Programming with Miranda

Chris Clack, Colin Myers and Ellen Poon

August 29, 2011

Contents

Index **307**

Preface

The purpose of this book is to teach structured programming skills using the functional programming language Miranda. It may be used as an introductory textbook for people with little programming experience or as an intermediate textbook for more advanced programmers who wish to learn a functional programming language. Both novice and advanced programmers will learn how to use a functional language to develop sound software design and code management techniques. Additionally, advanced programmers will find that knowledge of a functional language provides a foundation for further studies in the theory and implementation of programming languages and formal specification.

Miranda is one of the most popular functional languages employed in education and research, used in over 500 sites, and it is also increasingly having an impact on industry. Miranda has evolved from earlier languages such as SASL and KRC, which were developed in 1976 (Turner, 1976) and 1982 (Turner, 1982), respectively. It is a general-purpose programming environment, with a compiler embedded in an interactive system (implemented under UNIX[1]) providing access to a screen editor, an on-line reference manual and an interface to UNIX. Separate compilation is supported (to an intermediate code, which is subsequently interpreted), with automatic recompilation of altered source files. The Miranda programming system is a product of Research Software Limited; this textbook is based on Miranda *release two*.[2]

This is a practical programming book. Theoretical issues are avoided unless they are essential for a proper understanding of the use of the language. However, some formal definitions are introduced in order to aid the reader when referring to other texts. The boundary between those theoretical matters that are discussed and those that are not is, unavoidably, somewhat arbitrary. For example, Chapter 3 discusses the use of inductive reasoning to help with software design; however, we

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

[2]The Miranda system can be obtained directly from Research Software Ltd of 23 St Augustines Road, Canterbury, Kent, CT1 1XP, UK. email: mira-request@ukc.ac.uk.

do not extend this to the use of denotational semantics to reason about a program. Similarly, we provide a gentle introduction to the theory of strong typing, but we do not discuss type inference mechanisms. We do not discuss the lambda calculus, formal program transformation, numerical analysis, efficiency or any implementation methods.

The approach we adopt in this book is to show good software engineering principles by discussion of both correct and incorrect design decisions, using realistic examples. There is no Ackermann function and there are no miracles.

Acknowledgements

This book was developed during the teaching of functional programming courses to undergraduate and postgraduate classes at University College London and the University of Westminster. We thank our students for their many helpful recommendations and corrections.

Many thanks also to Don Clack, Simon Courtenage, Michael Fourman, Mark Hardie, Neil Harris, Hessam Khoshnevisan, Dave Pitt, Mike Poon, Mark Priestley, Will Richardson and David N. Turner for their positive suggestions concerning the companion volume to this book, *Programming with Standard ML*. These comments have proven equally valuable in the writing of this book.

We also thank Anthony Davie, Mark d'Inverno and Dr John Sharpe for their many helpful suggestions and corrections. Finally, thanks to David Turner for his many constructive comments and to Research Software Limited for permission to include information about the Miranda system in this book, and for allowing us to quote from the Miranda on-line reference manual.

Chris Clack
Colin Myers
Ellen Poon

Introduction

Programming languages may be grouped into several "families" with similar characteristics. Two of the most important families are the *imperative* languages (also known as "procedural" languages) and the *functional* languages. The imperative family includes such languages as BASIC, Pascal, C, Fortran, Ada, Modula-2 and COBOL. The functional family includes Miranda,[1] SML, Lispkit,[2] FP and Haskell; a good survey is provided in (Hudak, 1989).

The basic difference between functional and imperative languages is that functional languages are concerned with *describing a solution to a problem*, whereas imperative languages are concerned with *giving instructions to a computer*. Rather than attempt an explanation in isolation, this introduction *compares* the functional and imperative styles of programming. A number of claims will be made for the functional style and these will be substantiated in the main body of the text.

The reader should notice that this chapter compares *styles* and not *languages*: it is often possible to use a functional style when programming in an imperative language. Readers with no previous programming experience may safely skip this chapter on first reading.

Implications of the imperative style of programming

A program written in an imperative language describes in detail a sequence of actions that the computer must execute (these are sometimes called commands—hence the name "imperative"). This means that the imperative programmer must be familiar with the way the computer works. In general, a programmer must think in terms of the computer hardware—the conceptual model is one of memory locations containing values which may change during the execution of a program.

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

[2]Lispkit is a pure functional subset of the well-known language LISP which was the first important language with functional features.

The major consequence of this conceptual model is that a programmer is forced to muddle together the three separate activities of:

1. Describing the solution to a problem.
2. Organizing that solution into a set of instructions for a computer.
3. Administering low-level storage allocation.

A programmer should only be concerned with the first of these.

To a great extent this complexity derives from one particular feature—the assignment statement (Backus, 1978). The dangers inherent in the assignment statement include ambiguous pointer assignment, forgetting to initialize variables or incorrectly changing loop control variables (possibly causing an infinite loop). The availability of global variables also offers the opportunity of bad programming style, since it becomes very difficult to keep track of what value the variable is meant to have at any time and which parts of the program are changing the value of the variable. Assignment to pointers serves to compound these problems.

Consider the following Modula-2 solution to select all the items in an integer array that are less than 10:

```
j := 1;
FOR i := 1 TO LineLength DO
     IF line[i] < 10 THEN
        newline[j] := line[i];
        j := j + 1;
     END
END
```

The code has a relatively difficult interpretation; it is not easy to read compared with the simplicity of the natural language specification. Even in this small program extract, the major concern has been with the assignment to storage and maintaining indexes; that is, making the solution fit the hardware model.[3] This intellectual overhead is magnified if the problem were to select all the items less than 10 in a dynamic linked list. The emphasis would be even more towards the manipulation of memory rather than the representation of the data structure or the selection of items from it.

In general, the following observations hold for most existing imperative languages:

1. They have a limited set of control structures for expressing iteration and for combining subcalculations and hence it is often difficult to take advantage of a high-level modular design.
2. They have a limited set of built-in data structures. Although it is possible to model other data structures, the process often involves complex manipulation of pointers and a lot of code.

[3]To keep this example simple, the problem of retaining knowledge of the array newline's size has been ignored.

3. They have complex semantics, which makes it difficult to apply formal reasoning and hence more difficult to arrive at a correct solution.

Benefits of the functional programming style

Functional languages are an example of the *declarative* style of programming, whereby a program gives a description (or "declaration") of a problem to be solved together with various relationships that hold for it. It is the responsibility of the language implementation (perhaps a compiler or an interpreter) to convert this description into a list of instructions for a computer to run.[4] There are a number of implications of this alternative computational model:

1. A programmer does not have to worry about storage allocation. There is no assignment statement—a functional programmer does not assign values to storage locations, but instead has the ability to give names to the values of expressions; these names may then be used in other expressions or passed as parameters to functions.
2. The fact that in a functional language a name or an expression has a unique value that will never change is known as *referential transparency.* As a consequence of referential transparency, subcomputations always give the same result for the same arguments. This means that the code is safer and may often be reused in similar contexts; the nett result is a higher level of quality assurance and faster programming.
3. Functions and values are treated as mathematical objects which obey well-established mathematical rules and are therefore *well suited to formal reasoning.* This allows the programmer a high level of abstraction and so greater flexibility in defining control structures and data structures.
4. The syntax and semantics of functional languages tend to be simple and so they are relatively easy to learn. Furthermore, the resultant programs tend to be more concise and have fewer mistakes. Brooks (Brooks, 1975) observes that *"productivity seems constant in terms of elementary statements"*; if this applies equally to functional programmers then the conciseness of functional programs should lead to greater programmer efficiency.

It is not possible to justify all of these assertions in this brief introduction but, as a sample of the functional style, a typical solution to the problem discussed in the previous section (of selecting items from a sequence of numbers) would be:

```
filter (lessthan 10) number_sequence
```

[4]Many functional languages are based on a mathematical theory known as the "lambda calculus" (Revesz, 1988). At advanced levels, it may be useful for a programmer to understand this theory but, in general, it is not necessary for the novice or intermediate programmer.

There is no need to manage storage, no danger of forgetting to initialize or increment variables and no need to worry about where the results are kept! The code is clearly much nearer to the specification—indeed sometimes functional programs are considered as "executable specifications" (Turner, 1985a).

The functional style is also more flexible. For example, changing the Modula-2 code to select all items not equal to `"fred"` from a sequence of strings requires writing a new function—even though the structure of the solution is identical. Changing the functional code is straightforward:[5]

```
filter (notequal "fred") string_sequence
```

Finally, there is the question of efficiency. It is claimed that the higher the level of the programming language (that is, the further away from the hardware model), the less efficient the language is in terms of speed and memory utilization. To a certain extent this may be true,[6] however two very important points are worth making:

1. The first concern of a programmer is that a a program is written on time and is *correct*: performance is generally a secondary concern.
2. Functional languages are not tied to the von Neuman architecture (Backus 1978). This facilitates implementations on other hardware configurations such as multi-processors.

Consider the following:

```
sumofsquares := square (x) + square (y)
```

In an imperative language it is necessary to have an ordering of instructions; typically `square (x)` is executed before `square (y)`. Logically, however, it does not matter whether `square (x)` is executed first or second; actually both `square (x)` and `square (y)` could be calculated simultaneously. The very fact that functional languages are not based on any particular hardware conceptual model helps them to take advantage of parallel and multi-processor hardware (Peyton Jones, 1987; Kelley, 1989). In summary, the programmer is freed from the burden of implementation.[7]

However, the above discussion should not lead the reader to the idea that functional languages are the cure for all programming evils. There is still the need to

[5]In Miranda the function `filter` is built into the system, but could actually be written by the programmer with just three short statements. The reason why the code is so flexible derives directly from the ability to treat functions as parameters to other functions; `filter` takes any function which returns a Boolean value.

[6]Although modern implementations of functional languages rival the speed of compiled imperative languages such as Pascal and C.

[7]Because of this freedom, functional languages are also well suited to program transformation techniques. These allow programs to be mathematically derived from their specifications, or an inefficient algorithm to be converted to an efficient one by the compiler (Darlington *et al.*, 1982). Program transformation techniques also guarantee that new errors are not introduced into the code. However, such optimization techniques are outside the scope of this book.

recognize problems and find sensible solutions! There are also a number of software engineering issues that arise from large-scale programming tasks; in the main these are beyond the scope of this book.

Miranda in context

Miranda is not an acronym but a woman's name possibly first used by Shakespeare in *The Tempest*, and is the Latin for "to be admired". It has evolved from earlier languages such as SASL and KRC, which were developed by David Turner at (respectively) St. Andrews University in the late seventies and the University of Kent in the early eighties. An overview of Miranda's conceptual framework and key features is given in (Turner, 1985b).

Miranda is a purely functional language, with no imperative features of any kind. Following the style of SASL and KRC, Miranda has *lazy* semantics; this permits the use of potentially infinite data structures and supports an elegant style of problem decomposition. Miranda is now a general purpose programming language and is used for a wide variety of activities, especially in the areas of proof systems and specification.

Miranda shares with most other functional languages the notion that functions and data values have "equal citizenship"; that is, they can be manipulated in the same manner (Landin, 1966). Other Miranda characteristics of note include polymorphism (which gives flexibility to the strong type system), pattern matching (to provide an elegant selection control structure), list comprehension, partial function application and new type construction. These topics are dealt with in depth in the main text.

Miranda is an example of a *lazy* functional language, which means that a function's parameters are normally evaluated only if they are needed. This is in contrast to the alternative *strict* style of evaluation, whereby a function's parameters are normally evaluated before the function body itself. (A good example of the latter style is Standard ML (Myers, Clack and Poon, 1993)). The Bibliography contains references which will provide further discussion of strict and lazy implementations.

How to use the book

The first two chapters of this book provide a simple introduction to Miranda syntax and the functional approach to programming, whilst also serving to reorientate the imperative programmer. Some of the features shown will be quite familiar to anyone with imperative programming experience; these include the standard mathematical operators, the clear distinction made between different types of data and the idea of function definition and application. Novel features are dealt with in greater depth; these include pattern matching, polymorphism and the use of recursion for program control.

