one and the same value. For example, the above expressions are just four of infinitely many possible representations for the abstract integer value 30.

Expressions in a functional language may contain names which stand for unknown quantities, but, as in mathematics, different occurrences of the same name refer to the same unknown quantity, for example x in `double(`x`) + `$\frac{x}{2}$. Such names are usually called *variables*.

2.8 Summary

- A functional program consists of a collection of function definitions. To run a program one presents the evaluator with an expression and it will evaluate it. This expression can contain references to functions defined in the program, as well as to built-in functions and constant values.
- Functions are defined in a notation which is very close to mathematical notation.
- *Functional composition* (that is, passing the output of one function as an argument to another function) is used to define more complex functions in terms of simpler ones.
- Evaluation of an expression is by *reduction*, meaning simplification. The expression is repeatedly simplified until it has no more functions left to be applied.
- The meaning of an expression is the value which it *represents*. This value cannot be changed by any part of the computation. Evaluating an expression only alters its form, never its value.

2.9 Exercises

1. Define a function `hypotenuse` which, given the lengths of the two shorter sides of a right-angled triangle, returns the length of the third side. (There is a built-in function `sqrt` that calculates square roots.)
2. Write a function `addDigit` of two arguments which will concatenate a single-digit integer onto the right-hand end of an arbitrary-sized integer. For instance, `addDigit 123 4` should give `1234`. Ignore the possibility of the number becoming too large (called integer overflow).
3. Define a function `celsius_to_fahr` that converts celsius temperatures to Fahrenheit. Show for any value of `temp` that the following hold:

```
celsius_to_fahr(fahr_to_celsius temp) = temp
fahr_to_celsius(celsius_to_fahr temp) = temp
```

Chapter 3

Specifications

A conscientious programmer wants the customer to be entirely satisfied with the program, so their aims are the same overall: they both want to see a satisfactory and useful product at the end. None the less, there are certain tensions between the programmer's wish for an easy task, and the customer's desire for a powerful and comprehensive ('All singing, all dancing') program. This may boil down to money. A more powerful program will cost more to produce, the customer must balance his needs against his budget and the programmer must be able to make plain the difference between the more and the less powerful specifications.

In this vague sense, the specification represents part of a contract between programmer and customer. The full contract says 'Programmer will implement software to this specification, and customer will pay such-and-such amount of money.' However, there is also a sense in which *the specification itself* represents a contract.

3.1 Specification as contract

PUNTER (the customer) and HACKER (the programmer) have done business together before, and usually find they understand each other.

<u>Act 1</u>
PUNTER: Can you write me a program to calculate the square root of a real number?
HACKER: Can I assume it is non-negative?
PUNTER: Yes.
HACKER: OK, I can do that.
[*Shake hands and exeunt*]

They now have a gentlemen's agreement that HACKER will write a square root program: this is an oral contract between PUNTER as software purchaser, and HACKER as software producer. But there is also a more subtle contract involved, between PUNTER as software user, and HACKER 'as software'. This says that if PUNTER uses the program, then, *provided that* the input is a non-negative number, the output will be a square root of it. This is a contract because it embodies some interlocking rights and obligations governing the way in which the program is to be used.

First, the input must be non-negative. This is an obligation on PUNTER, but a right for HACKER, who is entitled to expect, for the sake of his implementation, that the input is non-negative. But, then, he is obliged to calculate a square root, and PUNTER has the right to expect that this is what the output will be.

A specification such as this can be divided into the following two parts:

- The *pre-condition* is the condition on the input that the user guarantees: for example, the input is a non-negative real number.
- The *post-condition* is the condition on the output that the programmer guarantees: for example, the output is the square root of the input.

There is a certain asymmetry here (which is due to the fact that the input comes first). The pre-condition can only refer to the input, whereas the post-condition will probably refer to both the input and the output.

Note also the underlying tension: the customer would like weak pre-conditions (so the program works for very general input) and strong post-conditions (so it computes many, or very precise, answers). The programmer, on the other hand, would like the reverse.

Therefore, there must be some kind of dialogue in order to agree on the terms of the contract. Figure 3.1 shows these tensions with springs. The programmer would have the pre- and post-conditions the same, then he doesn't have to do anything at all. The customer's spring pulls the conditions apart.

The strength of the springs depends on various factors. For instance, if the customer is prepared to pay a lot of money for some software, or if the software is a procedure that is called very often, then it is worth putting a lot of work into programming it — 'the customer's spring is powerful'.

3.2 Formalizing specifications

Let us introduce a very specific format for writing such specifications as part of a Miranda program. It has three parts, namely typing information and the pre- and post-conditions.

