lies closer to the original vague intentions, not the code. So the first step should always be to think carefully about your intentions and try to refine them to a more precise specification.

After that, the next step is to convert globality (specification) into locality (code), and this is much easier after the initial thought. In fact, there are specific mathematical techniques, which we shall discuss later, that make much of this process automatic. At the same time, they tie the specification and code carefully together so you know *as part of the coding process* that the link between them is made. Figure 1.2 illustrates the progression from vague intention to precise code via precise specification.

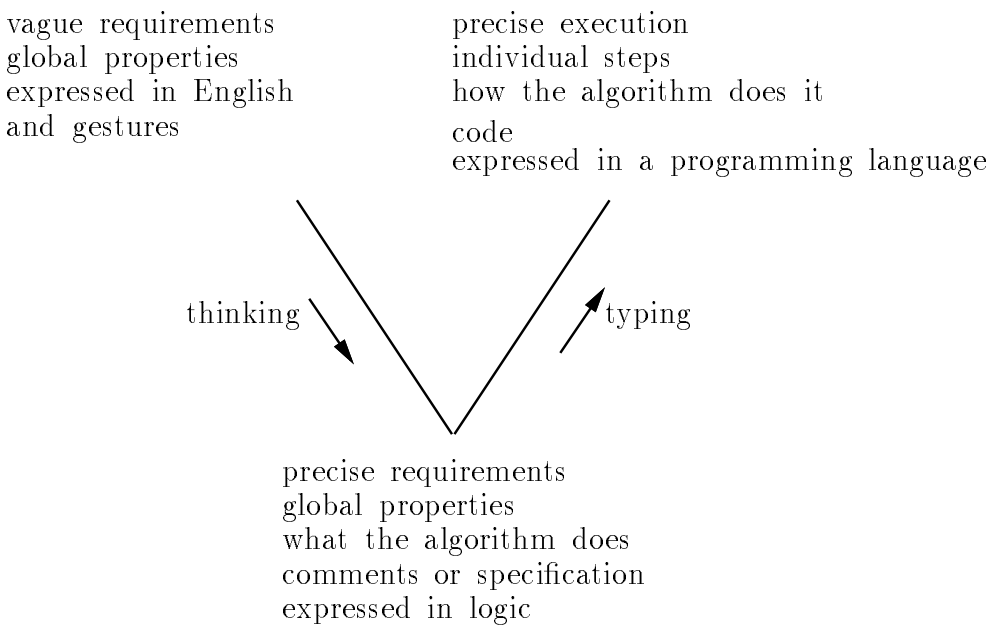


Figure 1.2

1.7 Modules

This distinction that we have made between specification and code, corresponding to users and computer, also makes sense inside a program. It is common to find that part of the program, with a well-defined task, can be made fairly self-contained and it is then called — in various contexts — a *subprogram*, or *subroutine*, or *procedure* or *function*, or, for larger, more structured pieces of program, a *module*. The idea is that the overall program is a composite thing, made up *using* components: so it takes on the role of *user*.

A module can be specified, and this describes how its environment, the rest of the program, can call on it and what that achieves. The specification

describes all that the rest of the program needs to know about the module. The implementation of the module, the code that it contains, its inner workings, is hidden and can be ignored by the rest of the program.

Modularization is crucial when you want to write a large program because it divides the overall coding problem into independent subproblems. Once you have specified a module, you can code up the inside while forgetting the outside, and vice versa. The specifications of the modules also act as *bulkheads*, like the partitions in the hold of a ship that stop water from a hole spreading everywhere and sinking the ship. The specifications compartmentalize the program so that if an error is discovered in one module you can easily check whether or not correcting it has any consequences for the others. This helps to avoid the 'Hydra' problem, in which correcting one error introduces ten new ones.

1.8 Programming in the large

This book makes a significant simplifying assumption, namely that *specifications can be got right first time.* This is usually (though not always) realistic for small programs, and so the techniques that we shall present are called those of programming *in the small.* The underlying idea, of understanding the users' point of view through a specification, is still important in large-scale programs, but the techniques cannot be applied in such a pure form (specify first, then code). To understand why, you must understand what could possibly be wrong with a specification.

The ultimate test — in fact the definition — of *quality* of software is that it is fit for its purpose. To be sure, the specification is supposed to capture formally this idea of fitness, and if that has been done well then a *correct* program, one for which the code satisfies the specification, will indeed be a quality one. But, conversely, *specifications can have mistakes in them,* and this will manifest itself in unexpected and unwanted features in a formally correct program. Hence *correctness* is only an approximation to *quality.*

Now there are many advantages to forgetting quality and working for correctness. For instance, we have precise objectives (write the code to satisfy the specification) that are susceptible to mathematical analysis, and we can modularize the program and work for correctness of small, easy parts, forgetting the wider issues. The widget manufacturer who takes an order for 2000 blue, size 15 widgets will find life easier if he does not ask himself whether they are really the right colour, let alone whether or not their end use is to help train dolphins to run suicide missions smuggling cocaine.