The third and fourth chapters explore in depth the heart of the functional style. They are core to Miranda (and functional) programming and will probably appear quite new to a programmer only familiar with an imperative programming language. In Chapter 3, the list aggregate type is introduced as the most fundamental Miranda data structure and various styles of recursive programming using lists are investigated. The chapter ends with the design of a program similar to the UNIX utility *grep*, which is used in subsequent chapters to highlight the expressive power of new features. Chapter 4 introduces the important concept of partial application and demonstrates that functions can be treated as data. It also shows how recursive solutions to programming tasks can be generalized to facilitate control structure abstraction.

Chapter 5 explores two important concepts: controlling the availability of identifiers and functions; and lazy evaluation. The former is essential for any production programming since it gives the ability to make blocks of code private to other blocks of code—thereby facilitating safer and more reusable software. The latter provides an important mechanism for combining functions and structuring programs.

The next two chapters explore the relationship between data and process by showing Miranda's powerful facilities to create new types and data structures, and to encapsulate a type and its associated operations into a single abstraction.

Chapter 8 introduces the programmer to file handling and interactive programming, and Chapter 9 provides tools for medium to large-scale program construction.

It is recommended that the main body of the book is studied sequentially, and that the reader should attempt the exercises as they are encountered. The exercises serve both as a review of the student's current knowledge and also as a commentary on the text; sample solutions are given at the end of the book.

Chapter 1

Operators, Identifiers and Types

This chapter starts by showing some elementary Miranda programs, where only simple expressions involving built-in arithmetic and relational operators are used. Since these simple expressions can only make simple programs, more enhanced features are needed in order to create real-life applications. These include the access to built-in functions, such as those to calculate the sine or square root of a number, and more importantly, the ability to associate names either with immediate data values or with expressions which are evaluated to some data value. Once a name is defined, it can be recalled in all subsequent expressions within a program. This makes programs more readable and makes it easier to create more complex expressions.

In order to promote good programming, Miranda has a strong type system whereby a function can only be applied to an argument of the expected type. Any attempt to give an argument of a type other than the one it expects will result in a "type error". In practice, the strong type system is a useful debugging tool.

Good style is further encouraged by the use of comments in order to document the code.

1.1 A Miranda session

Typically, the Miranda system will be entered from the host system by typing the command `mira`. Miranda responds by displaying an initial message, such as:

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

             version 2.009 last revised 13 November 1989

              Copyright Research Software Ltd, 1989
```

This message may be surrounded by site and version specific details and may be followed by the Miranda system prompt, `Miranda`.

Exit from Miranda systems is achieved by typing `/quit` or `/q`, to which Miranda responds: `Miranda logout` and then returns control to the UNIX shell.

Using the Miranda system

The Miranda interactive system issues a prompt and waits for the programmer to type something. The programmer may type either an expression to be evaluated or an instruction to the Miranda system; all instructions start with a slash (for example, `/q` to quit from Miranda, or `/e` to ask Miranda to edit a file). Miranda will either evaluate the expression or obey the instruction; it then issues a fresh prompt and waits for the programmer to type something else.

The simplest use of Miranda is to type in an expression to be evaluated; in this way, Miranda acts rather like a desk calculator.

All Miranda expressions and instructions, when entered at the prompt, must be terminated by a newline. The following is an example of an expression with the Miranda system response:

```
Miranda 3 + 4
7
```

Miranda responds to the expression by simply displaying the result: `7`.

An expression cannot be split across more than one line; if it seems incomplete then the system will give an error message, such as:

```
Miranda 3 +
syntax error - unexpected newline
```

Standard operators

Miranda provides built-in functions for the standard arithmetical and relational operations. The rest of this section discusses their general characteristics, more specific detail being provided in subsequent sections.

Examples:

```
Miranda 34 + 56
90

Miranda 2.0 * 3.5
7.0
```

```
Miranda 3 > 4
False

Miranda 3 = 3
True
```

All of the arithmetic operations can be used several times in an expression or in combination with other operators. For example:

```
Miranda 2 + 3 + 5
10

Miranda 2 + 3 * 5
17

Miranda (2 + 3) * 5
25
```

The above behave as expected, with brackets being used to enforce the order of evaluation.

An interesting, although perhaps surprising, feature of Miranda is that the relational operators `>`, `>=`, etc., can also be chained together to form "continued relations". For example, the expression `(2 < 3 < 4)` evaluates to `True`. However, the expression `((2 < 3) < 4)` would give an error, as explained later in this chapter.

Simple function application

Miranda provides a number of useful functions, full details of which are given in Section 28 of the On-line Manual (see below). To use one of these functions, it must be applied to an argument. Function application is denoted by giving the name of the function, followed by the argument enclosed in brackets. If the argument is a single value then the brackets may be omitted.

Examples:

```
Miranda sqrt (4.0 + 12.0)
4.0

Miranda sqrt (25.0)
5.0

Miranda sqrt 9.0
3.0
```

```
Miranda (sqrt 9.0) + (sqrt 25.0)
8.0

Miranda sqrt (sqrt 81.0)
3.0
```

1.1.1 On-line help

Miranda offers a brief *help* screen and a more detailed on-line *manual*. By typing `/help` or `/h` the help screen appears with a short explanation of Miranda's interactive features. The on-line manual can be accessed by typing `/man` or `/m`. This gives rise to the help menu as shown in Figure 1.1. Notice that the on-line manual is extensive and contains detailed information for experienced users as well as novices; furthermore, the manual is well structured for on-line browsing and it is consequently neither necessary nor advisable to print a hard copy.

```
 Miranda System Manual   Copyright Research Software Limited 1989

 1. How to use the manual system    20. Algebraic types
 2. About the name "Miranda"        21. Abstract types
 3. About this release              22. Placeholder types
 4. The Miranda command interpreter 23. The special function {\bf show}
 5. Brief summary of main commands  24. Formal syntax of Miranda scripts
 6. List of remaining commands      25. Comments on syntax
 7. Expressions                     26. Miranda lexical syntax
 8. Operators                       27. The library mechanism
 9. Operator sections               28. The standard environment
10. Identifiers                     29. Literate scripts
11. Literals                        30. Some hints on Miranda style
12. Tokenisation and layout         31. UNIX/Miranda system interface
13. Iterative expressions           32. -->> CHANGES <<--
14. Scripts, overview               33. Licensing information
15. Definitions                     34. Known bugs and deficiencies
16. Pattern matching                35. Notice
17. Compiler directives
18. Basic type structure            99. Create a printout of the manual
19. Type synonyms                   100. An Overview of Miranda (paper)

::please type selection number (or return to exit):
```

Figure 1.1 The menu for the Miranda on-line manual.

1.2 Identifiers

In Miranda it is possible to give a name or *identifier* to the value of an expression. This is achieved by typing `/e` or `/edit` to enter the editor and then modifying the default *script file* (which has the name `script.m`). An expression may then be given a name, using the format:

identifier = *expression*

The simplest kind of expression is a basic data value, examples of this being:[1]

```
hours   = 24
message = "Hello World"
```

The value associated with the name `hours` is now `24` and the value associated with the name `message` is `"Hello World"`.

Once the editor has been exited, the system checks the syntax of the script file and, if it is valid, then the value given to a name can readily be recalled by entering the name as an expression to be evaluated:

```
Miranda hours
24
```

Similarly, the value returned by the expression `((4 * 30) + (7 * 31) + 28)` may be given a name, within the script file, as follows:

```
days = ((4 * 30) + (7 * 31) + 28)
```

and recalled by entering `days` at the system prompt:

```
Miranda days
365
```

In general, any name may appear on the left-hand side of the equals sign and any expression may appear on the right-hand side. Note that a name is a kind of expression, but an expression is not a name, so `twentyfour = hours` is a legal definition but `(3 + 4) = hours` is not. Giving a name to a value is useful because that name can then be used in subsequent expressions; the choice of meaningful names will make a program easier to read.

The following is a simple example of an expression that itself involves names that have been previously defined:

```
hours_in_year     = (days * hours)

hours_in_leapyear = ((days + 1) * hours)
```

[1]In the rest of this book, script files are presented within a box.

The values of these two identifiers can be recalled as follows:

```
Miranda hours_in_year
8760

Miranda hours_in_leapyear
8784
```

Legal and sensible identifiers

In practice there is a limitation to the choice of names. There are three restrictions:

1. Certain words are reserved by the Miranda system and therefore *cannot* be used:[2]

   abstype div if mod otherwise
   readvals show type where with

2. Certain words have already been defined within the Miranda *Standard Environment* and should similarly be avoided. Appendix A contains a list of those definitions provided by the Miranda system.
3. An identifier must begin with an alphabetic character and may be followed by zero or more characters, which may be alphabetic, digits or underscores (_) or single quotes ('). For names of simple expressions such as described in this chapter, the first character *must be in lower case.*

1.2.1 Referential transparency and reusing names

The functional programming style is that a name is given to a value rather than giving a name to a memory location. In an imperative programming language, the memory location referred to by a name is constant, but the value stored in that location may change; in a functional programming language, the value itself is constant and the programmer is not concerned with how or where this value is stored in memory.

The functional style has the important consequence that the values of names do not change and therefore the result of a given expression will be the same wherever it appears in a program. Any program written in this style is said to have the property of *referential transparency.*

Programs exhibiting referential transparency have many benefits, including the fact that it is easier to reason about these programs and hence easier to debug them. For example, one part of a program may be tested independently of the rest.

[2]Throughout this book, the use of a **bold** font indicates a reserved name.

Miranda adopts this style and hence it is not possible to use the same name more than once.[3] Thus constructing the following definition sequence within the script file is illegal:

```
message = "Hello World"

message = "Goodbye Cruel World"
```

On exit from the editor, the Miranda system will report a syntax error and abandon compilation.

1.3 Types

Types are a way of classifying data values according to their intended use. A data type may be specified either by enumerating the values that data items of that type may have, or by the operations that may be performed upon data items of that type. Conscientious use of a type classification system aids the construction of clear, reliable, well-structured programs.

Miranda has a number of built-in data types, including the simple types—integers, fractional (or real) numbers, strings, characters and Booleans—together with fixed-length aggregate types which are discussed in this chapter. Chapter 3 introduces the variable-length list aggregate type, whereas the facility to define new data types is presented in Chapter 6.

Miranda is a *strongly typed* language. This means that the system uses information about the types of data values to ensure that they are used in the correct way. For example, the predefined function `sqrt` expects to be applied to an argument that is a number. In Miranda, numbers are not surrounded by quotation marks and so the following is detected as an error (the error message, itself, will be explained in the next subsection):

```
Miranda sqrt '4'
type error in expression
cannot unify char with num
```

This principle extends to the infix arithmetic operators, in that an error will arise if the wrong type of value is used. For example, in the following session Miranda expects the function **div** to be used with two numbers and gives an error if used with a number and a Boolean value:

```
Miranda 2 div True
type error in expression
cannot unify bool with num
```

[3]See, however, Chapter 5.

1.3.1 Error messages

On exit from the editor, Miranda checks the script file (assuming that it has changed) and, if the script file is an error-free Miranda program, then the following message will be printed:

```
compiling script.m
checking types in script.m
Miranda
```

By contrast, if the program contains errors then Miranda will issue one or more error messages and will refuse to accept *any* of the definitions in the program.

Error messages highlight two kinds of error; syntax errors and type errors. Here is an example of a syntax error, where the value `"hello"` has been wrongly entered as `"hello`:

```
wrong_message = "hello
```

On exit from the editor, Miranda gives the following report, including the line number where the syntax error was found (which is useful in large programs):

```
compiling script.m
syntax error: non-escaped newline encountered inside string quotes
error found near line 1 of file "script.m"
compilation abandoned
```

Note that the error message assumes that the fault is the newline occurring immediately after the word `"hello`; this is because a newline is not allowed unless the terminating quote character has been given.

Here is an example of a type error:

```
wrong_value = sqrt 'A'
```

On exit from the editor, Miranda gives the following report, including the line number where the error was found (which is useful in large programs):

