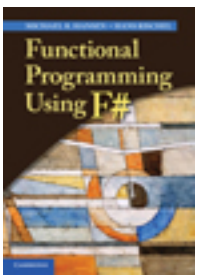


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

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

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

5 - Collections: Lists, maps and sets pp. 93-120

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.006

Cambridge University Press

5

Collections: Lists, maps and sets

Functional languages make it easy to express standard recursion patterns in the form of higher-order functions. A collection of such higher-order functions on lists, for example, provides a powerful library where many recursive functions can be obtained directly by application of higher-order library functions. This has two important consequences:

1. The functions in the library correspond to natural abstract concepts and conscious use of them supports high-level program design, and
2. these functions support code reuse because you can make many functions simply by applying library functions.

In this chapter we shall study libraries for lists, sets and maps, which are parts of the collection library of F#. This part of the collection library is studied together since:

- It constitutes the *immutable part* of the collection library. The list, set and map collections are finite collections programmed in a functional style.
- There are many similarities in the corresponding library functions.

This chapter is a natural extension of Chapter 4 since many of the patterns introduced in that chapter correspond to higher-order functions for lists and since more natural program designs can be given for the two examples in Section 4.6 using sets and maps.

We will focus on the main concepts and applications in this book, and will deliberately not cover the complete collection library of F#. The functions of the collection library do also apply to (mutable) arrays. We address this part in Section 8.10.

5.1 Lists

This section describes the library functions `map`, various library functions using a predicate on list elements plus the functions `fold` and `foldBack`. Each description aims to provide the following:

1. An intuitive understanding of the objective of the function.
2. Examples of use of the function.

The actual declarations of the library functions are not considered as we want to concentrate on how to *use* these functions in problem solving. Declarations of `fold` and `foldBack` are, however, of considerable theoretical interest and are therefore studied in the last part of the section. An overview of the `List`-library functions considered in this section is found in Table 5.1.

Operation Meaning
`map: ('a -> 'b) -> 'a list -> 'b list`, where `map` $f\ xs = [f(x_0); f(x_1); \ldots; f(x_{n-1})]$
`exists: ('a -> bool) -> 'a list -> bool`, where `exists` $p\ xs = \exists x \in xs.p(x)$
`forall: ('a -> bool) -> 'a list -> bool`, where `forall` $p\ xs = \forall x \in xs.p(x)$
`tryFind: ('a -> bool) -> 'a list -> 'a option`, where `tryFind` $p\ xs$ is `Some` x for some $x \in xs$ with $p(x) =$ `true` or `None` if no such x exists
`filter: ('a -> bool) -> 'a list -> 'a list`, where `filter` $p\ xs = ys$ where ys is obtained from xs by deletion of elements $x_i : p(x_i) =$ false
`fold: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`, where `fold` $f\ a\ [b_0; b_1; \ldots; b_{n-2}; b_{n-1}] = f(f(f(\cdots f(f(a, b_0), b_1), \ldots), b_{n-2}), b_{n-1})$
`foldBack: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`, where `foldBack` $f\ [a_0; a_1; \ldots; a_{n-2}; a_{n-1}]\ b = f(a_0, f(a_1, f(\ldots, f(a_{n-2}, f(a_{n-1}, b)) \cdots)))$
`collect: ('a -> 'b list) -> 'a list -> 'b list`, where `collect` $f\ [a_0; a_1; \ldots; a_{n-1}] = (f\ a_0)@(f\ a_1)@ \cdots @(f\ a_{n-1})$

These operations are found under the names: `List.map`, `List.exists`, and so on.
We assume that $xs = [x_0; x_1; \ldots; x_{n-2}; x_{n-1}]$.

Table 5.1 *A selection of functions from the* `List` *library*

The `map` *function*

The library function

```
List.map: ('a -> 'b) -> 'a list -> 'b list
```

works as follows:

$$\texttt{List.map}\ f\ \texttt{[}x_0\texttt{;}x_1\texttt{;}\cdots\texttt{;}x_{n-1}\texttt{]}\ =\ \texttt{[}f\ x_0\texttt{;}f\ x_1\texttt{;}\cdots\texttt{;}f\ x_{n-1}\texttt{]}$$

In words:

> The function application `List.map` f is the function that applies the function f to each element $x_0, x_1, \ldots, x_{n-1}$ in a list $[x_0; x_1; \cdots; x_{n-1}]$

It is easy to use `List.map`:

- The function `addFsExt` adds the F# file extension ".fs" to every string in a list of file names.
- The function `intPairToRational` converts every integer pair in a list to the string of a rational number on the basis of the declarations in Section 3.7.
- The function `areaList` computes the area of every shape in a list on the basis of the declarations in Section 3.8.

```
let addFsExt = List.map (fun s -> s + ".fs");;
val addFsExt : (string list -> string list)

let intPairToRational = List.map (toString << mkQ);;
val intPairToRational : ((int * int) list -> string list)
```

```
let areaList = List.map area;;
val areaList : (shape list -> float list)
```

since

- `addFsExt` applies the function that concatenates the suffix "`.fs`" to a string, to every element in a `string list`,
- `intPairToRational` applies the function that converts an integer pair to the string representation of the corresponding rational number to every element in a list of integer pairs, and
- `areaList` applies the `area` function to every element in a `shape list`.

