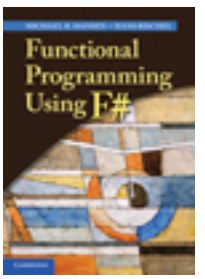


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

4 - Lists pp. 67-92

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.005

Cambridge University Press

4

Lists

Lists are at the core of functional programming. A large number of applications can be modelled and implemented using lists. In this chapter we introduce the list concept, including list values, patterns and basic operations, and we study a collection of recursion schemas over lists. We end the chapter introducing a model-based approach to functional programming on the basis of two examples. The concept of a list is a special case of a collection. In the next chapter, when we consider collections more generally, we shall see that the F# library comprises a rich collection of powerful functions on lists.

4.1 The concept of a list

A *list* is a finite sequence of values

$$[v_0; v_1; \ldots; v_{n-1}]$$

of the same type. For example, `[2]`, `[3; 2]`, and `[2; 3; 2]` are lists of integers. A list can contain an arbitrary number of elements.

A list $[v_0; v_1; \ldots; v_{n-1}]$ is either empty (when $n = 0$), or it is a non-empty list and can be characterized by the first element v_0 called its *head*, and the rest $[v_1; \ldots; v_{n-1}]$ called its *tail*.

Figure 4.1 shows the graphs for the lists `[2; 3; 2]` and `[2]`. The list `[2; 3; 2]` is

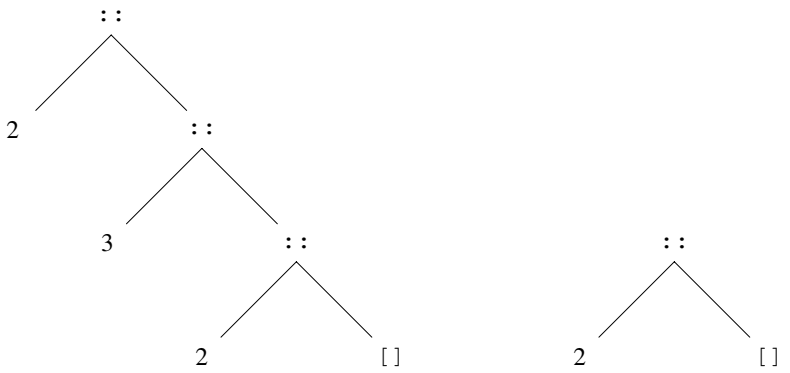


Figure 4.1 Graphs for two lists

hence a tagged pair with tag `::` where the first component, the head of the list, is the integer `2`, while the second component, the tail of the list, is the list `[3; 2]` with just two elements. This list is again a tagged pair with tag `::`, head `3` and tail `[2]`. Finally the head of the list `[2]` is the integer `2`, while the tail is the empty list `[]`.

List constants in F#

Lists can be entered as values:

```
let xs = [2;3;2];;
val xs : int list = [2; 3; 2]

let ys = ["Big"; "Mac"];;
val ys  : string list = ["Big"; "Mac"]
```

The types `int list` and `string list`, containing the *type constructor* `list`, indicate that the value of `xs` is a list of integers and that the value of `ys` is a list of strings.

We may have lists with any element type, so we can, for example, build lists of pairs:

```
[("b",2);("c",3);("e",5)];;
val it : (string * int) list = [("b", 2);("c", 3);("e", 5)]
```

lists of records:

```
type P = {name: string; age: int}
[{name = "Brown"; age = 25}; {name = "Cook";  age = 45}];;
val it : P list =
  [{name = "Brown"; age = 25}; {name = "Cook"; age = 45}]
```

lists of functions:

```
[sin; cos];;
val it : (float -> float) list = [<fun:it@7>; <fun:it@7-1>]
```

or even lists of lists:

```
[[2;3];[3];[2;3;3]];;
val it : int list list = [[2; 3]; [3]; [2; 3; 3]]
```

Furthermore, lists can be components of other values. We can, for example, have pairs containing lists:

```
("bce", [2;3;5]);;
val it : string * int list = ("bce", [2; 3; 5])
```

The type constructor `list`

The type constructor `list` has higher *precedence* than `*` and `->` in *type expressions*, so the type `string * int list` means `string * (int list)`. The type constructor `list` is used in postfix notation like the factorial function _! in 3! and associates to the left, so

`int list list` means `(int list) list`. Note that `int (list list)` would not make sense.

All elements in a list must have the same type. For example, the following is *not* a legal value in F#:

```
["a";1];;
-----^
stdin(8,6): error FS0001:
This expression was expected to have type
    string
but here has type
    int
```

Equality of lists

Two lists `[`x_0`;`x_1`; ...;`x_{m-1}`]` and `[`y_0`;`y_1`; ...,`y_{n-1}`]` (of the same type) are *equal* when $m = n$ and $x_i = y_i$, for all i such that $0 \leq i < m$. This corresponds to equality of the graphs represented by the lists. Hence, the order of the elements as well as repetitions of the same value are significant in a list.

The equality operator = of F# can be used to test equality of two lists provided that the elements of the lists are of the same type and provided that the equality operator can be used on values of that element type.

For example:

```
[2;3;2] = [2;3];;
val it : bool = false

[2;3;2] = [2;3;3];;
val it : bool = false
```

The differences are easily recognized from the graphs representing `[2; 3; 2]`, `[2; 3]` and `[2; 3; 3]`.

Lists containing functions cannot be compared because F# equality is not defined for functions.

For example:

```
[sin; cos] = [];;
_^^^
... The type '( ^a ->  ^a ) when
^a : (static member Sin :  ^a ->  ^a)' does not support
the 'equality' constraint because it is a function type
```

Ordering of lists

Lists of the same type are ordered *lexicographically*, provided there is an ordering defined on the elements:

$$[x_0; x_1; \ldots; x_{m-1}] < [y_0; y_1; \ldots; y_{n-1}]$$

exactly when

$$[x_0; x_1; \ldots; x_k] = [y_0; y_1; \ldots; y_k] \quad \text{and} \quad \left(\begin{array}{ll} & k = m-1 < n-1 \\ \text{or} & k < \min\{m-1, n-1\} \text{ and } x_{k+1} < y_{k+1} \end{array} \right)$$

for some k, where $0 \leq k < \min\{m-1, n-1\}$.