```
compiling script.m
checking types in script.m
type error in definition of wrong_value
(line   1 of "script.m") cannot unify char with num
Miranda
```

The last line of the error message states that Miranda "cannot unify char with num". This means that Miranda realizes that the built-in function `sqrt` expects to be applied to a number (a "num") and not to a character (a "char"). In general, type error messages may include obscure information for advanced programmers as well as information that is useful for relative beginners. These type error messages will become clearer in later chapters.

1.3.2 Numbers

Miranda recognizes two sorts of number: integers and fractional (or real) numbers, and is generally happy for these two subtypes to mix. Notice that fractional numbers are indicated by decimal points (trailing zeros are required) or by using exponential format. The precision cannot be specified by the programmer; it is determined by the machine in use, but it is guaranteed to be at least the equivalent of 16 decimal places.

Data values of both sorts of number are indicated by the type name `num` and may be manipulated by the operators shown in Table 1.1. Notice that whenever a fractional number appears as one of the operands to these operators then the result will be a fractional number. The only exceptions are **div** and **mod** which expect two integer operands, and will produce an error message otherwise.

Table 1.1 Operations on numbers.

`+`	`addition`
`-`	`subtraction`
`*`	`multiplication`
`/`	`real number division`
`div`	`integer division`
`mod`	`integer remainder`
`^`	`exponentiation`

For example:

```
Miranda 365.0 / 7.0
52.142857142857

Miranda 365 / 7.0
52.142857142857

Miranda 365 / 7
52.142857142857

Miranda 365 div 7
52

Miranda 365 mod 7
1

Miranda (1 / 3) * 3
1.0
```

```
Miranda 3 ^ 3
27

Miranda 4.0 ^ -1
0.25

Miranda days div 7
52
```

where `days` has already been defined as having the value 365.

Notice, that the fractional number divide operator, `/`, allows integers and fractional numbers to mix but that the **div** operator expects *both* of its operands to be integers:

```
Miranda 365 div 7.0
program error: fractional number where integer expected (div)
```

Negative numbers

It is possible to denote negative numbers by prefixing them with a minus sign: `-`. For example:

```
x = -33
```

However, the `-` character is actually a built-in prefix operator that takes a single argument and negates the value:

```
Miranda - (-33.0)
33.0

Miranda - (- (33.0))
33.0

Miranda -x
33
```

The last of the above examples shows that there need not be a space between the `-` and a name.

The fact that `-` can be used in this prefix manner is important, as illustrated by the following session which uses the built-in function `abs` (which takes a positive or negative number, and returns its absolute or unsigned, value):

```
Miranda abs -33
type error in expression
cannot unify num->num with num
```

```
Miranda abs -x
type error in expression
cannot unify num->num with num
```

Neither expression succeeds because the `-` appears as the first object after the function name `abs` and therefore Miranda believes that `-` is the argument for `abs` (which obviously gives an error). The intended meaning can be enforced by bracketing the `-x` or the `-33` thus forcing Miranda to interpret the whole subexpression as the argument to the function `abs` :

```
Miranda abs (-x)
33

Miranda abs (- 33)
33
```

To avoid confusion, the Miranda Standard Environment also provides the function `neg`:

```
Miranda abs (neg 33)
33
```

Integers and fractional numbers

Miranda generally treats integers and fractional numbers in the same manner. However, it is sometimes necessary to distinguish between integers and fractional numbers. Miranda provides the built-in predicate `integer` in its Standard Environment to test whether a number is a whole number:

```
Miranda integer 3
True

Miranda integer 3.0
False
```

It may also be necessary to convert from fractional numbers to integers. This can be achieved by the Standard Environment function `entier`, which returns the largest whole number which is less than the given fractional number:

```
Miranda entier 3.2
3

Miranda entier 3.7
3
```

```
Miranda entier (-3.2)
-4

Miranda entier (-3.7)
-4
```

1.3.3 Characters and strings

Miranda differentiates between single characters and strings of characters. As will be seen in Chapter 3, strings of characters are special instances of the type `list`.

Characters

A character is a single token, denoted by the type name `char`. For example:

```
Miranda 'a'
'a'

Miranda '1'
'1'
```

However, there is no “empty character”:

```
Miranda ''
syntax error: badly formed char const
```

Notice also that characters and numbers do not mix. For example:

```
Miranda '1' + 1
type error in expression
cannot unify char with num
```

Special characters

Miranda follows the C language philosophy of implementing “special characters” by a two-character sequence (the first character of which is a backslash). The special characters include the newline character (`'\n'`), the tab character (`'\t'`), backslash itself (`'\\' (backslash`) and quotes (`'\''`). It is also possible to use the ASCII decimal codes for characters; for example `\065` for the character `A` (in this format, a three-digit sequence is used after the backslash character).

Character to number type conversion

Miranda assumes an underlying character set of 256 characters, numbered from 0 to 255, where the first 128 characters correspond to the ASCII character set. The built-in function `code` returns the ASCII number of its character argument, whilst `decode` decodes its integer argument to produce a single character. For example:

```
Miranda code 'A'
65

Miranda decode 65
'A'
```

It is to be noted that an attempt to apply `decode` to a number not in the range `0` to `255` will result in an error:

```
Miranda decode 256
CHARACTER OUT-OF-RANGE decode(256)
```

Strings

Character strings are represented by any sequence of characters surrounded by double quotations marks and are denoted by the type `[char]`. Hence there is a difference between the character `'a'` and the string `"a"`. Strings may be operated on by the functions given in Table 1.2.

Table 1.2 Operations on strings.

`++`	`concatenation`
`--`	`subtraction`
`#`	`length`
`!`	`indexing`

The following examples show that Miranda responds by representing an input string *without* the quotation marks:

```
Miranda "functional programming is fun"
functional programming is fun

Miranda "aaa" ++ "rgh" ++ "!!!!"
aaargh!!!!

Miranda # "a"
1
```

```
Miranda # "a string"
8

Miranda "abc" ! 0
'a'

Miranda "abc" ! 1
'b'

Miranda "abc" ! 999
program error: subscript out of range
```

Notice that the index operator `!` considers the first position in the string as position 0.

Concatenation treats the empty string in the same way as addition treats the constant value zero, as illustrated below:

```
Miranda "" ++ ""


Miranda "1" ++ ""
1

Miranda "" ++ "anything"
anything
```

The string difference operator `--` returns a value which is derived from its first operand with some characters removed. The deleted characters are all of those which appear in the second operand (the actual sequence of characters being irrelevant):

```
Miranda "abc" -- "ab"
c

Miranda "abc" -- "ccar"
b

Miranda "abc" -- "def"
abc

Miranda "aa" -- "a"
a

Miranda "miranda" -- "mira"
nda
```

Special characters and strings

The convention for special characters also extends to strings, with `"\""` representing a string containing the double quotes single character. Similarly, the string `"\"Buy us a drink\""` will appear as the characters `"Buy us a drink"`. Notice that the two-character sequence is considered to be a single character, hence:

```
Miranda # "\"Buy us a drink\""
16
```

Number to string type conversion

It is possible to represent a number as a string of characters using the general purpose **show** keyword or by the more specific `shownum` function. For example:

```
Miranda "mir" ++ (show 123) ++ "anda"
mir123anda

Miranda "mir" ++ (shownum 123) ++ "anda"
mir123anda

Miranda (shownum (12 * 3)) ++ "anda"
36anda
```

1.3.4 Booleans

Boolean values are truth values, denoted by the type name `bool`. Data values of type `bool` are represented by one of the two constant values `True` and `False`.

Data values of type `bool` may be operated on by special logical operators shown in Table 1.3 or be produced as a result of the relational operators (`>`,`>=`,`<`,`<=`,`=`, `~=`).

Table 1.3 Logical operations on Booleans.

`~`	`logical negation`
`&`	`logical conjunction`
`\/`	`logical disjunction`

The following presents some simple examples of their use. Especial notice should be taken of the fact that the sign `=` is used both as a way of giving an identifier to a value, within a script file, and to check if two values are equal. This is shown in the following sample session (where `x` already has the value 3):

```
Miranda 3 = 4
False

Miranda 7.8 < 9.2
True

Miranda x <= 7
True

Miranda ~ True
False

Miranda (2 < 3) = (~ (~ True))
True

Miranda 2 < 3 < 4
True
```

The last example recalls that it is not necessary to bracket the relational operators to form more extended expressions. Indeed, if brackets were to be used, such as `((1 < 2) < 3)`, then this would give rise to an error because `(1 < 2)` evaluates to the Boolean value `True`, which cannot legally be compared to the number `3`.

There is no built-in conditional expression—the ability to denote alternatives is provided by conditional guards on the right-hand side of function definitions, which will be discussed in Chapter 2. These conditional guards will return Boolean values.

Boolean to string type conversion

As with values of type `num`, it is possible to convert values of type `bool` to their string representation, using the **show** keyword. For example:

```
Miranda "mir" ++ (show True) ++ "anda"
mirTrueanda
```

Lexicographic ordering

The relational operators will work with numbers and also with characters, Booleans and strings: Booleans are ordered such that `False` is less than `True`; characters take the ASCII ordering; and strings take the normal lexicographic ordering:

```
Miranda 'a' < 'b'
True
```

```
Miranda "A" < "a"
True

Miranda "A" < "A1"
True

Miranda "B" < "A1"
False
```

As expected, different types cannot be compared, hence it is meaningless to attempt to compare a character with a string:

```
Miranda 'a' < "ab"
type error in expression
cannot unify [char] with char
```

Fractional numbers and equality tests

Note that comparing two fractional numbers for *equality* is highly suspect, since many real numbers do not have a terminating decimal representation. Miranda allows fractional numbers to be tested for equality, but it is assumed that the programmer understands the limitations of fractional representations. Using the built-in function `abs` (which gives the absolute value of a number, regardless of its sign), it is possible to determine whether two fractional numbers `x` and `y` (which have been defined in the script file) are the same to within a given precision (say 0.1):

```
Miranda abs(x - y) < 0.1
True
Miranda
```

Conjunction and disjunction

The Miranda operators for both conjunction `&` (logical and) and disjunction `\/` (logical or) are sometimes known as "lazy" operators because they do not strictly need to evaluate their second operand. The left operand is always evaluated, but sometimes it is not necessary to evaluate the right operand in order to determine the overall truth value.[4]

The special operator `&` evaluates to `True` when both of its operands evaluate to `True`, otherwise it evaluates to `False`. Similarly, the special operator `\/` evaluates

[4]In this respect they are similar to the LISP conditional "cond" and the C operators "&&" and "||" but dissimilar to the Pascal "and" and "or" functions (which always evaluate each subexpression in a compound Boolean expression before deciding the overall truth value).

to `False` when both of its operands evaluate to `False`, otherwise it evaluates to `True`. Thus, in the expression `False & (x = y)` it is not necessary to evaluate the right operand because, regardless of the value of the right operand, the overall result will always be `False`. The current implementation of Miranda takes advantage of this fact and *does not* evaluate the right operand if the result can be determined from the left operand alone.

Example session, given x has the value 0 and y has the value 42:

```
Miranda (y = 42) & (x = 0) & (y = x)
False

Miranda (y = 42) & ((x = 0) & (y = x))
False

Miranda (x = 0) \/ ((y div x) > 23)
True
```

The last of these examples shows that the lazy evaluation of the right operand is quite useful when it is not desirable for it to be evaluated. In this case a divide by zero error has been avoided.[5]

1.3.5 Determining the type of an expression

The type of any expression may be determined by typing either the expression or its name followed by *two colons.* This invokes Miranda's type checker and reports the type. For example:

```
Miranda (3 + 4) ::
num

Miranda 'A' ::
char

Miranda x ::
num

Miranda True \/ True ::
bool
```

If the name is undefined, Miranda returns the answer *, which indicates that the value could have any type.

[5]Notice that the lazy evaluation means that `\/` and `&` are not the precise equivalent of logical disjunction and conjunction.

1.4 Tuples

This section introduces the *tuple* which combines values to create an *aggregate type*. As an example, it is possible to give a name to the combination of an integer, string and integer that represents a given date:

```
date = (13, "March", 1066)
```

Formally, the tuple represents the *Cartesian product* (or *cross product*) of the underlying components.

Domains and Cartesian products

In order to understand the type information in error messages (and to create new types as will be shown in Chapter 6), it is useful to understand the concept of *type domains*. The following is an informal introduction to this topic. The reader is referred to (Gordon, 1979) and (Thompson, 1991) for more details.

The collection of values that a type can have is known as its *domain*. For example, the domain of type `bool` is the collection of built-in values `True` and `False`. Similarly, the collection of values of the type `int` is the theoretically infinite range of whole numbers from *minus infinity* to *plus infinity*.[6]

During evaluation, it is possible for an error to occur: either an error message will be produced, or the program will enter an infinite loop. The formal treatment of type domains treats such an error as a valid ("undefined") result and a special "Error" value (representing both kinds of error) is added to the domain of every type. This error value is often given the special symbol $\perp$ and called "bottom". Thus, the formal expression of the type domain `bool` will contain the values {`True, False,` $\perp$}.

Some programming languages allow the program to recognize an error value as the result of a calculation and to take action accordingly (although it is normally impossible for the program to detect an infinite loop!). However, Miranda does not do this—for example, there is no special pattern which can be used in pattern-matching to detect an error value[7]—and so this informal presentation of type domains does not include $\perp$ as a domain value.

The type `(bool,bool)` contains the Cartesian product operator "`,`". This indicates that the domain contains all possible combinations which combine one value from the first domain (for type `bool`) with one value from the second domain (for type `bool`). Therefore the domain for the type `(bool,bool)` is fully represented by the four tuples:

[6]Of course, in practice, there is a machine-dependent restriction on this range.

[7]Although the programmer can express such behaviour with user-defined types—see Chapter 6.