However, the true proof of the program is, despite all we have said, its behaviour in real life, and ultimately no programmer should forget that. The specification and reasoning are merely a means to an end. Never forget the

possibility that the specification is faulty. This will be obvious if correct code plainly gives undesirable behaviour, but earlier warning signs are when the coding is unexpectedly complicated or perhaps even impossible.

If the specification is faulty, then it can be revised, which will involve checking existing code against the revised specifications. Alternatively, the specification can be left as it is for the time being, with the intention of revising it for future versions or in the light of future experience. This is often quite reasonable, and provides some stability to the project, but it should be chosen after consideration and not out of inertia. The universal experience is that the later corrections are left, the more expensive it is to make them (A Stitch in Time Saves Nine), and large software projects have been destroyed by the accumulation of uncorrected errors.

For programming in the large, many of the practical techniques that people use can be seen as being there to help to correct specificational faults as early as possible, while they are still cheap to fix. For instance, *requirements elicitation* is about how to communicate as effectively as possible with the users, to find out what they really do need and want; then a number of design methodologies help to obtain a good specification before coding starts; and prototyping produces some quick, cheap code in order to find those faults (such as difficulty of use in practice) that are best exposed by a working version. All of these are important issues but they are ignored in the rest of this book.

1.9 Logical notation

English is not always precise and unambiguous — that is why computer programming languages were invented. In general, the fewer things that a language needs to talk about, the more precise it can be.

In our specifications, we are going to make use of *logic* to make precise one particular aspect of what we want to say, namely how different properties connect together. In English there are connecting words such as 'and', 'or', 'but', 'not', 'all', 'some', and so on, and in logic these are systematized and given individual symbols. The reason for the importance of these *connectives* is that it is the logical connections between the given properties that allow us to deduce new ones.

For instance, suppose an instruction manual tells you:

> 'If anyone envelops the distal pinch-screw parascopically, then the pangolin will unbundle.'

Suppose you also know that the anterior proctor has just enveloped the distal pinch-screw parascopically. You do not need to be an expert on pangolins to realize that it is likely to unbundle. The reason is that you have spotted

the underlying *logical* structure of these facts, and it does not depend on the nature of pinch-screws, pangolins or proctors.

This logical structure shows up best if we introduce some abbreviations:

> $E(x)$ (where x stands for any person or thing) stands for 'x envelops the distal pinch-screw parascopically'.
> A stands for 'the anterior proctor'.
> P stands for 'the pangolin unbundles'.

As a special case of this notation, if we substitute A for x then:

> $E(A)$ stands for 'the anterior proctor envelops the distal pinch-screw parascopically'.

These abbreviations are not in themselves logical notation; that comes in when we connect these statements together. Logic writes —

> $\wedge$ for 'and'
> $\forall$ for 'for all' (reflecting 'anyone')
> $\rightarrow$ for 'implies' (reflecting 'if ... then ...')

Now our known facts appear as

$$\forall x.\ [E(x) \rightarrow P] \wedge E(A)$$

and just from this logical structure we can deduce P.

Much of logic is about making such deductions based on the logical structure of statements. The general pattern is that we start from some statements A called the *premisses*, and then deduce a *conclusion* B. The argument from A to B is *valid* if in any situation where A is true, it follows inevitably that B is true, too. Logic gives *formal* rules — that is to say, rules that depend just on the *form* of the statements and not on their content or meaning — for making valid deductions, and if these rules give us an argument from A to B then we write $A \vdash B$ (A *entails* B).

For example,

> If I have loads of money, then I buy lots of goods.
> I have loads of money.
> $\vdash$ I buy lots of goods.

is a valid argument There are situations where the premisses are false (for instance, if, as it happens, I am a miser then the first premiss is false), but that does not affect the validity of the argument. *So long as* the premisses are true, the conclusion ('I buy lots of goods') will be, too. However,

> If I have loads of money, then I buy lots of goods.
> I go on a spending spree.

$\vdash$ I have loads of money.

is not a valid argument. Even if the premisses are true, the conclusion *need not* be since I might be making imprudent use of my credit card.

In this book we shall use logic to help us with, broadly speaking, two kinds of deduction related to a given specification of a program: first, deducing new facts about how the program will behave when we come to use it; and, second, deducing that the program code, or implementation, really does meet the specification. Part II of this book is entirely devoted to logic itself.

1.10 The need for formality

English, and natural language in general, is tremendously rich and can express not only straightforward assertions and commands but also aspects of emotion, time, possibility and probability, meaning of life, and so on. But there is a cost. Much of it relies on common understanding and experience, and on the context. Look at the following three examples, and see how they contain progressively more that is unspoken:

1. 'She sang like her sister.'
2. 'She sang like a nightingale.'
3. 'He sang like a canary.'

