementation. A distinct advantage here is that the specifications can be checked for correctness by the Miranda compiler, which can then be executed to observe the behaviour of the system which I am specifying.

5.3 *Detecting an expression within text*

When an implementation of the sort required by this project is undertaken, The Miranda system has to be able to access the underlying UNIX operating system so that any file creation or interaction that has to take place can do so transparently. In other words, the programmer using the system shouldn't be aware of any file creation or manipulation that is taking place.

To gain access to a file that exists in a directory, a command known as `system` is used, which when called with a [char] argument (pathname of the file that is required), returns a triple, one element of which contains the contents of the desired file. For example, to assign the contents of a file, `foo`, to an identifier, `bar`, the specification may look something like :

```
bar = stuff
      where
      (stuff, x, y)
      = system "cat foo"
```

The identifier, `bar`, can now be investigated in any way by specifying functions which act on its contents.

A UNIX command `grep` exists, which given some regular expression and a piece of text, will tell you if that regular expression occurs within the text. This `grep` facility does not exist as a built-in function in Miranda, but may be written so as to simulate its UNIX counterpart.

This function would be extremely useful in many ways and would play a major role in preparing a file for investigation by removing the Miranda start-up header and non-error information, and for detecting 'unique' parts of errors that have occurred within an actual error message, e.g. "unexpected end of file" etc. An implementation for `grep` in Miranda is presented in Programming with Miranda [7].

`grep` can be written in Miranda to work element-by-element on a regular expression and a piece of text. That is, given any regular expression, `regexp`, and a piece of text, `line`, `grep` will check to see if the first element of `regexp` equals the first element of `line` and if it does, it will compare the second element of `regexp` with the second element of `line`. It will keep doing this until a comparison fails, i.e. an element of `regexp` doesn't match its corresponding element in `line`, or until the whole of `regexp` has been matched in `line` and so returns True.

If the former of the two options above is the case, then `regexp` will be compared with the tail of `line`, as the regular expression may occur further into the piece of text. When `regexp` has been tested on all tails of tails.....of tails of `line` until `line` is just one element, if no match of `regexp` in `line` has been made, then the overall return value is False.

```
grep :: ([char], [char]) → bool
grep (regexp, line) = sublist (regexp, line)
sublist :: ([char], [char]) → bool
sublist ([], any) = True
sublist (any, []) = False
```

```
sublist (regexp, line)
     = startswith (regexp, line) \/ sublist (regexp, tl line)

startswith :: ([*], [*]) → bool
startswith ([], anylist) = True
startswith (anylist, []) = False
startswith (front1 : rest1, front2 : rest2)
     = (front1 = front2) & startswith (rest1, rest2)
```

5.3.1 Breaking up text into chunks

When desired pieces of information have been extracted from a piece of text ([char]), it is often useful to be able to analyse them, for example, check the first word that occurs in the text is *x* or *y* etc. The best way to prepare the text for this is by breaking it up into smaller pieces of text, each piece being a word or symbol. The delimiting character for breaking text up could be a space, ' '. Below is a specification for a function, `tokenise`, which does exactly that.

```
tokenise :: [char] → [[char]]
tokenise [] = []
tokenise (x : xs)     = [] : tokenise xs, if x = ' '
                      = (x : x1) : xrest, otherwise
                      where
                      (x1 : xrest) = tokenise xs, if xs ~= []
                       = [] : [], otherwise
```

Further functions can be specified which tokenise many [char] (`parse`) and return a [char] back into a [[char]], putting a space (' ') between each [char] (`un_toke` and `un_break`). `un_toke` and `un_break` are the opposite in functionality to `parse` and `tokenise`.

```
parse :: [[char]] → [[char]]
parse [] = []
parse (x : xs) = [tokenise x] ++ parse xs

un_toke :: [[char]] →  [char]
un_toke [] = []
un_toke (x : xs) = [un_break x] ++ un_toke xs

un_break :: [char] → [char]
un_break [] = []
un_break (x : xs) = (x ++ " ") ++ un_break xs
```

Also when analysing a piece of text, it is useful to remove any white space (spaces and tabs) that may occur in the [char], along with the preceding '>' that denotes that a line is actually part of the code.

```
min_s in = min_t (filter (~= ' ') in)
min_t in = (filter (~= '\t') in) -- ">"
```

5.4 Removing non-error information

This section deals explicitly with the specification of functions involved in stage 1 of the design diagram given at the start of this chapter.

Having specified the `grep` function above, it now possible to specify an algorithm for extracting all error information. The part of the system that does this can be thought of as a pre-parser, which prepares the error information so that it can be interrogated to provide a more meaningful error message.

When Miranda is started with a file, the file is compiled and the result of compilation is sent to the users screen. It is possible to capture this output by redirecting it to a file. For example, to redirect the output from starting Miranda with the default file script.m, assuming that script.m contains code (the same as in section 4.1) to be compiled, at the UNIX prompt you would type :

```
UNIX prompt : mira script.m > outfile        <return>
```

In Miranda, `outfile` effectively has the type :: [char] (a list of characters). A built-in function exists in Miranda called `lines`, which turns a [char] into a [[char]], separating a [char] into a further [char] every time a newline is encountered. Applying `lines` to `outfile` makes it easier to take away an unwanted line (one with non-error information) from `outfile`, which now contains the Miranda header and details of the compilation of script.m.

```
err_output :: [[char]]
err_output = lines break
             where
             (break, x, y)
             = system "cat outfile"
```

`err_output` now contains :

```
["T h e   M i r a n d a   S y s t e m", "version 2.014 last revised
24 May 1990", "Copyright Research Software Ltd, 1990","(1000000
cells)", "compiling script.m", "syntax error: nameclash, \"foo\"
already defined", "error found near line 9 of file \"script.m\",
"compilation abandoned", "for help type /help", "Miranda"]
```

`grep` can now be used on each of the 'lines' in `err_output` to detect non-error information. If such information is found, the line in which it is present can be discarded. Eventually, all lines that aren't of use in helping create a meaningful error message will be removed. The result of removing all non-error information can be assigned to an identifier, `detect`.

```
get_errors [] = []
get_errors (x : xs)   || each 'x' is a line in err_output, passed as an argument
      = get_errors xs, if (x = "\r") \/ (#x = 0) \/
        grep("T h e   M i r a n d a   S y s t e m", x) \/
        grep("version 2.014 last revised 24 May 1990", x) \/
        grep("Copyright Research Software Ltd, 1990", x) \/
        grep("cells)", x) \/ grep("for help type /help", x) \/
        grep("Miranda", x) \/ grep("miranda logout", x) \/
        grep("\027", x) \/ grep("compilation abandoned", x) \/
        grep ("checking types in", x) \/ grep ("compiling", x)
      = x : get_errors xs, otherwise

detect = get_errors err_output
```

The resulting [[char]] which is produced when `get_errors err_output` is assigned to `detect` is as follows :

```
["syntax error: nameclash, \"foo\" already defined", "error found
near line 9 of file \"script.m\"]
```