```
    (False, False)
    (True, False)
    (False, True)
    (True, True)
```

Similarly, the infinite domain `(num,[char],bool)` contains all the tuples `(n,s,b)` where `n` is any value drawn from the number domain, `s` is any value drawn from the string domain and `b` is any value drawn from the Boolean domain.

Tuple equality

Any simple type has at least two operations defined upon its elements; that of equality (`=`) and inequality (`~=`). Thus any two tuples of the same type (whose elements are simple types) may be tested for equality, as is shown in the following session (a script followed by an interaction with the Miranda command interpreter):

```
date = (13, "March", 1066)
```

```
Miranda date = (13, "March", 1066)
True

Miranda (3,4) = (4,3)
False

Miranda date = (abs (-13), "March", 1065 + 8 mod 7)
True
```

As might be expected, tuples with different types cannot be compared:

```
Miranda (1,2) = (1, True)
type error in expression
cannot unify (num,bool) with (num,num)

Miranda (3,4,5) = (3,4)
type error in expression
cannot unify (num,num) with (num,num,num)
```

Tuple composite format

The *composite format* of a tuple can be used on the left-hand side of an identifier definition, thereby providing a convenient shorthand for multiple definitions:

```
(day, month, year) = (13, "March", 1066)
```

The names `day`, `month` and `year` can now be used as single identifiers:

```
Miranda day
13

Miranda month
March

Miranda year
1066
```

This shorthand notation is in fact a simple example of the powerful mechanism known as *pattern matching*, which will be discussed in Section 2.5. This general mechanism extends to complex patterns:

```
((day, month, year), wine, price)
           = ((13, "May", 1966), "Margaux", 60)
```

1.5 Properties of operators

An understanding of the properties of operators may significantly assist debugging and may aid the design of better functions, as will be seen in Chapter 4. This section explains the following key concepts:

1. *Precedence* and *order of association* determine the correct meaning of any expression which includes multiple occurrences of operators and/or function applications.
2. *Associative* and *commutative* functions and operators present an opportunity for the programmer to reorganize an expression without changing its meaning.
3. *Overloading* refers to the fact that some operators may manipulate values of different types.

1.5.1 Precedence

In an expression such as `(3 + 4 * 5)`, there is an inherent ambiguity as to whether the expression means `((3 + 4) * 5)` or `(3 + (4 * 5))`. This ambiguity is resolved by a set of simple rules for arithmetic operators; in the above example it is clear that the expression really means `(3 + (4 * 5))` because the rule is that multiplication should be done before addition. This is known as a rule of *precedence*; multiplication has a higher precedence than addition and therefore the subexpression `4 * 5` should be treated as the right operand of the addition operator. If it is necessary to override the precedence rules then brackets are used to indicate priority.

For example:

```
((3 + 4) * (5 + 6))
```

All Miranda operators are given a precedence level—a table of precedence levels (referred to as "binding powers") is given in Section 8 of the Miranda On-line Manual. However, it is important to remember that function application has a very high precedence, so that:

```
(sqrt 3.0 + 4.0)
```

is interpreted as:

```
((sqrt 3.0) + 4.0)
```

In general, the use of brackets is encouraged for reasons of safety and good documentation.

1.5.2 Order of association

Some operators (for example, `*` and `/`) have the same level of precedence; this means that there is still ambiguity in the meaning of some expressions. For example, the expression `(4.0 / 5.0 * 6.0)` is ambiguous—it might mean either `((4.0 / 5.0) * 6.0)` or `(4.0 / (5.0 * 6.0))`. Furthermore, repeated use of the same operator leads to similar ambiguity, as with the expression `(4.0 / 5.0 / 6.0)`. This ambiguity may be resolved by determining the *order of association*. In Miranda, all operators (except for `++` `--` `^` and `:`, as will be seen in Chapter 3) associate to the *left*. That is, `(4.0 / 5.0 * 6.0)` is interpreted as `((4.0 / 5.0) * 6.0)` and `(4.0 / 5.0 / 6.0 / 7.0)` is interpreted as `((4.0 / 5.0) / 6.0) / 7.0)`. This left-associating behaviour also extends to function application so that `(sqrt abs 1)` is interpreted as `((sqrt abs) 1)` and gives an error—the programmer should therefore take particular care to bracket function applications correctly. To summarize:

1. Precedence is always considered first; the rules for association are only applied when ambiguity cannot be resolved through precedence.
2. The rules governing the order of association only apply to operators at the same level of precedence; this may be two or more different operators, or perhaps two or more occurrences of the same operator (such as `(5 - 6 - 7)`).
3. Brackets are used for priority, to override precedence and the order of association. For example, although it is now clear that the expression `(4.0 / 5.0 * 6.0)` means `((4.0 / 5.0) * 6.0)`, it is possible to bracket the multiplication, to ensure that it happens first: `(4.0 / (5.0 * 6.0))`.

Associativity

For some operators, it makes no difference whether they associate to the right or to the left. These are called *associative* operators, for example multiplication:

```
Miranda ((4.0 * 5.0) * 6.0) = (4.0 * (5.0 * 6.0))
True
```

However, some operators are *not* associative:

```
Miranda ((4.0 / 5.0) / 6.0) = (4.0 / (5.0 / 6.0))
False
```

This property of *associativity* only refers to repeated use of the same operator in an expression; notice that an operator can only be associative if its result type is the same as its argument type(s).

Commutativity

Some operators are *commutative*; that is, it makes no difference in what order their parameters appear. For example:

```
Miranda (4 * 5) = (5 * 4)
True
```

However, some operators are *not* commutative:

```
Miranda (4 - 5) = (5 - 4)
False
```

Sometimes the non-commutativity of an operator is due to it being lazy in one of its operands:

```
Miranda True \/ ((3 div 0) = 3)
True

Miranda ((3 div 0) = 3) \/ True

program error: attempt to divide by zero
```

The recognition of non-associative and non-commutative operators often assists debugging. The associativity of operators is given in Section 8 of the On-line Manual.

1.5.3 Operator overloading

The idea of an *overloaded* operator such as `>` or `<` is that the same symbol may be used for two operations or functions that are internally dissimilar but semantically similar. For example, the internal representation of real numbers and strings on most computers is different and thus the implementation of comparison operators is also quite different; it would perhaps be logical to have two operators *num*< and *string*< to mirror this. However, the behaviour of the two operations appears to the programmer as identical and Miranda supports this semantic congruence by using the single operator symbols `<`, `<=`, `>`, `>=` for both number and string comparison. When an overloaded operator is used incorrectly, the resulting error message reports that it cannot unify the type of the right operand with that of the left. It is not legal to mix numbers, characters and strings:

```
Miranda 3 < "hello"
type error in expression
cannot unify [char] with num

Miranda True >= 0
type error in expression
cannot unify num with bool
```

1.6 Programs, layout and comments

Direct interaction with the Miranda command interpreter may be used for simple desk calculator operations. However, the inability to define names means that even for relatively small programs it is necessary to use the `/e` or `/edit` commands which allow definitions to be edited in the script file and automatically read by the system on exit from the editor. If no filename is given, Miranda assumes that the program file is called *script.m*. Miranda's notion of the current program can be changed by using the `/f` or `/file` command followed by the name of the new file. For example, given a file called "spending_money.m", typing `/f spending_money.m` will cause the system to read this new file from the current directory. The new file is automatically compiled; if it contains syntax errors then Miranda will abandon compilation and not incorporate any of the definitions into the current environment (consequently no previous script file definitions will be available).

1.6.1 Program layout

It is not legal to enter more than one expression at the Miranda prompt. By contrast, it is permitted to have several definitions on one line within a script file, provided that each is terminated by a semicolon. However, this generally results in code that is cramped and difficult to read or to modify, and is not recommended.

Similarly, whilst it is not legal to enter a long expression over more than one line at the Miranda prompt, this is permitted within a script file (with a restriction, known as the *offside rule*, discussed in Chapter 2.1.1). Rather than present a long expression all on one line, it is recommended to layout the program by using spaces, tabs and newlines (as illustrated by the examples in this book).

1.6.2 Program documentation

To enhance readability, all programs in a file should be documented; in Miranda this is achieved by using *comments* or by using a *literate script*.

Comments

Within a Miranda script file, commented text is anything to the right of two parallel bars `||` and before the end of the line (the end of the line terminates the comment). Figure 1.2 provides an example of this style of commenting.

```
|| Program "spending_money.sml".
|| Calculates the amount of holiday spending money.
|| Only deals with three different currencies.

amount_of_sterling = 200.0    || amount of money available

|| different exchange rates
uk_to_us = 1.70
uk_to_dm = 2.93
uk_to_fr = 10.94

|| amount of spending money in different currencies
us_spending_money = amount_of_sterling * uk_to_us
dm_spending_money = amount_of_sterling * uk_to_dm
fr_spending_money = amount_of_sterling * uk_to_fr
```

Figure 1.2 Simple commenting style.

Literate scripts

Miranda recognizes that it is often the case that the comments to a program are more extensive than the actual program code and offers a *literate script* facility—whereby the default text is comment and code must be emphasized. Within a literate script, the *first line* and *all code lines* must commence with a > sign. Additionally, blank lines (that is lines without text) must separate text and code.

Reworking the program in Figure 1.2 as a literate script gives the program in Figure 1.3. Notice that the || comment convention is also allowed.

```
>|| literate script version of above program

Program "spending_money.m".
Calculates the amount of holiday spending money.
Only deals with three different currencies.