(1) is fairly literal, but (2) is not — the comparison is not of the songs themselves but of their beauty, and the compliment works only because everyone knows (even if only by repute) that nightingales sing beautifully. As for (3), in a gangster film "He" might well be a criminal who, on arrest, told the police all about his accomplices. But it is extremely inexplicit, and would be hard to understand out of context.

Different people lead different lives, so these unspoken background assumptions of experience and understanding are imprecise, and this leads to an imprecision in English. To say anything precisely and unambiguously you must drastically restrict the range of what can be said, to the point where any background assumptions can also be made explicit. Then there is a direct correspondence between the language and its meaning, and you can treat the language 'formally', that is, as symbols to be manipulated (which, after all, is what a computer has to do), and be confident that such manipulations are reflected validly in the meaning.

An important example of a formal language that you must already know is algebra, the formal language of numbers. Problems can often be solved *symbolically* by algebraic manipulations without thinking about the numbers behind the symbols, and you still obtain correct answers. An extension of this is calculus. Again the symbolic manipulations — the various rules for

differentiating and integrating — can be carried through without you having to remember what the derivatives and integrals really mean.

In fact, this is only a particular application of the word 'calculus', which is Latin for 'little stone'. In ancient times, one method of *calcul*ating was by using little stones roughly like an abacus, and the idea is that you can obtain correct answers about unmanipulable things through surrogate manipulations of the little stones. We now use formal symbols instead of little stones, but the word 'calculus' is still often used for such a formal language — for instance, one part of logic is often called the 'predicate calculus'.

The other formal languages that you will see in this book are as follows:

logic This is the language of logical connections between statements. This is a very narrow aspect of the statements and so the logical notation will usually need to be combined with other notations, but once we have the logical symbols expressing the logical structure, we can describe what are logically correct arguments. Another point is that the logical symbols are more precisely defined than English words. For instance, there is a logical connective '$\vee$' that, by and large, means 'or': 'A or B' has the logical structure '$A \vee B$'. But sometimes the English 'or' carries an implicit restriction 'but not both' (the so-called *exclusive or*), and then the logic must take care to express this, as $(A \vee B) \wedge \neg(A \wedge B)$.

programming languages These are the languages of computer actions (roughly — this is more true for the imperative language Modula-2 than for the functional language Miranda). Once they are made formal then one can work with them by symbolic manipulation and this is exactly what computers do when they compile and interpret programs.

1.11 Can programs be proved correct?

We have already distinguished between quality and correctness, and explained how 'correctness', conformance to the specification, is only relative: if the specification is wrong (that is, not what the user wanted) then so, too, will be the code, however 'correct' it is. But at least the specification and code are both formal, so there is the possibility of giving formal proofs of this relative correctness — one might say that this is the objective of *formal methods* in computer software.

It is worth pointing out that what you will see in this book are really only 'informal formal methods'. There are two main reasons for this.

The first is that to give a formal correctness proof you need a formal *semantics* of your programming language, a mathematical account of what the programs actually mean in relation to the specifications. We shall not attempt to do this at all, but instead will rely on your informal understanding of what the programming constructs mean.

The second is that true formal reasoning has to include every last detail. This might be fine if it is a computer (via a software tool) that is checking the reasoning, but for humans such reasoning is tedious to the point of impracticability, and hides the overall shape of the argument — you cannot see the wood for the trees. Even in pure mathematics, proofs are 'rigorous' — to a high standard that resolves doubts — but not formal. Our aim is to introduce you to rigorous reasoning.

Now even rigorous reasoning runs the risk of containing errors, so if in this book we cannot claim unshakable mathematical correctness you might wonder what the point is. We do not seem to be working to a Reasoned Program as an error-free structure. Nevertheless, the structure of the Reasoned Program, with its specification and reasoning included, is much more stable than an Unreasoned Program, that is, code on its own. We have a clearer understanding of its working, and this helps us both to avoid errors in the first place and, when errors do slip through, to understand why we made them and how to correct them.

1.12 Summary

- The *code* is directed towards the computer, giving it its instructions.
- The *specification* is directed towards the users, describing what they will get out of the program. It is concerned with *quality* (fitness for purpose).
- By reasoning that the code satisfies the specification, you link them together into a *reasoned program*.
- By putting the specification first, as objectives to be achieved by the code, you engage in *reasoned programming*. Coding is then concerned with *correctness* (conformance with specification).
- This separation also underlies *modularization*. The specification of a module or subroutine is its interface with the rest of the program, the coding is its (hidden) internal workings.
- This book is about *programming in the small*. It makes the simplifying assumption that specifications can be got right first time.
- In practice, specifications can be faulty — so that correctness does not necessarily produce quality. Be on your guard against this.
- The earlier faults are corrected, the better and cheaper.
- There are numerous practices aimed at obtaining good specifications early rather than late, for instance talking to the customer, thinking hard about the design and prototyping, but this book is not concerned with these.
- To match the formality of the programming language, we use formal logical notation for specifications. It is also possible to use formal semantics to link the two, but we will not do this here.