The *typing information* gives the types of the input (argument) and output (result), and also the name of the program (function), and there is a standard Miranda notation for this. In many programming languages this is an essential part of the program definition, required by the compiler. In Miranda, it is

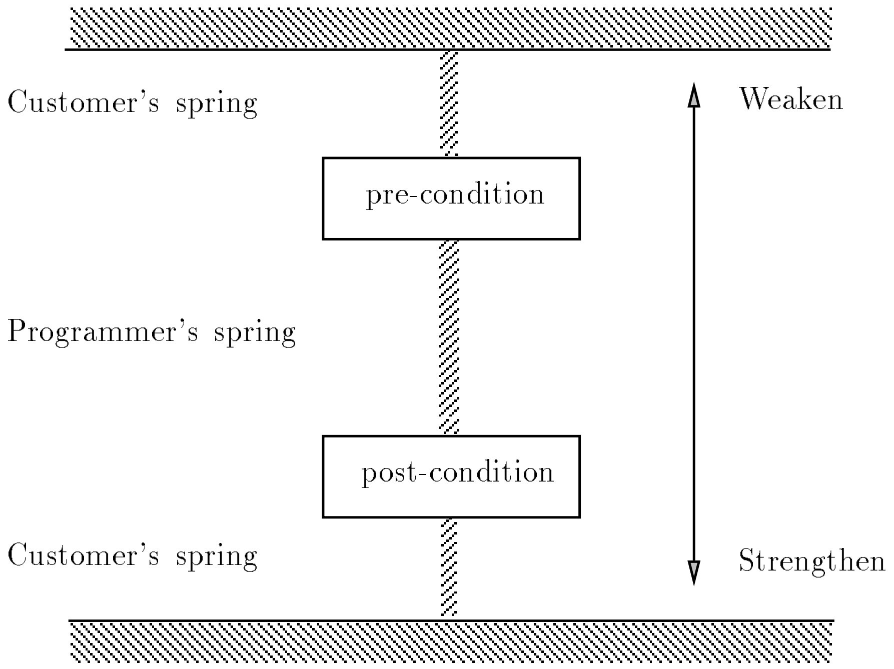

Figure 3.1

optional because the compiler can deduce the types from the rest of the definition. However, they may not be the intended types so type declarations should always be included in all programs.

The *pre- and post-conditions* are written using English and logical notation, and made into comments (that is, they follow the symbols '`||`'). Note that in Miranda any part of your program which starts with `||`, together with all the text to its right (on the same line), is regarded as being a comment and hence is ignored by the evaluator.

The square root function could be specified by

```
sqrt :: num -> num
||pre:  x >= 0
||post: (sqrt x)^2 = x
```

`sqrt x=` HACKER *must fill this in*

3.3 Defensive specifications — what happens if the input is bad?

This is a relatively convenient specification for HACKER because he doesn't have to worry about the possibility of negative input. That worry has been passed over to PUNTER, who must therefore be careful. If, by mistake, he gives a negative input, then the contract is off and there is no knowing what might happen. He might obtain a sensible answer, or a nonsense answer, or an apparently sensible but actually erroneous answer, or an error message, or an infinite loop, or a system crash, or World War III, or anything. ('Garbage in, garbage out'; the contract itself is 'Non-garbage in, non-garbage out'.)

This balance of worry is usually sensible if the only way in which PUNTER uses HACKER's square root program is by calling it from a program of his own. He needs to look at every place where it is called, and convince himself that he would never use a negative input. Thus in exchange for some care at the programming stage, the `sqrt` function can run efficiently without checking inputs all the time.

On the other hand, PUNTER may intend to use the program at an exposed place (for instance, in a calculator) where any input at all may conceivably be provided. In that case, PUNTER would prefer a 'defensive specification' for a function that defends itself against bad arguments. When HACKER asks if he can assume that the input is non-negative, PUNTER replies: 'No. If it is negative, stop and print an error message.'

```
defensivesqrt :: num -> num
||pre:  none
||post: (x < 0 & error reported) \/
||      (x >= 0 & (defensivesqrt x)^2 = x)
```

(We would like to use the logical notation $\wedge$ and $\vee$ for 'and' and 'or' but for a program comment, where it is impossible to type logical symbols we use the notation `&` and `\/` instead. This matches Miranda's own notation.) The point is that different ideas about how to handle erroneous input must be reflected in different specifications.