> amount_of_sterling = 200.0  || amount of money available

different exchange rates

> uk_to_us = 1.70
> uk_to_dm = 2.93
> uk_to_fr = 10.94

amount of spending money in different currencies

> us_spending_money = amount_of_sterling * uk_to_us
> dm_spending_money = amount_of_sterling * uk_to_dm
> fr_spending_money = amount_of_sterling * uk_to_fr
```

Figure 1.3 A literate script.

1.7 Summary

It is possible to treat the Miranda command interpreter as a simple desk calculator which evaluates expressions entered by the programmer. However, for the purposes of real programming, four additional features have been introduced:

1. The provision of built-in functions; user-defined functions will be introduced in Chapter 2.
2. The classification of data values into different types; the ability to create new types is discussed in Chapter 6.
3. The ability to give names to expressions and thereby use their values in other expressions; Chapter 5 shows how a programmer can specify where a name can and cannot be used.
4. The ability to combine built-in types to form tuples; another important aggregate type is introduced in Chapter 3.

It is worth emphasizing that the classification of data values ensures that they are used correctly and therefore that many common programming errors are detected early in the software development process.

Chapter 2

Functions

Miranda functions may either be built-in (for example `show`), predefined in the Standard Environment (for example `sqrt`) or they may be defined by the programmer. This chapter explains how to define new functions and then use these functions by applying them to values. Functions can be defined in terms of built-in functions (and operators) and/or other user-defined functions. The result of a function application may either be saved (by giving it an identifier) or it may be used immediately in a bigger expression (for example as an argument to another function).

Tuples may be used to package more than one argument to a function. Functions that only operate on the structure of the tuple and not on its components belong to the class of *polymorphic* functions. An example is a function to select the first component of a tuple, regardless of the values of its other components.

Two powerful tools, *pattern matching* and *recursion*, are introduced in this chapter. Pattern matching allows programmers to give alternative definitions for functions for different values of input data. This forces consideration of all inputs to functions and thus eliminates many potential errors at the design stage. Recursion is used to provide iteration and is an essential mechanism for flow control in programs.

Functions are important because they provide a basic component from which to build modular programs. This facilitates the technique of *top-down design* for problem solving; that is, a problem may be broken down into smaller problems which can easily be translated into Miranda functions. Programs structured in this way are easier to read and easier to maintain.

This chapter ends with a slightly larger example, which shows the use of top-down design to solve a problem.

2.1 Simple functions

This section introduces the simplest form of function definition and discusses how functions may be applied to arguments (or "parameters"[1]) and how Miranda evaluates such function applications.

2.1.1 Function definition

In Miranda, functions are defined within the script file. The simplest function definitions have the following template:

function_name parameter_name = *function_body*

Miranda knows that this is a function definition because the left-hand side of the expression consists of more than one name. Both the *parameter_name* and the *function_name* may be any legal identifier as described in Chapter 1 (Section 1.2) of this book (and must similarly start with a lower case character). The *parameter_name* is known as the *formal parameter* to the function; it obtains an actual value when the function is applied. This name is local to the function body and bears no relationship to any other value, function or formal parameter of the same name in the rest of the program. The function body may contain any valid expression, including constants, value identifiers and applications of built-in functions and user-defined functions.

In the following example `twice` is the function name, `x` is the parameter name and `x * 2` is the function body:

```
twice x = x * 2
```

At the Miranda prompt `twice` can be used in three ways:

1. Most importantly, as a function applied to an actual argument, for example:

   ```
   Miranda twice 2
   4
   ```

   This will be discussed further in the next section.
2. In conjunction with the special type indicator keyword `::`

   ```
   Miranda twice ::
   num->num
   ```

   The arrow in the system response indicates that the name `twice` is a function. The type preceding the arrow is known as the *source type* of the function and states the expected type of the argument to which the function will be applied. The type following the arrow is known as the *target type* and states the type of the value returned by the function. This is further illustrated in Figure 2.1 for the function `isodd`.

[1]In the rest of this book the words "argument" and "parameter" will be used interchangeably.

3. As an expression:

   ```
   Miranda twice
   <function>
   ```

   This gives the information that `twice` is a function rather than a value identifier or something not yet defined.

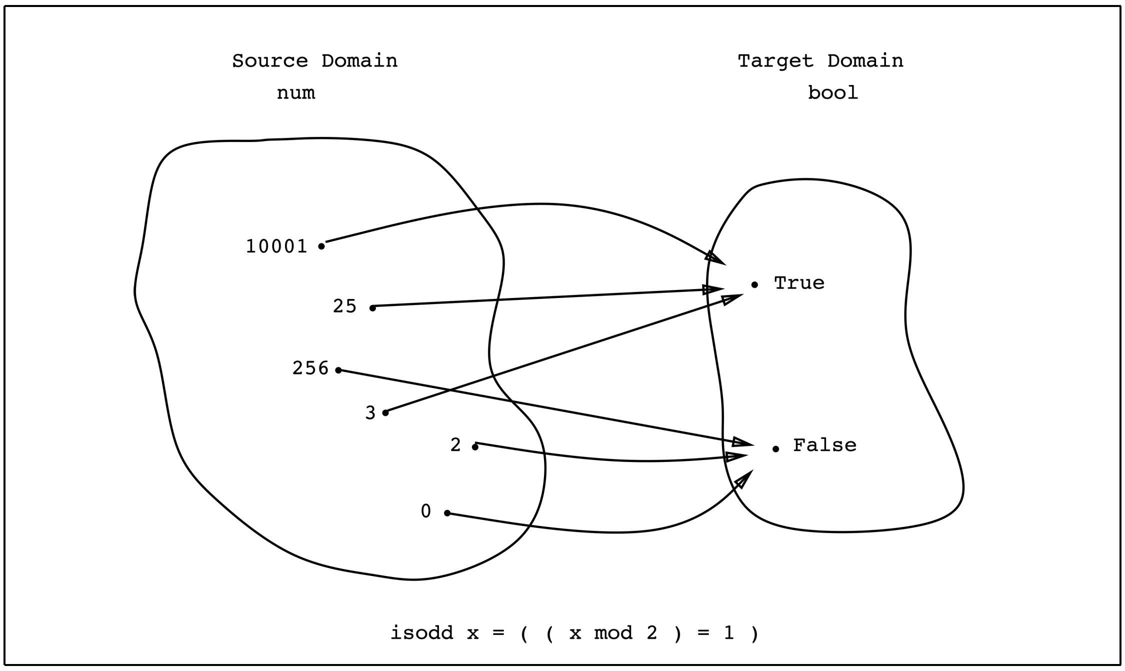

Figure 2.1 Function source domain and target domain.

In general, a function *translates* (or "maps") a value from a source type (or "argument type") to another value from a target type (or "result type"). Though many values from the source type can have the same target value, normally each value from the source type will have just one target result. The source and target types need not be the same. Both of these points are illustrated in the next example, which checks if a character is upper case:

```
isupper c = (c >= 'A') & (c <= 'Z')
```

```
Miranda isupper ::
char->bool
```

The following example makes use of the built-in functions `code` and `decode` to convert a lower case alphabetic into its upper case equivalent; all other characters being left unaltered. The calculation relies upon three facts:

1. Subtracting the ASCII code for lower case 'a' from the ASCII code for a lower case alphabetic character will give a number within the range 0 to 25.
2. Adding the ASCII code for upper case 'A' to a number within the range 0 to 25 will give the ASCII code for an upper case alphabetic character.
3. It is assumed that the argument `c` will be a character in the range 'a' to 'z'.

```
toupper c = decode ((code c) - (code 'a') + (code 'A'))
```

```
Miranda toupper ::
char->char
```

The offside rule

The "offside rule" is a syntactic constraint for actions that cannot be written on one line; its template is of the form:

function parameter = *start of action on first line*
next line of action must NOT be to the left
of the start of the action on the first line

For example:

```
wrong_timestamp (time, message) =
        message ++ " at "
   ++ (show time) ++ " o'clock"

correct_timestamp1 (time, message) =
      message ++
      " at " ++ (show time) ++ " o'clock"

correct_timestamp2 (time, message)
      = message ++ " at " ++
        (show time) ++ " o'clock"
```

2.1.2 Function application

The function `twice` can be applied to an actual value, in exactly the same manner as system-defined functions:

```
Miranda twice 3
6
```

Here, the formal parameter `x` has obtained the actual value `3`.

The integer value returned by the above application may also be used as the actual parameter for another function application. There is no restriction on the number of the times this principle may be employed, as long as the types match. However, it is important to remember that function application associates to the left; for example, the application:

```
abs twice 3
```

is interpreted by Miranda to mean:

```
((abs twice) 3)
```

which is clearly an error because `abs` expects an integer as an argument rather than the name of another function. It is therefore essential to use brackets to give the intended meaning, as shown in the following sample session:

```
Miranda abs (twice (-3))
6

Miranda twice (twice 3)
12

Miranda (sqrt (abs (twice 8)))
4.0
```

An application of a previously user-defined function, such as `twice`, may also appear inside a function body:

```
quad x = twice (twice x)
```

```
Miranda quad ::
num->num

Miranda quad 5
20
```

```
islower c = ~(isupper c)
```

```
Miranda islower ::
char->bool
```

Exercise 2.1

Provide a function to check if a character is alphanumeric, that is lower case, upper case or numeric.

2.1.3 Function evaluation

The example below presents a model of how the application of the function `quad` to an argument may be evaluated by a Miranda system. The example is interesting in that the argument is itself an expression involving an application of a function (the built-in function `sqrt`). Note that the `==>` sign is used to indicate the result of a conceptual evaluation step and has no special meaning in Miranda; this device is used in the rest of the book to indicate a "hand evaluation" of an expression.

```
quad (sqrt (22 + 42))

==> twice (twice (sqrt (22 + 42)))
==> (twice (sqrt (22 + 42))) * 2
==> ((sqrt (22 + 42)) * 2) * 2
==> ((sqrt 64) * 2) * 2
==> (8.0 * 2) * 2
==> 16.0 * 2
==> 32.0
```

The above example illustrates several important points about function evaluation and about evaluation in general:

1. In order to evaluate the function body, effectively a new copy of the function body is created with each occurrence of the formal parameter replaced by a copy of the actual parameter.
2. It is impossible for any function application to affect the value of its actual parameters.
3. The argument to a function is only evaluated when it is required (this is sometimes known as "call-by-name"). Furthermore, if an argument is used more than once inside the function body then it is only evaluated *at most once*—all occurrences of the formal argument name benefit from a single evaluation of the actual argument (this is sometimes known as "call-by-need" or "lazy evaluation").
4. In general, the evaluation of function applications (and indeed evaluation of any expression) may be viewed as a sequence of substitutions and simplifications, until a stage is reached where the expression can no longer be simplified.

It should be noted that in the above discussion there has been no mention of *how* the evaluation mechanism is implemented. Indeed, functional languages offer the considerable advantage that programmers need not pay any attention to the underlying implementation.

2.2 Functions as values

Perhaps surprisingly, a new name may also be given to a function. The name then has the same properties as the function:

```
tw = twice
```

```
Miranda tw ::
num->num

Miranda tw 4
8
```

This shows that functions are not only “black boxes” that translate values to other values, but are values in themselves. They may also be passed as parameters to other functions, as will be shown in Chapter 4.

Thus, entering a function’s name without parameters is the equivalent of entering a constant value. However, because the function has no meaningful value (it would not generally be useful for the system to print out the code definition for the function), the system responds with an indication that the value of a function has been requested:

```
Miranda twice
<function>
```

This principle explains the error message for the following incorrect attempt to apply `twice`:

```
Miranda twice twice 3
type error in expression
cannot unify num->num with num
```

The error arises because the argument to `twice` is a function, which contradicts the expected type of `num`. The above error message highlights the fact that `twice` is a value itself, of type `(num->num)`.

The major difference between a function and other values is that two functions may not be tested for equality—even if they have precisely the same code, or always result in the same value. For example, the expression `(tw = twice)` will give rise to an error. This difference is reflected in that the **show** keyword merely reports that a function is a function rather than echoes its code.

```
Miranda show twice
<function>
```

Redefining functions

It is important to remember that Miranda does not allow identifiers, and hence functions, to be given more than one value within a script file.[2]

2.3 Function parameters and results

Normally, functions translate from a single argument into a single result. However, it is often necessary to translate from more than one input argument or even to have functions that do not require any input value to produce a result. This section shows how Miranda caters for these types of function and also for functions that produce an aggregate result.

2.3.1 Functions with more than one parameter

At first sight, the notion of a function having more than one parameter seems to violate the constraint that all functions are translations from a source type to a target type (and so a function to evaluate the greater of two values would be impossible). Miranda has two ways of coping with this apparent problem—currying and tuples. The currying approach is introduced in Chapter 4, whereas the tuple approach is presented below.

Employing tuples to represent the formal parameter of functions with more than one argument is a natural consequence of the fact that a tuple will have a value drawn from an aggregate type (as discussed in Chapter 1). The formal parameter can either use a single name to represent the entire tuple or the tuple composite form to give names to each of the tuple's values.

The following example shows a simple function that checks the entire parameter and therefore uses a single formal parameter name (`date`) for the tuple:

```
ismybirthday date = (date = (1, "April", 1900))
```

```
Miranda ismybirthday ::
(num,[char],num)->bool
```

The system response makes explicit the fact that the formal parameter `date` has an aggregate type with three components.

In the next example, which embellishes an input message, it is necessary to extract the individual elements of the tuple and so the composite form is used for the function parameter; the corresponding domains are shown in Figure 2.2.

```
timestamp (time, message)
     = message ++ " at " ++ (show time) ++ " o'clock"
```

[2]See, however, Chapter 5 for a discussion of the Miranda rules of scope.