There are two cases in this definition of $xs < ys$:

1. The list xs is a *proper prefix* of ys:

```
[1; 2; 3] < [1; 2; 3; 4];;
val it : bool = true

['1'; '2'; '3'] < ['1'; '2'; '3'; '4'];;
val it : bool = true
```

   The examples illustrate comparisons of integer lists and character lists. Furthermore, the empty list is smaller than any non-empty list:

```
[] < [1; 2; 3];;
val it : bool = true

[] < [[]; [(true,2)]];;
val it : bool = true
```

2. The lists agree on the first k elements and $x_{k+1} < y_{k+1}$. For example:

```
[1; 2; 3; 0; 9; 10] < [1; 2; 3; 4];;
val it : bool = true

["research"; "articles"] < ["research"; "books"];;
val it : bool = true
```

   because `0 < 4` and `"articles" < "books"`.

The other comparison relations can be defined in terms of = and < as usual. For example:

```
[1; 1; 6; 10] >= [1; 2];;
val it : bool = false
```

The `compare` function is defined for lists, provided it is defined for the element type. For example:

```
compare [1; 1; 6; 10] [1; 2];;
val it : int = -1

compare [1;2] [1; 1; 6; 10];;
val it : int = 1
```

4.2 Construction and decomposition of lists

The `cons` *operator*

The infix operator `::` (called "cons") builds a list from its head and its tail as shown in Figures 4.2 and 4.3 so it adds an element at the front of a list:

```
let x = 2::[3;4;5];;
val x : int list = [2; 3; 4; 5]

let y = ""::[];;
val y : string list = [""]
```

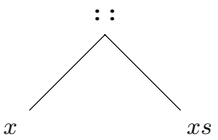


Figure 4.2 Graph for the list $x::xs$

The operator *associates* to the *right*, so $x_0::x_1::xs$ means $x_0::(x_1::xs)$ where x_0 and x_1 have the same type and xs is a list with elements of that same type (cf. Figure 4.3) so we get, for example:

```
let z = 2::3::[4;5];;
val z : int list = [2; 3; 4; 5]
```

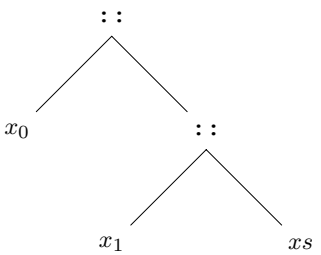


Figure 4.3 Graph for the list $x_0::(x_1::xs)$

List patterns

While the cons operator can be used to construct a list from a (head) element and a (tail) list, it is also used in *list patterns*. List patterns and pattern matching for lists are used in the subsequent sections to declare functions on lists by using the bindings of identifiers in a list pattern obtained by matching a list to the pattern.

There is the list pattern `[]` for the empty list while patterns for non-empty lists are constructed using the cons operator, that is, `x::xs` matches a non-empty list.

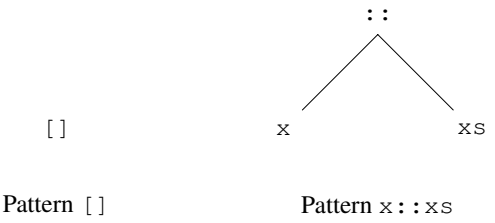


Figure 4.4 Graphs for list patterns

The patterns `[]` and `x::xs` denote the graphs in Figure 4.4. The pattern `[]` matches the empty list only, while the pattern `x::xs` matches any non-empty list $[x_0; x_1; \ldots; x_{n-1}]$. The latter matching gives the bindings $\texttt{x} \mapsto x_0$ and $\texttt{xs} \mapsto [x_1; \ldots; x_{n-1}]$ of the identifiers `x` and `xs`, as the list $[x_0; x_1; \ldots; x_{n-1}]$ denotes the graph in Figure 4.5.

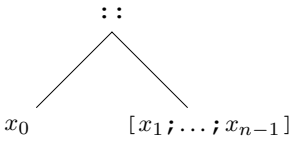


Figure 4.5 Graph for the list $[x_0; x_1; \ldots; x_{n-1}]$

For example, the execution of the declarations:

```
let x::xs = [1;2;3];;
val xs : int list = [2; 3]
val x : int = 1
```

will simultaneously bind `x` to the value 1 and `xs` to the value `[2;3]` by matching the value `[1;2;3]` to the pattern `x::xs`.

A list pattern for a list with a fixed number of elements, for example, three, may be written as `x0::x1::x2::[]` or in the shorter form `[x0;x1;x2]`. This pattern will match any list with precisely three elements $[x_0; x_1; x_2]$, and the matching binds `x0` to x_0, `x1` to x_1, and `x2` to x_2. For example:

```
let [x0;x1;x2] = [(1,true); (2,false); (3, false)];;
  let [x0;x1;x2] = [(1,true); (2,false); (3, false)];;
  ____^^^^^^^^^^
stdin(1,5): warning FS0025: Incomplete pattern matches on this
expression. For example, the value '[_;_;_;_]' may indicate a
case not covered by the pattern(s).
val x2 : int * bool = (3, false)
val x1 : int * bool = (2, false)
val x0 : int * bool = (1, true)
```

This generalizes to any fixed number of elements. (The F# compiler issues a warning because list patterns with a fixed number of elements are in general not recommended, but the bindings are, nevertheless, made.)

List patterns may have more structure than illustrated above. For example, we can construct list patterns that match lists with two or more elements (e.g., `x0::x1::xs`), and list patterns matching only non-empty lists of pairs (e.g., `(y1,y2)::ys`), and so on. For example:

```
let x0::x1::xs = [1.1; 2.2; 3.3; 4.4; 5.5];;
val xs : float list = [3.3; 4.4; 5.5]
val x1 : float = 2.2
val x0 : float = 1.1

let (y1, y2)::ys = [(1,[1]); (2, [2]); (3, [3]); (4,[4])];;
val ys : (int * int list) list =
         [(2, [2]); (3, [3]); (4, [4])]
val y2 : int list = [1]
val y1 : int = 1
```

We shall see examples of more involved patterns in this chapter and throughout the book.

Note the different roles of the operator symbol `::` in patterns and expressions. It denotes decomposing a list into smaller parts when used in a pattern like `x0::x1::xs`, and it denotes building a list from smaller parts in an expression like `0::[1; 2]`.

Simple list expressions

In F# there are special constructs that can generate lists. In this section we will just introduce the two simple forms of expressions called *range expressions*:

$$[b\,..\,e] \qquad\qquad [b\,..\,s\,..\,e]$$

where b, e and s are expressions having number types.

The range expression $[b\,..\,e]$, where $e \geq b$, generates the list of consecutive elements:

$$[b;\ b+1;\ b+2; \ldots;\ b+n]$$

where n is chosen such that $b + n \leq e < b + n + 1$. The range expression generates the empty list when $e < b$.