3.4 How to use specifications: `fourthroot`

Suppose PUNTER wants to write a Miranda function to calculate fourth roots:

```
fourthroot :: num -> num
||pre:  x >= 0
||post: (fourthroot x)^4 = x
```

Essentially, he wants to apply HACKER's `sqrt` twice, but he also notices a nuisance — the specification of `sqrt` doesn't specify the *positive* square root. So he splits the function definition into two cases:

```
fourthroot x = sqrt y,        if y>=0
             = sqrt(-y),      otherwise
               where y = sqrt x
||Would help if sqrt gave positive square roots.
```

PUNTER now wishes to show that this definition of `fourthroot` satisfies its specification. It is important to understand that he does not need to know anything at all about how HACKER calculates square roots. He just assumes that `sqrt` satisfies *its* specification. The specification is all that PUNTER knows, or is entitled to assume, about the `sqrt` function.

Note there is something important for PUNTER to do. He uses `sqrt` in three places, and at each one he must check that the pre-condition holds: that the argument of `sqrt` is non-negative.

3.5 Proof that `fourthroot` satisfies its specification

We want to *prove* (or explain why) `fourthroot` works correctly, that is,

$$\forall x : num.\ (x \geq 0 \rightarrow (\texttt{fourthroot}\ x)^4 = x)$$

We do this on the *assumption* that `sqrt` works correctly, that is,

$$\forall x : num.\ (x \geq 0 \rightarrow (\texttt{sqrt}\ x)^2 = x)$$

(Of course, it is possible that it doesn't, but `fourthroot` should not have to worry about that. It is the responsibility of `sqrt` to get its answer right.)

We shall put the reasoning in a framework where the assumptions go at the top, the conclusion (what is to be proved) goes at the bottom and the proof goes in the middle, as in Figure 3.2. What we want to end up with is a

$\forall x : num.\ (x \geq 0 \rightarrow (\texttt{sqrt}\ x)^2 = x)$ assumption

$\vdots$

proof

$\vdots$

$\forall x : num.\ (x \geq 0 \rightarrow (\texttt{fourthroot}\ x)^4 = x)$ conclusion

Figure 3.2

proof that, as you read down through it, steadily accumulates more and more true consequences of the assumptions until it reaches the desired conclusion. That is how the proof can be read, but we can see already that writing it does not go straight down from top to bottom — we are going to have an interplay between working forwards from the assumptions and backwards from the desired conclusions.

The method is fully investigated in Chapter 16. In this example we give a rather informal introduction to it.

Here is a typical backward step. To prove the conclusion, we must show that if someone gives us a number — and we don't care what number it is

— as long as it is non-negative, `fourthroot` will calculate a fourth root of it. Once we have the number it is fixed, so let us give it a different name c to indicate this. So we are now working in a hypothetical context where

1. we have been given our c
2. c is a number
3. $c \geq 0$

and given all these assumptions we must prove $(\mathtt{fourthroot}\ c)^4 = c$. Figure 3.3 shows a box drawn around the part of the proof where these temporary assumptions are in force. For the final conclusion we have left c behind, so we

$$\begin{array}{lll}
 & \forall x : num.\ (x \geq 0 \rightarrow (\mathtt{sqrt}\ x)^2 = x) & \text{assumption} \\ \hline
\multicolumn{1}{|l}{c : num} & c \geq 0 & \multicolumn{1}{l|}{\text{temporary assumptions}} \\
\multicolumn{1}{|l}{} & \vdots & \multicolumn{1}{l|}{} \\
\multicolumn{1}{|l}{} & \text{proof} & \multicolumn{1}{l|}{} \\
\multicolumn{1}{|l}{} & \vdots & \multicolumn{1}{l|}{} \\
\multicolumn{1}{|l}{} & (\mathtt{fourthroot}\ c)^4 = c & \multicolumn{1}{l|}{\text{to prove}} \\ \hline
 & \forall x : num.\ (x \geq 0 \rightarrow (\mathtt{fourthroot}\ x)^4 = x) & \text{conclusion}
\end{array}$$

Figure 3.3

can come out of the box. What this purely logical, and automatic, analysis has given us is a context (the box) where we can begin to come to grips with the programming issues. Since $c \geq 0$, we can use our original assumption (that `sqrt` works) to deduce that `sqrt` c gives an answer y (with $y^2 = c$), and either $y \geq 0$ or $y < 0$.

