Chapter 4

Curried and Higher Order Functions

Chapter 2 has shown that functions are values in themselves and can be treated as data items (see Section 2.2). This chapter shows that functions may in fact be passed as parameters to other functions; a function that either takes another function as an argument or produces a function as a result is known as a *higher order function*.

Two further concepts are introduced in this chapter: *currying* and *function composition*. Currying (named in honour of the mathematician Haskell Curry) enables functions to be defined with multiple parameters without the need to use a tuple. A curried function also has the property that it does not have to be applied to all of its parameters at once; it can be *partially applied* to form a new function which may then be passed as a parameter to a higher order function. Function composition is used to chain together functions (which may themselves be partial applications) to form a new function.

Several powerful "families" of higher order functions on lists are shown; these functions can be combined and composed together to create new higher order functions. Higher order functions may be used to avoid the use of explicit recursion, thereby producing programs which are shorter, easier to read and easier to maintain.

4.1 Currying

It is sometimes tedious to use tuple notation to deal with functions that require more than one parameter. An alternative way of writing the function definition and application is to omit the tuple notation as shown in the definition for `get_nth` (which selects the nth item from a list, and is similar to the built-in `!` operator):[1]

[1]Note that the brackets around `(front : any)` are not tuple brackets but part of the list construction and must not be removed.

```
get_nth  any  []              =  error "get_nth"
get_nth  1    (front :  any)  =  front
get_nth  n    (front :  rest) =  get_nth (n - 1) rest
```

The type indication is new:

```
Miranda get_nth ::
num->[*]->*
```

This is different from previous responses for function definition, which had a single arrow in the type. In the curried definition shown above, there are two arrows which implies that two functions have been defined. These type arrows always associate to the right,[2] so that an equivalent type response is:

```
num->([*]->*)
```

This illustrates that `get_nth` is a function which has a number source type and generates an intermediate result which is itself a function; this second, but anonymous, function translates a polytype list into a polytype.

The presence of multiple arrows in the type indicates that `get_nth` is a *curried* function. This curried version of `get_nth` may now be applied to two arguments, without using tuple brackets, giving the desired result:

```
Miranda get_nth 2 ["a","bb","ccc"]
bb
```

In practice, any uncurried function could have been defined in a curried manner, regardless of the number of components in its argument. *From now on most functions will be presented in a curried form.*

4.1.1 Partially applied functions

Although currying allows a simpler syntax, its real advantage is that it provides the facility to define functions that can be *partially applied*. This is useful because it enables the creation of new functions as specializations of existing functions. For instance, the function `get_nth` may be used with just one argument to give rise to new functions:

```
get_second  =  get_nth 2
get_fifth   =  get_nth 5
```

```
Miranda get_second ::
[*]->*
```

[2]Remember that function *application* associates to the left.

```
Miranda get_fifth ::
[*]->*

Miranda get_second ["a","bb","ccc"]
bb
```

A hand evaluation of the above application of `get_second` shows:

```
get_second ["a","bb","c"]
==> (get_nth 2) ["a","bb","ccc"]
==> get_nth 2 ["a","bb","ccc"]
==> get_nth 1 ["bb","ccc"]
==> "bb"
```

The partial application of `(get_nth 2)` thus generates an intermediate function, waiting for application to a final argument in order to generate a result. A partial application may also be used as an actual parameter to another function; the expressive power of this feature will become evident later in this chapter during the discussion of *higher order functions*.

4.1.2 Partially applied operators

The Miranda arithmetic and relational operators all have an underlying infix format. This means that they cannot be partially applied. For example, both of the following definitions will fail:

```
wrong_inc = 1 +

wrong_inc = + 1
```

It is, of course, possible to define simple, prefix, curried functions which do the same as their operator equivalents and can then be partially applied:

```
plus ::  num -> num -> num
plus x y = x + y

inc = plus 1
```

As there are not many operators, this approach provides a brief, simple and pragmatic solution. An alternative approach is to use the Miranda *operator section* mechanism. With this approach *any* dyadic operator (that is, an operator taking two arguments) may be used in a prefix, curried manner by surrounding it with brackets.

For example, the following two expressions are equivalent:

```
1 + 2
(+) 1 2
```

The latter demonstrates firstly, that the syntactic form `(+)` is a prefix function and secondly, that it takes its two distinct arguments and hence is curried.

The following definitions further demonstrate the power of the bracketing notation:

```
inc = (+) 1

twice = (*) 2
```

The unrestricted use of operator sections can sometimes lead to cluttered code, and so it may be preferable to provide names for prefix, curried versions of the operators. Using these names will lead to code that is longer, but more comprehensible. The following definitions provide suggested names which may be used (where appropriate) in subsequent examples throughout the book:

```
plus              = (+)
minus             = (-)
times             = (*)
divide            = (/)

both              = (&)
either            = (\/)

append            = (++)
cons              = (:)

equal             = (=)
notequal          = (~=)

isitgreaterthan   = (<)
greaterthan       = (>)

isitlessthan      = (>)
lessthan          = (<)
```

Notice that number comparison has two variants, for example `isitlessthan` and `lessthan`.[3] These two functions are quite different:

1. `isitlessthan` is normally used in a partially applied manner to provide a predicate, for example:

[3]The following discussion applies equally to the functions `isitgreaterthan` and `greaterthan`.

```
is_negative ::  num -> bool
is_negative = isitlessthan 0
```

```
Miranda is_negative (-1)
True
```

2. `lessthan` is normally used as a prefix replacement for the infix operator `<`:

```
Miranda lessthan 0 (-1)
False
```

   The partial application `(lessthan 0)` is interpreted as "is 0 less than some integer?".