For example, the list of integers from -3 to 5 is generated by:

```
[ -3 .. 5 ];;
val it : int list = [-3; -2; -1; 0; 1; 2; 3; 4; 5]
```

and a list of floats is, for example, generated by:

```
[2.4 .. 3.0 ** 1.7];;
val it : float list = [2.4; 3.4; 4.4; 5.4; 6.4]
```

Note that `3.0 ** 1.7` = `6.47300784`.

The expression s in the range expression `[`b `..` s `..` e`]` is called the step. It can be positive or negative, but not zero:

$$\texttt{[}b\ \texttt{..}\ s\ \texttt{..}\ e\texttt{]} = \texttt{[}b\texttt{;}\, b+s\texttt{;}\, b+2s\texttt{;}\, \ldots\texttt{;}\, b+ns\texttt{]}$$
$$\text{where} \quad \begin{cases} b+ns \le e < b+(n+1)s & \text{if } s \text{ is positive} \\ b+ns \ge e > b+(n+1)s & \text{if } s \text{ is negative} \end{cases}$$

The generated list will be either ascending or descending depending on the sign of s. For example, the descending list of integers from 6 to 2 is generated by:

```
[6 .. -1 .. 2];;
val it : int list = [6; 5; 4; 3; 2]
```

and the float-based representation of the list consisting of $0, \pi/2, \pi, \frac{3}{2}\pi, 2\pi$ is generated by:

```
[0.0 .. System.Math.PI/2.0 .. 2.0*System.Math.PI];;
val it : float list =
     [0.0; 1.570796327; 3.141592654; 4.71238898; 6.283185307]
```

An exception is raised if the step is 0:

```
> [0 .. 0 .. 0];;
System.ArgumentException: The step of a range cannot be zero.
Parameter name: step
 ......
Stopped due to error
```

4.3 Typical recursions over lists

In this section we shall consider a collection of archetypical recursive function declarations on lists.

Function declarations with two clauses

Let us consider the function `suml` that computes the sum of a list of integers:

$$\texttt{suml [}x_0\texttt{;}\, x_1\texttt{;}\, \ldots\texttt{;}\, x_{n-1}\texttt{]} = \sum_{i=0}^{n-1} x_i = x_0 + x_1 + \cdots + x_{n-1} = x_0 + \sum_{i=1}^{n-1} x_i$$

We get the recursion formula:

$$\texttt{suml [}x_0\texttt{;}\, x_1\texttt{;}\, \ldots\texttt{;}\, x_{n-1}\texttt{]} = x_0 + \texttt{suml [}x_1\texttt{;}\, \ldots\texttt{;}\, x_{n-1}\texttt{]}$$

We define the value of the "empty" sum, that is, `suml []`, to be 0 and we arrive at a recursive function declaration with two clauses:

```
let rec suml = function
    | []    -> 0
    | x::xs -> x + suml xs;;
val suml : int list -> int
```

In evaluating a function value for `suml` xs, F# scans the clauses and selects the first clause where the argument matches the pattern. Hence, the evaluation of `suml [1;2]` proceeds as follows:

```
    suml [1;2]
⇝   1 + suml [2]          (x::xs matches [1;2] with x ↦ 1 and xs ↦ [2])
⇝   1 + (2 + suml [])     (x::xs matches [2] with x ↦ 2 and xs ↦ [])
⇝   1 + (2 + 0)           (the pattern [] matches the value [])
⇝   1 + 2
⇝   3
```

This example shows that patterns are convenient in order to split up a function declaration into clauses covering different forms of the argument. In this example, one clause of the declaration gives the function value for the empty list, and the other clause reduces the computation of the function value for a non-empty list `suml`$(x$`::`$xs)$ to a simple operation (addition) on the head x and the value of `suml` on the tail `xs` (i.e., `suml` xs), where the length of the argument list has been reduced by one.

It is easy to see that an evaluation for `suml` $[x_0$`;` $\ldots$`;` $x_{n-1}]$ will terminate, as it contains precisely $n+1$ recursive calls of `suml`.

The above declaration is an example of a typical recursion schema for the declaration of functions on lists.

Function declarations with several clauses

One can have function declarations with any number (≥ 1) of clauses. Consider, for example, the alternate sum of an integer list:

$$\texttt{altsum}\ [x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}] = x_0 - x_1 + x_2 - \cdots + (-1)^{n-1}x_{n-1}$$

In declaring this function we consider three different forms of the argument:

1. empty list: `altsum []` $= 0$
2. list with one element: `altsum` $[x_0] = x_0$
3. list with two or more elements:

$$\texttt{altsum}\ [x_0\texttt{;}x_1\texttt{;}x_2\texttt{;}\ldots\texttt{;}x_{n-1}] = x_0 - x_1 + \texttt{altsum}\ [x_2\texttt{;}\ldots\texttt{;}x_{n-1}]$$

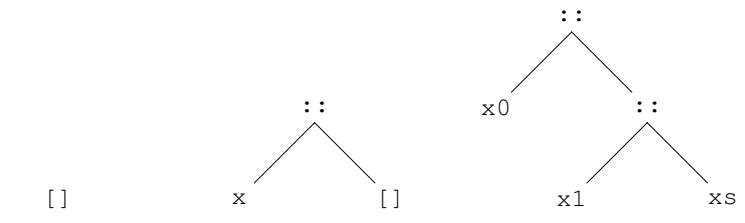


Figure 4.6 List patterns for `altsum` declaration

These cases are covered by the patterns in Figure 4.6. Thus, the function can be declared by:

```
let rec altsum = function
    | []          -> 0
    | [x]         -> x
    | x0::x1::xs -> x0 - x1 + altsum xs;;
val altsum : int list -> int

altsum [2; -1; 3];;
val it : int = 6
```

It is left as an exercise to give a declaration for `altsum` containing only two clauses.