5.5 *Specific error details*

In order to provide a formal description of the strategy required to elicit and provide more information on errors that can be resolved by inspection, certain type synonyms can be listed that make the overall algorithm easier to read.

```
ls_errs == [errors]    || 'errors' instantiated later
in_file == [[char]]
one_err == [[char]]
all_errs == [one_err]
line_num == [char]
problem == [char]
function == [char]
specified == [char]
inferred == [char]
```

5.5.1 *Data Structure for an error*

Regardless of whether a Fatal, Serious or Warning error has occurred during the compilation of a Miranda script, the one thing that they all have in common is that they are errors! For this reason, I have chosen to represent an error with an abstract data type (ADT) `err` (this is known as the type declaration). It is worth pointing out here that 'error' is a reserved word in Miranda, so it is impossible to redefine it within a script without getting a 'nameclash' error, hence `err`!

```
abstype err
with
      spec_errs :: in_file → all_errs
      get_stwlinum :: one_err → line_num
      get_flinum :: one_err → line_num
      search_linum :: one_err → line_num
      search_spec :: one_err → specified
      search_inf :: one_err → inferred
      search_prob :: one_err → problem
      get_error :: all_errs → ls_errs
      search_error :: one_err → ls_errs
```

Now that a new type `err` has been created, it must be instantiated in order to give it some meaning. For example, a 'num' type is useless unless it is defined to represent the numbers that we use in everyday life.

Given the four variations of errors that I am considering that can occur (syntax, type, function definition and warning), an algebraic type, `errors`, can be created which defines the underlying structure of an individual error. To make the interface to the abstype declaration complete, abstype `err` is linked to algebraic type `errors`. Also in the definition of `errors`, is

`Unknown`. This accounts for errors encountered that are not recognised by this prototype. The definition of `errors` can be extended if further work is undertaken by somebody else.

```
errors ::= Syntax (line_num, problem) |
           Fn_def (line_num, specified, inferred) |
           Type (line_num, function, problem) |
           Warning (line_num, problem) |
           Unknown

err == errors
```

For the rest of this section, I am dealing with functions which are specific to stage 2 in the design diagram. This stage deals with obtaining all information attributed to the occurrence of each particular error and grouping this information into the relevant algebraic type pattern specified above.

Based on the table presented in section 4.1, Isolating an error, it was shown that there were so many relevant lines of an error message that were specific to that particular error. Using the built-in Miranda function, `take`, it is possible to 'take' so many relevant lines of an error message, which will depend on the nature of the error itself. As an example, if you see a 'syntax error' in `detect` (remember that `detect` contains *all* error information), then you want to take that line and the next line as well - in all you want to take 2 lines of `detect`, which will give you all the information required on that one particular error.

```
ses = spec_errs detect
spec_errs [] = []
spec_errs (x : xs)
     = [[x]] ++ spec_errs xs, if grep("SPECIFIED ", x)
     = [take 2 (x : xs)] ++ spec_errs (drop 2 (x : xs))
       , if grep("syntax error", x) \/
            grep("undeclared typename", x) \/
            grep("type error in", x) \/
            grep("warning, script contains", x)
     = [take 3 (x : xs)] ++ spec_errs (drop 3 (x : xs))
       , if grep("badly formed type", x) \/
            grep("incorrect declaration ", x)
     = spec_errs xs, otherwise
```

Having defined the type `errors`, it remains to be specified how the relevant information for each error is obtained, e.g. line number, the problem itself etc. The following specifications are for the functions whose signatures were presented in the interface to the abstype `err`.

Depending on whether a syntax, type, function definition or warning error has been detected, the location of the line number on which the error has occurred will be in different places. For syntax / type errors and warnings, if a line number is provided, it will be in the last line of relevant information for that error. If a function definition error has occurred, the line number is in the first line of relevant information.

Bearing this in mind, two functions can be specified that prepare the relevant line of information to be searched for this line number by tokenising it, i.e. breaking the line into a further [char] every time a space is encountered. The main function, `search_linum` looks at each of these chunks of text and takes digits only, which will represent the line number.

```
get_stwlinum x = search_linum (tokenise (last x))
get_flinum x = search_linum (tokenise (hd x))

search_linum [] = []
```

```
search_linum (x : xs)
     = (takewhile digit x) ++ search_linum xs
```

If a function definition error has occurred during compilation, it is desirable to locate what was specified and what was inferred by the compiler with relation to the erroneous function. `search_spec` and `search_inf` do exactly this. By looking at the [[char]] which contains all the information on one error, each function can pick out the specified and inferred types of a function by using grep, testing against "specified," and "inferred ," for each line in the error information.

When the appropriate line is identified, either "specified," or "inferred ," is removed from this line to leave only the type information, which is returned by each function.

```
search_spec [] = []
search_spec (x : xs)
     = x -- "specified, ", if grep("specified,", x)
     = search_spec xs, otherwise

search_inf [] = []
search_inf (x : xs)
     = x -- "inferred,  ", if grep("inferred,", x)
     = search_inf xs, otherwise
```

Given specific information relating to one particular error, extracting the actual problem that has occurred, e.g. nameclash etc. is important so that this can be detected later on and some advice may be given on how to solve the error. As with the location of line numbers within an error message, the location of the actual problem that the error message is relaying is also dependant on the nature of the error.

```
search_prob [] = []
search_prob (x : xs)
     = x ++ " " ++ hd xs, if grep("type error in", x)
     = x, if grep("SPECIFIED ", x)
     = last (tokenise (hd xs))
       , if grep("warning, script contains", x)
     = x -- "syntax error", if grep ("syntax error", x)
     = ": " ++ x, if grep ("undeclared typename", x)
     = ": " ++ x ++ "\n\t  " ++ hd xs
       , if grep("badly formed type", x)
     = "Unknown", otherwise
```

When a type error occurs, certain 'extra' information is provided by the compiler : the name of the function in which the type error has been detected. It would not be sensible for my system not to include this information - I am trying to create more meaningful, easier to understand messages, not harder ones! The first line of a type error message is "type error in definition of ..." The sixth word of this line, without exception, is the name of the function in which the error has been detected. `find_tfn` obtains this name from the problem itself, making use of the built-in function `last`.

```
find_tfn x = last (take 6 (tokenise x))
```

When the actual nature of a type error that has occurred is obtained with the function `search_prob`, also included in this is the line number of the error and the reported fact that it is a type error, i.e. "type error in definition of...". As the line number has already been extracted using `get_stwlinum` and I know that I am working with a type error, the actual problem itself is the type(s) that are in conflict. The function `edit` goes through the whole error and returns

just these type conflicts, which are always after the reported line number, i.e. `edit` returns all text in a message after (line ... of ??.m"). Using `grep` to determine where the actual type conflicts are listed, the specification for `edit` is as follows :

```
edit x = un_break (alter (tokenise x))
alter (x : xs)
      = alter xs, if ~grep(".m:")", x)
      = xs, otherwise
```