This kind of distinction applies to all *non-commutative* operators.

In general, Miranda allows both *presections* (for example, `(1+)`) and *postsections* (for example, `(+1)`). There is one important exception: it is not possible to define a postsection for the subtraction operator. This is because `(-1)` already has a special meaning—minus one!

4.2 Simple higher order functions

Higher order functions are a powerful extension to the function concept and are as easy to define and use as any other function. The rest of this section shows a number of simple higher order functions whilst the next section extends the principle to higher order functions over lists.

4.2.1 Function composition

The built-in operator `.` (pronounced "compose") is different to previously shown built-in operators in that it takes two functions as its parameters (and hence is also a higher order function). A frequent programming practice is to apply a function to the result of another function application. For instance, using the function `twice` (defined in the previous section):

```
quad x = twice (twice x)

many x = twice (twice (twice (twice x)))
```

In this sort of function definition, the use of bracket pairs is tedious and can lead to errors if a bracket is either misplaced or forgotten. The operator `.` enables most of the brackets to be replaced:

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

many x = (twice .  twice .  twice .  twice) x
```

Not only is this notation easier to read but it also emphasizes the way the functions are combined. The outermost brackets are still necessary because `.` has a lower precedence than function application. The compose operator is specified by *(f . g) x = f (g x)*, where the source type of the function *f* must be the same as the result type of the function *g*.

Function composition provides a further advantage beyond mere "syntactic sugaring", in that it allows two functions to be combined and treated as a single function. As shown in Chapter 1, the following intuitive attempt at naming `many` fails because `twice` expects a number argument rather than a function translating a number to a number:

```
wrong_many = twice twice twice twice
```

```
type error in definition of wrong_many
cannot unify num->num with num
```

The correct version uses function composition:

```
many ::  num -> num
many = twice .  twice .  twice .  twice
```

```
Miranda many 3
48
```

The use of `.` is not limited to combining several instances of the same function (which itself must have identical source and target types); it can be used to combine any pair of functions that satisfy the specification given above. For example:

```
sqrt_many = sqrt .  many

sqrt_dbl = sqrt .  (plus 1) .  twice
```

Note that in the final example `(plus 1)` is a monadic function (it takes just one parameter); the use of `plus` on its own is inappropriate for composition because it is dyadic.

Composing operators

If an operator is to be used in a function composition then it must be used in its sectional form. For example:

```
Miranda ((+ 2) . (* 3)) 3
11
```

This rule also applies to ~ and #, which must be treated as prefix operators rather than functions:

```
Miranda ((~) . (< 3)) 4
True

Miranda ((#) . tl) [1,2,3]
2
```

Exercise 4.1
Give the types of the following compositions:

```
tl . (++ [])
abs . fst
code . code
show . tl
```

4.2.2 Combinatorial functions

The following functions are similar to . in that they facilitate the use of higher order functions. These functions are known as *combinators* (Diller, 1988; Field and Harrison, 1988; Revesz, 1988; Turner, 1979) and, traditionally, are given single upper case letter names (which is not possible under Miranda naming rules). This text only introduces versions of the combinators B, C and K, which are of general utility. There are many more possible combinators which have mainly theoretical interest and serve as the basis for many implementations of functional languages. For more information the reader is referred to the Bibliography.

Compose

The B combinator is the prefix equivalent of . and is defined below as the function "compose":

```
compose = (.)  || B combinator
```

The function compose can now be used in a similar way and with the same advantages as its built-in equivalent.

Swap

The `C` combinator is known as "swap" and serves to exchange the arguments of a (curried) dyadic function:

```
swap ff x y = ff y x || C combinator
```

Note that the Miranda Standard Environment provides the function `converse`, which has the same functionality.

Cancel

The `K` combinator is known as "cancel" because it always discards the second of its two arguments and returns its first argument:

```
cancel x y = x || K combinator
```

Note that the Miranda Standard Environment provides the function `const`, which has the same functionality as `cancel`.

Exercise 4.2

Theoreticians claim that all of the combinators (and consequently all functions) can be written in terms of the combinator K (`cancel`) and the following combinator `S` (`distribute`):

```
distribute f g x = f x (g x)
```

Define the combinator `identity` (which returns its argument unaltered) using only the functions `distribute` and `cancel` in the function body. Provide a similar definition for a curried version of `snd`.

4.2.3 Converting uncurried functions to curried form

The user-defined functions presented in the first three chapters of this book have been defined in an uncurried format. This is a sensible format when the functions are intended to be used with all of their arguments, but it is not as flexible as the curried format. If curried versions are required then it would be possible to rewrite each function using curried notation; however, this would involve a lot of programmer effort (and consequently would be prone to programmer error). A more pragmatic approach is to write a function that will generate curried versions of uncurried functions—this function could be written once, then used as and when necessary.

This section presents a function, `make_curried`, which will convert any uncurried, dyadic function (that is, a function which takes a tuple with two components) to curried format. The programmer can then use this function as a template for further conversion functions as required.

The conversion function `make_curried` is itself in curried form and takes three parameters. The first parameter represents the uncurried function and the next two parameters represent that function's arguments. All that the function body needs to do is apply the input function to these last two parameters collected into a tuple (since an uncurried function only works on a single argument):