```
Miranda timestamp ::
(num,[char])->[char]

Miranda timestamp (8, "important meeting today")
important meeting today at 8 o'clock
```

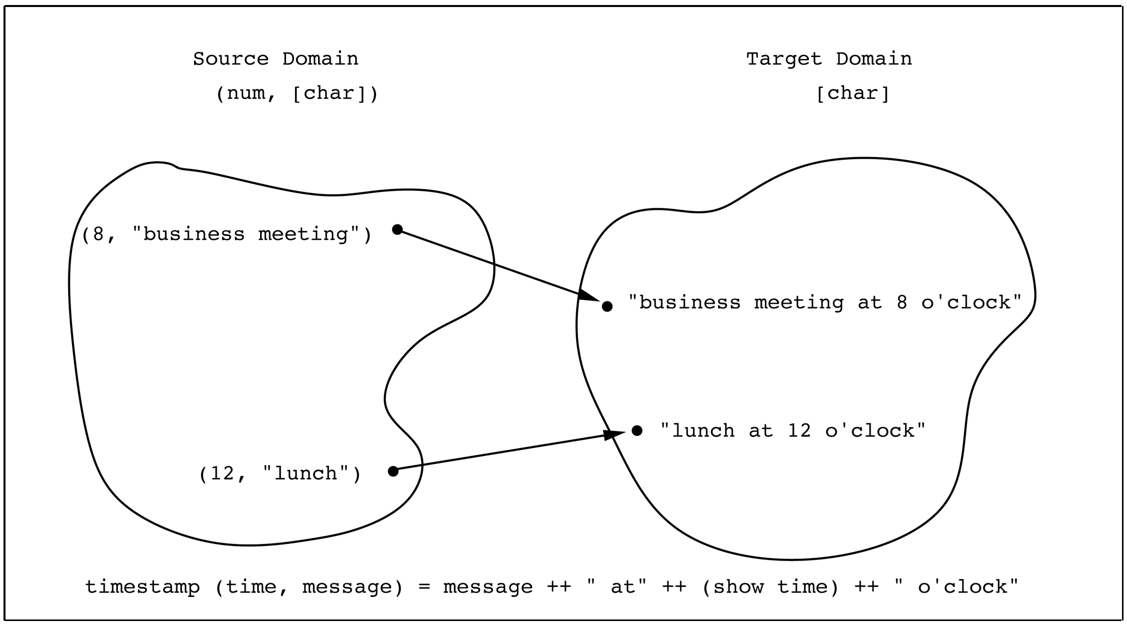
Source Domain
(num, [char])
Target Domain
[char]
(8, "business meeting")
(12, "lunch")
"business meeting at 8 o'clock"
"lunch at 12 o'clock"
timestamp (time, message) = message ++ " at" ++ (show time) ++ " o'clock"

Figure 2.2 Source and target domains for `timestamp`.

2.3.2 Functions with more than one result

It is possible for a function to return more than one result because a tuple may also be the target type of a function. This is demonstrated in the following function `quotrem` (and in Figure 2.3) whose result holds both the quotient and remainder of dividing the source tuple's first component by its second component:

```
quotrem (x, y) = ((x div y), (x mod y))
```

```
Miranda quotrem ::
(num,num)->(num,num)

Miranda quotrem (7, 3)
(2,1)
```

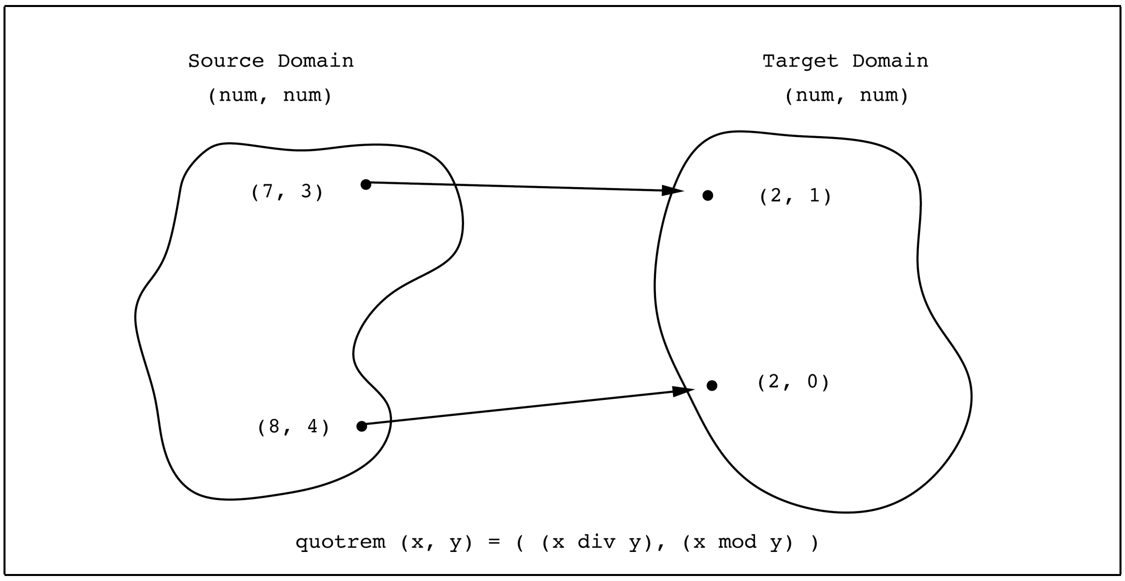

Figure 2.3 Source and target domains for `quotrem`.

2.3.3 Functions without parameters

Sometimes functions do not need any parameters. As such, they have no obvious source type—which unfortunately violates the constraint that all functions are translations from a source type to a target type!

In general, a function can only return differing results if it is applied to differing arguments. Thus, by definition a parameterless function must always return the same result. Normally, it is adequate to cater for such a requirement by means of a constant value definition, as described in Chapter 1. An alternative is to use the explicit "null" parameter, `()`, which is the only value of the special empty tuple type:

```
message () = "Buy us several drinks"
```

2.4 Polymorphic functions

It can be useful to define functions that provide general operations on tuples regardless of their component types. For example, the functions `myfst` and `mysnd`, respectively, extract the first item and second item from a pair:

```
myfst (x,y) = x
mysnd (x,y) = y
```

```
Miranda myfst ::
(*,**)->*

Miranda mysnd ::
(*,**)->**
```

These two functions mimic the action of the built-in functions `fst` and `snd`, which are defined in exactly the same way (see Section 28 of the on-line manual).[3]

The type of `myfst` is different to previously defined functions in that it takes a pair of values of *any* two types and returns a value of the first type. It does not operate on the elements in the pair but on the shape or construction of the pair itself; therefore `myfst` does not care about the type of `x` nor about the type of `y`. This is indicated by the system response which gives the general purpose type names `*` and `**`, which are known as *polytypes*. It should be noted that the actual values of the tuple could be of different types, which is indicated by the two different polytype names (a single star and a double star). However, because `*` and `**` can stand for *any* type, the actual components could also be of the same type.

```
Miranda myfst (3,4)
3

Miranda myfst (3,("a",True))
3

Miranda mysnd (3,("a",True))
("a",True)
```

The above are examples of *polymorphic* functions. In general, a function is said to be polymorphic in the parts of its input which it does not evaluate.[4] Thus, in the above example `myfst` is polymorphic in both `x` and `y`, whereas in the following example `g` is only polymorphic in `x`:

```
g (x,y) = (-y,x)
```

```
Miranda g ::
(*,num)->(num,*)
```

Polymorphic functions need not take a tuple argument. For example, both of the following functions are polymorphic in their argument `x`:

[3]Remember that it is not possible to reuse names which are already defined in the Standard Environment. However, since it is recommended that the reader tries out these example definitions for themselves, this book adopts the practice of prefixing the names of Standard Environment functions with the letters `my`.

[4]There is, in theory, no restriction on the number of polytypes in the tuple parameter to a function.

```
id x = x
three x = 3
```

```
Miranda id ::
*->*

Miranda three ::
*->num
```

Exercise 2.2

What happens in the following application and why?

```
myfst (3, 4 div 0)
```

Exercise 2.3

Define a function `dup` which takes a single element of any type and returns a tuple with the element duplicated.

2.5 Pattern matching

One of the more powerful features of Miranda is *pattern matching*, which allows the format of one value to be compared with a template format (known as a "pattern"). For example, the appearance of a tuple composite form as the formal parameter to a function is an instance of pattern matching. Pattern matching is used both in constant definitions and in function definitions; the former has already been introduced in Section 1.4. In this section, the principle of pattern matching is extended to enable alternative definitions for a function depending on the format of the actual parameter. This facility has several advantages:

1. As a means of taking different actions according to the actual input.
2. As a design aid to help the programmer to consider all possible inputs to a function.
3. As an aid to the deconstruction of aggregate types such as tuples, lists and user-defined types (the last two will be introduced in Chapters 3 and 6, respectively).

In most cases, it will be seen that the use of pattern matching reduces the risk of programming errors and enhances program readability.

2.5.1 Alternative patterns

Pattern matching template

The general template for achieving pattern matching in functions is:

```
function_name   pattern_1   =   function_body_1
function_name   pattern_2   =   function_body_2
      .             .                 .
      .             .                 .
function_name   pattern_N   =   function_body_N
```

When the function is applied to an argument, Miranda sequentially scans the definitions, from the topmost definition, until the actual parameter matches one of the patterns. The associated function body is then evaluated. At its simplest, a function will have only one pattern. Since a formal parameter is an instance of a pattern, all the function definitions so far presented are also examples of this form.

The following implementation of the function `not` (the equivalent of the built-in function `~` — which inverts the truth value of its parameter) demonstrates a simple use of pattern matching:

```
not True = False
not False = True
```

Similarly:

```
solomonGrundy "Monday"  = "Born"
solomonGrundy "Sunday"  = "Buried"
solomonGrundy anyday    = "Did something else"
```

In this example, the `anyday` pattern will match any other input and will therefore act as a default case. Notice that this default must be the bottommost definition (see Section 2.5.2.).

Exercise 2.4

Modify the function `solomonGrundy` so that `Thursday` and `Friday` may be treated with special significance.

2.5.2 Legal patterns

Patterns may consist of constants (for example, integers but not fractional numbers, or the Boolean values `True` and `False`), tuples and formal parameter names.

Patterns containing arithmetic, relational or logical expressions are generally not allowed; the following are both wrong:

```
wrong (x + y) = "silly"
also_wrong ("a" ++ anystring) = "starts with a"
```

However, it is legitimate to have patterns of the form `(n + k)`, where `n` will evaluate to a non-negative integer value and `k` is a non-negative integer constant. Hence the following is legitimate, for all actual values of `n` greater than zero:

```
decrement (n + 1) = n
```

Unfortunately, this facility does *not* extend to the other operators or to values of `n` and `k` that are negative or fractional.

Duplicates

Unlike some other languages that have pattern matching, Miranda permits both constant and formal parameters to be replicated:

```
both_equal (x,x) = True
both_equal (x,y) = False

both_zero (0,0) = True
both_zero (x,y) = False
```

The occurrence of duplicated identifiers in a pattern instructs Miranda to perform an equality check. Thus, in the above example, `both_equal` will only return `True` if both the first and second components of its two-tuple argument are the same (are equal to each other), and `both_zero` will only return `True` if both components are equal to `0`.

The equality check is *only* performed if an identifier is duplicated in a single pattern: duplication of identifiers "vertically" across several patterns is commonplace and has no special meaning. Thus, in the following examples, the multiple occurrences of the identifier `x` do *not* cause an equality check to be performed:

```
step_up (0,x) = x
step_up (1,x) = x+1

plus (0,x) = x
plus (x,0) = x
plus (x,y) = x+y

twenty_four_hour (x,"a.m.") = x
twenty_four_hour (x,"p.m.") = x + 12

equals_ten (x, 10) = True
equals_ten (x, y) = False
```

Note that this cannot be used to compare functions for equality:

```
Miranda both_equal (sqrt, sqrt)
program error: attempt to compare functions
```

Non-exhaustive patterns

In the `not` function (shown in Section 2.5.1), both possible values for the Boolean type were checked. If the programmer has not considered all the possible input options then Miranda will still permit the script file but may generate a run-time error.