Part I

Programming

Chapter 2

Functions and expressions

2.1 Functions

From the specification point of view a function is a *black box* which converts input to output. 'Black box' means you cannot see — or are not interested in — its internal workings, the implementing code. Mathematically speaking, the input and output represent the *argument* to the function and the computed *result* (Figure 2.1).

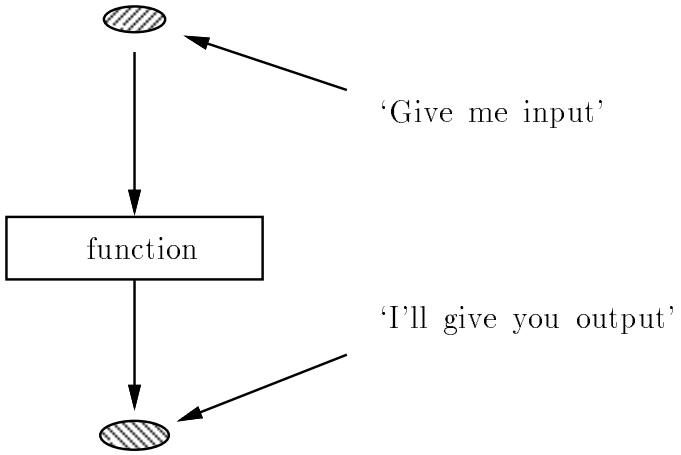


Figure 2.1

In Figure 2.2 the function `add1` simply produces a result which is one more than its given argument. The number 16 is called an argument or an *actual parameter* and the process of supplying a function with a parameter is called *function application*. We say that the function `add1` is *applied* to 16. Similarly, the function `capital` takes arguments which are countries and returns the capital city corresponding to the given country.

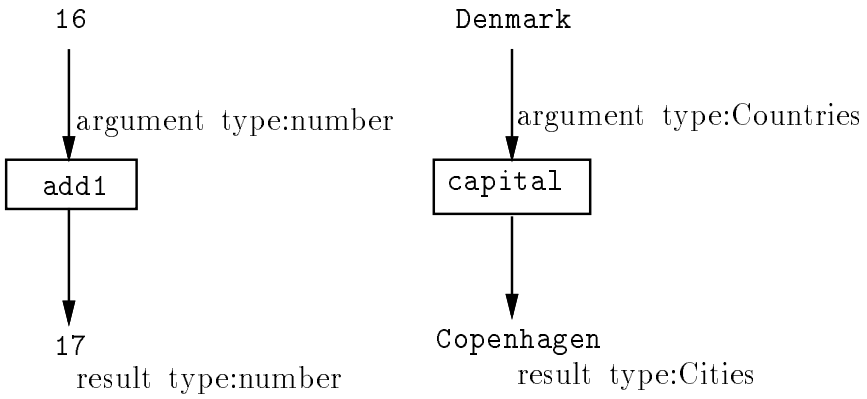


Figure 2.2

From mathematics we are all familiar with functions which take more than one argument. For example, functions `+` and `*` require two numbers as arguments. Figure 2.3 gives some examples of applications of multi-argument functions.

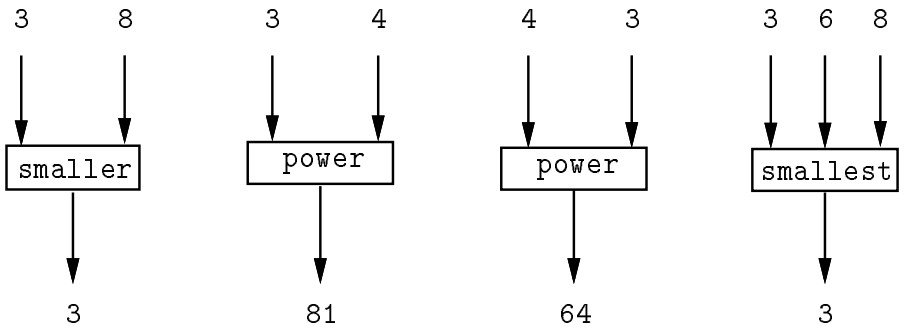


Figure 2.3

When we first define a function we need to pay attention both to the way it works as a rule for calculation (the code) and also to its overall global, external behaviour (the specification). But, when a function comes to be used, only its external behaviour is significant and the local rule used in calculations and evaluations becomes invisible (a black box). For example, whenever `double` is used the same external behaviour will result whether `double n` is defined as `2*n` or as `n+n`.