We thus — again by automatic logic, but working forwards this time — have two cases to work with, which again we put in boxes because each case has a temporary assumption ($y \geq 0$ for one, $y < 0$ for the other). In each case, we must prove $(\mathtt{fourthroot}\ c)^4 = c$, so this equation ends up by being written down three times. This can be seen in Figure 3.4. The two cases may then be argued by chains of equations as in the final box proof, Figure 3.5. Notice the following features of box proofs:

1. Each box marks the scope, or region of validity, of some names or assumptions. For instance, within the left-hand innermost box we are working in a context where
   - we have a number c
   - $c \geq 0$
   - $y = \mathtt{sqrt}\ c \geq 0$

	$\forall x : num.\ (x \geq 0 \rightarrow (\texttt{sqrt}\ x)^2 = x)$	assumption		
$c : num$	$c \geq 0$	assumption		
	$(\texttt{sqrt}\ c)^2 = c$	spec of `sqrt`		
	$y^2 = c$	write y for `sqrt` c		
	$y \geq 0 \vee y < 0$			
	$y \geq 0$		$y < 0$	
	$\vdots$	case 1	$\vdots$	case 2
	$(\texttt{fourthroot}\ c)^4 = c$	to prove	$(\texttt{fourthroot}\ c)^4 = c$	to prove
	$(\texttt{fourthroot}\ c)^4 = c$			
	$\forall x : num.\ (x \geq 0 \rightarrow (\texttt{fourthroot}\ x)^4 = x)$	conclusion		

Figure 3.4 Working forwards

	$F = \texttt{fourthroot}\ c$			
	$\forall x : num.\ (x \geq 0 \rightarrow (\texttt{sqrt}\ x)^2 = x)$	assumption		
$c : num$	$c \geq 0$	assumption		
	$(\texttt{sqrt}\ c)^2 = c$	spec of `sqrt`		
	$y^2 = c$	y for `sqrt` c		
	$y \geq 0 \vee y < 0$			
	$y \geq 0$		$y < 0$	
	$(F)^4 = ((F)^2)^2$	arithmetic	$(F)^4 = ((F)^2)^2$	arithmetic
	$= ((\texttt{sqrt}\ y)^2)^2$	def F	$= ((\texttt{sqrt}\ (-y))^2)^2$	def F
	$= y^2$	spec `sqrt`	$= (-y)^2$	spec `sqrt`
	$= c$	as required	$= y^2$	arithmetic
			$= c$	as required
	$(\texttt{fourthroot}\ c)^4 = c$			
	$\forall x : num.\ (x \geq 0 \rightarrow (\texttt{fourthroot}\ x)^4 = x)$	conclusion		

Figure 3.5 The final box proof for `fourthroot`

These are not permanent; for instance outside the boxes nothing is known of c, and the right-hand innermost box does not know that $y \geq 0$.

2. When you read a box proof, you can read it straight down from the top: each new line is either a temporary hypothesis or is derived from lines higher up. But when you *construct* a box proof you work both forwards, from assumptions, and backwards, from your goal. Hence there is a definite difference between *proof* and *proving*. (It is very similar to that between the Reasoned Program and Reasoned Programming.)

Box proofs can be translated into English as follows:

Let $c : num$ with $c \geq 0$, and let $y =$ `sqrt` c. Since $c \geq 0$ (pre-condition of `sqrt`), we know $y^2 = c$.

There are two cases.

If $y \geq 0$, then
$(\texttt{fourthroot}\ c)^4 = (\texttt{sqrt}\ y)^4 = ((\texttt{sqrt}\ y)^2)^2$
$= y^2$ (because $y \geq 0$ and so satisfies the pre-condition of `sqrt`)
$= c$

If $y < 0$, then
$(\texttt{fourthroot}\ c)^4 = ((\texttt{sqrt}\ (-y))^2)^2$
$= (-y)^2$ (because $-y \geq 0$)
$= y^2 = c$

Either way, we obtain the required result. □

However, the virtue of box proofs for beginners is that certain steps are automatic, and the box proofs give you a framework for making these steps. They take you to a context where you have disentangled the logic and have something to prove about concrete programs.

3.6 A little unpleasantness: error tolerances

Act 2

HACKER: I can't calculate exact square roots. There has to be an error tolerance.

PUNTER: But the programs I've just had ROMmed assume the roots *are* exact. It'll cost me £5m to have them changed.

HACKER: I'm sorry, you'll just have to.

PUNTER: You shall hear from my solicitors.

[*Exeunt scowling*]

The story has a happy ending. HACKER's legal department had prudently included the following general disclaimer clause in his software:

> This software might do anything at all, but there again it might not. Anything HACKER says about it is inoperative.

PUNTER stopped thinking in legal terms, and negotiated the following revised specification with HACKER:

```
sqrt :: num -> num
||pre:  x >= 0
||post: |(sqrt x)^2-x| < tolerance
```

where `tolerance` was a number still to be negotiated.

This is a perfectly common occurrence, that specifications must be revised in the light of attempts to implement them. It is a nuisance, but it happens, and you must understand how to deal with it.