```
not False = True
```

```
Miranda not False
True

Miranda not True
program error: missing case in definition of not
```

Although programmers may be tempted to ignore patterns that they believe will never arise, it is generally sensible to trap unexpected cases with an appropriate “default” pattern. For example, if all of the days of the week were catered for as patterns in the `solomonGrundy` function, it might still be desirable to have a final pattern that dealt with any string which did not represent a day of the week.[5]

[5] See Section 2.9 for how this pattern may be dealt with by the Miranda **error** function.

Order of evaluation

It must be stressed that Miranda checks each alternative sequentially from the top pattern to the bottom pattern, ceasing evaluation at the first successful match. If the patterns are mutually exclusive then it does not matter in which order they are defined. However, patterns may overlap and then the order of definition is *vital*, as is now shown in the following script file:

```
wrong_or x = True
wrong_or (False,False) = False
```

On exit from the editor Miranda will report:

```
syntax error: unreachable case in defn of "wrong_or"
error found near line 2 of file "script.m"
compilation abandoned
```

It is strongly advised that patterns should be mutually exclusive wherever possible. However, the guideline for designing functions with overlapping alternative patterns is to arrange the patterns such that the more specific cases appear first. In the above example `(False,False)` was less general than `x` and should have been the first pattern to be matched against.

Lazy evaluation of function arguments

Finally, a word of caution regarding the "lazy evaluation" of function arguments, as discussed in Section 2.1. Miranda uses a "call-by-need" style of argument evaluation, so that in the following program a divide-by-zero error is avoided:

```
myfst (x,y) = x

result = myfst (34, (23 / 0))
```

```
Miranda result
34
```

However, if `myfst` were to use pattern matching with constants, then the divide-by-zero error is *not* avoided:

```
myfst (x,0) = x
myfst (x,y) = x

result = myfst (34, (23 / 0))
```

```
Miranda result

program error: attempt to divide by zero
```

The error occurs because the pattern matching on the second component of the tuple (to see if it is equal to zero) forces Miranda to evaluate the division, so that it can decide whether to use the first or the second function body.

2.5.3 Guards

This section demonstrates how pattern matching may be made more selective and powerful by using the keywords **if** and **otherwise.**

if

In the example of `toupper` in Section 2.1.1 there was no check to see if the character to be converted was within the expected range. In the following example, use of Miranda's "guard" mechanism demonstrates how patterns may be constrained:

```
safe_toupper c
        = decode ((code c) - (code 'a') + (code 'A')),
          if (c >= 'a') & (c <= 'z')
safe_toupper c = c
```

Here, the first action is only executed when the guard condition (which follows the comma token (**,**) and the keyword **if**) evaluates to `True`.[6] In general, the guard can be any expression that evaluates to a Boolean result and more than one guard is permitted. For example:

```
whichsign n  = "Positive", if n > 0
             = "Zero", if n = 0
             = "Negative", if n < 0
```

otherwise

The use of the keyword **otherwise** demonstrates another way of trapping the default case. The action associated with it is only taken when the pattern does not satisfy any guard. The function `safe_toupper` could have been written:

[6]In fact the **if** can be omitted but it probably makes the code easier to read if it is included. To ensure good documentation, Miranda has an optional `/strictif` directive that actually enforces the use of the **if** keyword. The comma is mandatory in either case.

```
safe_toupper c
        = decode ((code c) - (code 'a') + (code 'A')),
          if (c >= 'a') & (c <= 'z')
        = c, otherwise
```

Note that this keyword can only appear as the default case to the last body for any given pattern.

Alternative and guarded patterns

There is no restriction on the combination of alternative patterns and guarded patterns. For example:

```
check_equiv (0,x)  = "Equivalent", if x="Zero"
                   = "Not equivalent", otherwise
check_equiv (1,x)  = "Equivalent", if x="One"
                   = "Not equivalent", otherwise
check_equiv (n,x)  = "Out of range of this check"
```

2.6 Type information

This section shows how Miranda can be helped to distinguish the type of formal parameters and introduces a mechanism for interaction with the Miranda type system.

2.6.1 Type declarations

For all the functions defined so far, the Miranda system has been able to work out the types of the function parameters from their context. However, this is not always the case. For example, the keyword **show** is overloaded and, on leaving the editor, it is not always possible for Miranda to determine the implicit type of a function that uses **show** in its function body. Thus, Miranda will raise a type error if given a script file containing the following attempted definition:

```
wrong_echo x = (show x) ++ (show x)
```

The solution is to employ the type indicator **::** which not only *reports* the type of a function or value identifier when used with the Miranda command interpreter, but can also be used to *declare* the intended type of a function or identifier inside a script file. The general template for its use is:

function_name :: parameter_type `->` *target_type*

The actual code can then be given. Hence `wrong_echo` can be corrected as shown in the function `right_echo`:

```
right_echo ::  num -> [char]
right_echo x = (show x) ++ (show x)
```

In addition to resolving any possible type ambiguity, the use of `::` is recommended for three other important reasons:

1. As a design aid, since the program designer is forced to consider the nature of the input and output before implementing any algorithm.
2. As a documentation aid; any programmer can immediately see the source and target types of each function.
3. As a debugging aid, since Miranda will indicate differences between inferred types and declared types.

Whilst some of the examples in the rest of this book may not contain explicit type constraints (for pedagogic reasons of simplicity and clarity), the approach is encouraged as being a good programming discipline and all the extended examples will adopt this style.

2.6.2 Polymorphic type constraint

A polytype may also be included as a type constraint. The following functions `third_same` and `third_any` are two versions of the same function which exemplify this property. In the first version, the use of the same polytypes constrains the actual parameters to be of the same type. The second version uses different polytypes to permit actual parameters to be of different types.
Same type version:

```
third_same ::  (*,*,*) -> *
third_same (x,y,z) = z
```

```
Miranda third_same (1,2,3)
3

Miranda third_same (1,2,'3')
type error in expression
cannot unify (num,num,char) with (num,num,num)
```

Any type version:

```
third_any ::  (*,**,***) -> ***
third_any (x,y,z) = z
```

```
Miranda third_any (1,2,3)
3

Miranda third_any (1,2,'3')
'3'
```

There is no way to constrain the parameters to have *different* polytypes.

Exercise 2.5

Define a function `nummax` which takes a number pair and returns the greater of its two components.

2.6.3 Type synonyms

As a further aid to program documentation, Miranda allows a type to be given a name, which may be any legal identifier as described in Chapter 1 (and which must also start with a lower case character). For example the tuple `(13, "March", 1066)` has the type `(num,[char],num)`. For clarity this type could be given the name `date` by using the `==` facility as follows:

```
date == (num, [char], num)
```

Unfortunately, if a value is constrained with the type name `date` then Miranda will respond with the underlying types rather than the new type name. Thus, though type synonyms are very useful for shortening type expressions in script files, they do not help to make type error messages clearer:

```
mybirthday ::  date
mybirthday = (1, "April", 1900)
```

```
Miranda mybirthday ::
(num,[char],num)
```

For all purposes other than documentation of the script file, the value `mybirthday` will be treated as a `(num,[char],num)` aggregate type. Using the `==` notation *does not introduce a new type.* Values of type `date` and type `(num,[char],num)` can be compared and otherwise mixed, *unlike* the strong type discipline imposed, for instance, between `nums` and `chars`. Thus, the following is legal:

```
abirthday = (1, "April", 1900)
```

```
Miranda mybirthday = abirthday
True
```

Polymorphic type synonyms

It is also legal to have type synonyms that are entirely or partially comprised of polytypes. If a polytype appears on the right-hand side of a declaration then it must also appear on the left-hand side, between the == token and the new type name. For example, `sametype_triple` can be declared to be a triple of any type such as numbers or characters by defining it as a polytype:

```
sametype_triple * == (*,*,*)
```

An alternative and more general type synonym would be:

```
triple * ** *** == (***,**,*)
```

This is more general than the first version because it does not require all the elements of the three-tuple to have the same type.

Polymorphic type synonyms can be used to declare the types of functions in much the same way as other type synonyms. The only constraint is that it is necessary to indicate whether the type of the function is intended to be polymorphic or have a specific monomorphic type:

```
revNumTriple ::  (sametype_triple num)
                    -> (sametype_triple num)
revNumTriple (x,y,z) = (z,y,x)
```

or

```
revAnyTriple ::  triple * ** *** -> triple *** ** *
revAnyTriple (x,y,z) = (z,y,x)
```

2.7 Simple recursive functions

Miranda, like other functional programming languages, uses *recursion* as its main iteration control structure; this general mechanism can achieve the same effect as the imperative language features such as “while”, “repeat” and “for” loops.[7] A recursive function definition is one where the name of the function being defined appears inside the function body. When the function is applied to an argument, an appearance of the function name in the function body causes a new copy of the function to be generated and then applied to a new argument. If the function name appears as part of an expression then the evaluation of the expression is suspended until the recursive function application returns a value.

Recursion is a very powerful and general mechanism which must be used carefully. At its simplest, it can be used to generate an infinite number of applications of the same function, for example:

[7] See also Chapter 4.

```
loop_forever () = loop_forever ()
```

If this function is ever evaluated, it will loop for ever, calling itself again and again and achieving nothing. The function `loop_forever` could also be defined to take a non-empty argument, but is equally useless:

```
loop_forever x = loop_forever x
```

```
Miranda loop_forever 2
BLACK HOLE
```

Evaluating this function by hand shows that whenever `loop_forever` is applied to the message `"BUY US A DRINK"` it immediately invokes another *copy* of the function `loop_forever` and applies it to another *copy* of the message.

```
loop_forever "BUY US A DRINK"
==> loop_forever "BUY US A DRINK"
==> loop_forever "BUY US A DRINK"
==> loop_forever "BUY US A DRINK"
....
```

To create a more useful recursive function definition, it is necessary to ensure that there is some *terminating condition* for the function and also that the recursive calls are successively applied to arguments that *converge* towards the terminating condition.

A number of recursive styles can be identified which achieve the above criteria, the rest of this section shows the most common simple recursive styles: *stack* and *accumulative* recursion.

2.7.1 Stack recursive functions

The following function `printdots` is an example of the recursive style known as *stack* recursion. This function prints the number of dots indicated by its parameter. Firstly, there is a terminating pattern `0`. Secondly, each recursive application of `printdots` is to a *different* `n`. Thirdly, each successive `n` decreases towards the terminating pattern `0`.

```
printdots ::  num -> [char]
printdots 0 = ""
printdots n = "." ++ (printdots (n - 1))
```

A hand evaluation reveals:

```
printdots 3
==> "." ++  (printdots (3 - 1))
==> "." ++  (printdots 2)
==> "." ++  ("." ++ (printdots (2 - 1)))
==> "." ++  ("." ++ (printdots 1))
==> "." ++  ("." ++ ("." ++ (printdots (1 - 1))))
==> "." ++  ("." ++ ("." ++ (printdots 0)))
==> "." ++  ("." ++ ("." ++ ("")))
==> "." ++  ("." ++ ("." ++ ""))
==> "." ++  ("." ++ ".")
==> "." ++  ("..")
==> "..."
```

This hand evaluation illustrates the appropriateness of the name *stack recursion*; the arguments of all the calculations being *stacked* (rather like a stack of playing cards) until the terminating pattern is met.

Exercise 2.6

Define a recursive function to add up all the integers from 1 to a given upper limit.

2.7.2 Accumulative recursive functions

An alternative style of recursive function is now shown. The following example `plus` exploits the fact that two positive numbers can be added, by successively incrementing one of them and decrementing the other until it reaches zero.