Finally, now that the abstype `err` functions have been specified, it is possible to go through the list of all errors, fitting each error into the data structure that was designed to hold error details. `ses`, from above, contains all errors from one compilation of a script file. `get_errors`, with `ses` as its argument goes through this list of errors and groups each one according to its nature, obtaining details on the error consistent with the algebraic type specification of each error that I am dealing with.

```
get_error [] = [Unknown]
get_error (x : xs)
      = search_error x ++ get_error xs

search_error x
      = [Syntax (get_stwlinum x, search_prob x)]
        , if grep("syntax error", hd x) \/
             grep("undeclared typename", hd x) \/
             grep("badly formed type", hd x)
      = [Fn_def (get_flinum x, search_spec x, search_inf x)]
        , if grep("incorrect declaration", hd x)
      = [Type (get_stwlinum x, find_tfn (search_prob x),
        edit (search_prob x))]
        , if grep("type error in", hd x)
      = [Warning (get_stwlinum x, search_prob x)]
        , if grep("warning, script contains", hd x)
      = [Warning ("<not specified>", search_prob x)]
        , if grep("SPECIFIED", hd x)
      = [Unknown], otherwise
```

5.6 *Determining conflicts in types*

In the case of fatal errors, function definition errors and warnings, enough information can be collected on the content of these errors through the use of the specified function `get_error`, above. From this information, it is reasonable to expect that a rule-based system can be specified to provide suitable analysis of the error and provide hints on how the error could be resolved. This rule-based part of my system, corresponding to stage 3b of the system design diagram, is discussed later on. At the moment, this section will consider the specification of functions that form stage 3a of my system dealing with type error analysis - the largest and most interesting part of my system

In chapter 4 it was discussed at some length how it would be possible to determine more specifically the factors that have caused a type error to occur, namely whether it was the formal parameters or return type of a function which was in conflict for different equations in a function definition.

Below is an explanation of the functions and their specifications which are required to create a new script file with the erroneous function replaced with several new functions for each equation in the function definition. When information has been collected on a type error that has

occurred using `get_error`, included in this is the name of the function that is in error. This function name is the only thing that is required to extract it from the original script.

Within the context of this specification, it can be assumed that this function name is assigned to an identifier called `func`. Subsequently, a list of all names that can be given to newly created functions in the new script file can be written. These are not necessarily unique names, as they are built from the original function name with a number appended to the end of it. This is a restriction on the use of the system - no functions names with numbers as the last character!

```
ls_of_new_names
 = [func, func++"2", func++"3", func++"4", func++"5", func++"6",
    func++"6", func++"7", func++"8", func++"9", func++"10",
    func++"11", func++"12", func++"13", func++"14", func++"15",
    func++"16", func++"17", func++"18", func++"19", func++"20"]
```

`get_def` gets the complete function definition of the desired function. This is achieved by locating the start of the function definition (using `grep` sub-function `startswith` and the function `min_s`) and taking every line of code until the function signature of a new function is encountered, or there is no text left in the input file.

Once the function definition has been extracted, it is broken into chunks - each word is a 'chunk', and assigned to the identifier `do` for use by other functions.

```
do = parse (get_def prog_code)

get_def [] = []
get_def (x : xs)
      = x : get_def xs, if startswith(func, min_s x) &
          ~startswith("::",  (min_s  (tl  x)))  ||  detect  new
signature
      = get_def xs, otherwise
```

Having got the definition of the erroneous function, it is possible to start to create the new functions from the equations of the original one. First of all, I will create a list of all the patterns of function arguments. `gen` takes `do` as its argument, and works through each of the chunks that make up the function definition, taking all those chunks that precede an '=' sign, whenever the '=' is not a comparison. Collectively, these chunks form one of the patterns of arguments. This is repeated for all of the contents of `do`. When this is complete, the chunks are returned to a readable form (unchunked!) using the previously specified function `un_toke`.

```
fst_phase = un_toke (gen do)

gen [] = []
gen (x : xs)
      = [newcode x []] ++ gen xs

newcode [] final = final
newcode (x : xs) final
      = newcode xs (final ++ [x]), if ~grep("=", x)
      = final ++ ["="], otherwise
```

The function `join` removes unwanted patterns in function arguments. For example, fst_phase may contain some patterns which are of no use in obtaining every function argument. When `fst_phase` is created, it will pick up on definitions which follow a layout as I have used for my specifications of functions, that is

<function name> < formal parameters>
`=` *<function definition>*

It could be the case that some of the `[[char]]` which should be argument patterns, in fact only contain `[">","","","="]`. These are removed here.

```
join [] = []
join (x : xrest)
     = x : join (tl xrest)
       , if ((last (min_s x) ~= '=') &
       (startswith (func, min_s x)) &
        startswith("=", (min_s (hd xrest))))
     = x : join (xrest), otherwise
```

The result of applying `join` to `fst_phase` will acquire all function patterns that exist within a function definition, but it may be the case that a pattern of arguments is used for more than one equation. `fn_patts` prepares a list of all function patterns that are required *for each* definition that occurs.

```
fn_patts [] final all = all
fn_patts (x : xs) final all
     = fn_patts xs x (all ++ [x])
       , if ((min_s x) ~= "=") & (grep(func, x)) & (x ~= final)
     = fn_patts xs final (all ++ [final]), if ((min_s x) = "=")
     = fn_patts xs final all, otherwise
```

Because of the way new definitions will be specified later on for each of the function patterns, the '=' symbols that succeed each of the patterns are removed.

```
mod_fn [] = []
mod_fn (x : xs) = (x -- "=") : mod_fn xs
```

Finally, the required function arguments for the new functions can be obtained combining the above functions, applied at the appropriate times. The arguments are assigned to the identifier `fn_args`.

```
fn_args = mod_fn (fn_patts (join fst_phase) [] [[]])
```

Having extracted the required function arguments which will go on and form the patterns of the arguments for each new function, a definition for each of the patterns must be obtained.

```
body [] = []
body (x : xs)
     = ([restcode x x []] -- [[]]) ++ body xs

restcode [] curr final = final -- [[]]
restcode (x : xs) curr final
     = restcode [] curr (final ++ (x : xs)), if grep("=", x) \/
       ((~startswith(func, min_s(hd(un_toke [curr])))) &
       (~startswith("=", min_s(hd(un_toke [curr])))))
     = restcode xs curr final, otherwise
```

```
link [] = []
link [xs] = [xs]
link (x : x1 : xs)
      = (x ++ (((hd x1) -- ">") : (tl x1))) : link xs
        , if grep(">", (hd x1))
      = x : link (x1 : xs), otherwise

fn_body = (un_toke (link (body do)))
```

`all`, which takes the list of each pattern of function arguments and each corresponding equation for each of these argument patterns, produces the new functions by taking the first argument pattern and joining it with the first definition and so on until each list is empty. Remember, the lists will always have the same number of elements - this is ensured when the lists are originally created.

```
all = newcode fn_args fn_body

newcode [] [] = []
newcode (x : xs) (y : ys)
      = (x ++ y) : newfile xs ys
```