```
make_curried ::  ((*,**) -> ***) -> * -> ** -> ***
make_curried ff x y = ff (x,y)
```

Now, given the definition:

```
maxnum ::  (num,num)->num
maxnum (x, y) = x, if x > y
              = y, otherwise
```

then clearly the application `maxnum 1 2` will fail, as the function `maxnum` expects a number pair. Using `make_curried` gets around this problem:[4]

```
make_curried maxnum 2 3
==> maxnum (2, 3)
==> 3
```

Similarly, new curried versions of existing functions can be created with the minimum of programmer effort:

```
newmaxnum = make_curried maxnum
```

The function `make_curried` is another example of a higher order function because it expects a function as one of its parameters. In general, any function that takes a function as at least one of its parameters or returns a function, as its result is known as a higher order function.

Exercise 4.3

Explain why `make_curried` cannot be generalized to work for functions with an arbitrary number of tuple components.

Exercise 4.4

Write the function `make_uncurried` which will allow a curried, dyadic function to accept a tuple as its argument.

[4]This is an excellent example of the fact that function application associates from the left, to give `((make_curried maxnum) 2) 3` rather than `make_curried (maxnum (2 3))` which would be an error.

Exercise 4.5
The built-in function `fst` can be written using `make_uncurried` and the `cancel` combinator:

```
myfst = make_uncurried cancel
```

Provide a similar definition for the built-in function `snd`.

4.2.4 Iteration

An important advantage of the functional approach is that the programmer can create a rich set of iterative control structures and hence be more likely to choose one which represents the problem specification. This subsection illustrates this point in the definition of a "repeat" loop construct. This function is *not* similar to the `repeat` function available in the Miranda Standard Environment—it is, however, similar to the `iterate` function.[5]

The following (non-robust) function repeatedly applies its second parameter to its final parameter, which serves as an *accumulator* for the final result. The parameter of recursion is `n` which converges towards zero:

```
myiterate ::  num -> (* -> *) -> * -> *
myiterate 0 ff state = state
myiterate n ff state = myiterate (n-1) ff (ff state)
```

The function `myiterate` can be used to give a non-recursive definition of any function that bases its recursion on a fixed number of iterations. For example, the function `printdots` (from Chapter 2) may be defined as:

```
printdots n = myiterate n ((++) ".") ""
```

In `printdots` the empty string `""` is the initial value of the `state` parameter (or accumulator), which changes at each recursive step. A hand evaluation of `(printdots 2)` reveals:

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

[5]The function could alternatively be defined as `myiterate n ff state = (iterate ff state) !  n`.

If the function to be repeated also requires the iteration count as a parameter, the following variant `myiterate_c` may be used (assuming the iteration counter counts down towards zero):

```
myiterate_c ::  num -> (* -> num -> *) -> * -> *
myiterate_c 0 ff state = state
myiterate_c n ff state
        = myiterate_c (n - 1) ff (ff state n)
```

Exercise 4.6

Explain why `myiterate` is non-robust and provide a robust version.

Exercise 4.7

Imperative programming languages generally have a general-purpose iterative control structure known as a "while" loop. This construct will repeatedly apply a function to a variable whilst the variable satisfies some predefined condition. Define an equivalent function in Miranda.

4.3 Higher order functions on lists

Many of the list-handling functions presented in Chapter 3 exhibit similar forms of recursion but use different operations to achieve their results. Miranda provides the facility to generalize these functions and removes the need to program with explicit recursion.

This section shows three families of curried, polymorphic, higher order functions on lists:

1. The "map" family, which retains the list structure but transforms the list items.
2. The "fold" family, which distributes an operator over a list, typically to produce a single value result.
3. The "select" family, which retains the list structure but may delete items from the list, according to a given predicate.

4.3.1 The map family

It is often useful to apply a function to each item in a list, returning a list of the consequences. For example, `map_inc` will apply `inc` to each item in a given number list and `map_twice` will apply `twice` to each item in a given number list:

```
inc = (+ 1)
twice = (* 2)

map_inc []                 = []
map_inc (front :  rest)    = (inc front) :  (map_inc rest)

map_twice []               = []
map_twice (front :  rest)  = (twice front) :  (map_twice rest)
```

A template for any function of this form is:
map_ff [] = []
map_ff (front : rest) = (ff front) : (map_ff rest)
It can be seen from this template that the only important difference between `map_inc` and `map_twice` is the name of the function that is applied to the `front` item (represented by *ff* in the template). Rather than having to define functions of this form using explicit recursion, it is possible to define them using the built-in higher order function `map`. This function has *ff* as its first argument and the list to be transformed as its second argument:

```
Miranda map inc [1,3,2,6]
[2,4,3,7]
```

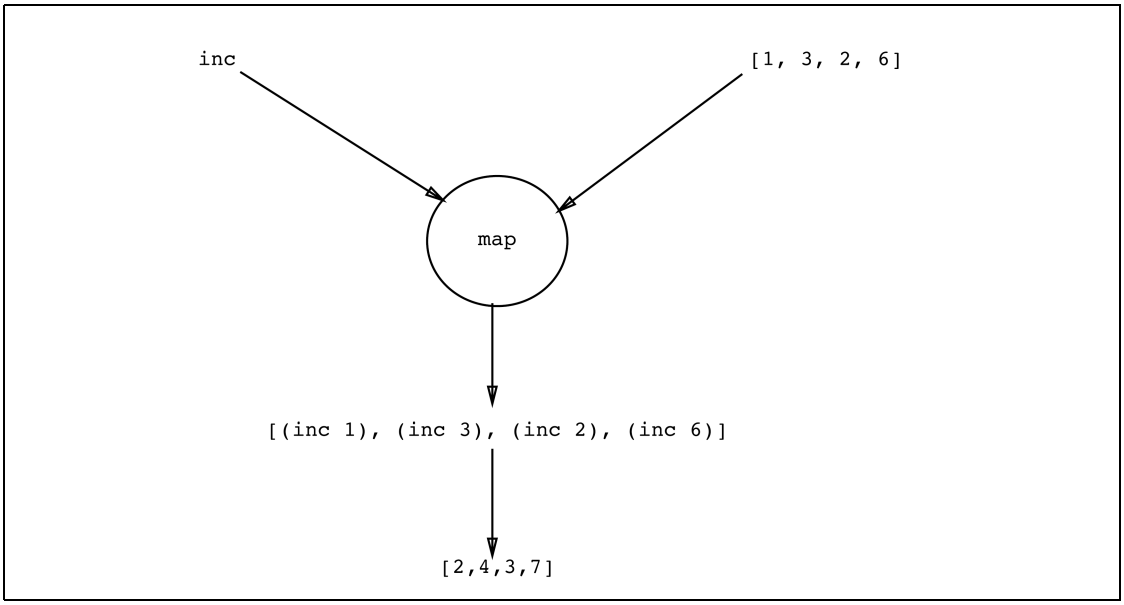

Figure 4.1 The behaviour of the function `map`.

The behaviour of the function `map` is further illustrated in Figure 4.1 and the following examples:

```
Miranda map (+ 1) [1,2,3]
[2,3,4]

Miranda map (twice . inc) [1,2,3]
[4,6,8]

Miranda map (plus 2) [1,2,3]
[3,4,5]
```

```
list_inc  = map inc
```

```
Miranda list_inc ::
[num] -> [num]

Miranda list_inc [1,2,3]
[2,3,4]
```

It is important to note that `(map inc)` is a partially applied function (where `inc` is a function argument to `map`) and is *not* a function composition:

```
Miranda (map inc) ::
[num] ->[num]

Miranda (map . inc) ::
cannot unify num->num with num->*->**
```

Furthermore, as in all other situations, the arguments to and the results produced by `map` are evaluated lazily:

```
Miranda map (+ 1) [1,2,3]
[2,3,4]

Miranda map (/ 0) [1,2,3]
[
program error: attempt to divide by zero

Miranda fst (3, map (/ 0) [1,2,3])
3

Miranda (hd.tl) (map (3 /) [1,0,3])
program error: attempt to divide by zero

Miranda hd (map (3 /) [1,0,3])
3.0
```

Designing map

In fact, `map` is very easy to write. All that is necessary is to generalize from the *map_ff* template by passing `ff` as an additional argument:

```
map ff []                = []
map ff (front :  rest)   = (ff front) :  (map ff rest)
```

The function `map` is polymorphic, as can be seen from its type:

```
Miranda map ::
(*->**)->[*]->[**]
```

The function can transform a list of any type provided that the source type of `ff` matches the type of the items in the list.

Mapping over two lists

The principle of list transformation using `map` can be extended to cater for more than one list. For example, the following function takes two lists and recursively applies a dyadic function to the corresponding front items:

```
map_two ::  (*->*->**)->[*]->[*]->[**]
map_two  ff [] [] = []
map_two  ff (front1 :  rest1) (front2 :  rest2)
     =  (ff front1 front2) :  (map_two ff rest1 rest2)
map_two  ff any1 any2
     =  error "map_two:  lists of unequal length"
```

```
Miranda map_two (make_curried max) [1,2,3] [3,2,1]
[3,2,3]
```

Exercise 4.8
In the definition of `map_two`, source lists of unequal length have been treated as an error. It is an equally valid design decision to truncate the longer list; amend the definition to meet this revised specification.

Exercise 4.9

Write a function `applylist` which takes a list of functions (each of type `*->**`) and an item (of type `*`) and returns a list of the results of applying each function to the item. For example: `applylist [(+ 10),(* 3)] 2` will evaluate to `[12,6]`.

Exercise 4.10
Explain why the following definitions are equivalent:

```
f1 x alist = map (plus x) alist
f2 x = map (plus x)
f3 = (map . plus)
```

4.3.2 List folding—reduce and accumulate

This subsection discusses how a dyadic function can be distributed so that it works over a list, typically evaluating to a single value, for example to give the sum of all the numbers in a list. As with the discussion of `map`, it will be shown how explicit recursion can be removed by means of a single higher order function. Two strategies are used:

1. Stack recursion, to define a function called `reduce` (known in the Miranda Standard Environment as `foldr`).
2. Accumulative recursion, to define a function called `accumulate` (known in the Miranda Standard Environment as `foldl`).

The higher order function reduce

On inspecting the structure of the following definitions of `sumlist`, `divall` and `anytrue`, it can be seen that they share a common structure:

```
sumlist []                = 0
sumlist (front :  rest)   = front  +   (sumlist rest)

divall []                 = 1.0
divall (front :  rest)    = front  /   (divall rest)

anytrue []                = False
anytrue (front :  rest)   = front  \/  (anytrue rest)
```

Functions of this form place the dyadic operator between each of the list items and substitute a terminating value for the empty list. For example, the sum of the list [1,2,5,2] may be thought of as the result of 1 + 2 + 5 + 2 + 0. Generalizing gives the template:

reduce_ff *[]* = *default*
reduce_ff *(front : rest)* = *(ff front)* $\oplus$ *(reduce_ff rest)*

This template is appropriate for all dyadic infix functions $\oplus$. In the above examples, the default value has been chosen such that the following specification holds:

$$any \oplus default = any$$

When the default value has this property, it is formally known as the *Identity* element of the dyadic function:

Table 4.1 Identity elements of common dyadic functions.

any	$\oplus$	*Identity*	=	*any*
`any_int`	`+`	`0`	=	`any_num`
`any_bool`	`\/`	`False`	=	`any_bool`
`any_string`	`++`	`""`	=	`any_string`
`any_list`	`++`	`[]`	=	`any_list`

Notice it is not always necessary to choose the identity as the default. For example:

```
product_times_ten [] = 10
product_times_ten (front :  rest)
     = front * (product_times_ten rest)
```

Furthermore, many dyadic functions do not have an identity and so great care must be taken in the choice of a sensible default value. For example, it is not obvious what would be a sensible default value for the operator `mod`.

Miranda provides a built-in higher order function called `foldr` to generalize *reduce_ff*. However, a user-defined version (here called `reduce`) can also be written, using a similar approach to that taken with the design of `map`.

Designing reduce

The above examples considered the distribution of built-in infix operators over lists. However, `reduce` will be designed to accept prefix functions, mainly because user-defined functions are normally defined in prefix form. The design proceeds by replacing $\oplus$ with a prefix version `ff` in the template:

reduce_ff [] = default
reduce_ff (front : rest) = (ff front) (reduce_ff rest)

All that is now necessary is to make *ff* and *default* become explicit arguments:

```
reduce  ::  (* -> ** -> **) -> ** -> [*] -> **

reduce  ff default [] = default
reduce  ff default (front :  rest)
     =  ff front (reduce ff default rest)
```

Examples of partial applications which use `reduce` are now presented (employing the curried functions defined in Section 4.1.2):

```
anytrue            = reduce either False
sumlist            = reduce plus 0
divall             = reduce divide 1.0
product_times_ten  = reduce times 10
```

```
Miranda reduce plus 0 [1,2,3]
6

Miranda sumlist [1,3,5]
9
```

A hand evaluation of the last application shows:

```
sumlist [1,3,5]
==> reduce (+) 0 [1,3,5]
==> (+) 1 (reduce (+) 0 [3,5])
==> (+) 1 ((+) 3 (reduce (+) 0 [5]))
==> (+) 1 ((+) 3 ((+) 5 (reduce (+) 0 [])))
==> (+) 1 ((+) 3 ((+) 5 0))
==> (+) 1 ((+) 3 5)
==> (+) 1 8
==> 9
```

The function `reduce` is stack recursive, in that it stacks a growing unevaluated expression until the empty list is encountered. At this point it unstacks from the innermost right to the outermost left, combining them pairwise by means of application of the function parameter `ff`. In the unstacking phase, this can be seen as "folding" the list from the right, usually into a single value; hence the alternative name, `foldr`, which is used by the equivalent Miranda Standard Environment function.

The function `reduce` is not restricted to a single value result; it can also return an aggregate type, as illustrated in the following example:

```
do_nothing ::  [*] -> [*]
do_nothing = reduce (:)  []
```

```
Miranda do_nothing [1,5,8]
[1,5,8]
```

A hand evaluation shows how this works:

```
do_nothing [1,5,8]
==> reduce (:) [] [1,5,8]
==> (:) 1 (reduce (:) [] [5,8])
==> (:) 1 ((:) 5 (reduce (:) [] [8]))
==> (:) 1 ((:) 5 ((:) 8 (reduce (:) [] [])))
==> (:) 1 ((:) 5 ((:) 8 [] ))
==> (:) 1 ((:) 5 [8])
==> (:) 1 [5,8]
==> [1,5,8]
```

The overall result is an aggregate type because the dyadic function being distributed (in this case `(:)`) returns an aggregate type. Notice that the target type of the function being distributed must be the same as the source type of its second argument (that is, it must have type `* -> ** -> **`). Also notice that `(:)` has no identity—however, `[]` was chosen as the sensible default value because it produces a list as required.

It is also possible to use `reduce` to distribute partial applications and function compositions across lists, as demonstrated below:

```
addup_greaterthan ::  num -> num -> num -> num
addup_greaterthan x y z  = y + z, if x < y
                         = z, otherwise
```