Layered patterns

We want to define a function `succPairs` such that:

`succPairs []` $= []$
`succPairs` $[x] = []$
`succPairs` $[x_0; x_1; \ldots; x_{n-1}] = [(x_0, x_1); (x_1, x_2); \ldots; (x_{n-2}, x_{n-1})]$

Using the pattern `x0::x1::xs` as in the above example we get the declaration

```
let rec succPairs = function
    | x0 :: x1 :: xs -> (x0,x1) :: succPairs(x1::xs)
    | _              -> [];;
val succPairs : 'a list -> ('a * 'a) list
```

This works OK, but we may get a smarter declaration avoiding the cons expression `x1::xs` in the recursive call in the following way:

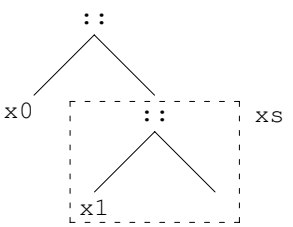


Figure 4.7 A pattern `x0::(x1::_ as xs)` containing a layered sub-pattern `x1::_ as xs`

```
let rec succPairs = function
    | x0::(x1::_ as xs) -> (x0,x1) :: succPairs xs
    | _                 -> [];;
val succPairs : 'a list -> ('a * 'a) list

succPairs [1;2;3];;
val it : (int * int) list = [(1, 2); (2, 3)]
```

The pattern `x1::_ as xs` is an example of a *layered pattern*. It is part of the pattern shown in Figure 4.7. A layered pattern has the general form:

$$pat \texttt{ as } id$$

with pattern *pat* and identifier *id*. A value *val* matches this pattern exactly when the value matches the pattern *pat*. The matching binds identifiers in the pattern *pat* as usual with the addition that the identifier *id* is bound to *val*. Matching the list $[x_0; x_1; \ldots]$ with the pattern `x0::(x1::_ as xs)` will hence give the following bindings:

$$\begin{array}{lcl} \texttt{x0} & \mapsto & x_0 \\ \texttt{x1} & \mapsto & x_1 \\ \texttt{xs} & \mapsto & [x_1; \ldots] \end{array}$$

which is exactly what is needed in this case.

Pattern matching on result of recursive call

The following example illustrates the use of pattern matching to split the result of a recursive call into components. The function `sumProd` computes the pair consisting of the sum and the product of the elements in a list of integers, that is:

$$\begin{array}{l} \texttt{sumProd } [x_0; x_1; \ldots; x_{n-1}] \\ \qquad\qquad = \quad (\, x_0 + x_1 + \ldots + x_{n-1} \,,\, x_0 * x_1 * \ldots * x_{n-1} \,) \\ \texttt{sumProd } [\,] \quad = \quad (0,1) \end{array}$$

The declaration is based on the recursion formula:

$$\texttt{sumProd } [x_0; x_1; \ldots; x_{n-1}] = (x_0 + \texttt{rSum}, x_0 * \texttt{rProd})$$

where

$$(\texttt{rSum}, \texttt{rProd}) = \texttt{sumProd } [x_1; \ldots; x_{n-1}]$$

This gives the declaration:

```
let rec sumProd = function
    | []   -> (0,1)
    | x::rest ->
          let (rSum,rProd) = sumProd rest
          (x+rSum,x*rProd);;
val sumProd : int list -> int * int

sumProd [2;5];;
val it : int * int = (7, 10)
```

Another example is the `unzip` function that maps a list of pairs to a pair of lists:

$$\begin{array}{l} \texttt{unzip}([(x_0, y_0); (x_1, y_1); \ldots; (x_{n-1}, y_{n-1})] \\ \qquad = ([x_0; x_1; \ldots; x_{n-1}], [y_0; y_1; \ldots; y_{n-1}]) \end{array}$$

The declaration of `unzip` looks as follows:

```
let rec unzip = function
    | []          -> ([],[])
    | (x,y)::rest ->
          let (xs,ys) = unzip rest
          (x::xs,y::ys);;
val unzip : ('a * 'b) list -> 'a list * 'b list

unzip [(1,"a");(2,"b")];;
val it : int list * string list = ([1; 2], ["a"; "b"])
```

The `unzip` function is found as `List.unzip` is the F# library.

Pattern matching on pairs of lists

We want to declare a function `mix` that mixes the elements of two lists with the same length:

$$\texttt{mix ([}x_0\texttt{; }x_1\texttt{; ...; }x_{n-1}\texttt{], [}y_0\texttt{; }y_1\texttt{; ...; }y_{n-1}\texttt{])} = \texttt{[}x_0\texttt{; }y_0\texttt{; }x_1\texttt{; }y_1\texttt{; ...; }x_{n-1}\texttt{; }y_{n-1}\texttt{]}$$

It is declared using pattern matching on the pair of lists:

```
let rec mix = function
    | (x::xs,y::ys) -> x::y::(mix (xs,ys))
    | ([],[])       -> []
    | _             -> failwith "mix: parameter error";;
val mix : 'a list * 'a list -> 'a list

mix ([1;2;3],[4;5;6]);;
val it : int list = [1; 4; 2; 5; 3; 6]
```

The corresponding higher-order function is defined using a `match` expression:

```
let rec mix xlst ylst =
    match (xlst,ylst) with
    | (x::xs,y::ys) -> x::y::(mix xs ys)
    | ([],[])       -> []
    | _             -> failwith "mix: parameter error";;
val mix : 'a list -> 'a list -> 'a list

mix [1;2;3] [4;5;6];;
val it : int list = [1; 4; 2; 5; 3; 6]
```

4.4 Polymorphism

In this section we will study some general kinds of polymorphism, appearing frequently in connection with lists. We will do that on the basis of three useful list functions that all can be declared using the same structure of recursion as shown in Section 4.3.

List membership

The member function for lists determines whether a value x is equal to one of the elements in a list `[`y_0`;` y_1`; ...;` y_{n-1}`]`, that is:

$$\begin{aligned} & \texttt{isMember}\ x\ \texttt{[}y_0\texttt{;}\ y_1\texttt{;}\ \ldots\texttt{;}\ y_{n-1}\texttt{]} \\ = \ & (x = y_0) \text{ or } (x = y_1) \text{ or } \cdots \text{ or } (x = y_{n-1}) \\ = \ & (x = y_0) \text{ or } (\texttt{isMember}\ x\ \texttt{[}y_1\texttt{;}\ \ldots\texttt{;}\ y_{n-1}\texttt{]}) \end{aligned}$$

Since no x can be a member of the empty list, we arrive at the declaration:

```
let rec isMember x = function
    | y::ys -> x=y || (isMember x ys)
    | []    -> false;;
val isMember : 'a -> 'a list -> bool when 'a : equality
```

The function `isMember` can be useful in certain cases, but it is not included in the F# library.

The annotation `'a : equality` indicates that `'a` is an *equality type variable*; see Section 2.10. The equality type is inferred from the expression `x=y`. It implies that the function `isMember` will only allow an argument x where the equality operator = is defined for values of the type of x. A type such as `int * (bool * string) list * int list` is an equality type, and the function can be applied to elements of this type.