```
plus ::  (num,num) -> num
plus (x,0) = x
plus (x,y) = plus (x + 1, y - 1)
```

Hand evaluating the application of `plus` to the tuple `(2,3)` gives:

```
plus (2, 3)
==> plus (2 + 1, 3 - 1)
==> plus (2 + 1, 2)
==> plus (2 + 1 + 1, 2 - 1)
==> plus (2 + 1 + 1, 1)
==> plus (2 + 1 + 1 + 1, 1 - 1)
==> plus (2 + 1 + 1 + 1, 0)
==> 2 + 1 + 1 + 1
==> 5
```

This function also meets the fundamental requirements of having a terminating condition and a convergent action. The terminating condition is met in the first pattern, which if matched halts the recursion and gives the result. The convergent action is satisfied by the body of the second pattern which will eventually decrement `y` towards the first pattern's terminating condition. It can be seen that the name *accumulative* is appropriate in that one of the component arguments is used as an *accumulator* to gather the final result.

Exercise 2.7

Write `printdots` in an accumulative recursive style. This will require more than one function.

Exercise 2.8

Write the function `plus` in a stack recursive style.

2.8 Who needs assignment?

Imperative programming languages use a memory-store model with "variables" whose names refer to memory locations. Central to the imperative style is the action of changing the values held in these variables through a process called "assignment". For example, in the C++ language the statement `int x;` gives the variable name `x` to a previously unused memory location which is big enough to store an integer. The value stored at this location might initially be set to `3` using the assignment statement `x = 3;` and subsequently overwritten with a new value `25` using the assignment statement `x = 25;`. Multiple assignment (reassignment) of values to the same memory location is vitally important to the imperative style, yet this mechanism seems absent in the functional style.

In Miranda there is *no assignment operator*. It is not possible for a programmer to make an explicit change to the value held at a given storage location and it is not possible for a function to change the value of its argument. To those who have some experience of programming in an imperative language, this may come as a shock.

In order to understand why assignment is not necessary in a functional language, it is worthwhile analysing its rôle in imperative programs:

1. To store input and output values.
   A memory location is set aside to receive incoming data; the program reads in that data and then uses assignment to save the data in the appropriate memory location.
2. To control iteration.
   A memory location is set aside to hold an integer which counts how many

times an iteration construct has looped. Each time around the loop, an assignment statement is used to increment or decrement the value held in the "counter" memory location.

3. To store the results of intermediate calculations.
   Often it is necessary to remember an intermediate value whilst the program does some other calculation; the intermediate result can then be used later. This is usually achieved by assigning the intermediate value to a memory location.
4. To store a history of intermediate results (known as "state" information).
   This is much the same as the previous rôle except that a collection of different memory locations may be required. This involves some additional memory management (using direct addressing of memory) to ensure that each value is stored (using assignment) in the appropriate location, possibly overwriting the existing value at that location. The collection of memory locations is often known as the "state" of the computation.

All of the above imply the need to consider the nature of data storage. In functional languages, it is not necessary to think in terms of data storage locations and so it is correspondingly unnecessary to think in terms of the alteration of the contents of data storage locations. The rôle taken by assignment is either provided transparently by the Miranda system or is a natural consequence of the recursive style of programming:

1. Input to a program appears as the actual parameter to its topmost function and output from the program is the result of the topmost function application. The allocation of storage locations to hold these values is automatically managed by the system and the output of the topmost function application is printed to the screen without the programmer having to specify this action.
2. Iteration control is provided by recursion; the number of iterative steps being controlled by the value of the initial argument.
3. The results of intermediate calculations need not be stored (although a result can be "remembered" as a definition for later use). The automatic storage allocation mechanism of Miranda hides store allocation and deallocation from the programmer, as seen in the following comparison of the imperative and functional treatments of the function `swap`.
   An ANSI C definition of `swap` is:

```
void  swap(int *xp, int *yp) {
      int t;

      t = *xp;
      *xp = *yp;
      *yp = t;
   }
```

An alternative imperative approach is to pass the two integers as an aggregate type, for example:

```
typedef  struct {
         int x;
         int y;
         } PAIR;

PAIR swap (PAIR arg){
         PAIR result;

         result.x = arg.y;
         result.y = arg.x;
         return result;
         }
```

For both imperative approaches it is necessary to assign to a specific storage variable—either to hold a temporary (intermediate) value or to hold the result. By contrast, the Miranda approach is much simpler:

```
swap (x,y) = (y,x)
```

4. The storage of state information is also dealt with automatically by a functional language implementation. For example, the accumulative recursive style "remembers" the changing state of values from one recursive application to the next by means of the "accumulative" parameter.

Program control

It is worth noting that the three classic program control structures of imperative programming languages are also available to the functional programmer:

1. Iteration: by means of recursion.
2. Selection: by means of pattern matching.
3. Sequencing: by means of the nesting of function applications.

2.9 Error messages

Some functions may not have a sensible result for all of their possible argument values. For example, if the built-in function `div` is applied with a zero denominator then it will terminate with the error message:

```
program error: attempt to divide by zero
```

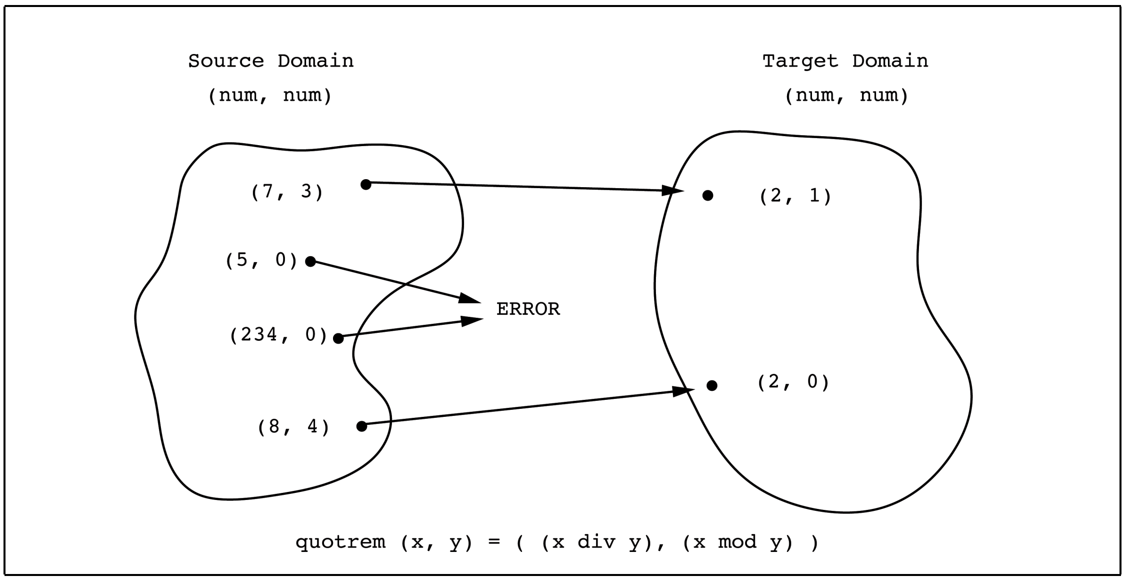

Figure 2.4 Example of a partial function.

A function which has no defined result for one or more of its input values is known as a *partial function*, as illustrated in Figure 2.4. It is often the case that a programmer will need to define a partial function. For example, the function `printdots` (given in Section 2.7.1) should give an error if applied to a value of `n` which is less than zero. Unfortunately, in the current definition the parameter of recursion, `n`, will never converge to its terminating condition if it is initially negative. There are three options available to the Miranda programmer to handle this situation:

1. Impose a dummy answer. A negative number of dots is clearly meaningless, but could be dealt with by indicating that negative values generate no dots.
2. Ignore it and let the Miranda system recurse for ever or generate some not very helpful error message, such as:

   ```
   <<not enough heap space -- task abandoned>>
   ```

   Unfortunately this will not necessarily pinpoint which function has failed inside a complex expression or program.
3. Force the Miranda system to give a more meaningful message, by stating which function has failed and why it has failed.

This last option can be achieved using the built-in function `error` as follows:

```
string == [char]

printdots ::  num -> string
printdots 0 = ""
printdots n
    = "." ++ (printdots (n - 1)), if n > 1
    = error "printdots:  negative input", otherwise
```

```
Miranda printdots (-1)
program error: printdots - negative input
```

The function `error` is polymorphic in its return type. It can take any input string and will evaluate to the return type of the function where it appears. As a *side-effect*, its parameter string is printed as a user-provided diagnostic. No further evaluation can occur after `error` is applied.

Exercise 2.9

Write a function called `int_divide` which divides one whole number by another; the function should not use any arithmetic operators except for subtraction, addition and unary minus.

2.10 Programming with functions

This section shows the design of a small program to convert an integer to a string value. This mirrors the function `shownum` for integers, though it must be noted that due to Miranda's strong type system it is not possible for the user to write a **show** program to convert all types. The design of this program is presented in a *top-down* manner. This technique is known as top-down design because each step, except the final step, assumes the existence of functions defined at a later step.[8]

Specification

The program will take an integer as its parameter and will evaluate to a string. The string will be a denary representation of the integer; for a negative integer the string will start with a minus sign character.

[8]Chapter 5 shows a way to more closely associate the functions within a program.

Design

Firstly, the sign of the input integer is determined and dealt with. Secondly, a function to process positive integers is developed. This breaks its input into two parts: an integer less than 10 (which can be represented as a single character), together with an integer greater than 10 which requires exactly the same processing.

Implementation

Step 1: Converting any integer to a string

Assuming that a positive number can be represented as a string, then a negative number can also be represented by taking its absolute value (using `abs`) and preceding the string representation by a `-` character. This description converts directly to the Miranda code for a top-level function `int_to_string` which will translate an integer value to a string value:

```
string == [char]

int_to_string ::  num -> string
int_to_string n = "-" ++ pos_to_string (abs n), if n < 0
                 = pos_to_string n, otherwise
```

Step 2: Converting a positive integer to a string

The next stage of the design is to define `pos_to_string` to convert a positive integer to a string. Since only integers less than 10 have a single character ASCII representation, the integer to be converted must either be less than 10 (the terminating condition of the function) or requires further processing.

The integer must be split into two parts: the least significant digit (which can be obtained using **mod**), and the integer without its least significant digit (which can be obtained using **div**). At each step, the least significant digit must be converted to a single character string using the function `int_to_char`. The rest of the integer forms the parameter to another application of `pos_to_string`. At each successive application, the parameter will reduce and eventually converge to a number less than 10.

Because individual digits are discovered in the reverse order to that which they must appear, it is necessary to concatenate them on the right of any recursive application of `pos_to_string`.

This description leads directly to the following Miranda code:

```
pos_to_string ::  num -> string

pos_to_string n
          = int_to_char n, if n < 10
          = pos_to_string (n div 10)
            ++ (int_to_char (n mod 10)), otherwise
```

Step 3: Converting an integer less than 10 to a string

The code for this function is based on the same rationale as for the function `toupper` shown in Section 2.1.1 and should be self-explanatory:

```
int_to_char ::  num -> string

int_to_char n = show (decode (n + code '0'))
```

Note that `int_to_char` is presented as a separate function because it is concerned with type *conversion*, whilst `pos_to_string` has an *iterative* purpose.

Putting it all together

In Miranda it does not matter in which sequence the definitions are written. However, it is good style to present the functions in the order of their top-down design.

One function per function

It must be stressed that each function in the above program has a single purpose. This recommended method allows *easier testing* and *easier modification* of individual parts of the program.

2.11 Summary

This chapter has discussed the definition and use of functions. Functions translate a value from a source type to a value in a target type, with *tuples* being used to represent composite domains. An alternative style that does not need tuples is discussed in Chapter 4.

There are many choices available to the designer of functions, including the following:

1. The types of the function parameters—this includes the choice of which types

may be used, whether they are *monomorphic* or *polymorphic* and whether the function is defined for all input values.
Functions may either be monomorphic, which means that their definition absolutely specifies the required types of the input data, or they may be polymorphic, whereby the actual types of the input data may be different for different applications of the function. Polymorphic function definitions are an important extension to the strong type system introduced in Chapter 1.
Functions that are only defined for some values of their input parameter are known as *partial functions*; if they are applied to an argument for which they have no definition, the programmer may signify that the program should terminate immediately with an error message.

2. The use of *recursion* as a general-purpose control structure.
   The mechanism of recursion has been shown to provide the iterative control structures of imperative programming languages (such as "for loops" and "while loops"). Recursive function definition is of fundamental importance to programming in Miranda and in this chapter the two most common forms of recursion (*stack* and *accumulative*) have been introduced. The nature of recursive definitions is further explored in Chapters 3 and 4.
3. The use of the *pattern matching* style of definition.
   The technique of pattern matching may be used when defining functions; this has many advantages, including the introduction of a selection mechanism and providing a way to ensure that a programmer gives consideration to all of the possible inputs to a function.