`se_and_rep` (search and replace) goes through each and every new definition of a function and replaces the names with unique ones from the list `ls_of_new_names`. This is achieved by parsing (using `tokenise` on each and every new definition) `all` (created above) and replacing each occurrence of the old function name with an appropriate new one. Remember, the function names for each new function have to be unique so that a 'nameclash' error isn't detected. While this is being done, needless information can be removed from the definition lists, like "", which are a side-effect of tokenising lists of characters.

```
se_and_rep = (un_toke (ch_nms (parse all) ls_of_new_names))

ch_nms [] any = []
ch_nms (x : xs) (n : ns)
      = [eachline (filter (~= "") x) n] ++ ch_nms xs ns

eachline [] n = []
eachline (x : xs) n
      = x : eachline xs n, if ~(grep(func, x))
      = n : eachline xs n, otherwise
```

The final integral functions which are required now, are those which extract all code apart from the erroneous function from the original script file. The identifier, `not_funct`, is the opposite of `funct`, as it extracts everything from the input code file except the erroneous function. This is done by taking every line of the code that forms the script file, up until the first line of the original erroneous function definition is encountered. At this point, control is handed to another function, `unrest_fn`, which discards every line of the erroneous function definition. When the start of a new type signature is detected, `get_undef`, takes the rest of the code in the script file. The overall outcome is that everything within the script file apart from the erroneous function is extracted.

```
not_funct = (get_undef prog_code)

get_undef [] = []
get_undef (x : xs)
      = x : get_undef xs, if ~startswith(func, min_s x) &
       ~startswith("::", (min_s (tl x))) || detect new signature
      = unrest_fn xs, otherwise
```

```
unrest_fn [] = []
unrest_fn (x : xs)
      = unrest_fn xs, if ~grep("otherwise", x)
      = get_undef xs, otherwise
```

Finally, using everything that has been specified above, it is possible to create a new script file (called new.m) which contains all error free code plus the newly created functions, built from the equations of the original function that was in error. This is achieved using the command `Tofile` which takes a filename to be created and the contents of the file, which is the newly created functions and the remaining functions in the file.

```
new_script = [Tofile "new.m" (lay (not_funct ++ se_and_rep))]
```

Assuming that this newly created file, `new.m` can be compiled and each of the new functions that have been written can be type checked using `::`, with output of the type check redirected to a file, `specfile`, it is possible for this file to now be interrogated to find out more about where the type error has occurred in the original function.

```
sigs = transpose (refine (parspec (lines specs)))
       where
       (specs, x, y)
       = system "cat specfile"
```

The identifier, `sigs`, holds the contents of `specfile`, each line of which is the result of invoking the type checker on a new function. The built-in function `transpose` has been used on `sigs`. What this does is group the first elements of each line in `specfile` into one list, the second elements into another list and so on. Therefore, each list contains corresponding elements representing the types assigned to each argument in the original function definitions.

By specifying two new functions which act similarly to `parse` and `tokenise`, `parspec` and `tokespec`, `spec` is broken into chunks each time a '-' symbol is seen. The '-' symbol forms the first part of the → sign, used when type expressions are reported.

```
tokespec :: [char] → [[char]]
tokespec [] = []
tokespec (x : xs) = [] : tokespec xs, if x = '-'
      = (x : x1) : xrest, otherwise
      where
       (x1 : xrest) = tokespec xs, if xs ~= []
      = [] : [], otherwise

parspec [] = []
parspec (x : xs) = [tokespec x] ++ parspec xs
```

Having done this, a complete type expression which may have looked like `["num → num → bool → [char]"]`, now looks like `["num",">num",">bool",">[char]"]`. A simple function, `refine`, can now be specified which removes the ">" symbols.

```
refine [] = []
refine (x : xs) = [del_unreq x] ++ refine xs

del_unreq [] = []
del_unreq (x : xs) = [x -- ">"] ++ del_unreq xs
```

If a group of types within one of the lists of sigs are all the same, then they must unify correctly and a conflict has not occurred. If there are any polymorphic types in the list, they can

be removed as they unify with anything. From each of the lists in `sigs`, smaller lists can be made which have had all duplicates in the list removed (built-in function `mkset`) and all polymorphic types removed. If a conflict has not occurred, the list should only contain one element (`num` or `char` or `bool` etc.) otherwise a conflict is present. The function `differing_in` reduces the lists in `sigs`, while the function `compare` creates a list of two tuples (`bool, [char]`) for each list analysed. If there is a conflict, the output is (`False, [char]` representing conflicts) and (`True, [char]` with no conflicts) otherwise.

```
compare [] = []
compare (x : xs)
      = [(True, x)] ++ compare xs
        , if # (differing_in (mkset x)) <= 1
      = [(False, x)] ++ compare xs, otherwise

differing_in [] = []
differing_in (x : xs) = differing_in xs, if member x '*'
                      = x : differing_in xs, otherwise
```

`analyse` is a function which takes this list of two tuples and instantiates a counter which keeps track of what types (corresponding to arguments) are being checked. If the first element of the tuple is `True` then nothing needs to be done. If it is `False` though, a statement is prepared which reports that a conflict has occurred, what the conflict is and whether it is in the formal parameters or return type. The latter of these three is achieved by pattern matching for when there is only one tuple left in the argument list - this must be the comparison of the return type of a function, as a function can only return a single type. When reporting what conflict has occurred, this is done using a function called `prep_out`, which takes each of the types that can't be unified and joins them together with the word 'with' in between each one.

```
analyse [] count = []
analyse ((False, x) : []) count
      = ["Conflict in Return type - " ++ prep_out x] ++
        analyse [] (count + 1)
analyse ((False, x) : rest) count
      = ["Conflict in Argument " ++ shownum count ++ " - " ++
        prep_out x] ++ analyse rest (count + 1)
analyse ((True, x) : rest) count
      = analyse rest (count + 1), otherwise

prep_out [x] = x
prep_out (x : xs) = x ++ " with " ++ (prep_out xs)
```

The result of applying the two previously specified functions are assigned to the identifier, check, which also instantiates the counter for analyse.

```
check = analyse (compare sigs) 1
```

5.7 *Outputting a new error message*

Having specified most of the functionality required to adequately prototype a system that fulfils the objective of generating more meaningful error messages than are already produced by the compiler, I can detail the specification of the final set of functions which deal with the output of the new error message to the screen.

