Chapter 3

Lists

As has been shown in previous chapters, data items of types such as `num` and `bool` can only hold one data value at a time. In order to collect more than one value, *tuples* and *strings* can be used; these are known as aggregate data types. However, since they are not recursive by nature, it is not easy to manipulate data items of these types in a recursive manner. The *list* data type is introduced in this chapter as a powerful aggregate data type which is recursively defined and is used to hold many values of a single type. Built-in functions are introduced to construct and deconstruct data items of this new type.

Functions which manipulate lists are naturally recursive. This chapter provides guidelines for the design of such functions using the techniques of case analysis and structural induction; case analysis ensures that the programmer considers all possible inputs to a function, whereas structural induction is a powerful tool for developing recursive algorithms.

Recursive function definition is of such fundamental importance to the manipulation of the list data structure that the general technique is analysed further and five common modes of recursion over lists are discussed. The chapter ends with a detailed example of program design utilizing the above techniques; the program *grep*, a textual search utility, which will be used as an extended example throughout the rest of the book.

3.1 The list aggregate type

Aggregate types

It is often useful to have a single name to refer to items of data that are related in some way. Any data type that allows more than one item of data to be referenced by a single name is called an *aggregate type* or a *data structure*, since the items are represented in a structured and orderly manner.

It is clear that objects of the simple types `bool`, `char` or `num` have one value and are not aggregate types. One example of an aggregate type is the tuple, which allows a fixed number of data items to be represented by a single name. This chapter introduces the *list* aggregate type, which is of particular interest to functional language programmers, because it is *recursively defined* and may be easily manipulated by recursive functions.

Lists

The important concept of the list data structure is analogous to a collection of nested boxes. There is a central ("empty") box which may be surrounded by a slightly larger box, which in turn may be surrounded by another box, and so on. The central box contains nothing and cannot be opened, whereas all other boxes contain a data item and another nested-box structure (see Figure 3.1).

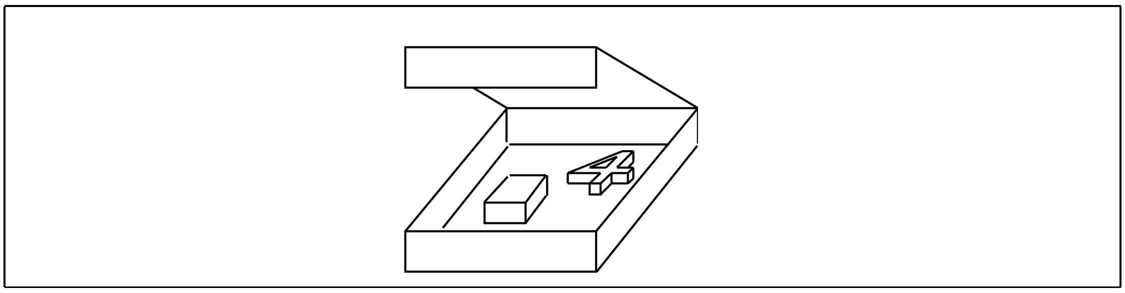

Figure 3.1 Box containing empty box and the data item "4".

To inspect the first item in this data structure, it is necessary to open the outermost box—inside will be the first data item and a smaller box (the rest of the data structure). In order to inspect the second item in the list, it is necessary to do two things—first open the outermost box, then open the box that is found inside. At that point, the second data item in the list is available, together with a smaller box representing the remainder of the data structure (see Figure 3.2).

In order to use such a data structure, the following operations must be provided by the language:

1. A way to construct a new structure (that is, the ability to put things inside a new box).
2. A way to deconstruct an existing structure (that is, the ability to "open a box").
3. A way to recognize the empty box so that there will be no attempt to deconstruct it (which would be an error because it contains nothing).

The following discussion of Miranda lists will first present the notation used and then explain how these three operations may be performed.

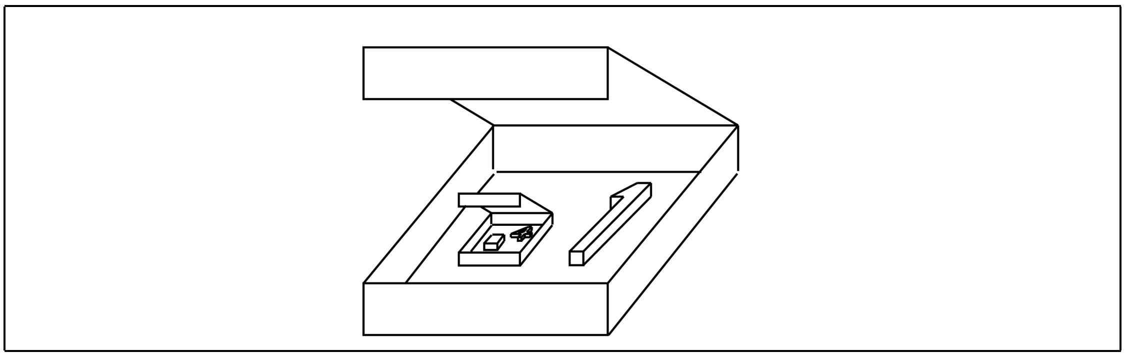

Figure 3.2 Box containing the data item “1” and another box.

3.1.1 List syntax

From the above analogy, it should be clear that the list data structure is naturally recursive in its definition. Informally, a Miranda list may be defined either as being empty or containing one element together with a list (which itself is empty or consists of an element of the same type as the previous element together with a list (which itself is empty or consists of an element of the same type as the previous element together with ... and so on ...)).

From the above, a more formal, recursive definition of a list may now be derived:

A list is either:

empty

or:

an element of a given type together with
a list of elements of the same type

There are two notations that Miranda uses to represent lists: an *aggregate* format and a *constructed* format. The aggregate format is introduced below, whereas the constructed format is presented in Section 3.2.

The empty list is represented in both formats by the characters `[]`—this corresponds to the “empty box” which cannot be opened. For lists with one or more elements, the empty box is implicit in the aggregate format and only the data items are given explicitly. For example, a box which contains the number `1` together with the empty box is represented by the aggregate format `[1]`. Further examples of lists entered at the prompt together with the type indicator now follow:

```
Miranda [1,2] ::
[num]

Miranda ["David Turner’s", "Miranda"]  ::
[[char]]
```

```
Miranda [(1,2),(3,1),(6,5)] ::
[(num,num)]

Miranda [[1],[2,2,2]] ::
[[num]]
```

Illegal lists

All the elements of a list *must* have the same type; the Miranda system will detect any attempt to define a list of mixed types. All of the following examples within the script file are therefore illegal:

```
wrong1 = [1, "1" ]
wrong2 = [1, [1] ]
wrong3 = [1, (1, 1) ]
wrong4 = [1, [] ]
```

In each case the first element of the list is a number and so, on exit from the editor, the Miranda type checker expects all the other elements of the list to be numbers as well:

```
type error in definition of wrong4
(line   4 of "script.m") cannot cons num to [[*]]
type error in definition of wrong3
(line   3 of "script.m") cannot cons num to [(num,num)]
type error in definition of wrong2
(line   2 of "script.m") cannot cons num to [[num]]
type error in definition of wrong1
(line   1 of "script.m") cannot cons num to [[char]]
```

The meaning of "cons" will be explained below.

3.1.2 The empty list—`[]`

The empty list is represented by the characters `[]`. Recursive functions which operate on lists will successively deconstruct each smaller "box" in the list structure and will converge on the empty list `[]`. However, it is important that such a function should not attempt to deconstruct `[]` since this would give an error.

The value `[]` is used as the starting point for constructing a list of any type; thus, `[]` must have the polytype `[*]`. For example, when `[]` is used to create a list of numbers it adopts the type `[num]`, but when it is used to create a list of Booleans it adopts the type `[bool]`. Notice that `[]` is a data item which can itself appear in lists; both of the following values have type `[[*]]`:

```
list1 = [ [], [], [] ]

list2 = [[]]
```

3.2 Built-in operations on lists

Miranda provides a number of built-in operations for list manipulation, including those for list construction and deconstruction. An empty list can simply be created by giving a name to the empty list constant `[]`. Other lists can be created by combining an item with an existing list, using the special list constructor `:` or by combining a list with another list using the built-in operator `++`. Two functions are also provided for list deconstruction: `hd` and `tl`. The former takes a list and returns its first element; the latter takes a list and returns the list without the first element.