Append and reverse. Two built-in functions

The infix operator `@` (called 'append') joins two lists of the same type:

`[`x_0`;` x_1`; ...;` x_{m-1}`] @ [`y_0`;` y_1`; ...;` y_{n-1}`] =` `[`x_0`;` x_1`; ...;` x_{m-1}`;` y_0`;` y_1`; ...;` y_{n-1}`]`

and the function `List.rev` (called "reverse") reverses a list:

`List.rev [`x_0`;` x_1`; ...;` x_{n-1}`] =` `[`x_{n-1}`; ...;` x_1`;` x_0`]`

These functions are predefined in F#, but their declarations reveal important issues and are therefore discussed here. The operator `@` is actually the infix operator corresponding to the library function `List.append`.

The declaration of the (infix) function `@` is based on the recursion formula:

$$\begin{aligned} \texttt{[]}\ \texttt{@}\ ys &= ys \\ \texttt{[}x_0\texttt{;}\ x_1\texttt{;}\ \ldots\texttt{;}\ x_{m-1}\texttt{]}\ \texttt{@}\ ys &= x_0\texttt{::}(\texttt{[}x_1\texttt{;}\ \ldots\texttt{;}\ x_{m-1}\texttt{]}\ \texttt{@}\ ys) \end{aligned}$$

This leads to the declaration:

```
let rec (@) xs ys =
    match xs with
    | []     -> ys
    | x::xs' -> x::(xs' @ ys);;
val (@) : 'a list -> 'a list -> 'a list
```

The evaluation of append decomposes the left-hand list into its elements, that are afterwards 'cons'ed' onto the right-hand list:

```
    [1;2]@[3;4]
⇝   1::([2]@[3;4])
⇝   1::(2::([]@[3;4]))
⇝   1::(2::[3;4])
⇝   1::[2;3;4]
⇝   [1;2;3;4]
```

The evaluation of xs `@` ys comprises $m+1$ pattern matches plus m cons'es where m is the length of xs.

The notion of polymorphism is very convenient for the programmer because one need not write a special function for appending, for example, integer lists and another function for appending lists of integer lists, as the polymorphic append function is capable of both:

```
[1;2] @ [3;4];;
val it : int list = [1; 2; 3; 4]

[[1];[2;3]] @ [[4]];;
val it : int list list = [[1]; [2; 3]; [4]]
```

The operators `::` and `@` have the same precedence (5) and both associate to the *right*. A mixture of these operators also associates to the right, so `[1]@2::[3]`, for example, is interpreted as `[1]@(2::[3])`, while `1::[2]@[3]` is interpreted as `1::([2]@[3])`:

```
[1] @ 2 :: [3];;
val it : int list = [1; 2; 3]

1 :: [2] @ [3];;
val it : int list = [1; 2; 3]
```

For the reverse function `rev`, where

$$\texttt{rev [}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]} = \texttt{[}x_{n-1}\texttt{;}\ldots\texttt{;}x_1\texttt{;}x_0\texttt{]}$$

we have the recursion formula:

$$\texttt{rev [}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]} = \big(\texttt{rev [}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}\big)\ \texttt{@}\ \texttt{[}x_0\texttt{]}$$

because

$$\texttt{rev [}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]} = \texttt{[}x_{n-1}\texttt{;}\ldots\texttt{;}x_1\texttt{]}$$

This leads immediately to a naive declaration of a reverse function:

```
let rec naiveRev xls  =
    match xls with
    | []    -> []
    | x::xs -> naiveRev xs @ [x];;
val naiveRev : 'a list -> 'a list
```

This declaration corresponds directly to the recursion formula for `rev`: the tail list `xs` is reversed and the head element `x` is inserted at the end of the resulting list — but it may be considered naive as it gives a very inefficient evaluation of the reversed list:

```
    naiveRev[1;2;3]
↝  naiveRev[2;3] @ [1]
↝  (naiveRev[3] @ [2]) @ [1]
↝  ((naiveRev[] @ [3]) @ [2]) @ [1]
↝  (([] @ [3]) @ [2]) @ [1]
↝  ([3] @ [2]) @ [1]
↝  (3::([] @ [2])) @ [1]
↝  (3::[2]) @ [1]
↝  [3,2] @ [1]
↝  3::([2] @ [1])
↝  3 :: (2 :: ([] @ [1]))
↝  3 :: (2 :: [1])
↝  3 :: [2;1]
↝  [3;2;1]
```

We will make a much more efficient declaration of the reverse function in a later chapter (Page 208). The library function `List.rev` is, of course, implemented using an efficient declaration.

4.5 The value restrictions on polymorphic expressions

The type system and type inference of F# is very general and flexible. It has, however, been necessary to make a *restriction* on the *use of polymorphic expressions* in order to ensure type correctness in all situations.

The formulation of this restriction is based on the concept of *value expressions*. A value expression is an expression that is not reduced further by an evaluation, that is, it has already the same form as its value. The following expressions are hence value expressions:

```
[]      Some []      (5,[])      (fun x -> [x])
```

while

```
List.rev []       [] @ []
```

do not qualify as a value expression as they can be further evaluated. Note that a function expression (a closure) is considered a value expression because it is only evaluated further when applied to an argument.

The restriction applies to the expression *exp* in declarations

`let` *id* = *exp*

and states the following

> At top level, polymorphic expressions are allowed only if they are value expressions. Polymorphic expressions can be used freely for intermediate results.

Hence F# allows *values* of polymorphic types, such as the empty list `[]`, the pair `(5,[[]])` or the function `(fun x -> [x])`:

```
let z = [];;
val z : 'a list

(5,[[]]);;
val it : int * 'a list list = (5, [[]]

let p = (fun x -> [x]);;
val p : 'a -> 'a list
```

On the other hand, the following is refused at top level:

```
List.rev [];;
^^^^^^^^^^
stdin(86,1): error FS0030: Value restriction.
The value 'it' has been inferred to have generic type
```

The restriction on polymorphic expressions may be paraphrased as follows:

- All monomorphic expressions are OK, even non-value expressions,
- all value expressions are OK, even polymorphic ones, and
- at top-level, polymorphic non-value expressions are forbidden,

where the type of a *monomorphic expression* does not contain type variables, that is, it is a *monomorphic type*.

The rationale for these restrictions will only become clear much later when imperative features of F# are introduced in Chapter 8. In the meantime, we just have to accept the restrictions, and they will really not do us much harm.

Remark: A list expression $a_0 :: a_1 :: \cdots :: a_k :: [a_{k+1}, \ldots, a_{n-1}]$ containing values $a_0, a_2, \ldots, a_{n-1}$ is considered a value expression with the value $[a_0, a_1, \ldots, a_{n-1}]$.