2.2 Describing functions

We can describe functions in a number of ways. We can specify the function value explicitly by giving one equation for each individual input element; or

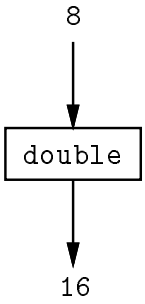
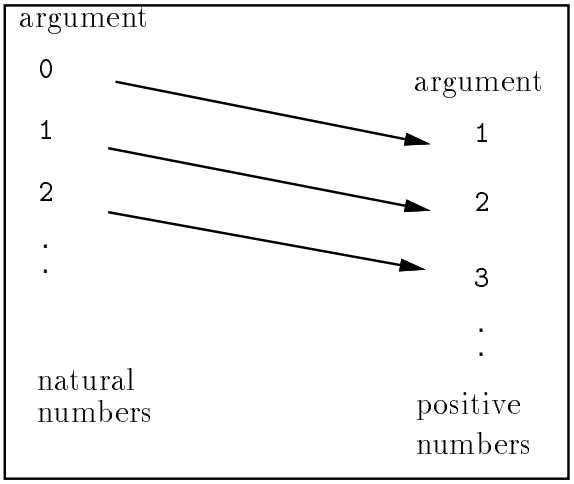


Figure 2.4

we can draw a diagram — a *mapping diagram* showing for each input element its corresponding result (Figure 2.5). However, often there will be many, even infinitely many, individual elements to consider and such methods will clearly be inconvenient.



showing individual mappings

```
add1 0 = 1
add1 1 = 2
     .
     .
```

an equation for each possible argument

```
add1 x = x+1
```

a few general equations

Figure 2.5

An alternative method is to describe the function using a few general equations (Figure 2.5). Here we can make use of *formal parameters*, which are names that we give to represent any argument to which the function will be applied. For example, the formal parameter `x` in the definition of `add1` stands for any number. The *right hand side* of the *rule* (or *equation*) describes the result computed in terms of the formal parameter. In the functional language Miranda, `add1` is described in a notation which is very close to the mathematical notation used above:

```
add1 :: num -> num
add1 x = x+1
```

The first line *declares* the function by indicating the function's argument and result types. The argument type, which precedes the arrow, states the expected type of the argument to which the function will be applied. The result type, which follows the arrow, states the type of the value returned by the function. A function is said to *map* a value from an argument type to another value of a result type. The second line is an equation which *defines* the function.

Now let us look at some more programs: for example, consider the problems of finding the area and circumference of a circle given its radius. We need the constant value π, which is built-in to the Miranda evaluator under the name `pi`, but note that even if `pi` were not built-in we could define our own constant as shown for `mypi`:

```
mypi :: num
mypi = 3.14159
circumference, areaofcircle :: num -> num
areaofcircle  radius =  pi * radius * radius
circumference  r = 2 * pi * r
```

(This also illustrates how a formal parameter is not restricted to a single letter such as `n`.) Similarly, we can define a function to convert a temperature given in degrees Fahrenheit to degrees Celsius by:

```
fahr_to_celsius :: num -> num
fahr_to_celsius temp = (temp - 32) / 1.8
```

Multi-argument functions can be defined similarly. For example, see the function below, which, given the base area and the height of a uniform object, computes the volume of the object:

```
volume :: num -> num -> num
volume  hgt area = hgt * area
```

The declaration is read as follows: `volume` is a function which takes two numbers as arguments and returns a number as its result. The reason for using `->` to separate the two number arguments will become clear later when we discuss typing in more detail. Each `->` marks the type preceding it as an argument type.

Joining functions together

More complex functions can be defined by *functional composition*, making the result of one function application an argument of another function application. This can be viewed pictorially as joining the output wire of one black box onto an input wire of another black box (Figure 2.6). In this way several functions can be combined, for example `double (4 * 6)` combines the functions `double` and `*`. This combination can be pictured by connecting up the wires:

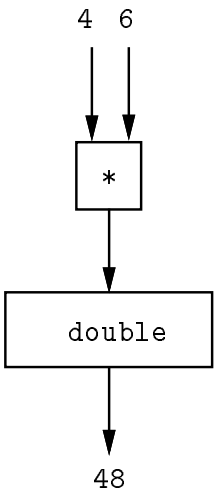


Figure 2.6

There is no restriction on the number of times this principle may be employed as long as the result and argument types of the various pairs of functions match.