A number of other list manipulating functions (such as extracting the n^{th} item of a list or finding its length) may also be provided as built-in functions. However, they could easily be written by the programmer, as will be seen later in this chapter.

3.2.1 List construction—:

Apart from `[]`, the only list operator that is actually required is the polymorphic operator `:` (pronounced "cons") which *prepends an item to an existing list of the same type as that item.* To use the nested-box analogy, : takes a data item and a box and puts both into a bigger box; this creates a new list. The type constraint is important—the second argument must either be empty or contain items of the same type as the first argument. A list of one item can be created from a single item and `[]` because `[]` is a polytype list and so can be combined with an item of any type. The type of `:` is:[1]

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

In the following example script file, all of the identifiers are of type `[[char]]`:

```
alist = "a" :  []
blist = "fed" :  "cb" :  alist
clist = "C" :  ["c", "c", "c"]
dlist = "d" :  "dd" :  "d" :  []
```

It is also possible to identify individual list components, using the pattern matching definition style introduced in Section 1.4:

[1]This represents a "mapping" or function type; for many purposes it is similar to a `(*,[*])->[*]` type, but see also the section on currying in Chapter 4.

```
(firstItem :  otherItems) = [3, 2, 1]
```

```
Miranda firstItem
3
Miranda otherItems
[2,1]
Miranda 4 : 2 : 3 : []
[4,2,3]
```

The use of `:` provides the list in its constructed format though the system responds by giving the list in its aggregate format.

The `(firstItem : otherItems)` definition is similar to that of tuple matching, shown in Chapter 1. The use of `:` in this manner is possible because it is not a normal function or operator but a special form of what is known as a *constructor*. It is not concerned with manipulating the values of its arguments but with constructing a new list. *One of the most important properties of constructors is that they provide a unique representation for each value of an aggregate type.* Thus, a new list can be constructed—and so deconstructed—in only one way. In the above example, the only possible value for `firstItem` is the number `3` and the only possible value for `otherItems` is the number list `[2,1]`.

The operator `:` is also different from other operators so far seen in that it associates to the right; that is `(1:2:[])` is really `(1:(2:[]))`. If `:` associated to the left giving `((1:2):[])`, then the construction `(1:2)` would fail because `2` is not a list.

List construction abuse

The following examples demonstrate common mistakes. The first example is wrong, in that the number `1` is *cons*-ed onto a list whose items are strings (whereas Miranda expects the items to be numbers, to be consistent with the first argument to `:`). In the second example, the first argument to `:` is a list of type number and therefore Miranda expects the second argument to be a list whose items are themselves lists of numbers. An error occurs because the second argument is in fact a list whose items are numbers. The third example has a similar error, in that the number `1` is expected to be prepended to a list of numbers, whereas the second argument is a number and not a list.

```
wrong_x = 1 :  ["a","bcd"]
wrong_y = [1] :  [2,3]
wrong_z = 1 :  2
```

```
type error in definition of wrong_z
(line   3 of "script.m") cannot cons num to num
type error in definition of wrong_y
```

```
(line   2 of "script.m") cannot cons [num] to [num]
type error in definition of wrong_x
(line   1 of "script.m") cannot cons num to [[char]]
```

Exercise 3.1
Give two possible correct versions of `wrong_y`.
Exercise 3.2
Which of the following are legal list constructions?

```
list1 = 1 : []
list2 = 1 : [] : []
list3 = 1 : [1]
list4 = [] : [1]
list5 = [1] : [1] : []
```

Append

Miranda provides another polymorphic list operator `++` (pronounced "append"). This takes two lists of the same type and concatenates them to create a new list of the same type. The terminating `[]` value of the first list is discarded. The type of `++` is:[2]

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

Sample session:

```
Miranda [1] ++ [2,3]
[1,2,3]

Miranda [] ++ [1]
[1]

Miranda [] ++ []
[]

Miranda 1 ++ [2,3]
type error in expression
cannot unify num with [num]
```

[2]This type is similar to `([*],[*])->[*]`, see also Chapter 4 on currying.

Properties of append

Two properties of `++` are worth noting:

1. The operator `++` could associate from the left or from the right and produce the same result. An advantage here is that expressions involving `++` do not require bracketing.
2. The operator `++` is not a constructor because it cannot provide a unique representation for a list. This is demonstrated by the fact that the comparison:

   ```
   ["A","A"] = firstList ++ secondList
   ```

   could be satisfied either by both `firstList` and `secondList` having the values `["A"]` or by one of them having the value `[]` and the other having the value `["A","A"]`.
   Theoretically `++` is unnecessary since it can be defined in terms of `:` (as shown in Section 3.7.2). Nevertheless, it is provided as a built-in operator because it is such a frequent programming requirement.

3.2.2 List deconstruction

Miranda provides two polymorphic functions to extract items from a list, called `hd` and `tl`.[3]

The function `hd` (pronounced "head") operates on a list to select its first element (that is, it "opens the box" and extracts the data item). The type of `hd` is therefore:

```
[*]->*
```

Examples:

```
Miranda hd [1]
1

Miranda hd [1,2,3,4]
1

Miranda hd ["Schoenfinkel","male","mathematician"]
Schoenfinkel

Miranda hd [[1,2,3],[8,9]]
[1,2,3]

Miranda hd [(1,2),(7,-1),(0,0)]
(1,2)
```

[3]Section 3.4 shows how easily they could be defined if they were not already provided.

The function `tl` (pronounced “tail”) is complementary to `hd` in that it operates on a list to yield that list with its first element removed (that is, it “opens the box” and extracts the inner box). The type of `tl` is therefore:

```
[*]->[*]
```

Thus, if `tl` is applied to a value of type `[num]` then it will return a value of type `[num]` (even if the resulting list is `[]`).

Examples:

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

Miranda tl ["Schoenfinkel","male","mathematician"]
["male","mathematician"]

Miranda tl [[1,2,3],[8,9]]
[[8,9]]

Miranda tl [(1,2),(7,-1),(0,0)]
[(7,-1),(0,0)]

Miranda (tl [1])
[]

Miranda (tl [1]) ::
[num]
```

The relationship between `hd`, `tl` and `:`

Both `hd` and `tl` give rise to errors when applied to empty lists:

```
Miranda hd []

program error: hd []

Miranda tl []
[
program error: tl []
```

Thus, when programming with lists, it is necessary to use either a guard or pattern matching to avoid such errors. This test often arises as a natural consequence of the fact that the empty list is the terminating condition for functions that are defined recursively over a list (as shown later in this chapter).

Apart from the case of the empty list, `hd` and `tl` and `:` are related in such a way that for any list:

```
(hd anylist) : (tl anylist) = anylist
```

Exercise 3.3
Miranda adopts the view that it is meaningless to attempt to extract something from nothing; generating an error seems a reasonable treatment for such an attempt. What would be the consequences if `hd` and `tl` were to evaluate to `[]` when applied to an empty list?

3.2.3 Other list operators

A number of other operators are provided for list manipulation. Three of these operators are familiar, being congruent to the string manipulation operators introduced in Chapter 1. These are: `--` (list subtraction), `#` (list length) and `!` (list indexing). For example:

```
Miranda ['c','a','t'] -- ['a']
ct

Miranda # [345,2,34]
3

Miranda ["chris", "colin", "ellen"] ! 0
chris
```

The other list operators are: "dotdot" notation, which is used as an abbreviation for lists representing a series of numbers, presented below; and "list comprehension", discussed in Chapter 5, which provides a tool for intensionally specifying the contents of a list through calculation (rather than extensionally stating each element).

3.2.4 Dotdot notation

The simplest form of "dotdot" notation is to place two numbers in a list (using the aggregate format), separated by two dots as follows:

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

The above example shows that Miranda responds by "filling in" the missing numbers between the first and the last. This abbreviated form may only be used for ascending lists—if a descending list is specified, the system returns the empty list. However, negative numbers may be specified:

```
Miranda [4..1]
[]

Miranda [-2..2]
[-2,-1,0,1,2]

Miranda [-2..-4]
[]
```

Lists of numbers which form an arithmetic series are also allowed. If the first number is followed by a comma and a second number (before the two dots) then this will specify the interval between each number in the series:

```
Miranda [1,3..9]
[1,3,5,7,9]

Miranda [-2,0..10]
[-2,0,2,4,6,8,10]