4.6 Examples. A model-based approach

In this section we will introduce a model-based approach to functional programming by means of two examples. The goal is to get a program directly reflecting the problem formulation. An important step in achieving this goal is to identify names denoting key concepts in the problem and to associate F# types with these names. We shall return to these two examples in the next chapter when we have a richer set of collection types with associated library functions.

Example: Cash register

Consider an electronic cash register that contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program that makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

Article code and article name are central concepts that are named and associated with a type:

```
type ArticleCode = string;;
type ArticleName = string;;
```

where the choice of the `string` type for `ArticleCode` is somewhat arbitrary. An alternative choice could be the `int` type.

The register associates article name and article price with each article code, and we model a register by a list of pairs. Each pair has the form:

$$(ac, (aname, aprice))$$

where ac is an article code, $aname$ is an article name, and $aprice$ is an article price. We choose (non-negative) integers to represent prices (in the smallest currency unit):

```
type Price       = int;;          // pr  where  pr >= 0
```

and we get the following type for a register:

```
type Register    = (ArticleCode * (ArticleName*Price)) list;;
```

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

A purchase comprises a list of items, where each item comprises a pair:

$$(np, ac)$$

describing a number of pieces *np* (that is a non-negative integer) purchased of an article with code *ac*:

```
type NoPieces    = int;;          // np  where np >= 0
type Item        = NoPieces * ArticleCode;;
type Purchase    = Item list;;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

A bill comprises an information list $infos$ for the individual items and the grand total sum, and this composite structure is modelled by a pair:

$$(infos, sum)$$

where each element in the list $infos$ is a triple

$$(np, aname, tprice)$$

of the number of pieces *np*, the name *aname*, and the total price *tprice* of a purchased article:

```
type Info        = NoPieces * ArticleName * Price;;
type Infoseq     = Info list;;
type Bill        = Infoseq * Price;;
```

The following value is an example of a bill:

```
([(3,"herring",12); (1,"cheese",25)],37)
```

The function `makeBill` computes a bill given a purchase and a register and it has the type:

```
makeBill: Register -> Purchase -> Bill
```

In this example, it is convenient to declare a auxiliary function:

```
findArticle: ArticleCode -> Register -> ArticleName * Price
```

to find the article name and price in the register for a given article code. This will make the declaration for the function `makeBill` easier to comprehend. An exception is raised when no article with the given code occurs in the register:

```
let rec findArticle ac = function
    | (ac',adesc)::_ when ac=ac' -> adesc
    | _::reg                    -> findArticle ac reg
    | _                         ->
           failwith(ac + " is an unknown article code");;
val findArticle : string -> (string * 'a) list -> 'a
```

Then the bill is made by the function:

```
let rec makeBill reg = function
  | []           -> ([],0)
  | (np,ac)::pur -> let (aname,aprice) = findArticle ac reg
                    let tprice         = np*aprice
                    let (billtl,sumtl) = makeBill reg pur
                    ((np,aname,tprice)::billtl,tprice+sumtl);;
val makeBill :
  (string * ('a * int)) list -> (int * string) list
                -> (int * 'a * int) list * int

makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

The declaration of `makeBill` uses the pattern introduced in Section 4.3 to decompose the value of the recursive call.

Note that the F# system infers a more general type for the `makeBill` function than the type given in our model. This is, however, no problem as the specified type is an instance of the inferred type – `makeBill` has the specified type (among others).

Example: Map colouring

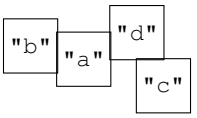
We shall now consider the problem of colouring a map in a way so that neighbouring countries get different colours. We will provide a model using named types, similar to what we did in the previous example. But the map colouring problem is more complex than the cash register example, and we use it to illustrate *functional decomposition* by devising a collection of simple well-understood functions that can be composed to solve the problem.

This problem is a famous mathematical problem and it has been proved that any (reasonable) map can be coloured by using at most four different colours. We will not aim for an "optimal" solution. Neither will we consider the trivial solution where each country always gets its own separate colour. We will assume that each country is connected. This is an oversimplification as Alaska and Kaliningrad, for example, are not connected to other regions of the United States and Russia, respectively.

A *country* is represented by its name, that is a string, and a *map* is represented by a *neighbour relation*, that is represented by a list of pairs of countries with a common border:

```
type Country = string;;
type Map     = (Country * Country) list;;
```

Consider the map in Figure 4.8 with four countries "a", "b", "c", and "d", where the country "a" has the neighbouring countries "b" and "d", the country "b" has the neighbouring country "a", and so on. The F# value for this map is given by the declaration of `exMap`.




```
let exMap = [("a","b"); ("c","d"); ("d","a")];;
```

Figure 4.8 Colouring problem with 4 countries

A *colour* on a map is represented by the set of countries having this colour, and a *colouring* is described by a list of mutually disjoint colours:

```
type Colour     = Country list;;
type Colouring  = Colour list;;
```

The countries of the map in Figure 4.8 may hence be coloured by the colouring:

```
[["a";"c"]; ["b"; "d"]]
```

This colouring has two colours `["a";"c"]` and `["b"; "d"]`, where the countries "a" and "c" get one colour, while the countries "b" and "d" get another colour.

An overview of the model is shown in Figure 4.9 together with sample values. This figure also contains meta symbols used for the various types, as this helps in achieving a consistent naming convention throughout the program.

Meta symbol: Type	Definition	Sample value
`c: Country`	`string`	`"a"`
`m: Map`	`(Country*Country) list`	`[("a","b");` `("c","d");("d","a")]`
`col: Colour`	`Country list`	`["a";"c"]`
`cols: Colouring`	`Colour list`	`[["a";"c"];["b";"d"]]`

Figure 4.9 A Data model for map colouring problem

Our task is to declare a function:

```
colMap: Map -> Colouring
```

that can generate a colouring of a given map. We will express this function as a composition of simple functions, each with a well-understood meaning. These simple functions arise from the *algorithmic idea* behind the solutions to the problem. The idea we will pursue here is the following: We start with the empty colouring, that is, the empty list containing no colours. Then we will gradually extend the actual colouring by adding one country at a time.

	country	old colouring	new colouring
1.	`"a"`	`[]`	`[["a"]]`
2.	`"b"`	`[["a"]]`	`[["a"];["b"]]`
3.	`"c"`	`[["a"];["b"]]`	`[["a";"c"];["b"]]`
4.	`"d"`	`[["a";"c"];["b"]]`	`[["a";"c"];["b";"d"]]`

Figure 4.10 Algorithmic idea

We illustrate this algorithmic idea on the map in Figure 4.8, with the four countries: "`a`", "`b`", "`c`" and "`d`". The four main algorithmic steps (one for each country) are shown in Figure 4.10. We give a brief comment to each step:

1. The colouring containing no colours is the empty list.
2. The colour `["a"]` cannot be *extended* by `"b"` because the countries `"a"` and `"b"` *are neighbours*. Hence the colouring should be extended by a new colour `["b"]`.
3. The colour `["a"]` can be extended by `"c"` because `"a"` and `"c"` are not neighbours.
4. The colour `["a","c"]` can not be extended by `"d"` while the colour `["b"]` can be extended by `"d"`.