If we use functional composition without explicit (that is, actual) arguments then the combination can be regarded as a new function (a *composition* of `double` and `*`), which we will call `doubleprod` (Figure 2.7). This new function

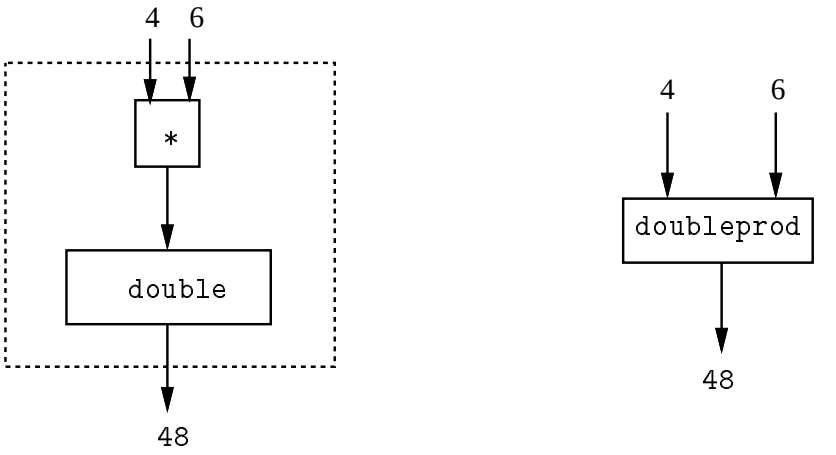


Figure 2.7

has the property that, for all numbers `x` and `y`,

```
doubleprod :: num -> num -> num
doubleprod x y = double(x*y)
```

As another example consider the following function, which computes the volume of a cylinder of height h and radius r (Figure 2.8). A cylinder is a particular kind of 'uniform object' whose volume we calculate by multiplying its height, which is h, by its base area, which we calculate using `areaofcircle`. Hence, assuming that `volume` and `areaofcircle` compute correctly (conform to their specifications), our function for the volume of a cylinder can be computed by

```
cylinderV :: num -> num -> num
cylinderV h r = volume h(areaofcircle r)
```

This is an example of *top-down* design. Ultimately, we want to implement

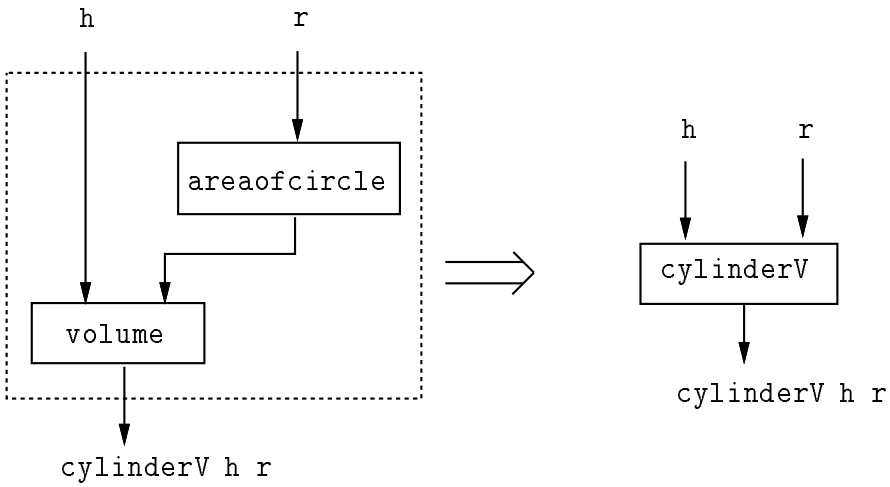


Figure 2.8

the high-level functions, the ones we really want, by building them up from the low-level, *primitive* functions, that are built-in to Miranda. But we can do this step by step from the top down, for instance by defining `cylinderV` using functions `volume` and `areaofcircle` that do not need to have been implemented yet, but do need to have been specified.

It can therefore be seen that black boxes (that is, functions) can be plugged together to build even bigger black boxes, and so on. The external, black box view of functions, which allows us to encapsulate the complicated internal plugging and concentrate on the specification, has an important impact on the cohesion of large programs. To see an example of this, suppose we had mistakenly defined

```
areaofcircle radius = pi*pi*radius
```

Of course, `cylinderV` will then give wrong answers. But we are none the less convinced that *the definition of* `cylinderV` *is correct*, and that is because our use of `areaofcircle` in it is based on the *specification* of `areaofcircle`, not on the erroneous definition. (`volume` asks for the base area, and `areaofcircle` is supposed to compute this.)

There may be lots of parts of a large program, all using `areaofcircle` correctly, and all giving wrong answers. As soon as `areaofcircle` has been corrected, all these problems will vanish.

On the other hand, someone might have been tempted to correct for the error by defining

```
cylinderV h r = volume h ((areaofcircle r)*r/pi)
```

This is a perfect recipe for writing code that is difficult to understand and debug: as soon as `areaofcircle` is corrected, `cylinderV` goes wrong. The rule is:

> When you use a function, rely on its *specification*, not its *definition*.