Miranda [1,7..49]
[1,7,13,19,25,31,37,43,49]

Miranda [1.1,1.2..2.4]
[1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2.0,2.1,2.2,2.3,2.4]
```

If the last number does not belong in the sequence then the list will stop before exceeding the last number:

```
Miranda [1,3..10]
[1,3,5,7,9]
```

With this second form of "dotdot" abbreviation, it is also possible to define descending sequences:

```
Miranda [3,1..-5]
[3,1,-1,-3,-5]

Miranda [3,1..-8]
[3,1,-1,-3,-5,-7]

Miranda  hd (tl (tl (tl [3,1..-5])))
-3
```

3.3 Lists and other aggregate types

This section briefly looks at the relationship between lists and the other Miranda aggregate types: strings and tuples.

Lists and strings

Strings and lists of strings are semantically related, since they both represent a sequence of characters. In fact, the string double-quote notation is just a convenient shorthand for a list of characters, as seen by the type indication for a string:

```
Miranda "sample" ::
[char]
```

This explains why all of the Miranda list operators work on both strings and lists. Notice that the above sample string could equally well have been written:

```
Miranda ('s' : 'a' : 'm' : 'p' : 'l' : 'e' : []) ::
[char]
```

or

```
Miranda ['s','a','m','p','l','e'] ::
[char]
```

Exercise 3.4
At first sight it would appear that **show** can be bypassed by defining a function that quotes its numeric parameter:

```
numbertostring :: num -> [char]
numbertostring n = "n"
```

Explain what the above function *actually* does.

Lists and tuples

It can be seen that lists and tuples are similar in that they are both aggregate data structures (that is, they collect together many data elements). However, they differ in three ways:

1. Their types: all the items in a list must be of the same type,[4] whereas tuple elements may be of differing types. However, lists and tuples can mix; it is legal to have tuples with list components and vice versa. In the latter case, it is necessary to ensure that the tuple elements of the list are of the same length and have the same component types.

```
wrong_list = [(1,2,3), (True,False)]
```

```
type error in definition of wrong_list
cannot cons (num,num,num) to [(bool,bool)]
```

```
good_tuple = ([1,2,3], [True,False])
good_list = [(12,"June",1752), (13,"March",1066)]
```

2. A list is a *recursively defined* type (it is defined in terms of itself), and therefore the type contains no information about the number of elements in the list. By contrast, a tuple is *not* recursively defined and the type of a tuple determines exactly how many elements it will contain. It is for this reason that lists may be constructed incrementally, whilst tuples may not.
3. Equality testing: tuple comparison can only be made against tuples of the same composite type and hence the same number of components, whereas lists of any length may be compared.
   Comparing tuples:

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

   Comparing lists:

```
Miranda [1,2,3] = [1,2]
False
```

3.4 Simple functions using lists

Functions over lists may be defined in the same manner as all other functions; they can be recursive, polymorphic and use pattern matching. The nature of recursive, and especially polymorphic recursive, functions is dealt with in later sections; this section overviews the use of pattern matching on lists. There are four things that can appear in function patterns involving lists:

1. Formal parameters—which can be substituted by any actual value.

[4]See Chapter 6 for ways of creating a new type of list which may have elements of different underlying types.

2. Constants—including the empty list [].
3. Constructed lists using : notation (the aggregate form is also legal, but is not recommended for general use).

Examples of simple functions using lists:

```
isempty ::  [*] -> bool
isempty [] = True
isempty anylist = False

isnotempty ::  [*] -> bool
isnotempty anylist = ~(isempty anylist)

bothempty ::  ([*],[**]) -> bool
bothempty ([],[]) = True
bothempty anything = False

startswithMira ::  [char] -> bool
startswithMira ('M' : 'i' :  'r' :  'a' :  rest) = True
startswithMira anylist = False
```

In the final example, it is necessary to bracket the `('M' : 'i' :  'r' :  'a' :  rest)` list construction so that it is treated as a single value, which can then be pattern matched. Further examples of pattern matching using list construction include possible implementations of the built-in functions `hd` and `tl`:

```
myhd ::  [*] -> *
myhd [] = error "myhd"
myhd (front :  rest) = front

mytl ::  [*] -> [*]
mytl [] = error "mytl"
mytl (front :  rest) = rest
```

The use of pattern matching in these examples emphasizes that there is no possible way of deconstructing `[]` to give component values.

3.5 Recursive functions using lists

Many of the functions over lists are recursive because the list itself is a recursive data structure. Whilst recursion involving numbers generally requires the recognition of zero to provide a terminating condition, list processing generally requires the recognition of the empty list. This section shows that the two recursive styles shown in Chapter 2 can also be used to manipulate lists.

Stack recursion over lists

The template for many functions on lists is very similar to that of stack recursive functions shown in Section 2.7.1:

template []	=	*some final value*
template (front: rest)	=	*do something with front and combine the result with a recursion on template applied to rest*

For example, the following function adds all the items in a number list. It specifies that the sum of an empty list is 0, whilst the sum of any other list is the value of the front item added to the sum of the rest of the list:

```
nlist == [num]

sumlist ::  nlist -> num
sumlist [] = 0
sumlist (front :  rest) = front + sumlist rest
```

Exercise 3.5
Write a stack recursive function to add all numbers less than 3 which appear in a list of numbers.

Accumulative recursion over lists

The following example reworks `sumlist` to show its equivalent accumulative recursive implementation:

```
nlist == [num]

sumlist ::  nlist -> num
sumlist any = xsum (any, 0)

xsum ::  (nlist, num) -> num
xsum ([], total) = total
xsum (front :  rest, total) = xsum (rest, front + total)
```

However, if it were desired to consider the sum of an empty list to be meaningless or an error, it would be sensible to validate the input list in `sumlist` and process it in the auxiliary function `xsum`. In the following version, if `sumlist` recognizes an empty list then it treats this as an error and does no further processing. In contrast, when `xsum` recognizes an empty list it treats this as the *terminating condition* for the recursion and returns the desired total:

```
nlist == [num]

sumlist ::  nlist -> num
sumlist []  = error "sumlist - empty list"
sumlist any = xsum (any, 0)

xsum ::  (nlist, num) -> num
xsum ([], total) = total
xsum (front :  rest, total) = xsum (rest, front + total)
```

It is worth noting that validation is only done once in this example and has been separated from the calculation. In contrast, the following version is undesirable in that an extra pattern is needed to detect the terminating condition and also that the validation is confused with the calculation. As discussed in Section 2.10, from a software design perspective it is generally poor practice to make a function do more than one thing.

```
nlist == [num]

muddledsum ::  nlist -> num
muddledsum any = xmuddledsum (any, 0)

xmuddledsum ::  (nlist, num) -> num
xmuddledsum ([], total)
       = error "muddledsum"
xmuddledsum ((item :  []), total)
       = (item + total)
xmuddledsum (front :  rest, total)
       = xmuddledsum (rest, front + total)
```

A template for accumulative recursive functions is:

```
main []  = some terminating condition
           or error condition
main any = aux (any, (initial value of accumulator))

aux ([], accumulator) = accumulator
aux ((front: rest), accumulator)
           = aux(rest, do something with front
                       and accumulator))
```

Exercise 3.6

The following function `listmax` is accumulative recursive. Rather than using an explicit accumulator, it uses the front of the list to hold the current maximum value.

```
numlist == [num]

listmax :: numlist -> num
listmax [] = error "listmax - empty list"
listmax (front : []) = front
listmax (front : next : rest)
    = listmax (front : rest), if front > next
    = listmax (next : rest), otherwise
```

Rewrite `listmax` so that it uses an auxiliary function and an explicit accumulator to store the current largest item in the list.

3.6 Polymorphic recursive functions on lists

Many list-handling functions are designed to explore or manipulate the list structure itself rather than the list elements. These are called "polymorphic" functions because the elements of the source lists may be of any type; otherwise they are the same as non-polymorphic functions, as has already been shown with the simple functions `isempty` and `bothempty` in Section 3.4. Polymorphic recursive functions on lists are particularly interesting, in that they are the basis for a rich variety of tools of general utility for list processing. This section presents two of these functions over lists: `length`, and `mydrop`. The treatment of `mydrop` will also demonstrate that not all software design principles are set in stone.

length

The following definition of `length` follows the template provided to describe stack recursive functions. It provides the same functionality as the built-in # operator. Informally, the length of a list can be seen as one of two possibilities: the length of an empty list (which is 0) or the length of a non-empty list. The latter can be seen to be 1 plus the length of a list containing one less item.