The task is now to make a program where the main concepts of this algorithmic idea are directly represented. The concepts emphasized in the above discussion are:

- Test whether a colour can be extended by a country for a given map.
- Test whether two countries are neighbours in a given map.
- Extend a colouring by a country for a given map.

The function specification of each of the main concepts documents the algorithmic idea. These specifications are shown in Figure 4.11. We have added the specification of a function `countries` for extracting the list of countries occurring in a given map and the specification of a function `colCntrs` which gives the colouring for given country list and map.

Type Meaning
`areNb: Map -> Country -> Country -> bool` Decides whether two countries are neighbours
`canBeExtBy: Map -> Colour -> Country -> bool` Decides whether a colour can be extended by a country
`extColouring: Map -> Colouring -> Country -> Colouring` Extends a colouring by an extra country
`countries: Map -> Country list` Computes a list of countries in a map
`colCntrs: Map -> Country list -> Colouring` Builds a colouring for a list of countries

Figure 4.11 Functional break-down for map colouring problem

We now give a declaration for each of the functions specified in Figure 4.11.

1. First we declare a predicate (i.e., a truth-valued function) `areNb` to determine for a given map whether two countries are neighbours:

```
let areNb m c1 c2 =
    isMember (c1,c2) m || isMember (c2,c1) m;;
```

   This declaration makes use of the `isMember`-function declared in Section 4.4.

2. Next we declare a predicate to determine for a given map whether a colour can be extended by a country:

```
let rec canBeExtBy m col c =
    match  col with
    | []       -> true
    | c'::col' -> not(areNb m c' c) && canBeExtBy m col' c;;

canBeExtBy exMap ["c"] "a";;
val it : bool = true

canBeExtBy exMap ["a"; "c"] "b";;
val it : bool = false
```

3. Our solution strategy is to insert the countries of a map one after the other into a colouring, starting with the empty one. To this end we declare a function `extColouring` that for a given map extends a partial colouring by a country:

```
let rec extColouring m cols c =
    match cols with
    | []         -> [[c]]
    | col::cols' -> if canBeExtBy m col c
                    then (c::col)::cols'
                    else col::extColouring m cols' c;;


extColouring exMap [] "a";;
val it : string list list = [["a"]]
```

```
extColouring exMap [["c"]] "a";;
val it : string list list = [["a"; "c"]]

extColouring exMap [["b"]] "a";;
val it : string list list = [["b"]; ["a"]]
```

Note that the first of the three examples exercises the base case of the declaration, the second example the `then`-branch, and the last example the `else`-branch (the recursion and the base case).

4. In order to complete our task, we declare a function to extract a list of countries without repeated elements from a map and a function to colour a list of countries given a map:

```
let addElem x ys = if isMember x ys then ys else x::ys;;

let rec countries = function
    | []           -> []
    | (c1,c2)::m -> addElem c1 (addElem c2 (countries m));;

let rec colCntrs m = function
    | []    -> []
    | c::cs -> extColouring m (colCntrs m cs) c;;
```

The function giving a colouring for a given map is declared by combination of the functions `colCntrs` and `countries`.

```
let colMap m = colCntrs m (countries m);;

colMap exMap;;
val it : string list list = [["c"; "a"]; ["b"; "d"]]
```

Comments

In these two examples we have just used types introduced previously in this book, and some comments could be made concerning the adequacy of the solutions. For example, modelling a data register by a list of pairs does not capture that each article has a unique description in the register, and modelling a colour by a list of countries does not capture the property that the sequence in which countries occur in the list is irrelevant. The same applies to the property that repeated occurrences of a country in a colour are irrelevant.

In Chapter 5 we shall introduce maps and sets and we shall give more suitable models and solutions for the two examples above.

Summary

In this chapter we have introduced the notions of lists and list types, and the notion of list patterns. A selection of typical recursive functions on lists were presented, and the notions of polymorphic types and values were studied. Furthermore, we have introduced a model-based approach to functional programming, where important concepts are named and types are associated with the names.

Exercises

4.1 Declare function `upto: int -> int list` such that `upto` $n = [1; 2; \ldots; n]$.

4.2 Declare function `downto1: int -> int list` such that the value of `downto1` n is the list $[n; n-1; \ldots; 1]$.

4.3 Declare function `evenN: int -> int list` such that `evenN` n generates the list of the first n non-negative even numbers.

4.4 Give a declaration for `altsum` (see Page 76) containing just two clauses.

4.5 Declare an F# function `rmodd` removing the odd-numbered elements from a list:

$$\texttt{rmodd [}x_0\texttt{;}x_1\texttt{;}x_2\texttt{;}x_3\texttt{;} \ldots \texttt{]} = \texttt{[}x_0\texttt{;}x_2\texttt{;} \ldots \texttt{]}$$

4.6 Declare an F# function to remove even numbers occurring in an integer list.

4.7 Declare an F# function `multiplicity` x xs to find the number of times the value x occurs in the list xs.

4.8 Declare an F# function `split` such that:

$$\texttt{split [}x_0\texttt{;}x_1\texttt{;}x_2\texttt{;}x_3\texttt{;} \ldots \texttt{;}x_{n-1}\texttt{]} = (\texttt{[}x_0\texttt{;}x_2\texttt{;} \ldots \texttt{]}, \texttt{[}x_1\texttt{;}x_3\texttt{;} \ldots \texttt{]})$$

4.9 Declare an F# function `zip` such that:

$$\begin{aligned}&\texttt{zip([}x_0\texttt{;}x_1\texttt{;} \ldots \texttt{;}x_{n-1}\texttt{],[}y_0\texttt{;}y_1\texttt{;} \ldots \texttt{;}y_{n-1}\texttt{])}\\ &= \texttt{[(}x_0, y_0\texttt{);(}x_1, y_1\texttt{);} \ldots \texttt{;(}x_{n-1}, y_{n-1}\texttt{)]}\end{aligned}$$

The function should raise an exception if the two lists are not of equal length.