In this case, the post-condition has been weakened for HACKER's benefit, and this makes extra work for PUNTER. He must look at every place where he has called `sqrt` and check whether his reasoning still works with the revised specification. If it does still work, PUNTER is happy. If not, PUNTER may yet be able to modify his program and his reasoning to cope. But in this case, PUNTER realizes that he cannot compute exact fourth roots after all, so he must go back apologetically to *his* customer and negotiate a revised specification for `fourthroot`.

The reasoning is the same if HACKER and PUNTER are collaborators, or even the same person (a programmer calling his own procedures).

3.7 Other changes to the contract

The error tolerance was a *weakened post-condition.* Other possibilities are as follows:

- *Strengthened pre-condition:* HACKER might decide he needs to assume more for his routine to work. Again, PUNTER must check every call to ensure that the new pre-conditions are still set up properly.
- *Weakened pre-condition or strengthened post-condition:* Now the specification is better for PUNTER, so he has no checking to do. This time it is HACKER who must check his routine to ensure that it still satisfies the new conditions. He might find that it does, or that he can modify it so that it does, or that he has to go to the people who wrote the functions he calls and negotiate a revised specification for them.

Either way, we see that when a specification is changed, programs have to be checked to make sure that they still fit the revised specification. This checking is boring, but routine: because of the way the specifications have given *logical* structure to the program, you know exactly which parts of the program you need to examine, and exactly what you are checking for. If you don't bother, you are likely to run into the *Hydra* problem: for every mistake you correct, you make ten new ones.

There are also mixed changes to specification, for instance if you strengthen both pre- and post-conditions. This might happen if the customer wants a strengthened post-condition but the programmer needs a strengthened pre-condition before he can deliver it. The customer may be happy with that. Perhaps by chance his existing applications already set up the strong pre-condition. On the other hand, he may find that the new pre-condition requires too much work to be worth while. Thus the new specification may or may not be a good idea. The customer and programmer must negotiate the best compromise.

Simultaneously weakened conditions are similar.

3.8 A careless slip: positive square roots

The story so far: PUNTER and HACKER have agreed a specification for `sqrt` with error tolerances.

Act 3, Scene 1

PUNTER: The result of `sqrt` always seems to be non-negative. Is that right?

HACKER: [*looks at code*] Yes.

PUNTER: Good. That's useful to know.

[*exeunt*]

This is how, validly, coding may feed back into the specification. If they agree on a new, strengthened post-condition:

$$\mid (\texttt{sqrt}\ x)^2 - x \mid \leq \textit{tolerance} \ \wedge \ (\texttt{sqrt}\ x) \geq 0$$

then this is better for PUNTER, so he is happy, and HACKER is no worse off because his code does it anyway. PUNTER thinks they have agreed, but unfortunately HACKER never wrote it into the comments for the `sqrt` function.

Act 3, Scene 2

[*It is very late at night.* HACKER *sits in front of a computer terminal.*]

HACKER: Eureka! I can make `sqrt` go 0.2% faster by making its result negative.

[*Erases old version of* `sqrt`]

Act 3, Scene 3

PUNTER: My programs have suddenly stopped working.

HACKER: [*looks at code*] It's not my fault. `sqrt` satisfies its specification.

[*exeunt*]

This kind of misunderstanding is just as common *when you are your own customer* (that is, when you write your own procedure). It is easy to assume that you can understand a simple program just by looking at the code; but this is dangerous. The code can only tell you what the computer does, not what the result was meant to be. Avoid the problem with a strong specification discipline: only assume what is specified. Equivalently, everything that is assumed must be in the specification.

3.9 Another example, `min`

The minimum function is easily enough defined as

```
min ::  num -> num -> num
min x y = x,   if x <= y
        = y,   otherwise
```

However, there is an unnatural asymmetry in the way the cases are divided between `x≤y` and `x>y`, when they could equally well have been `x<y` and `x≥y`. This case division is not part of what you need to know to be able to use `min`. Perhaps a more natural specification would be

```
min :: num -> num -> num
||pre:  none
||post: ((min x y) = x \/ (min x y) =y) &
||      ((min x y) <= x & (min x y) <=y)
```

Proposition 3.1 The definition of `min` satisfies the specification.