```
length ::  [*] -> num
length [] = 0
length (front :  rest) = 1 + length rest
```

mydrop

The polymorphic function `mydrop` returns part of its second argument (a list of items of any type) by removing the first `n` items, where `n` is the first argument. It provides similar functionality to the `drop` function provided in the Miranda Standard Environment:

```
mydrop ::  (num, [*]) -> [*]
mydrop (0, anylist)         = anylist
mydrop (any, [])            = error "mydrop"
mydrop (n, (front :  rest))  = mydrop (n - 1, rest)
```

This function has been presented in the way that most functional programmers would probably write it. However, it goes against the advice given for the function `sumlist`. A better design might be:

```
mydrop ::  (num, [*]) -> [*]
mydrop (n, anylist)
    = error "mydrop:  negative input", if n < 0
    = error "mydrop:  input too big", if n > (# anylist)
    = xdrop (n, anylist), otherwise

xdrop ::  (num, [*]) -> [*]
xdrop (0, anylist)      = anylist
xdrop (n, front :  rest) = xdrop (n - 1, rest)
```

This definition satisfies the principle of a function only doing one kind of activity, because it separates the validation from the rest of the processing. However, it is algorithmically unsatisfying in that the entire length of `anylist` must be calculated, even though `xdrop` will work as desired on any list with at least `n` elements. For example, the length of a million item list may have to be calculated, even though the programmer only wanted to discard its first element!

An alternative solution is to replace # in the **if** guard with a more finely tuned function which only checks that `anylist` is of the minimum necessary length:

```
mydrop  ::  (num, [*]) -> [*]
mydrop  (n, anylist)
     =  error "drop:  negative input", if n < 0
     =  error "drop:  input too big",
        if shorterthan (n, anylist)
     =  xdrop (n, anylist), otherwise
```

Exercise 3.7

What happens if a negative value of `n` is supplied to the first version of `mydrop`?

Exercise 3.8

Write the function `shorterthan` used by the final version of `mydrop`.

A further example is the function `mymember` which checks whether an item appears in a list. This function also provides similar functionality to the function `member` provided in the Miranda Standard Environment. The specification is simple:

1. The terminating condition where the list is empty—hence nothing can appear in it, and so `mymember` evaluates to `False`.
2. The terminating condition where the item to be found matches the head of the list—and so `mymember` evaluates to `True`.
3. The item does not match the head of the list—and so it is necessary to see if the item appears in the rest of the list.

Translation to Miranda is as simple as the function specification:

```
mymember ::  ([*],*) -> bool
mymember ([], item) = False
mymember (item :  rest, item) = True
mymember (front :  rest, item) = mymember (rest, item)
```

3.7 Thinking about lists

The list-handling functions introduced so far have been simple, to write because their Miranda code follows directly from their natural language specification. Unfortunately not all functions are so intuitively defined. To make the programmer's task easier, this section discusses two important tools for designing functions, and list-handling functions in particular. The first tool is commonly known as *case analysis*, which stresses the importance of looking at each expected function parameter. The second tool is known as *structural induction* and offers an important technique to aid the design of recursive algorithms.

3.7.1 Case analysis

It is impossible to consider every possible argument value to a list-handling function. In practice, it is only necessary to consider a limited number of cases:

1. The empty list `[]` which must *always* be considered because it is either a terminating value or an illegal option.
2. The general list `(front : rest)` which also must *always* be considered; the function body corresponding to this pattern is where the recursive application is normally found.
3. Specific "n item" lists. For example the single item list `(item : [])`, since there is a class of functions (for example `listmax` in Exercise 3.6) that require at least one element in the list for a meaningful result.

List reversal

This section explores the design of the function `myreverse`, which reverses the order of the items in a list (and therefore has the type `[*] -> [*]`).[5] The discussion also highlights some important properties of list manipulation using `:` and `++`.

The function definition can be derived by case analysis of possible input values:

1. The empty list.
2. A list of one item.
3. A list of two items, which is the simplest case where the list is transformed.
4. A general list.

List reversal for the first two cases is trivial:

```
myreverse []     = []
myreverse [item] = [item]
```

Considering the third case, the desired result can be seen as simply reversing the two elements of the list. This means that the list must be deconstructed to give a name to each of the two elements and then reconstructed with the elements reversed.

The deconstruction of the list can be done using pattern matching with either the aggregate format or the constructed format for the list:

```
myreverse1 [front, final]  = ???
myreverse2 (front : rest) = ???
```

Remember that `final` and `rest` are not the same thing—the former refers to an item, whereas the latter refers to a list (which contains both the final item and the empty list).

Similarly, there are three ways to reconstruct the list—using either the `:` operator or using the list aggregate format or using the `++` operator. Here are the six possibilities:

[5]Note that the Miranda Standard Environment also provides a function called `reverse` which has the same functionality.

```
myreverse11 [front, final] = final : front : []
myreverse12 [front, final] = [final, front]
myreverse13 [front, final] = [final] ++ [front]
myreverse21 (front : rest) = (hd rest) : front : []
myreverse22 (front : rest) = [(hd rest), front]
myreverse23 (front : rest) = rest ++ [front]
```

Notice that both `myreverse21` and `myreverse22` require `rest` to be further broken down by use of the `hd` function, and so in this situation the aggregate pattern seems to be more useful. However, `myreverse13` requires both `front` and `final` to be converted into lists before they can be appended. This conversion is achieved by enclosing each in square brackets, which is equivalent to using `:` and the empty list `[]`.

As a guiding principle, the use of `:` in the pattern is preferred for the general recursive case (as will be seen below), and in the base cases either format may be used. For `myreverse`, the most straightforward of the correct definitions given above is:

```
myreverse [front, final] = [final, front]
```

The reader should be wary of the following two errors:

```
wrong_myreverse1 [front, final] = final : front
wrong_myreverse2 [front, final] = final ++ front
```

Both of the above definitions fail to compile because the operator (either `:` or `++`) does not get arguments of the correct types. The latter definition is actually legal for a list whose items are themselves lists, but it is semantically wrong because the list items are inappropriately compressed:

```
wrong_myreverse2 [[1], [2]]
==> [2] ++ [1]
==> [2,1]
```

Considering the final case, the recursive nature of list processing becomes more obvious. As a first step, reversing, for example, [1,2,3] can be treated as reversing [2,3] appended to [1], that is `(myreverse [2,3]) ++ [1]`. Reversing [2,3] is, of course, covered by the third case and the list [2,3] can easily be extracted from [1,2,3] using pattern matching.

The above discussion leads to the provisional function definition:

```
myreverse ::  [*] -> [*]
myreverse []             = []
myreverse [item]         = [item]
myreverse [front, final] = [final, front]
myreverse (front :  rest) = (myreverse rest) ++ [front]
```

The provisional function evaluates in the following manner:

```
myreverse [1,2,3]
==> (myreverse [2,3]) ++ [1]
==> [3,2] ++ [1]
==> [3,2,1]
```

Rationalization of list reversal

In practice, the above definition may be simplified, in that the final three patterns can all be catered for by the final function body. Applying the final pattern to a single or double item list shows that both the patterns for single and double item lists are redundant. This is because the reverse of the single item list is a special case of the final pattern `[item]` is just `(item :  [])`. The function `myreverse` thus simplifies to:

```
myreverse [*] -> [*]
myreverse [] = []
myreverse (front :  rest) = (myreverse rest) ++ [front]
```

A hand evaluation reveals:

```
myreverse [1,2]
==> (myreverse [2]) ++ [1]         || using the final pattern
==> ((myreverse []) ++ [2]) ++ [1] || using the final pattern
==> ([] ++ [2]) ++ [1]             || using the first pattern
==> [2] ++ [1]                     || ([] ++ [item]) == [item]
==> [2,1]
```

3.7.2 Structural induction

Case analysis is a useful method of ensuring that a programmer considers all possible inputs to a function. This gives immediate solutions to all "base" cases which are directly provided for in the function specification and also highlights those cases which require further processing. In the previous example of `myreverse` an intuitive recursive solution was evolved for the general list `(front:rest)`; however not all problems are as amenable to such intuition. There is, therefore, a need for a more systematic method of analysis. The nature of this analysis will depend on the nature of the parameter of recursion (that is the parameter which converges towards a terminating condition). For lists, a technique known as *structural induction* is recommended because it provides a way to reason about a list's recursive structure.

Structural induction requires two steps:

1. Consider all the base cases as for case analysis.
2. Consider the general case. The design of the function body may be facilitated by the following technique: assume that a function body exists for a list `rest` and then construct a definition for the case `(front:rest)`. The assumption is normally known as the *induction hypothesis.*

The use of structural induction to design two functions (`startswith` and `append`) is now presented. The first function `startswith` will be used in the extended example later in this chapter.

Specification of startswith

The function `startswith` takes two lists and returns a Boolean value. If the first list is an initial sublist of the second list then the function evaluates to `True`, otherwise it evaluates to `False` (for example, `startswith ([1,2,3],[1,2,3,4,5])` evaluates to `True`). By definition, an empty list is an initial sublist of all lists.

Design of startswith

The function `startswith` will take a pair of lists whose elements may be compared for equality. It therefore has the type `([*],[*]) -> bool`.

The design will consider the general case and then the base cases. In this example, it helps to consider the general case first, in order to determine the parameter of recursion. However, the program designer may consider the base cases first if appropriate.

The general case is where neither list is empty—there is no direct solution from the specification and so this requires further processing. The general form of the first list is `(front1 : rest1)` and the general form of the second list is `(front2 : rest2)`. In this general case, the induction hypothesis is that the application `startswith (rest1, rest2)` will evaluate to the appropriate truth value. Given this assumption, the creative step is to realize that the general case should evaluate to `True` if `startswith (rest1, rest2)` evaluates to `True` and also `front1` is the same as `front2`. This highlights the fact that each list converges and both are parameters of recursion. The design translates directly to the following incomplete Miranda code:

```
startswith (front1 : rest1,  front2 : rest2)
         = (front1 = front2) & startswith (rest1, rest2)
```

The base cases are:

1. An empty first list—by definition this case always returns `True`. This is the terminating case for successful matches:

   ```
   startswith ([], anylist) = True
   ```

2. An empty second list—which always evaluates to `False`. This is the terminating case for unsuccessful matches because there is still some of the first list to be compared. If the first list had also been empty, this would have been matched by the first base case:

   ```
   startswith (anylist, []) = False
   ```

The complete definition is:

```
startswith ::  ([*],[*]) -> bool

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

List append

The following discussion emphasizes the advantages of using structural induction rather than case analysis, especially when a large number of cases arise, since the philosophy of structural induction is to generalize rather than specialize.

The infix operator `++` may itself be defined as a prefix function, in terms of `:`. The specification of the following function `append` is very similar to that of `++` as stated earlier in this chapter; it takes a pair of lists of the same type and concatenates them to create a new list of that type; that is, it has the tuple type `([*],[*]) -> [*]`, which differs only slightly from `++` which has the curried type `[*]->[*]->[*]` (as will be explained in Chapter 4). The `[]` component of the first list is always discarded; if the first list is empty the function evaluates to the second list.

An attempt at defining `append` using case analysis is to write down the possible combinations of types of lists to be appended:

```
append ([],             [])             =  Body_1
append ([],             item2 : [])     =  Body_2
append ([],             front2 : rest2) =  Body_3
append (item1 : [],     [])             =  Body_4
append (item1 : [],     item2 : [])     =  Body_5
append (item1 : [],     front2 : rest2) =  Body_6
append (front1 : rest1, [])             =  Body_7
append (front1 : rest1, item2 : [])     =  Body_8
append (front1 : rest1, front2 : rest2) =  Body_9
```

It is clear that providing function bodies for each of these possible cases will involve a lot of programmer effort. Much of this effort can be saved if structural induction is employed.

A definition for the general case `Body_9` is based on the induction hypothesis that there is already a definition for `append (rest1, front2 : rest2)`. The induction step is to produce a definition for `append (front1 : rest1, front2 : rest2)` which uses the property that `front1` must become the first item of the resultant list:

```
append (front1 : rest1, front2 : rest2)
     = front1 : append (rest1, front2 : rest2)
```

It is clear that the parameter of recursion is the first list—the second list never alters. Hence `front2:rest2` could be written more simply as `anylist`. There is now only one base case to consider: when the first list is empty (`Body_3`). By definition this evaluates to the second list. The full Miranda code is thus:

```
append  ::  ([*],[*]) -> [*]

append  ([], anylist)
     =  anylist
append  (front1 :  rest1, anylist)
     =  front1 :  append (rest1, anylist)
```

All the other cases can be removed from consideration; in particular, it can be seen that the single item list is not a special case. *In general, single (or n-item) lists should only be treated in a special manner by the function definition if they are treated as special cases by the function specification.*

Exercise 3.9

Use structural induction to design the function `mytake`, which works similarly to `mydrop` but takes the first `n` items in a list and discards the rest.

Exercise 3.10

Write a function `fromto` which takes two numbers and a list and outputs all the elements in the list starting from the position indicated by the first number up to the position indicated by the second number. For example:

```
Miranda fromto (3, 5, ['a','b','c','d','e','f'])
['d','e','f']
```

3.7.3 Problem solving using structural induction and top-down design

This section looks at a slightly larger problem, the solution to which illustrates the use of structural induction in conjunction with the top-down design technique (discussed in Section 2.10).

The problem to be considered is that of sorting information in an ascending order. There is a wealth of literature on how this may be best achieved (for example, Standish, 1980); one of the simplest methods, known as *insertion sort*, is now presented.

Sorted list specification

1. An empty list is defined as already sorted.
2. A list of only one element is defined as already sorted.
3. The list `(front:rest)` is sorted in ascending order if `front` is less than all items in `rest` and `rest` is sorted in ascending order.
4. For the purposes of this example, only number lists will be considered.

Top-down design

There are a number of strategies to achieve insertion sort; the approach taken here is to start with an unsorted list and an empty list and then insert the items of the former into the latter one at a time, ensuring that the latter list is always sorted. This approach makes the assumption that it is possible to insert one item into an already sorted list to give a new sorted list. For example, to sort the list `[3,1,4,6,2,4]`, the following changes will occur to the two lists:

	Unsorted list	*Sorted list*
Initially	*[3,1,4,6,2,4]*	*[ ]*
First pass	*[1,4,6,2,4]*	*[3]*
Second pass	*[4,6,2,4]*	*[1,3]*
Third pass	*[6,2,4]*	*[1,3,4]*
Fourth pass	*[2,4]*	*[1,3,4,6]*
Fifth pass	*[4]*	*[1,2,3,4,6]*
Final pass	*[ ]*	*[1,2,3,4,4,6]*

Insertion sort implementation

To meet the above design, the function `isort` must employ an accumulator (which is initialized to be empty) to build the final sorted list. There is also the need for a function to insert each element from the unsorted list into the accumulator. This leads directly to the Miranda code:

```
nlist == [num]

isort ::  nlist -> nlist
isort anylist = xsort (anylist, [])

xsort ::  (nlist, nlist) -> nlist
xsort ([], sortedlist)
    = sortedlist
xsort (front :  rest, sortedlist)
    = xsort (rest, insert (front, sortedlist))
```

Insert design

The design of the function `insert` is simple; the item to be inserted is compared against each element of the sorted list in turn, until its correct position is found.

Insert implementation

The base case is that of inserting an item into an empty list which just gives a singleton list of that item:

```
nlist == [num]
insert :: (num, nlist) -> nlist
insert (item, []) = [item]
```

The general case is that of inserting an item into a non-empty sorted list:

```
insert (item, (front : rest)) = ???
```

This involves finding the first item in that list which is greater than the item to be inserted and placing the new item before it.

There are two subcases to consider:

1. The front of the list is greater than the new item. The new sorted list is now the new item constructed to the existing sorted list:

```
    = item : front : rest, if item <= front
    = ???, otherwise
```

2. The front of the list is not greater than the new item, and so it is necessary to place the new item somewhere in the rest of the sorted list. The inductive hypothesis is to assume that `insert` works correctly for the smaller list `rest`. The inductive step is to use this assumption to form the general function body so that `insert` will work correctly for the larger list `(front : rest)`. This gives:

```
front : insert (item, rest)
```

Note that if the new item is larger than any existing list member then eventually the rest of the list will converge towards `[]`; this has already been covered in the base case.