Depending whether errors have been detected during compilation or not, determines what is output to the programmer. If compilation is 'clean', i.e. no errors have been detected, then this

is reported, otherwise the list of errors that was produced in the file `get_dets.m(prod_errs)` is passed to the function `myshow`, which passes each error in turn to the `display` function.

```
message = "\nErrors have been found during compilation.....\n"

go    = "\nNo errors were detected at compile time.\n"
        , if prod_errs = [Unknown]
      = message ++ (myshow prod_errs) , otherwise

myshow :: [errors] → [char]
myshow [] = []
myshow (x : xs) = display x ++ myshow xs
```

Depending on which error has occurred during compilation, `display` has several alternative argument patterns to cope with. `display` is pattern matched to deal with all errors that my system is capable of detecting. A function exists, `diagnose`, which takes a problem as its argument and returns a (more meaningful) analysis of the error that has occurred. This function is defined in appendix C and forms the rule-based system of stage 3a of the system design. This rule-based part of my system is called from within `display`, the result of which forms the diagnosis output from display.

```
display (Syntax (a, b))
   = "\nA SYNTAX ERROR has occurred on line " ++ a ++
     ".\nProblem " ++ b ++
     "\nDiagnosis :" ++ diagnose b

display (Fn_def (a, b, c))
   = "\nA FUNCTION DEFINITION ERROR has occurred on line " ++ a ++
     ".\nFunction is specified as  : " ++ b ++
     "\nFunction is inferred to be : " ++ c ++
     "\nDiagnosis :" ++ diagnose b

display (Type (a, b, c))
   = "\nA TYPE ERROR has occurred on line " ++ a -++ " in function \""
     ++ b ++ "\".\nProblem : " ++ c ++
     "\nDiagnosis :" ++ diagnose c

display (Warning (a, b))
   = "\nA WARNING is reported referring to line " ++ a ++
       ".\nProblem : script contains unused local definitions - " ++ b
++
       ".\nDiagnosis :" ++ diagnose "script contains"
        , if ~grep("SPECIFIED BUT", b)
   = "\n***WARNING***\n" ++ "\nProblem : " ++ b ++
       ".\nDiagnosis :" ++ diagnose b
        , otherwise

display (Unknown) = ""
```

Through the specifications that have been shown in the previous sections, the integral functionality for a prototype system that produces more meaningful error messages has been presented. The four main stages in the design diagram at the start of this chapter are now complete and it is possible to progress to the testing of my system.

6. Test plan

As it stands, the prototype that I have implemented cannot be used from start to finish in terms of compiling a file on its own, analysing the errors and outputting a more meaningful error message. This is mainly due to parts of the Miranda/UNIX interface that I was unable to incorporate into my implementation because of time constraints. Having said this, the core functions exist within my implementation that allow an error to be analysed and more meaningful error messages to be output.

For this reason, the main operations that my prototype system does perform will be tested independently. Where something should have happened that has not been implemented, this can be simulated by doing it manually. The stages that are involved in producing more meaningful error messages and the main alterations that the original error message undergoes are detailed in the following table, along with what my system can and can't do depending on the state the system is in.

State within system	What my system should do	What actually happens
1. A user compiles a Miranda script and the compilation is either clean or errors are detected	Detect that a compilation has occurred and redirect output to a file ready for interrogation by my system.	Compilation has to be manually redirected to an output file ready for interrogation
2. A file containing the Miranda header and any errors that have occur exists.	Remove all headers and non error information, leaving only errors that have occurred.	As left.
3.1 Error information exists in a file. No type errors have occurred.	Detect errors that have occurred and collect relevant information about the error.	As left.
3.2 Error information exists in a file, including type errors.	As above, and create a new script file with the function in which a type error has occurred rewritten as separate functions.	As left.
3.2.1 A new file exists which can be investigated to find out more about the type error that has occurred.	Automatically compile this new script file and invoke to Miranda type checker on each of the new functions, redirecting output to a file.	Script file has to be compiled manually, and the type checker invoked at the Miranda prompt. Output from the type checker is also redirected manually.
3.2.2 A file exists with the specifications of the new functions that have been written.	Analyse each of the specifications, detecting where conflicts in types have occurred, preparing output for programmer.	As left.
4. All error information has been collected.	Diagnose errors where necessary and output to the programmer.	As left.

I propose to test my system according to the table on the previous page. Where the columns on the right of the table mention *As left*, i.e. the system does what is supposed to, a test can be performed on these parts. Where the system differs between what it does and should actually do, files can be prepared where required to simulate a complete implementation.

6.1 Test data

The first stage of the testing of my project involves creating some files (simulation of existence of part of the system that automatically completes stage 1 in the table shown), the contents of which are the results of compiling some Miranda scripts. These files will contain various errors of differing categories. Because of the large number of errors that can occur during compilation, especially fatal ones - the compiler only being able to detect one of these at a time - I have decided to generate all errors but then group them into two different files together - one containing all fatal errors, function definition errors and warnings, and one containing a type error. There are two different reasons for this decision.

1. The testing process will be dramatically sped up if you don't have to show the details of *every* test on the system for *every* error that can occur. Remember, many errors can occur.

2. When investigating a type error, the code from which the error came has to be manipulated. If this code contains a fatal error, it will have to be resolved before anything further can be done - remember, fatal errors render *all* functions useless. When the code is duplicated, the fatal error will still be present and consequently, invoking the type checker on the new functions will render results of no use.

During testing, I do not plan to show the code that has caused the errors as it will be similar to that shown in chapter 3, when a detailed description of Miranda errors was undertaken. The following sections detail the tests that my system will undergo, yet the actual test data will not be shown. This data will be shown in the test log in Appendix D.

6.1.1 Test 1

Test : Removing all headers and non error information, leaving only errors that have occurred.

Expected output : See a list of all errors that were included in the test file. All non-error information should be removed.

Test files :

- One file containing fatal errors, function definition errors and warnings to be tested.
- One file containing a type error(s).
- One file with no errors.

6.1.2 Test 2

Test : Detection of errors that have occurred and collect relevant information about the error.

Expected output : See output of type, `errors`, detailing the nature of the error any relevant information about the error that has been detected.

Test file :

- Results from test 1 for fatal errors, function definition errors and warnings. Separate test on file containing type errors.

6.1.3 Test 3

Testing : As above, and create a new script file with the function in which a type error has occurred rewritten as separate functions.

Expected output : As for test 2, and also see a file, `new.m`, which contains the rewritten functions for the original function which contained a type error.

Test file :

- Results from test 1 for a type error.

Again, simulating part of the system that doesn't exist is required here. This file new.m which is created above, can be compiled manually and the type checker invoked on the new functions. Results of the type check can be redirected to a file as would be done by the system. This file is called `specfile`.

6.1.4 Test 4

Test : Analyse each of the specifications, detecting where conflicts in types have occurred, preparing output for programmer.

Expected output : Diagnosis of whether conflicts are in argument or return types, what the conflicts are and the arguments they represent.

Test file :

- Results from test 3, above.

6.1.5 Test 5

Test : Diagnose errors where necessary and output to the programmer.

Expected output : Meaningful error message for each fatal and function definition errors and warnings that were produced during compilation.

Test files :

- Results from tests 2, above.

6.2 Testing a complete system

If a full implementation for my project had taken place, it is reasonable to expect that someone could create a Miranda script file and have my system deal with the compilation completely automatically, making use of what I had implemented as well as the Miranda compiler.

The testing of such a system would be more formal than the way my prototype is to be tested as there are considerably more issues to be considered - field trials with *real* users, analysis of how users react to the system and testing of the whole system as opposed to modules of it. Such testing would be likely to yield far more useable results in terms of deciding what improvements needed to be made to it.

In terms of the number of errors that my prototype system was designed to deal with, every effort was made to make these as exhaustive as possible. David Turner, the author of Miranda, was contacted in the hope that he would be able to provide a full error list. Unfortunately, he was unable to do so. This limits the ability to test the absolute robustness of my system in that every error that the system is designed to recognise are the only ones that the system can be given, simply because I was the programmer and I am the tester! This problem can be detoured by having somebody else test the system, but I decided against this as the prototype is not complete.