Proof Suppose x and y are real numbers. There are two cases — either $x \leq y$, or $x > y$.

case 1: $x \leq y$, then $(\texttt{min}\ x\ y) = x$. This immediately proves $((\texttt{min}\ x\ y) = x \vee (\texttt{min}\ x\ y) = y)$ and $(\texttt{min}\ x\ y) \leq x$; and $(\texttt{min}\ x\ y) \leq y$ because $x \leq y$.

case 2: $x > y$, then $(\texttt{min}\ x\ y) = y$. Immediately, $((\texttt{min}\ x\ y) = x \vee (\texttt{min}\ x\ y) = y)$ and $(\texttt{min}\ x\ y) \leq y$; and $(\texttt{min}\ x\ y) \leq x$ because $y < x$. □

We can now prove properties of `min` solely from the specification.

Proposition 3.2 $(\texttt{min}\ x\ y)$ is uniquely determined by the specification.

Proof Let $m1$ and $m2$ be two possible values of $(\texttt{min}\ x\ y)$ according to the specification (not the definition). We wish to show that $m1 = m2$. We know that

$$(m1 = x \vee m1 = y) \wedge (m1 \leq x) \wedge (m1 \leq y) \wedge$$
$$(m2 = x \vee m2 = y) \wedge (m2 \leq x) \wedge (m2 \leq y)$$

We first show that $m1 \leq m2$. From $(m2 = x \vee m2 = y)$, there are two cases, two possible values for $m2$, and, either way, $m1 \leq m2$. By symmetry, $m2 \leq m1$, so $m1 = m2$. □

Specifications do not have to specify uniquely; there may be several different possible answers, equally satisfactory. But uniqueness of specification is a useful property, as is illustrated by the next result.

Proposition 3.3 (*Commutativity*) (`min` x y) $=$ (`min` y x).

Proof The specification of (`min` x y) is symmetrical in x and y, so it is also satisfied by (`min` y x). Hence, by uniqueness (the previous proposition), (`min` y x) $=$ (`min` x y). □

3.10 Summary

- A specification of a procedure can be expressed as typing information, pre-condition and post-condition.
- You can write these down as part of a Miranda program using logical notation.
- To show that a function definition satisfies the specification you assume that you are given arguments satisfying the pre-condition, and show that the result satisfies the post-condition.
- When you use a function, you rely on its *specification*, not its *definition.*
- Any change to a specification requires a methodical examination of the function definition, and all calls of the function. This may entail no changes, or changes to the program only, or to other specifications, or to both.

3.11 Exercises

1. Write pre- and post-conditions for the functions (both in the text and the exercises) in Chapter 2. Try to get to the heart of what each function is meant to achieve.
2. Use pre- and post-conditions to write a specification for calculating square roots. Try to think of as many ideas as possible for what the customer might want. Choosing one interpretation rather than another may be a *design decision*, or it may call for clarification from the customer.
3. Suppose you want a procedure to solve the quadratic equation $ax^2 + bx + c = 0$:

```
solve :: num -> num -> num -> (num, num)
||pre:  ?
||post: x1 and x2 are the solutions of a*x^2+b*x+c = 0,
          where (x1,x2) = (solve a b c)
```

Assume that you intend to use the formula

$$x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

What are suitable pre- and post-conditions? Try to write them in logic. (NOTE: the result type `(num, num)` is the type of *pairs* of numbers, such as `(19, 2.6)`.)

4. Using the uniqueness property of `min` prove the associative property, that is,

$$(\textit{Associativity}) \qquad (\texttt{min}\ x\ (\texttt{min}\ y\ z)) = (\texttt{min}(\texttt{min}\ x\ y)\ z)$$

5. Directly from the *definition* of `min` prove associativity.
6. Use pre- and post-conditions to write specifications for the standard Miranda functions `abs` and `entier`. (Of course, these are already coded unalterably. Your 'specification' expresses your understanding of what the standard functions do.)
   `abs` takes a number and makes it non-negative by removing its sign: for instance, `abs` -1.3 = `abs` 1.3 = 1.3.
   `entier` takes a number x and returns an integer, the biggest that is no bigger than x. For instance,

   `entier` 3 = 3 `entier` 2.9 = 2 `entier` -3 = -3 `entier` -2.9 = -3

7. (a) Specify a function `round :: num -> num` that rounds its argument to the nearest integer. Try to capture the idea that, of all the integers, `round` x is as close as you can get to x.

   (b) Show that the definition `round1` satisfies the specification of `round`:

```
round1 x = e,    if abs (e-x)< abs (e+1-x)
                 || i.e.  if e is closer to x than e+1
         = e+1, otherwise
              where e = entier x
```

   (c) Show that this definition `round2` computes the same function as `round1`.

```
round2 x = entier (x+0.5)
```

   (HINT: express the condition $\texttt{abs}(e - x) < \texttt{abs}(e + 1 - x)$ without using `abs`.)