Piecing all this together gives:

```
nlist == [num]

insert ::  (num, nlist) -> nlist
insert (item, [])
    = [item]
insert (item, front :  rest)
    = item :  front :  rest, if item <= front
    = front :  insert (item, rest), otherwise
```

Insertion sort limitations

This sorting algorithm is not particularly efficient, nor is it very general (it will only sort numbers in ascending order). Chapter 4 presents a more general purpose version that will sort a list of any type in any order. Chapter 5 presents a more elegant algorithm known as *quicksort*.

3.7.4 Lazy evaluation of lists

When a function is applied to a data structure, it only evaluates that data structure as far as is necessary in order to produce the required result. This is similar to the behaviour of the built-in operators for logical conjunction (`&`) and logical disjunction (`\/`). For example, applying the function `hd` to the list `[1,2,3]` only requires the value of the first item in the list. If the following items were not yet fully evaluated, for example if the list were `[1,(1 + 1),(1 + 1 + 1)]`, then evaluating `hd [1,(1 + 1),(1 + 1 + 1)]` would *not* cause the additions to be done. The following Miranda session illustrates this behaviour:

```
Miranda hd [23, (4 / 0), 39]
23

Miranda [23, (4 / 0), 39] ! 0
23

Miranda [23, (4 / 0), 39] ! 1

program error: attempt to divide by zero

Miranda [23, (4 / 0), 39] ! 2
39
```

This lazy behaviour provides the basis for elegant yet efficient solutions as will be shown in Section 5.4.

3.8 Modes of recursion

This section reviews the two familiar styles of recursive functions over lists (stack and accumulative recursion) and introduces three new styles (filter, tail and mutual recursion). The different styles of recursion may often be mixed to form even more expressive control structures. However, the programmer should choose the style which most closely mirrors the natural specification of the problem.

Stack recursive functions

Stack recursion was first introduced in Chapter 2, where the `printdots` function was used as an example. In general, stack recursive functions such as `length` have a growing stage where evaluation is suspended, and a reducing stage wherein the final result is evaluated. The reducing stage can only arise when the parameter of recursion reaches the value that causes the non-recursive option to be taken. At this point, the parameter of recursion is said to have *converged* and the reducing stage is triggered by the fact that the non-recursive option returns a value instead of causing another function application.

A further example of a stack recursive function is `occurs` which counts how many times a particular item appears in a given list. There are three cases to consider:

1. The terminating condition of the empty list which has no occurrences of any item and so evaluates to zero.
2. The item matches the head of the list and so the number of occurrences is one plus the number of occurrences in the rest of the list.

3. The item does not match the head of the list and so the number of occurrences is zero plus the number of occurrences in the rest of the list.

Translating the above specification to a stack recursive implementation gives:

```
occurs ::  ([*],*) -> num
occurs ([], item)            = 0
occurs ((item : rest), item)  = 1 + occurs (rest, item)
occurs ((front : rest), item) = 0 + occurs (rest, item)
```

Two alternative recursive implementations of the above specification are now discussed.

Filter recursive functions

A variation on the stack recursive approach is filter or partial stack recursion, which is demonstrated in the following implementation of `occurs`. The definition just presented is inelegant, in that the artificial value `0` was invented to denote the non-occurrence of an item in a list. For example, the application `occurs ([0,22,3,5,0,1,1,9,101,0],0)` recursively expands to `(1 + 0 + 0 + 0 + 1 + 0 + 0 + 0 + 0 + 1 + 0)`, whereas it would be more elegant to have an algorithm that recursively expanded to just `(1 + 1 + 1)`, thereby dropping the superfluous zeros.

The stack recursive algorithm for the function `occurs` can be modified to adopt a filter recursive approach as follows:

1. If the list is empty, the result is the same as for stack recursion.
2. If the item matches the head of the list, the result is the same as for stack recursion.
3. If the item does not match the head of the list, the number of occurrences is just the number of occurrences in the rest of the list.

As with the stack recursive version, the new version eventually reaches an empty list and evaluates to zero; thus there is no danger of a “dangling” addition:

```
occurs ::  ([*],*) -> num
occurs ([], item)            = 0
occurs ((item : rest), item)  = 1 + occurs (rest, item)
occurs ((front : rest), item) = occurs (rest, item)
```

Applying this version of `occurs` on the above example almost filters out all of the unwanted zeros—there is still the trivial case of a final addition for the empty list!

Accumulative recursive functions

The recursive style using *accumulators* (introduced in Section 2.7.2) is also applicable to list-handling functions and, as already shown in the definitions of `sumlist` and `xsum` (in Section 3.5), it is often necessary to provide a main and auxiliary function definition. The main function normally provides any necessary preliminary validation and initializes the accumulator. Rewriting the function `occurs` in this style gives:

```
occurs  ::  ([*],*) -> num
occurs  (any, item) = xoccurs (any, item, 0)

xoccurs ::  ([*],*,num) -> num
xoccurs ([], item, total) = total
xoccurs ((front :  rest), item, total)
    = xoccurs (rest, item, total + 1), if front = item
    = xoccurs (rest, item, total), otherwise
```

This version of `occurs` will achieve the same results as the two previous versions, but is more difficult to read and reason about, probably because its definition is more procedural in nature and further away from its natural language specification.

Tail recursive functions

The polymorphic definition of the function `mymember` in Section 3.6 is an example of a *tail recursive* function. A tail recursive function is one wherein at no stage is evaluation suspended.[6] Another example is the function `mylast`, which selects the final item in a non-empty list or raises an `error` if the list is empty:

```
mylast [*] -> *
mylast [] = error "mylast:  empty list"
mylast (front :  []) = front
mylast (front :  rest) = mylast rest
```

Note that this function is so commonly required that it is also provided in the Miranda Standard Environment (with the name `last`).

[6]This has an important consequence for functional language implementation because many implementations detect tail recursive functions and produce code which uses constant stack storage space (Field and Harrison, 1988; MacLennan, 1990).

The functions `mymember` and `mylast` are “pure” tail recursive functions; they exhibit neither stack nor accumulative recursive style. Accumulative recursive functions are also tail recursive when they do not suspend any evaluation. By contrast, stack recursive functions can never be tail recursive.

3.8.1 Mutual recursion

Sometimes it is tempting to define functions in terms of each other (that is, these functions are mutually recursive). For example, the following (not recommended) program deletes all text within brackets from a given string:

```
string == [char]

nasty_skipbrackets :: string -> string

nasty_skipbrackets [] = []
nasty_skipbrackets ('(' : rest)
    = nasty_inbrackets rest
nasty_skipbrackets (front : rest)
    = front : nasty_skipbrackets rest

nasty_inbrackets :: string -> string

nasty_inbrackets []
    = error "text ends inside a bracket pair"
nasty_inbrackets (')' : rest)
    = nasty_skipbrackets rest
nasty_inbrackets (front : rest)
    = nasty_inbrackets rest
```

Is mutual recursion desirable?

Mutually recursive solutions are often considered undesirable because it is generally unclear which function is calling the other with any particular values. Furthermore, it is also impossible to test each individual function separately. Fortunately, the mutual dependencies can often be eliminated.

In the following program, the function `skipbrackets` scans the text until it finds the start of brackets. At this point it passes the remainder of the input text to `inbrackets`, which strips the rest of the current bracketed expression and returns the remaining text. The function `skipbrackets` can now use this result to find the next bracketed expression:

```
string == [char]

skipbrackets ::  string -> string

skipbrackets []
     = []
skipbrackets ('(' :  rest)
     = skipbrackets (inbrackets rest)
skipbrackets (front :  rest)
     = front :  skipbrackets rest

inbrackets ::  string -> string

inbrackets []
     = error "text ends inside a bracket pair"
inbrackets (')' :  rest)
     = rest
inbrackets (front :  rest)
     = inbrackets rest
```

Exercise 3.11
Modify the `skipbrackets` program to cater for nested brackets.

3.9 Extended example—grep

This section introduces the development of the UNIX program *grep* which will be used as an extended example throughout this book. The rest of this chapter shows how the Miranda facilities already demonstrated may be used to develop a real software tool. The program will be developed incrementally as new features of Miranda are introduced in subsequent chapters to make the design easier to understand and of more general utility.

Grep specification

Grep is a UNIX program which displays each line from its input that contains an instance of a given pattern. As such, it is clearly very useful for a number of different tasks, including the location of keywords and predefined names within a program or selecting interesting parts of a document.

Table 3.1 Options for the UNIX *grep* facility.