7. Evaluation and Conclusion

The last six months spent working on this project have been very rewarding for me, both in terms of the report that I have compiled here and the amount of learning and investigation that had to be done before I could even start any implementation of a prototype system. Ideally, everybody would like to be able to implement a fully operational system to accompany their project report, but in my case, I feel that the initial amount of background reading that had to be done fully justifies the level of implementation that has been undertaken. All things considered though, major functionality has been provided which goes a long way to providing more meaningful error messages to the functional programmer.

7.1 Further work

Although the system that I have prototyped is limited in its ability to analyse type errors completely, it is important to remember that ideas have been presented that could improve the systems capabilities when it comes to error analysis and resolution. Overall, I have identified three main areas in which worthwhile further work could be embarked upon to improve the quality, robustness and usability of my system.

7.1.1 Extensions to Functionality

The ideas that have been presented for a system in previous chapters could be extended to capture more errors that can occur from different aspects of the Miranda programming language, for example, higher order functions and various user-defined types, e.g. abstract types or algebraic types. Also, the further research and development of ideas that lead to the specification of functions which analyse type errors more thoroughly would increase the power of the system.

Undertaking such extensions would allow further implementation, until eventually a system existed that covered as many errors that could occur as possible, providing concise and accurate information on the cause and possible resolution for each one.

7.1.2 Evaluation on Students

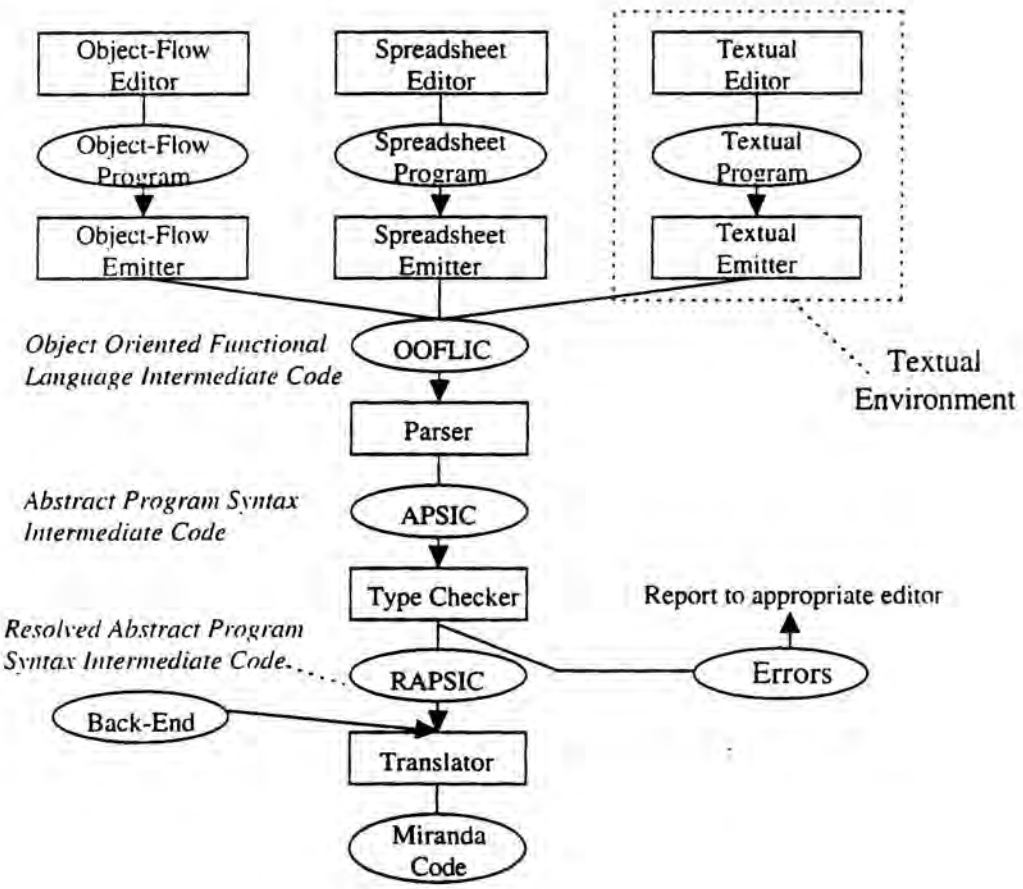
One of my main inspirations in taking on this project was to try and develop something that would be of benefit to Computer Science undergraduates learning a functional programming language, namely Miranda.

Many methods exist that allow you to gauge the success of a product, and in this case the most appropriate would be a field trial of my system on the people for which the system was originally intended. Such a trial would only really be feasible when the extensions to functionality mentioned above (7.1.1) had been fulfilled, thus allowing the system to cope with most errors that could occur. Based on how the students found the system, this could determine the next steps in the system's development.

7.1.3 Integration with CLOVER

Currently under development at UCL is CLOVER [1, 2, 3, 4] - **C**ontrolled **L**azy **O**bject-flow **V**isual **EnviR**onment - which has been dubbed the complete *software development environment* and already has attracted interest from major players in the commercial and financial markets, most notable of whom is Andersen Consulting.

CLOVER is being designed for the development and management control of large scale applications, and does so by integrating Object Oriented Design and Programming, Functional Programming and Visual Programming. The CLOVER development environment has three different editors : textual, spreadsheet and object-flow. However, all of these editors emit code written in a universal language which goes through several stages before being converted to the functional language Miranda. The general overview of the CLOVER system is such :