4.10 Declare an F# function `prefix: 'a list -> 'a list -> bool when a : equality`. The value of the expression `prefix` $\texttt{[}x_0\texttt{;}x_1\texttt{;} \ldots \texttt{;}x_m\texttt{]}$ $\texttt{[}y_0\texttt{;}y_1\texttt{;} \ldots \texttt{;}y_n\texttt{]}$ is `true` if $m \leq n$ and $x_i = y_i$ for $0 \leq i \leq m$, and `false` otherwise.

4.11 A list of integers $\texttt{[}x_0\texttt{;}x_1\texttt{;} \ldots \texttt{;}x_{n-1}\texttt{]}$ is *weakly ascending* if the elements satisfy:

$$x_0 \leq x_1 \leq x_2 \leq \ldots \leq x_{n-2} \leq x_{n-1}$$

or if the list is empty. The problem is now to declare functions on weakly ascending lists.

1. Declare an F# function `count: int list * int -> int`, where `count`(xs, x) is the number of occurrences of the integer x in the weakly ascending list xs.
2. Declare an F# function `insert: int list * int -> int list`, where the value of `insert`(xs, x) is a weakly ascending list obtained by inserting the number x into the weakly ascending list xs.
3. Declare an F# function `intersect: int list * int list -> int list`, where the value of `intersect`(xs, xs') is a weakly ascending list containing the common elements of the weakly ascending lists xs and xs'. For instance:

```
intersect([1;1;1;2;2], [1;1;2;4]) = [1;1;2]
```

4. Declare an F# function `plus: int list * int list -> int list`, where the value of `plus`(xs, xs') is a weakly ascending list, that is the union of the weakly ascending lists xs and xs'. For instance:

```
plus([1;1;2],[1;2;4]) = [1;1;1;2;2;4]
```

5. Declare an F# function `minus: int list * int list -> int list`, where the value of `minus`(xs, xs') is a weakly ascending list obtained from the weakly ascending list xs by removing those elements, that are also found in the weakly ascending list xs'. For instance:

```
minus([1;1;1;2;2],[1;1;2;3]) = [1;2]
minus([1;1;2;3],[1;1;1;2;2]) = [3]
```

4.12 Declare a function `sum`(p, xs) where p is a predicate of type `int -> bool` and xs is a list of integers. The value of `sum`(p, xs) is the sum of the elements in xs satisfying the predicate p. Test the function on different predicates (e.g., $p(x) = x > 0$).

4.13 Naive sort function:

1. Declare an F# function finding the smallest element in a non-empty integer list.
2. Declare an F# function `delete: int * int list -> int list`, where the value of `delete`(a, xs) is the list obtained by deleting one occurrence of a in xs (if there is one).
3. Declare an F# function that sorts an integer list so that the elements are placed in weakly ascending order.

Note that there is a much more efficient sort function `List.sort` in the library.

4.14 Declare a function of type `int list -> int option` for finding the smallest element in an integer list.

4.15 Declare an F# function `revrev` working on a list of lists, that maps a list to the reversed list of the reversed elements, for example:

```
revrev [[1;2];[3;4;5]] = [[5;4;3];[2;1]]
```

4.16 Consider the declarations:

```
let rec f = function
    | (x, [])     -> []
    | (x, y::ys) -> (x+y)::f(x-1, ys);;

let rec g = function
    | []       -> []
    | (x,y)::s -> (x,y)::(y,x)::g s;;

let rec h = function
    | []    -> []
    | x::xs -> x::(h xs)@[x];;
```

Find the types for `f`, `g` and `h` and explain the value of the expressions:

1. `f`$(x, [y_0, y_1, \ldots, y_{n-1}]), n \geq 0$
2. `g`$[(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})], n \geq 0$
3. `h`$[x_0, x_1, \ldots, x_{n-1}], n \geq 0$

4.17 Consider the declaration:

```
let rec p q = function
    | []    -> []
    | x::xs -> let ys = p q xs
               if q x then x::ys else ys@[x];;
```

Find the type for `p` and explain the value of the expression:

$$\texttt{p}\ q\ [x_0; x_1; x_3; \ldots; x_{n-1}]$$

4.18 Consider the declaration:

```
let rec f g = function
    | []    -> []
    | x::xs -> g x :: f (fun y -> g(g y)) xs;;
```

Find the type for `f` and explain the value of the expression:

$$\texttt{f}\ g\ [x_0; x_1; x_2; \ldots; x_{n-1}]$$

4.19 Evaluation of the expression `areNb` $m\, c_1\, c_2$ may traverse the map m twice. Explain why and give an alternative declaration for `areNb` which avoids this problem.

4.20 Most of the auxiliary functions for the map-colouring program just assume an arbitrary, but fixed, map. The function `canBeExtBy`, for example, just passes `m` on to `areNb`, which again passes `m` on to `isMember`. The program can therefore be simplified by declaring (most of) the auxiliary functions locally as sketched here:

```
...
let colMap m =
    let areNb c1 c2 = ...
    let canBeExtBy col c = ...
    ...
```

Revise the program by completing this skeleton.

4.21 Revise the map-colouring program so that it can cope with countries which are islands (such as Iceland) having no neighbours.

4.22 We represent the polynomial $a_0 + a_1 \cdot x + ... + a_n \cdot x^n$ with integer coefficients $a_0, a_1, ..., a_n$ by the list $[a_0, a_1, ..., a_n]$. For instance, the polynomial $x^3 + 2$ is represented by the list $[2, 0, 0, 1]$.

1. Declare an F# function for multiplying a polynomial by a constant.
2. Declare an F# function for multiplying a polynomial $Q(x)$ by x.
3. Declare infix F# operators for addition and multiplication of polynomials in the chosen representation. The following recursion formula is useful when defining the multiplication:

$$\begin{aligned} 0 \cdot Q(x) \quad &= 0 \\ (a_0 + a_1 \cdot x + ... + a_n \cdot x^n) \cdot Q(x) \\ &= a_0 \cdot Q(x) + x \cdot \big((a_1 + a_2 \cdot x + ... + a_n \cdot x^{n-1}) \cdot Q(x)\big) \end{aligned}$$

4. Declare an F# function to give a textual representation for a polynomial.

4.23 A dating bureau has a file containing name, telephone number, sex, year of birth and themes of interest for each client. You may make a request to the bureau stating your own sex, year of birth and themes of interest and get a response listing all matching clients, that is, clients with different sex, a deviation in age less than 10 years and with at least one common theme of interest. The problem is to construct a program for generating the responses from the dating bureau.