Character	Meaning
c	any non-special character *c* matches itself
\c	turn off any special meaning of character *c*
^	beginning of line
$	end of line
.	any single character
`[...]`	any one of characters in ...; ranges like a-z are legal
`[^...]`	any one of characters not in ... ; ranges are legal
*r**	zero or more occurrences of a regular expression *r*
r1r2	regular expression *r1* followed by regular expression *r2*
No regular expression matches a newline	

For example, assuming the file called "textbook" contains the lines:

```
Programming in Miranda
by
C.Clack, C.Myers and E.Poon
```

then typing the following in response to the UNIX prompt:

```
grep Miranda textbook
```

will result in the first line of the file being displayed.

In practice *grep* is more than a simple pattern matching tool; it also caters for what are known as *regular expressions*; (Aho *et al.*, 1974; Kernighan and Pike, 1984). In a regular expression certain characters may be *meta-characters*, which have special meanings. In the case of *grep* the meanings of characters, in order of precedence, are given in Table 3.1.[7]
Thus, it is possible to search for a pattern that is anchored at the beginning or end of a line and more importantly to search for patterns that contain "wild cards". For example, `^M.R` will match any line starting with `M` followed by any character, followed by `R` as the third character.

Similarly, `M.*R` will match:

```
MIR or M.R or MXR or MAAAAAAR or MABCR or MR
```

anywhere on the searched line.

[7]This table was adapted from Section 4.1 of the *Unix Programming Environment* (Kernighan and Pike, 1984). The actual UNIX tool has several more options.

Furthermore,

```
[a-zA-Z][a-zA-Z_1-9]*
```

will match any line that starts with an alphabetic character, followed by zero or more occurrences of an alphabetic character, digit or underscore, *in any combination*, such as:

```
fun
Miranda
legal_name
aux1
aux2_fun
```

It should be clear from the above why *grep* is sometimes known as a "filter" program; only the part of the input that matches the pattern is output, all other parts are filtered out.

Grep development considerations

The development of *grep* will be presented in a number of stages:

1. A simple program to indicate if a given pattern appears in a line. This is *grep* without any meta-characters. This version of *grep* will deal with just one line of input, and merely return the Boolean value `True` or `False`, depending on whether the given pattern has been matched or not.
2. A program to allow the * meta-character to appear in the pattern. The rationale for considering this as the next case is; if zero or more occurrences of a normal character can be matched then it is likely that zero or more occurrences of a wild card can also be matched.
3. An extended program to cater for many lines of input returning the text of each line that has a successful match. This is presented in Chapter 4.
4. A safer version using enumerated types, is presented in Chapter 6.
5. A version that deals with file handling is presented in Chapter 8.
6. A program to cater for the other regular expression meta-characters, including wild cards. This is presented in Chapter 9.

3.9.1 A simple version of grep

Specification

The *grep* program will return the truth value of whether the first component of its tuple pair argument is a substring of the second component. Therefore it has the type:

```
([char],[char]) -> bool
```

Design

The requirement for a successful *grep* search is that the regular expression (the pattern) either matches from the front of the searched line, or it matches from the front of the rest of the searched line. This is the general case of the structural induction. The terminating cases are:

1. The empty regular expression, which matches any line.
2. The empty searched line, which only matches an empty regular expression.

Implementation

Since it is the intention to deal with regular expressions, it is worthwhile providing a simple front-end to perform this action before any further processing:

```
string == [char]

grep ::  (string, string) -> bool
grep (regexp, line) = sublist (regexp, line)
```

The rest of the design translates directly to the Miranda code:

```
string == [char]

sublist ::  (string, string) -> bool
sublist ([], any) = True
sublist (any, []) = False
sublist (regexp, line)
     = startswith (regexp, line)
       \/ sublist (regexp, tl line)
```

It is important that the two base cases are defined in the above sequence. The cases are not mutually exclusive; if they were swapped then matching an empty regular expression and an empty input line would incorrectly evaluate to `False`.

Design of startswith

This has already been done in Section 3.7.2.

Exercise 3.12
It would appear that `sublist` no longer needs its first function pattern because this is checked as the first pattern in `startswith`. Explain why this is incorrect, and also whether the second pattern of `sublist` can safely be removed.

Exercise 3.13

An incorrect attempt to optimize the `startswith` program would combine `startswith` and `sublist` in one function:

```
stringpair == ([char], [char])

sublist :: stringpair -> bool
sublist ([], any) = True
sublist (any, []) = False
sublist ((regfront : regrest), (lfront : lrest))
      = ((regfront = lfront) & sublist (regrest, lrest))
         \/ sublist (regfront : regrest, lrest)
```

This follows the general inductive case that the result is true if the front two items of the lists are equal and the result of a sublist search of the rest of the two lists is also true. Alternatively, the entire regular expression matches the rest of the search line. Show why this approach is wrong.

3.9.2 Incorporating the zero or more occurrences

The next stage of the development is to allow for zero or more occurrences of a given character to appear in the searched line. For example:

```
A*BC     appears in          AAABC
                             ABC
                             BC
                             AAABAAABC

A*A      appears in          AAAAAA
                             AA
                             A
```

At this point it is also worth recalling that * itself can be matched by preceding it with a \which turns off its special meaning:

```
A\*B    appears in          A*B

A\*B    does not appear in  AABC
                            ABC
                            BC
                            AAABAAABC
```

The need for lexical analysis

The design of *grep* is now made more complex by the necessity of always checking for the * meta-character following any other character. There are two approaches to handling this. One approach is that the auxiliary function `startswith` could look ahead for the meta-character *. However, this *confuses* the activity of recognizing the meta-characters in the regular expression with the activity of comparing the regular expression with the current search line. This makes the resultant function `startswith` very complicated and will go against the advice in Chapter 2 that a function should only do one thing.

In preference, the regular expression could be preprocessed to attribute every character with a "match-type token" to indicate whether it is to be matched *once only*, as a normal character, or whether it is to be matched *zero or more* times.[8] For this version of *grep*, a new function called `lex` is used to do the preprocessing; this function will initially recognize `\` and `*` but will be extended later to deal with the other meta-characters.[9] The easiest way of representing the two attributes is probably a tuple of the form `(MATCH_TYPE, actual_value)`. For example `("ONCE", ch)` or `("ZERO_MORE", ch)`, as shown in Figure 3.3.

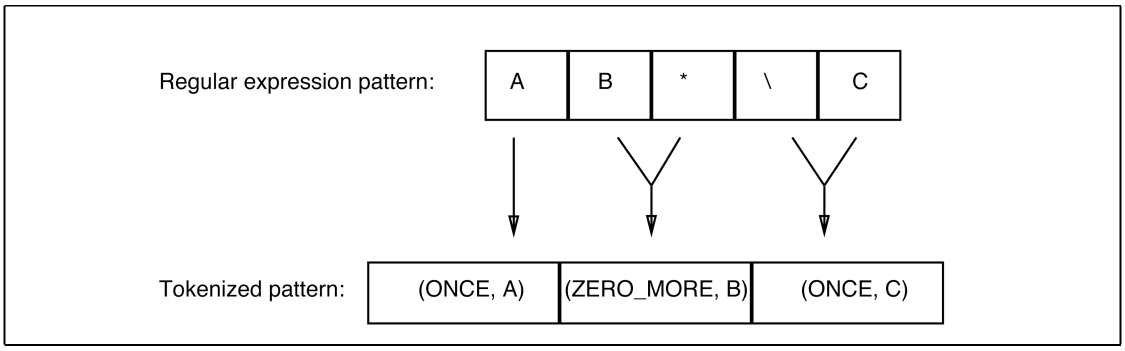

Figure 3.3 Example of a regular expression in tokenized form.

Pattern matching, by case analysis, can now be employed to convert the raw regular expression into a list of regular expression elements:[10]

[8]With this approach, `startswith` is also saved the bother of checking for the escape sequence `\c`, where c is any character.

[9]This technique of transforming the regular expression into a sequence of actual values and their match-type token is known as lexical analysis (Aho *et al.*, 1974; Kernighan and Pike, 1984) hence the name of the function `lex`.