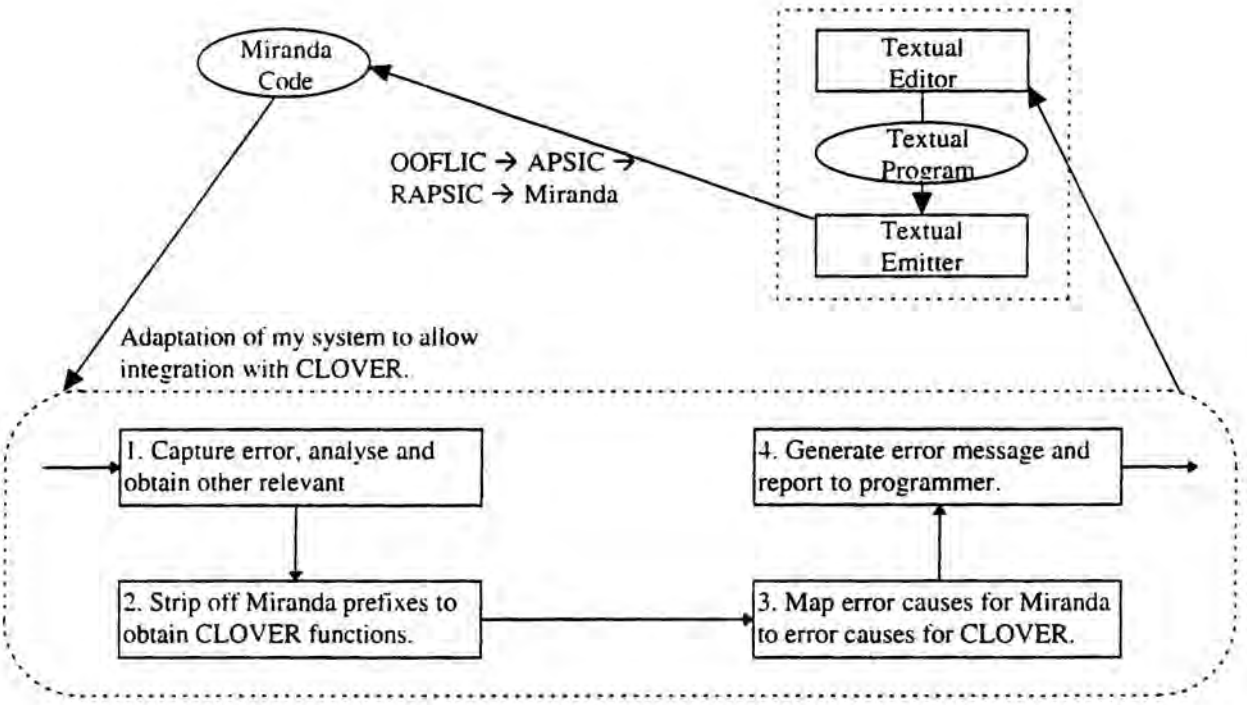


The current development of CLOVER is being undertaken by a Ph.D. student and due to time constraints, the area of providing error feedback to the textual editor is not within the scope of his work. However, this has been identified as a useful add-on and a *fully* implemented system of the type that I have described throughout this report could be adapted to fit into the CLOVER software development environment.

Errors can actually occur at several points within the CLOVER system. My project would fit into the system as feedback between some generated Miranda code and the boxed Textual Environment at the top of the diagram. The operation that a system such as mine would perform in the CLOVER system would be the tracing of an error from Miranda code back to the CLOVER function from which it originated, and suitable error analysis could be provided to the programmer so that an error could be understood and resolved.

An extra module would have to be added to the system that I have specified which copes with the actual tracing of the Miranda function to the CLOVER one. CLOVER functions, when transformed into Miranda functions are prefixed with `m_`, e.g. the CLOVER function, `foo` becomes the Miranda function `m_foo`. Other prefixes deal with other aspects of the target language, e.g. identifiers and functions that are written within Miranda 'where' blocks are prefixed `l_`.

This extra part to be added to my system would deal with stripping off the appropriate prefixes to obtain the actual CLOVER specification. There would also have to a mapping between the cause of the error in Miranda and the corresponding cause in CLOVER. With my system incorporated into CLOVER, a link could be added to the general overview detailing the new error feedback feature that is supported.



7.2 *Summary*

This report has dealt with the problem that programmers dealing with Miranda have to face when they are confronted with an error message after compilation of a Miranda script. What I have presented in the various chapters preceding this one are methods that could help programmers understand these errors quickly and possibly allow them to go on and resolve them with minimal effort. Chapter 5, which saw the specification of a prototype system that implemented some of these "understanding and resolution" methods, included the recognition of as many errors that I am aware of though my time dealing with Miranda.

My main motivation for tackling this project was to do with my enjoyment of functional programming, but also because of the sympathy felt for the new functional programmer, trying to resolve errors that are very hard to understand without experience. Ideally, I would like the work that I have done to be continued by somebody else and eventually have a system that is being use by Miranda programmers to help then with their programming and general understanding and resolution of errors.

Bibliography

References

[1] Braine. L. and Clack, C 1997. Object-Flow. *In Proceedings IEEE Symposium on Visual Languages pp. 418-419, IEEE Computer Society*

[2] Clack. C. and Braine. L. 1997. Object-Oriented Functional Spreadsheets. *In Proceedings 10th Glasgow Workshop on Functional Programming (GlaFP'97).*

[3] Clack, C. and Braine. L. 1996. Introducing CLOVER: an Object-Oriented Functional Language *In Proceedings of the Eighth International Workshop on Implementation of Functional Languages (IFL'96), pp. 21-38. Bonn. September 1996*

[4] Braine. L. and Clack, C. 1997. An Object-Oriented Functional Approach to Information Systems Engineering. *In Proceedings CAiSE'97- Fourth Doctoral Consortium on Advanced Information Systems Engineering, Barcelona, June 1997.*

[5] Clack, C. Clayman, S. and Parrott, D. Lexical Profiling: Theory and Practice. *Journal of Functional Programming 5(2), pp 225-277.*

[6] Clack, C. and Myers, C. 1995. The Dys-functional student. *Lecture Notes in Computer Science 1022, pp 289-309*

[7] Clack. C. Myers, C. Poon, E. 1995. Programming with Miranda. Prentice Hall.

[8] Clack, C. and Peyton Jones S. 1985. Generating Parallelism from Strictness Analysis. *In Proceedings Implementation of Functional Languages 1985, Chalmers University Report 17, pp 92-131.*

[9] Hudak. P. 1989. Conception, Evolution and Application of Functional Programming Languages. *ACM Computing Surveys. Vol. 21. no 3, pp 359-411.*

[10] Landin. P.J. 1966. The next 700 programming languages. *Communications of the ACM. Vol. 9, no. 3.*

[11] Milner, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Science. Vol. 17, pp 348-375.*

[12] Myers, C.. Clack, C. & Poon, E. 1993. Programming with Standard ML. Prentice Hall.

[13] Peyton Jones, S. 1986. The Implementation of Functional Programming Languages. Prentice Hall

[14] Peyton Jones, S. Clack, C. and Salkild, J. 1987. GRIP: A parallel graph reduction machine. *ICL Technical Journal 5(3), pp 595-599.*

[15] Research Software Limited. 1990. The Online Miranda System Manual.

[16] Turner, D. 1985. An Introduction to Miranda. *In: The Implementation of Functional Programming Languages, 1986 by Peyton Jones. Appendix A. Prentice Hall*

[17] Turner, D. 1987. Functional Programs As Executable Specifications. *In Hoare, Shepherdson (Eds.), Mathematical Logic and Programming Languages, Prentice-Hall.*

Appendix A : User Manual

This user manual has been written with my prototype system in mind as it stands. Parts of the system have not been implemented, especially those which deal with the Miranda/UNIX interface. To use my system, the following stages must be completed. Before the stages involved, some useful information about using Miranda :

- **starting Miranda with a script file** : type `mira` followed by the script name.
- **changing to another script from within Miranda** : type `/f` followed by the script name.
- **editing a script in the UNIX shell** : type `jove` followed by script name. To save and exit type `<control-X-S-X-C>`
- **editing a script in Miranda** : type /e at the Miranda prompt. Same as above to save and exit.
- quitting Miranda : type `/q` at the Miranda prompt.

Stage 1

Create a Miranda script though not from within the Miranda environment but by using a standard text editor in a UNIX shell. The Miranda script must have a `.m` extension. Redirect the results of compiling this script file by typing the following at the UNIX prompt.

```
mira "filename.m" > errs
<type control-C after a few seconds to give a UNIX prompt>
```

A file will be created in your working directory called errs. At the UNIX prompt display the contents of `errs` using the 'more' or 'cat' commands. ('more' is the preferred command here as err may be quite large depending on the errors that occurred during compilation. 'cat' could simply send everything off of the screen.)