Chapter 4

Functional programming in Miranda

In the preceding chapters, where we were illustrating rather general issues of programming, we did not probe too deeply into the details of Miranda but relied on its closeness to mathematical notation to make the meaning clear. We now turn to a more careful description of Miranda itself.

4.1 Data types — `bool`, `num` and `char`

Every value in Miranda has a type; the simplest are `num` (which you have already seen), `bool` and `char`.

The data type `num` includes both whole numbers (or *integers*) and fractional numbers (or *reals*, or floating-point numbers). A whole number is a number whose fractional part is zero. Here are some data values of type `num`:

```
56  -78  0  -87.631  0.29e-7  4.68e13  -0.62e-4  12.891
```

Although there are infinitely many numbers, computers have finite capacity and can only store a limited range. Similarly, within a finite range, there are infinitely many fractional numbers, so not all of them can be stored exactly. Although such practical limitations can be important when you are doing numerical calculations, especially when you are trying to obtain a fractional answer that is as accurate as possible, we shall largely ignore them here. The theory of *numerical analysis* deals with these questions.

Booleans are the truth-values `True` and `False` and their Miranda type is called `bool`. Truth-values are produced as a result of the application of the *comparison* operators (for example, `>`, `>=`, `=`, `<`). They can also be returned by user-defined functions, for example the function `even`. Expressions of type `bool` are really, rather, like logical formulas, and on this analogy functions that return a `bool` as their result are often called *predicates*.

If the evaluator is presented with an expression which is already in its normal form, then it will simply echo back the same expression since it cannot reduce the expression any further. For example,

```
Miranda False
False
```

`char` is the type of *characters*, the elements of the ASCII character set. They include printable symbols such as letters (`'a'`, `'A'`, ...), digits (`'0'` to `'9'`), punctuation marks (`','`, ...) and so on, as well as various layout characters such as newline `'\n'`. Obviously, characters are most useful when strung together into lists such as `"Reasoned Programming"` (note the double quotes for strings, single quotes for individual characters), so we shall defer more detailed consideration until the chapter on lists (Chapter 6).

4.2 Built-in functions over basic types

Values of the basic built-in types can be manipulated by a host of built-in functions and operators. Most such built-in functions and operators are binary (that is, operate on two arguments) and can be used in infix form.

Arithmetic

These operations are on numbers. Each is used as a *binary infix* operator. The minus sign can also be used as a *unary prefix* operator.

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

All except `/` return exact integer results when arguments are integers, provided that the integers are in the permitted range. Representation for floating-point numbers may not be exact, so operations on fractional numbers may not produce the same results as ordinary arithmetic. For example, `(x*y)/y` and `x` may not be equal. `div` and `mod` can be specified in tandem by

```
div :: num -> num -> num
mod :: num -> num -> num
||pre:  int(x) & int(y) & y ~= 0
||      (where int(x) means x is a whole number and ~ means not)
||post: x = (x div y) * y + (x mod y)
||      & y>0 -> (0 <= (x mod y) < y)
||      & y<0 -> (y < (x mod y) <= 0)
```

Arithmetic expressions can be entered directly into the evaluator, for example after the computer has displayed the `Miranda` prompt:

```
Miranda 14 div 5
2
Miranda 14 mod 5
4
Miranda 2^4
16
```

The relative precedence of these operators is as follows:

`+ -`	
`* / div mod`	⇓ increasing precedence
`^`	

Function application always binds more tightly than any other operator! Parentheses are used when one is not sure of binding powers or when one wishes to force a different order of grouping, for example,

`Miranda` $\underline{\texttt{double 5}}$ `+` $\underline{\texttt{8 mod 3}}$ = $\underline{\texttt{10+2}}$

`12`

`Miranda` $\underline{\underline{\texttt{double }\underline{\texttt{(5 + 8)}}}}$ `mod 3` = $\underline{\texttt{double 13}}$ `mod 3` = `26 mod 3`

`2`

Comparisons

`=`	equals	
`~=`	not equals	All have the same level of precedence.
`<`	less than	
`>`	greater than	
`<=`	less than or equal	Their precedence is lower
`>=`	greater than or equal	than that of the arithmetic operators.