```
Miranda reduce (addup_greaterthan 3) 0 [1,7,3,9,8,4,1]
28

Miranda reduce ((:) . inc) [] [1,2,3]
[2,3,4] : int list
```

The final example above is particularly interesting as it has the same action as `map inc [1,2,3]`. A hand evaluation shows how it works:

```
reduce ((:) . inc) [] [1,2,3]
==> ((:) . inc) 1 (reduce ((:) . inc) [] [2,3])
==> (:) (inc 1) (reduce ((:) . inc) [] [2,3])
==> (:) 2 (reduce ((:) . inc) [] [2,3])
==> (:) 2 (((:) . inc) 2 (reduce ((:) . inc) [] [3]))
==> (:) 2 ((:) (inc 2) (reduce ((:) . inc) [] [3]))
==> (:) 2 ((:) 3 (reduce ((:) . inc) [] [3]))
==> (:) 2 ((:) 3 (((:) . inc) 3 (reduce ((:) . inc) [] [])))
==> (:) 2 ((:) 3 ((:) (inc 3) (reduce ((:) . inc) [] [])))
==> (:) 2 ((:) 3 ((:) 4 (reduce ((:) . inc) [] [])))
==> (:) 2 ((:) 3 ((:) 4 []))
==> (:) 2 ((:) 3 [4])
==> (:) 2 [3,4]
==> [2,3,4]
```

Exercise 4.11
Rewrite the function `map` in terms of `reduce`.

The higher order function accumulate

It is possible to define a very similar function to `reduce` that makes use of the accumulative style of recursion. Miranda provides the built-in higher order function `foldl`, which starts evaluating immediately by recursing on the tail of the list with the default value being used as the accumulator. A user-defined version called `accumulate` is given below:

```
accumulate  ::  (*->**->*) -> * -> [**] -> *

accumulate  ff default [] = default
accumulate  ff default (front :  rest)
         =  accumulate ff (ff default front) rest
```

Equivalence of reduce and accumulate

On many occasions `reduce` can be substituted with `accumulate`, as shown in the next three examples which rework the previous `reduce` examples. However, this is not always the case, as will be demonstrated in the subsequent treatment for `reverse`.

```
anytrue           = accumulate either  False
sumlist           = accumulate plus    0
product_times_ten  = accumulate times   10
```

A hand evaluation of `sumlist [1,3,5]` reveals that the same answer is obtained as for the `reduce` version, but in a very different manner:

```
sumlist [1,3,5]
==> accumulate (+) 0 [1,3,5]
==> accumulate (+) ((+) 0 1) [3,5]
==> accumulate (+) ((+) ((+) 0 1) 3) [5]
==> accumulate (+) ((+) ((+) ((+) 0 1) 3) 5) []
==> ((+) ((+) ((+) 0 1) 3) 5)
==> ((+) ((+) 1 3) 5)
==> ((+) 4 5)
==> 9
```

By contrast, it is not possible to substitute `accumulate` for `reduce` in the `divall` example, nor `reduce` for `accumulate` in the following definition of `myreverse`:

```
myreverse = accumulate (swap (:))  []
```

Note that the above user-defined version is directly equivalent to the definition of the built-in function `reverse`, as presented in Section 28 of the Miranda on-line manual:

```
reverse = foldl (converse (:)) []
```

Comparing reduce and accumulate

The reason why `reduce` cannot be substituted for `accumulate` in the above definition of `reverse` is best illustrated through a diagrammatic comparison of the two functions. The following example compares the distribution of the infix `+` operator across the list `[1,2,3]`, with a default value of `0`. If the `reduce` function is used then this may be considered diagrammatically as placing the default value at the right-hand end of the list and bracketing the expression from the right (hence the alternative name `foldr`):

$$\begin{array}{ccccccc} 1 & : & 2 & : & 3 & : & [\,] \\ \downarrow & & \downarrow & & \downarrow & & \\ 1 & + & (2 & + & (3 & + & 0)) \end{array}$$

By contrast, if the `accumulate` function is used then this may be considered diagrammatically to place the default value at the left-hand end and bracketing the expression from the left (hence the alternative name `foldl`):

$$\begin{array}{ccccccc} & 1 & : & 2 & : & 3 & : & [\,] \\ & \downarrow & & \downarrow & & \downarrow & & \\ ((0 + & 1) & + & 2) & + & 3 & & \end{array}$$

Of course, `reduce` and `accumulate` are defined to take curried, prefix functions rather than infix operators and so the actual diagrams would be slightly more complex. For example, `(reduce (+) 0 [1,2,3])` would produce `((+) 1 ((+) 2 ((+) 3 0)))` and `(accumulate (+) 0 [1,2,3])` would produce `((+) ((+) ((+) 0 1) 2) 3)`. However, the above two diagrams provide a better visual mnemonic.

In general, it is only safe to substitute one of the *fold* functions with the other if the function parameter `ff` is associative and also commutative (at least with its identity). As explained in Chapter 1, an infix operator $\oplus$ is associative if the following holds for all possible values of x, y and z:

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

Similarly, a prefix function *ff* is associative if the following holds for all possible values of x, y and z:

ff x (ff y z) = ff (ff x y) z

A prefix function is commutative with its identity value if the following holds for all possible values of x:

ff Identity x = ff x Identity

It can now be seen that substituting `reduce` for `accumulate` in the definition of `reverse` will fail because both of the two conditions given above are violated. The function `(swap (:))` is neither associative nor does it have an identity value.

The relevance of these two criteria can also be illustrated diagrammatically. By using the two rules of *associativity* and *commutativity with the identity*, and by reference to the diagrams used above, it is possible to transform the diagram for `reduce (+) 0 [1,2,3]` into the diagram for `accumulate (+) 0 [1,2,3]` (once again infix form is used in the diagram for clarity):

		1 + (2 + (3 + 0))
by rule 1 =		*(1 + 2) + (3 + 0)*
by rule 1 =		*((1 + 2) + 3) + 0*
by rule 2 =	*0 +*	*((1 + 2) + 3)*
by rule 1 =	*(0 +*	*(1 + 2))+ 3*
by rule 1 =	*((0 +*	*1)+ 2) + 3*

Exercise 4.12

Some functions cannot be generalized over lists, as they have no obvious default value for the empty list; for example, it does not make sense to take the maximum value of an empty list. Write the function `reduce1` to cater for functions that require at least one list item.

Exercise 4.13

Write two curried versions of `mymember` (as specified in Chapter 3.6), using `reduce` and `accumulate`, respectively, and discuss their types and differences.

4.3.3 List selection

There are a large number of possible list selection functions, which remove items from a list if they do not satisfy a given predicate (that is, a function that translates its argument to a Boolean value). This subsection presents two functions which are typical of this family of functions.

List truncation

The following function takes a list and a predicate as arguments and returns the initial sublist of the list whose members all satisfy the predicate:

```
mytakewhile  ::  (* -> bool) -> [*] -> [*]

mytakewhile  pred [] = []
mytakewhile  pred (front :  rest)
          =  front :  (mytakewhile pred rest), if pred front
          =  [], otherwise
```

Example:

```
Miranda mytakewhile (notequal ' ') "how_long is a string"
how_long
```

This function behaves in the same manner as the Miranda Standard Environment `takewhile` function.

List filter

The next example is the function `myfilter` which uses a predicate that is based on the item's value. This function may be specified in terms of filter recursion, as shown in Section 3.8:

```
myfilter  ::  (* -> bool) -> [*] -> [*]
myfilter  pred [] = []
myfilter  pred (front :  rest)
       =  front :  (myfilter pred rest), if pred front
       =  myfilter pred rest, otherwise
```

Examples:

```
Miranda myfilter (isitlessthan 3) [1,7,2,9,67,3]
[1,2]
```

```
rm_dups ::  [*] -> [*]
rm_dups [] = []
rm_dups (front :  rest)
   = front :  (rm_dups (myfilter (notequal front) rest))
```

This function behaves in the same manner as the Miranda Standard Environment `filter` function.

Grep revisited

Using `filter`, the *grep* program shown in Section 3.9 can now be extended to mirror the UNIX *grep* behaviour, so that only those lines which match the regular expression are printed from an input stream. It should be noted that the code is presented in curried and partially applied form and assumes that `sublist` is also curried:

```
string == [char]

grep ::  string -> [string] -> [string]
grep regexp = filter (xgrep regexp)

xgrep ::  string -> string -> bool
xgrep regexp line = sublist (lex regexp) line
```

Exercise 4.14

Define the function `mydropwhile` which takes a list and a predicate as arguments and returns the list without the initial sublist of members which satisfy the predicate.

Exercise 4.15

The *set* data structure may be considered as an unordered list of unique items. Using the built-in functions `filter` and `member`, the following function will yield a list of all the items common to two sets:

```
intersection ::  [*] -> [*] -> [*]
intersection aset bset = filter (member aset) bset
```

Write a function `union` to create a set of all the items in two sets.

4.4 Program design with higher order functions

This section shows how programming with higher order functions leads to more general purpose programs. Higher order functions eliminate explicit recursion and so lead to programs that are more concise and often nearer to the natural specification of a problem.

4.4.1 Making functions more flexible

Many functions can be made more general by substituting explicit predicate functions with a parameter; the decision as to which predicate is actually employed is

thereby deferred until the function is applied. In the following example, *insertion sort* (presented in Section 3.7.3) is generalized so that it will sort a list of any type in either ascending or descending order. The only alteration that is required to the original specification is to substitute a comparison function in place of the infix < operator for `insert` and `isort`. The following code reworks the example presented in Section 3.7.3 in curried form:

```
ordertype * == * -> * -> bool
isort ::  ordertype * -> [*] -> [*]
isort order anylist
     =  xsort order anylist []

xsort ::  ordertype * -> [*] -> [*] -> [*]
xsort order [] sortedlist
     =  sortedlist
xsort order (front :  rest) sortedlist
     =  xsort order rest (insert order front sortedlist)

insert  ::  ordertype * -> * -> [*] -> [*]
insert  order item []
     =  [item]
insert  order item (front :  rest)
     =  item :  front :  rest, if order item front
     =  front :  (insert order item rest), otherwise
```

The extra parameter `order` provides the comparison function, whilst also dictating the type of the list. In this manner, the functionality of `isort` has been increased significantly since a different `order` can be slotted in place by a simple partial application:

```
desc_sort ::  [*] -> [*]
desc_sort = isort greaterthan

stringsort ::  [char] -> [char]
stringsort = isort lessthan
```

Notice that the type declaration allows the overloaded relational operators to be truly polymorphic (as with `desc_sort`) or forces them to be monomorphic (as with `stringsort`).

As a bonus, it is possible to remove some of the explicit recursion from these function definitions by replacing the explicitly defined `isort` with `reduce` as follows:

```
isort order = reduce (insert order) []
```

Exercise 4.16

An equivalent version of `stringsort` using `accumulate` would require that the arguments to `(insert lessthan)` be reversed. Why is this the case?

4.4.2 Combining higher order functions

This subsection briefly looks at some of the many ways of combining higher order functions.

Example—combining accumulate and map

The following is a simple example using `map` and `accumulate` to convert a string to an integer:

```
string_to_int ::  [char] -> num
string_to_int astring
  = accumulate (plus .  (times 10)) 0 (map ctoi astring)

ctoi ::  char -> num
ctoi x = (code x) - (code '0')
```

This may be rewritten, using further function composition, as follows:

```
string_to_int ::  [char] -> num
string_to_int astring
  = ((accumulate (plus .  (times 10)) 0) .  (map ctoi))
                                              astring

ctoi ::  char -> num
ctoi x = (code x) - (code '0')
```

Notice that for any definition $f\ x$ = *expression* x, where *expression* contains no reference to x, then f and *expression* must be exactly equivalent (they both operate on x to produce a result and the two results are equal by definition). Thus, the given definition can be shortened to one which simply states that f and *expression* are equivalent: f = *expression*. This does not alter the type of f; it is still a function of one argument. This optimization may be applied to the above definition of `string_to_int` as follows:

```
string_to_int ::  [char] -> num
string_to_int
  = (accumulate (plus .  (times 10)) 0) .  (map ctoi)

ctoi ::  char -> num
ctoi x = (code x) - (code '0')
```

This example reinforces the observation that function composition is the equivalent of sequencing in an imperative style of programming, though here the program sequence should be read from the rightmost composition towards the leftmost.

Example—combining map and filter

The functions `filter` and `map` may be combined to give a function which takes a list of items, each of type `(student, grade)`, and outputs the names of all students with a grade less than 50%:

```
student  == [char]
grade    == num
results  == [(student, grade)]
mapping  == results -> [student]

weak_students ::  mapping
weak_students
      =  (map fst) .  filter ((isitlessthan 50) .  snd)
```

Different approaches to combining functions

There are a number of approaches to combining functions and higher order functions; to illustrate this point, two of the many possible versions of the function `length` are now shown.

The first version has an underlying design that considers the length of a list as the sum of a list which has had all of its items transformed into the number 1. Hence the implementation must *first* transform each of its argument's list elements into a 1 and *second* perform the summation. This can be written using `accumulate`, `map` and the combinator `cancel` as follows:

```
length = (accumulate (+) 0) .  (map (cancel 1))
```

An alternative design is to think in terms of function composition and consider the length of a list as the *ongoing* summation of each list element transformed into the number 1:

```
length = reduce ((+) .  (cancel 1)) 0
```

There are no rigid rules concerning which style to use. The best advice is to choose the definition which most closely mirrors the natural specification; however this will differ from problem to problem and from program designer to program designer. Probably the biggest danger is to be tempted to go too far with the facilities shown in this chapter:

```
guesswhat  = (foldr (+) 0).(foldr ((:).((#).(:[]))) [])
```

Exercise 4.17

A function `foldiftrue` which `reduce`s only those elements of a list which satisfy a given predicate could be defined as:

```
foldiftrue :: (* -> bool) -> (* -> ** -> **) -> ** -> [*] -> **
foldiftrue pred ff default []
    = default
foldiftrue pred ff default (front : rest)
    = ff front (foldiftrue pred ff default rest), if pred front
    = foldiftrue pred ff default rest, otherwise
```

Write this function in terms of a composition of `reduce` and `filter`.

Exercise 4.18

What is the purpose of the function `guesswhat`?

4.5 Summary

Much of the power of Miranda derives from its ability to treat functions themselves as objects to be manipulated as easily as single values and data structures; a function that either takes a function as an argument or returns a function as its result is known as a *higher order function*.

All the functions introduced in this chapter are higher order functions. They make extensive use of a functional programming language feature known as *currying*; this technique allows the programmer to express the *partial application* of a function to less than all of its arguments. Since a partial application is itself a function, this provides a simple and uniform mechanism for treating functions as values; they may be passed as arguments to other functions and they may be returned as the result of a function.

In order to emphasize the treatment of functions as values, a number of *combinators* have been introduced. These combinators are higher order functions that manipulate other functions. They may often help to simplify code by providing simple, general-purpose manipulation of functions and their arguments. In particular, *functional composition* encapsulates the common practice of using the result

of one function application as the input parameter to a second function. The composition operator facilitates the repeated use of this technique, producing a "pipeline" style of programming.

Higher order functions provide an elegant mechanism to remove the need for explicit recursion. To demonstrate this facility for lists, three families of curried, polymorphic higher order functions on lists have been introduced:

1. The `map` family, which retains the list structure but transforms the list items.
2. The `fold` family, which distributes an operator over a list, generally to produce a single value result.
3. The `select` family, which may select items from a list, according to a given predicate.