If you see that no errors have occurred during compilation of your script, then there is no need to go any further using my system. If errors have been detected, then you will notice them here and you should progress with stage 2.

Stage 2

Using a text editor or from within the Miranda system, edit the file rm_header.m and change line 4 of the definition of `err_output`, which is near the start of the file. The line which reads `= system "cat ....."` should be edited to include the path of the file, `errs`, which was created above. For example, if errs is contained in the directory ~zcacppk/Proj/, then this line should read

```
= system "cat ~zcacppk/Proj/errs"
```

Having done this, it is not required that the file rm_header.m be compiled, although this will be done automatically if it is edited from within the Miranda environment.

Stage 3

Enter the Miranda environment with the file output.m, by typing the following at the

UNIX prompt :

```
mira output.m
```

Now, at the Miranda prompt, typing the word 'go' will result in the errors that have occurred during compilation of your script file to be analysed and meaningful error messages be output.

Stage 4

If all of the errors that have occurred have a diagnosis provided for them from my system, then my system can serve no more use. If however, type errors have been detected, then more must be done to analyse them. Make a note of the name of the function in which the error has occurred. If there is more than one type error, then each of the function names must be noted and stages 5, 6 and 7 repeated for each *different* function in error.

Stage 5

Edit the file "find_fn.m", using a standard text editor or from within the Miranda system, just as before. At the start of the file, change the definition of prog_code (around line 6) to contain the path of the file that contains the type error to be resolved. For example, if the script file was called `dummy.m`, the path of which was ~zcacppk/Proj/, then this line should read

```
= system "cat ~zcacppk/Proj/dummy.m | grep '^>'"
```

Below this is the identifier `func`, which should be assigned to the name of the function that is in error. E.g. `func = "foo"`. Compile find_fn.m by saving and exiting the text editor if you are already in Miranda or by starting up Miranda with `find_fn.m` otherwise.

Type `at_last` at the Miranda prompt and after a few seconds, the Miranda prompt will reappear. Miranda can now be exited. In your working directory, you should have a new file called new.m.

Stage 6

For every equation within the original erroneous function, a new function has been written from each equation in new.m. Each of these functions has the name of the original erroneous function but with a number appended onto the end of their name, apart from the first function - this retains its original name. For example, if the erroneous function `foo` contained 4 equations, then 4 new functions relating to each equation in `foo` would be in new.m : `foo`, built from the first equation in foo, `foo1` from the second, foo2 and foo3 in just the same way.

By starting up Miranda with the file new.m (`mira new.m` from UNIX or `/f new.m` if already within Miranda) the type checker can be invoked on each of these new functions. For every new function, invoke the type checker (e.g. `foo ::`) and make a note of the type signature that is produced. When complete, each of these signatures should be written to a file called specfile. This file contains only the type signatures and nothing else.

Stage 7

Start Miranda with the file compare.m. At the Miranda prompt, type `check`. Analysis of the type error should be produced, with information on whether conflicts have occurred in arguments or return types or both. Repeat from stage 5 for any other error that has occurred.

Appendix B : System Manual

The actual code that forms the prototype of my system is stored in the following directory and can be accessed by typing :

cd ~zcacppk/newproj/

Changing into this directory and typing `ls` at the UNIX prompt, you will get the following listings for the contents of this directory, which relate to the files which are presented in appendix B. Files with a `.m` extension are *editable* Miranda scripts and files with `.x` extensions are the *non-editable* compiled versions for the appropriate `.m` files with the same name.

```
Testfiles       find_fn.m       get_dets.x      rm_header.m
compare.m       find_fn.x       output.m        rm_header.x
compare.x       grep.m          output.x
diagnosis.m     grep.x          parser.m
diagnosis.x     get_dets.m      parser.x
```

A further directory is contained within ~zcacppk/newproj/ directory, which contains files relating to the testing of the prototype system. It can be accessed by typing :

cd Testfiles

Files within this directory are all readable and used for the testing of the system.

```
cleancomp       errfile.m       errs        fafdwerrors
```

Any file in this and previous directories may be accessed from within the Miranda environment as long as it has a .m extension. A file can either be started up at the Miranda prompt or changed to once Miranda has been started. If 'mira' is typed at the UNIX prompt without a filename, a default script.m is created and this is the work file until changed by the programmer.

UNIX `mira <filename.m to start with>` from the UNIX shell, at the UNIX prompt

or

Miranda `/f <filename.m to change to>` from within Miranda, at the miranda prompt

Having edited a Miranda script file, when the file is saved and exited the Miranda compiler is automatically invoked on the file. Compilation of the `.m` script creates a new file in the current working directory with an identical name and a `.x` extension. Although these files can be loaded into a text editor, it is not recommended and this would be of little use anyway.

Each of the files involved in the operation of the prototype system have their own specific function. Starting with the `.m` files found in the ~zcacppk/newproj directory, here follows a brief description of the contents of each one.

compare.m

This file contains the code which deals with the comparison of type signatures of rewritten functions which originally contained type errors.

diagnosis.m

Containing the rules for each of the errors that my system caters for, this file should be edited if a new error is to be added to my system. A new 'rule' can be added following the same format as the other rules. There are certain formatting characters in the output, as the output from this file is designed for UNIX, running x-windows. There are no guidelines for how big a window is and tabs and new-lines should be used for the formatting of output carefully. It is often the case that trial and error is the best method to get output looking right.

find_fn.m

This file should be edited if it is required to change the way the new file with the rewritten functions is to be generated. As the file currently stands, it is not specified in the functionality how generation of this new file, new.m, can be done automatically. If this feature were to be added to my system, the specification would go in this file.

grep.m

This file contains the specification for the functions which detect regular expressions within text. It is unlikely that this file should ever be edited unless it is required that the functionality of grep be extended.

get_dets.m

This file contains the functionality which captures the details of each error that has occurred during compilation. The specification of the main data structure that describes the type 'error' is also included in this file. The main functionality within this file includes the functions which obtain the line number that an error has occurred on, the type of the error and the nature of the error.

If the functionality of my system were to be extended to include dealing with errors concerning the Miranda/UNIX interface, user defined types etc., the abstype err - the data structure for an error - would have to be altered to cater for these. If these new errors could be resolved by inspection, the file diagnosis.m would have to altered to include the rule for solving the error.

output.m

This file contains the functions which compile the meaningful error message for the user. This is the top-level function that is run at the Miranda prompt to access my system. Again, this file would have to be altered if new errors were to be added to the scope of my system.

parser.m

parser.m contains the functions which break text up into chunks, the delimiting character being a space (' '). Any like functions that deal with breaking up functions into chunks required for processing by other functions should be put in this file.

rm_header.m

Removes the Miranda start-up header from the results of compiling a script, leaving only the errors that have occurred. Beware - whenever the Miranda system is updated, parts of the start-up header change, specifically the date of release of the new update. Whenever the release changes, rm_header.m has to be altered.