2.3 Some properties of functions

Functions map each combination of elements of the argument types to *at most one* element of the result type: when there is a result at all, then it is a well-defined, unique result. There *may* be argument combinations for which the result is not defined, and then we call the function *partial*. An example is `bitNegate`, which is undefined for all numbers other than 0 and 1 (Figure 2.9):

```
bitNegate ::  num -> num
bitNegate x = 1,   if x=0
            = 0,   if x=1
```

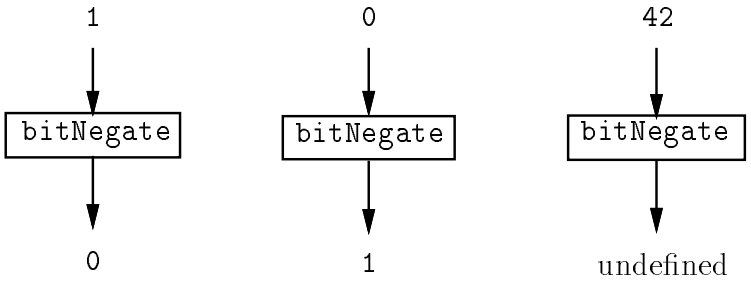


Figure 2.9

Similarly, division is a partial function and is said to be undefined for cases where its second argument is zero. A function that is not partial, one for which the result is always defined (at least for arguments of the right type), is called *total*.

Just as an illustration of some different possible behaviours of functions, here are two more kinds:

1. A function is *onto* if every value of the result type is a possible result of the function.
2. A function is *one-to-one* if two different combinations of arguments must lead to two different results.

For instance, `double` is one-to-one (if $x \neq y$ then `double` $x \neq$ `double` y) but not onto (for example, 3 is not a possible result because the results are all even). On the other hand, `volume` is onto (for example, any number z is a possible result because $z =$ `volume` z 1) but not one-to-one (for example, `volume` 2 3 = 6 = `volume` 3 2 — different argument combinations (2,3) and (3,2) lead to the same result).

2.4 Using a functional language evaluator

In order to construct a program in a functional language to solve a given problem one must define a function which solves the problem. If this definition involves other functions, then those must also be defined. Thus a functional program is just a collection of function definitions supplied by the programmer. To run a program one simply presents the functional language evaluator with an expression and it will do the rest. This expression can contain references to functions defined in the program as well as to built-in functions and constant values.

The functional language evaluator will have a number of *built-in* (or *primitive*) functions, together with their definitions: for example, the basic arithmetic functions `+`, `-`, `*`, `/` etc. The computer will evaluate your expression using your function definitions and those of its primitive functions and then print the result. Therefore, the computer just acts as a giant calculator.

Expressions that do not involve user-defined functions can be evaluated without using any program (just like a calculator). The evaluator, however, is more powerful than an ordinary calculator since you can introduce new function definitions in addition to those already built-in. Expressions can involve the name of these functions and are evaluated by using their definitions. This view of a functional language evaluator is illustrated in Figure 2.10.

2.5 Evaluation of expressions

When you present a functional evaluator with an expression, you can imagine it reducing the expression through a sequence of equivalent expressions to its 'simplest equivalent form' (or *normal* form), with no functions left to be

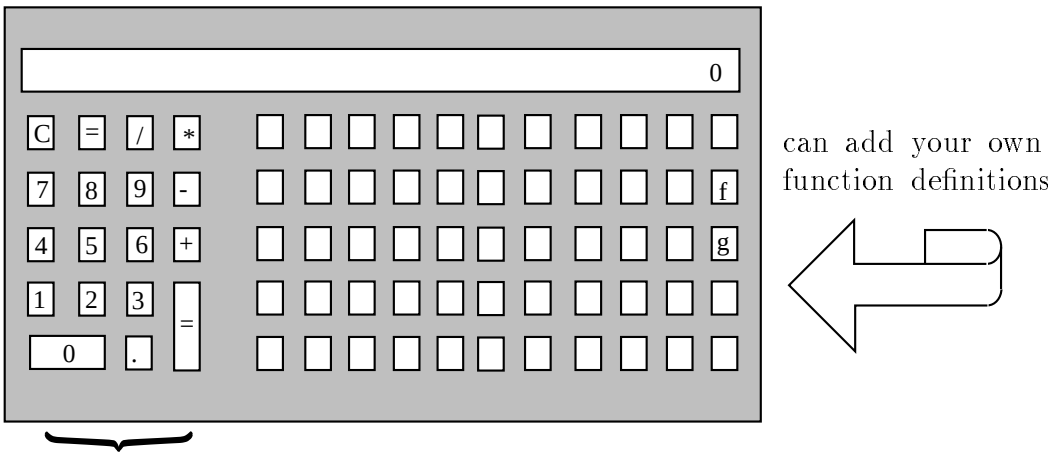


Figure 2.10 One view of a functional language evaluator

applied. This is the answer, which is then displayed. You can mimic this by 'hand evaluation', as in

```
double(3 + 4)
=   double 7              by built-in rules for +
=   7 + 7                 by the rule for double
=   14                    by built-in rules for +
```

`14` will be printed by the evaluator. (At each stage we have underlined the part that gets reduced next.) Other reduction sequences are possible, though of course they lead to the same answer in the end. Here is another one:

```
double (3 + 4)
=   (3 + 4) +(3 + 4)      by the rule for double
=   7 +(3 + 4)            by built-in rules for +
=   7 + 7                 by built-in rules for +
=   14                    by built-in rules for +
```

Thus evaluation is a simple process of *substitution* and *simplification*, using both primitive rules and rules (that is, definitions) supplied by the programmer. In order to simplify a function application, a *new copy* of the right-hand side of the function definition is created with each occurrence of the formal parameter replaced by a copy of the actual parameter. Function applications of the resulting expression are then simplified in the same manner until a normal form is reached.