[10]The reader is referred to Chapter 6 for a more elegant approach using *enumerated types*.

```
string  == [char]
mtype   == string
regtype == (mtype, char)
reglist == [regtype]

lex ::  string -> reglist

lex [] = []

lex ('\\' :  ch :  '*' :  rest)
        = ("ZERO_MORE", ch) :  lex rest
        || note the two char '\\'
        || is treated as the single char '\'
lex ('\\' :  ch :  rest) = ("ONCE", ch) :  lex rest
lex ('\\' :  [])      = [("ONCE", '\\')]
lex (ch :  '*' :  rest)  = ("ZERO_MORE", ch) :  lex rest
lex (ch :  rest)      = ("ONCE", ch) :  lex rest
```

3.9.3 Redesign of startswith

Matching strategy

Incorporating *ZERO_MORE* match-types into the `startswith` function requires the careful consideration of two additional factors:

1. Each element of the regular expression list may be either a *ONCE* match-type or a *ZERO_MORE* match-type. It is necessary to treat them separately because they lead to different inductive hypotheses.
2. There are many possible sub-lines within a line that can match a regular expression, especially one involving a *ZERO_MORE* match-type; for example, *A*A* matches both *A* and *AAAAAAAAAA*. In order to continue the design, it is necessary to adopt a strategy to deal with all possibilities in a consistent manner. The most common strategies are:

   (a) Look for the shortest possible match; for example, to match the regular expression *A*A* it is only necessary to inspect the first character in the string *AAAAAAA*. (Notice, however, that matching *A*B* against *AAAAAAAB* still requires the inspection of the entire second string.)

   (b) Look for the longest possible match; for example, to match the regular expression *A*A* it is necessary to inspect all of the characters in the string *AAAAAAA*.

Since `grep` only returns a Boolean value, either strategy is valid; the efficiency of the strategy depends mainly on the nature of the regular expression and the searched line. The program presented in this book will follow the strategy of the shortest possible match.[11]

Design and implementation

The design proceeds by identifying the important cases for analysis:

1. The general case, where both the searched list and the regular expression list are non-empty. This must be split into two subcases to cater for the two different match-types.
2. The base cases, for the empty regular expression list and for the empty searched list. In the latter case, once again there is a need to cater for the two different match-types.

This leads to the following five alternative patterns:

```
startswith ([], any)
    =  Body_1
startswith (("ONCE", ch) : regrest, [])
    =  Body_2
startswith (("ZERO_MORE", ch) : regrest, [])
    =  Body_3
startswith (("ONCE", ch) : regrest, (lfront : lrest))
    =  Body_4
startswith (("ZERO_MORE", ch) : regrest, (lfront : lrest))
    =  Body_5
```

Finally, in accordance with the advice given in Chapter 2, a default pattern is added to help with debugging:

```
startswith anyother = error "startswith: illegal pattern"
```

The definitions of `Body_1`, `Body_2` and `Body_4` are essentially the same as those for the previous version of `startswith` (as explained in Section 3.7.2). However, the other definitions need further attention.

[11]However, many text editors with a “search and replace” command adopt the longest possible match strategy.

Body_5—ZERO_MORE general case

Given the pattern `(("ZERO_MORE",ch) : regrest, (lfront : lrest))`, there are two successful matching possibilities: either to *zero* occurrences of `ch` or to the *first* of *one or more* occurrences of `ch` (which requires further processing to determine how many more occurrences of `ch` exist). The "shortest possible" match strategy first assumes a match to zero occurrences of `ch` and has the inductive hypothesis that `(startswith (regrest, lfront : lrest))` will operate correctly. If the recursion evaluates to `True` then the entire search evaluates to `True`. Otherwise, the strategy is to "backtrack" to consider the possibility of a single occurrence match. This search will be successful if `ch` matches `lfront` and also if all of the current regular expression matches the rest of the search line.[12]

This leads to the following Miranda code for Body_5:

```
startswith (regrest, (lfront : lrest))
\/
((ch = lfront) & startswith (("ZERO_MORE",ch) : regrest, lrest))
```

It is interesting to observe that if `\/` did not delay the evaluation of its second argument then the function would evaluate all possible matches for the regular expression to the searched list. By contrast, any actual evaluation is limited to a sequential search from the shortest to the longest, stopping when an overall match is found.

Body_5 at work

The following shows the code for `Body_5` at work on two exemplary cases:

1. A successful match of the regular expression `"A*A"` against the input line `"AAB"` which demonstrates the shortest possible match strategy.
2. A successful match of the regular expression `"A*B"` against the input line `"AAB"` which demonstrates the backtracking mechanism.

For clarity, the hand evaluation is shown using the "raw" regular expression and searched-line strings rather than their tokenized list versions.

1. Shortest match strategy

```
grep ("A*A", "AAB")
==> startswith ("A*A", "AAB")  || i.e. Pattern_5
==> startswith ("A", "AAB")    || i.e. Pattern_4
==> True
```

[12]Note that if the searched list contains a long sequence of `ch` characters then this could lead to a large number of delayed decisions and indeed backtracking may occur many times before the appropriate match is found.

2. Backtracking mechanism

```
grep ("A*B", "AAB")
==> startswith ("A*B", "AAB")            || i.e. Pattern_5
==> startswith ("B", "AAB")              || i.e. Pattern_4
==> False
     \/
    (
    "A" = "A"
        &
     startswith ("A*B", "AB")            || i.e. Pattern_5
        ==> startswith ("B", "AB")       || i.e. Pattern_4
        ==> False
            \/
             (
             "A" = "A"
                 &
              startswith ("A*B", "B")    || i.e. Pattern_5
              ==> startswith ("B", "B")  || i.e. Pattern_4
              ==> True
              )
        ==> True
    )
==> True
```

Body_3—ZERO_MORE at end of line

This leaves the case for `Body_3`, which can be considered as a successful zero-occurrence match. One of the parameters of recursion has reached termination (or was `[]` to start with), but the other still needs further processing to handle regular expressions of the form *A*B** (which can successfully match an empty search list). The inductive hypothesis is the same as for `Body_5`: the inductive step is therefore to recurse using the rest of the regular expression—the evaluation will either converge towards an empty regular expression and succeed or a *ONCE* match-type will be found in the regular expression and the overall search will fail. The code is:

```
startswith (("ZERO_MORE", ch) : regrest, [])
    = startswith (regrest, [])
```

Final code for startswith

Piecing this all together gives:

```
string      == [char]
mtype       == string
regtype     == (mtype,char)
reglist     == [regtype]

startswith ::  (reglist, string) -> bool

startswith ([], any) = True
startswith (("ONCE", ch) :  regrest, []) = False
startswith (("ZERO_MORE", ch) :  regrest, [])
    = startswith (regrest, [])
startswith (("ONCE", ch) :  regrest, lfront :  lrest)
    = (ch = lfront) & startswith (regrest, lrest)
startswith (("ZERO_MORE", ch) :  regrest, lfront :  lrest)
    = startswith (regrest, (lfront :  lrest)) \/
      ((ch = lfront) &
      startswith (("ZERO_MORE", ch) :  regrest, lrest))
startswith any = error "startswith"
```

In order to complete the implementation, the `grep` and `sublist` functions need amending to preprocess the raw regular expression:

```
sublist ::  (reglist, string) -> bool

sublist ([], any) = True
sublist (any, []) = False
sublist (regexp, line)
    = startswith (regexp, line) \/
         sublist (regexp, tl line)

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

Furthermore, the `sublist` function needs a little extra modification so that a regular expression such as `A*` matches the empty string (as shown in the answer to the final exercise in this chapter).

Exercise 3.14

Explain the presence of the final pattern in the function `startswith`, even though it should never be encountered.

Exercise 3.15

What would happen if the second and third pattern in `startswith` were swapped?

Exercise 3.16

Alter the `sublist` function so that `"A*"` matches the empty string.

3.10 Summary

This chapter has introduced the recursively defined aggregate list data type. A new list may be created from an existing list and a new value by use of the *constructor* **:**. Chapter 6 will extend this concept to user-defined recursive types with user-defined constructors.

The techniques of *case analysis* and *structural induction* were introduced as an aid to designing list-handling functions. Recursive function definition is of such fundamental importance to the manipulation of the list data structure that the general technique was analysed further and five common modes of recursion over lists were discussed: tail recursion, stack recursion, filter recursion, accumulative recursion and mutual recursion. Chapter 4 shows how higher order functions may be used to encapsulate common recursive forms.

This chapter ended with the design and development of the *grep* program, using the tools and techniques learned so far. Later in the book we will show how more advanced techniques can be applied to the same real-world example to produce a better design.