Comparison operators are made up of *relational* operators (`>`, `>=`, `<`, `<=`) and *equality* operators (`=`, `~=`) and their result is of type `bool`. The following are some examples:

```
Miranda  5 = 9
False
Miranda  6 >= 2+3
True
```

As the second example suggests, the precedence of comparison operators is lower than that of the arithmetic operators. Note that comparison operators cannot be combined so readily; for example, the expression `(2<3<4)` would give a type error since it would be interpreted as

$$\texttt{((}\underline{\texttt{2<3}}\texttt{)<4) = True<4}.$$

When operating on numbers '`=`' may not return the correct result unless the numbers are integers in the permitted range. This is because fractional numbers should be compared up to a specific tolerance. For example,

```
Miranda  sqrt(2)^2 = 2
False
```

We can define a function **`within`** as follows:

```
within eps x y = abs(x-y) < eps
```

`within` can then be used instead of '`=`' when comparing fractional numbers to a certain tolerance. For example, (**`within`** `0.001 a b`) can be used to see if `a` and `b` are closer than 0.001 apart.

Logical operators

Boolean values may be combined using the following logical operators:

`&`	conjunction (logical $\wedge$ 'and')	in order of
`\/`	disjunction (logical $\vee$ 'or')	$\Downarrow$ increasing precedence
$\sim$	negation (logical $\neg$ 'not')	

Their precedence is lower than that of comparisons. They can be defined in Miranda itself (not that you will need to do this) as in Figure 4.1. Defining these primitives in Miranda not only gives their meaning but also illustrates the use of pattern matching with Booleans. Exercise: we have used one equation to define **`and`** and two for **`or`**. Try writing **`and`** with two equations and **`or`** with one.

It is always a good idea to use parentheses whenever — as is often the case with logical connectives — there is the slightest doubt about the intended meaning:

```
Miranda 4>6 & (3<2 \/ 9=0)
False
```

```
and ::      bool -> bool -> bool
||pre:      none
||post:     and x y = x & y
or ::       bool -> bool -> bool
||pre:      none
||post:     or x y = x \/ y
not ::      bool -> bool
||pre:      none
||post:     not x = ~x

and x y     = y,       if x
            = False,   otherwise

or True x   = True
or False x = x

not True    = False
not False   = True
```

Figure 4.1

4.3 User-defined functions

Identifiers

Before introducing a new function the programmer must decide on an appropriate name for it. Names, also called *identifiers*, are subject to some restrictions in all programming languages.

Throughout a program, identifiers are used for variables, function names and type names. In Miranda, identifiers must start with a *lower case* letter. The remaining characters in the identifier can be letters, digits, _, or ' (single quote). However, not all such identifiers are valid as there are a number of special words (*reserved words*) which have a particular meaning to the evaluator, for example `where`, `if`, `otherwise`. Clearly, the programmer cannot use a reserved word for an identifier as this would lead to ambiguities. Furthermore, there are also a number of predefined names (for example, those of built-in functions such as `div`, `mod`) which must be avoided.

Meaningful identifiers for functions and variables will make a program easier to read. Longer names are usually better than shorter names, although the real criterion is *clarity*. For example, the identifier `record` is probably a better choice than `r`. But deciding whether it is better than, say, `rec` is not as straightforward. In fact, in most cases modest abbreviations need not reduce the clarity of the program.

A good rule is that identifiers should have long explanatory names if they are used in many different parts of the program. This is because it may be difficult to refer to the definition if it is a long way from the use. On the other hand, identifiers with purely local significance can safely have short names — such as `x` for a function argument. If the variable in question is a general purpose one then nothing is gained by having a long name such as `theBiggestNumberNeeded`; an identifier such as `n` may be just as clear. Finally, it is worth mentioning that it is best to avoid acronyms for identifiers. For example, `tBNN` is even worse than `theBiggestNumberNeeded`.

Defining values

It is often useful to give a name to a value because the name can then be used in other expressions. For example, we have already seen the definition of `mypi`:

```
mypi ::  num
mypi = 3.14159
```

As usual, the choice of meaningful names will make the program easier to read. The following is a simple example of an expression which involves names that have been previously defined using '`=`':

```
hours_in_day =  24
days  =  365
hours_in_year =  days * hours_in_day
```

If you are already familiar with imperative languages such as Pascal or BASIC, then it is important to understand that a definition like this is *not* like an assignment to a variable, but, rather, like declaring a constant. The identifier `days` has the value 365 and this cannot be changed except by rewriting the program. What is more, if you have conflicting definitions within a program, then only the first will ever have any effect. At this point it may also appear natural to be able to give names not only to values such as numbers or truth-values but also to functions, for example

```
      dd :: num -> num
      dd = double
```

`dd` behaves identically to `double` in every respect. This indicates that functions are not only 'black boxes' that map values to other values, but are *values* in themselves. Thus in functional languages functions are also first-class citizens (just like numbers, Booleans, etc.) which can be passed to other functions as parameters or returned as results of other functions. This is discussed in much more detail in Chapter 8.