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 much (if any) attention to the underlying implementation.

Some expressions do not represent well-defined values in the normal mathematical sense, for example any partial function applied to an argument for which it is undefined (for example, 6/0). When confronted with such expressions (that is, whose values are undefined), the computer may give an error message, or it may go into an infinitely long sequence of reductions and remain perpetually silent.

2.6 Notations for functions

So far we have seen functions in prefix and infix notations. In *prefix* notation the function symbol precedes its argument, as in `double 3` or `smaller x y`. *Infix* notation should also be familiar from school mathematics, where the function (also called operator) symbol appears between its arguments (also called operands), as in `2+6` or `x*y`.

In mathematics, $f(x,y)$ is written for the result of applying f to x and y. In Miranda, we can omit the parentheses and comma, and in fact it would be wrong to include them. Instead, we write `f x y` (with spaces) (Figure 2.11).

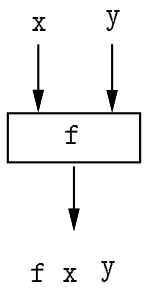


Figure 2.11

However, we cannot do without parentheses altogether, for we need them to package `f x y` as a single unit `(f x y)` when it is used within a larger expression. You can see this in

```
cylinderV h r = volume h(areaofcircle r)
```

Precedence

In expressions such as `(2+3*4)`, where there are several infix operators, it is ambiguous whether the expression is meant to represent `((2+3)*4)` or `(2+(3*4))`. Such ambiguities are resolved by a set of simple *precedence*

(priority) rules. For example, the above expression really means `(2+(3*4))` because, by long-standing convention, multiplication has a higher precedence relative to addition.

The purpose of precedence rules is to resolve possible ambiguity and to allow us to use fewer parentheses in expressions. Such rules of precedence will also be built-in to the evaluator to enable it to recognize the intended ordering. A hand evaluation example illustrating this is shown in Figure 2.12. Where necessary, the programmer can use extra parentheses to force a different order of grouping. For instance, `2*(3+4)*5 = 2*7*5 = 70`.

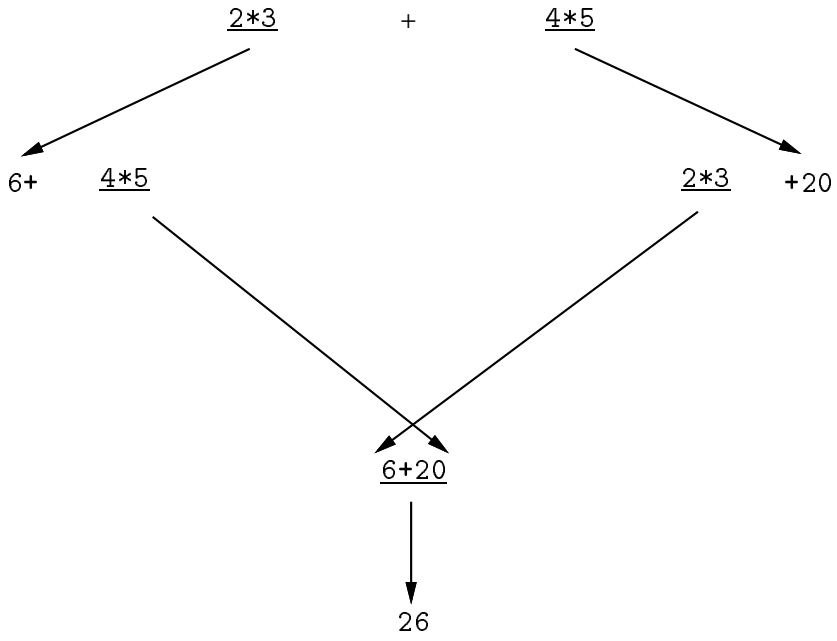


Figure 2.12 $2 * 3 + 4 * 5 = 6 + 20 = 26$

2.7 Meaning of expressions

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. For example, in the following evaluation sequence all expressions have the same value — the abstract integer value 30:

```
doubleprod 5 3 = double (5*3) = double 15 = 30
```

Note that an expression (in whatever form, even in its normal form) is not a value but, rather, a representation of it. There are many representations for