The functions work as follows:

```
addFsExt ["ListPrograms"; "AuxiliaryPrograms"];;
val it : string list =
       ["ListPrograms.fs"; "AuxiliaryPrograms.fs"]

intPairToRational [(2,6); (20,-8); (-12,-4)];;
val it : string list = ["1/3"; "-5/2"; "3/1"]

areaList [Circle 2.0; Square 2.0; Triangle(2.0, 3.0, 4.0)];;
val it : float list = [12.56637061; 4.0; 2.90473751]
```

Alternative ways of declaring `intPairToRational` using `List.map` are

```
let intPairToRational = List.map (fun p -> toString(mkQ p));;

let intPairToRational ps =
    List.map (fun p -> toString(mkQ p)) ps;;
```

where `fun p -> toString(mkQ p)` is an expansion of the function composition operator in `toString << mkQ` and `ps` is used as explicit list argument in the last declaration. Explicit list arguments could also be used in declarations of `addFsExt` and `areaList`.

Functions using a predicate on the list elements

The F# library contains a large number of functions using a predicate of type `'a -> bool` on elements in a list of type `'a list`.

We consider some of these functions here, namely (cf. Table 5.1):

```
List.exists  : ('a -> bool) -> 'a list -> bool
List.forall  : ('a -> bool) -> 'a list -> bool
List.tryFind : ('a -> bool) -> 'a list -> 'a option
List.filter  : ('a -> bool) -> 'a list -> 'a list
```

The value of the expression

$$\texttt{List.exists}\ p\ [x_0;x_1;\ldots;x_{n-1}]$$

is `true`, if $p(x_k) = \texttt{true}$ holds for some list element x_k, and `false` otherwise.

The value of the expression

$$\texttt{List.forall}\ p\ \texttt{[}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}$$

is `true`, if $p(x_k) = \texttt{true}$ holds for all list elements x_k, and `false` otherwise.

The value of the expression

$$\texttt{List.tryFind}\ p\ \texttt{[}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}$$

is `Some` x_k for a list element x_k with $p(x_k) = \texttt{true}$, or `None` if no such element exists.

The value of the expression

$$\texttt{List.filter}\ p\ \texttt{[}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}$$

is the list of those list elements x_k where $p(x_k) = \texttt{true}$.

Note that the evaluation of the expression

$$\texttt{List.exists}\ p\ \texttt{[}x_0\texttt{,}x_1\texttt{,}\ldots x_{i-1}\texttt{,}x_i\texttt{,}\ldots\texttt{,}x_{n-1}\texttt{]}$$

does not terminate if the evaluation of the expression $p(x_k)$ does not terminate for some k, where $0 \leq k \leq n-1$ and if $p(x_j) = \texttt{false}$ for all j where $1 \leq j < k$. A similar remark will apply to the other functions using a predicate on list elements.

Simple applications of the functions are:

```
List.exists (fun x -> x>=2) [1;3;1;4];;
val it : bool = true

List.forall (fun x -> x>=2) [1;3;1;4];;
val it : bool = false

List.tryFind (fun x -> x>3) [1;5;-2;8];;
val it : int option = Some 5

List.filter (fun x -> x>3) [1;5;-2;8];;
val it : int list = [5; 8]
```

The function `isMember` (cf. Section 4.4) can be declared using `List.exists`:

```
let isMember x xs = List.exists (fun y -> y=x) xs;;
val isMember : 'a -> 'a list -> bool when 'a : equality

isMember (2,3.0) [(2, 4.0) ; (3, 7.0)];;
val it : bool = false

isMember "abc" [""; "a"; "ab"; "abc"];;
val it : bool = true
```

The functions `fold` *and* `foldBack`

The library functions `List.fold` and `List.foldBack` are very powerful and rather useful in many circumstances, but they are somewhat difficult to understand at first glance. To ease the understanding we use a rather naive, almost grotesque, example to convey the ideas behind these functions.

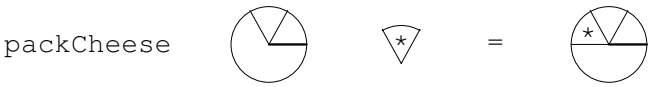
We consider small cheeses and a round package to contain small cheeses:

Cheeses and packages are considered elements of type `cheese` and `package`. A package may contain zero or more cheeses.

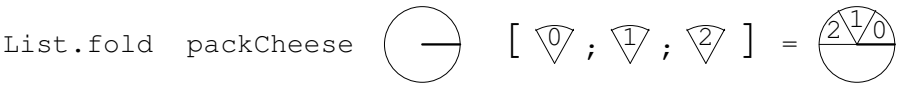
The function

```
packCheese: package -> cheese -> package
```

packs an extra cheese into a package:



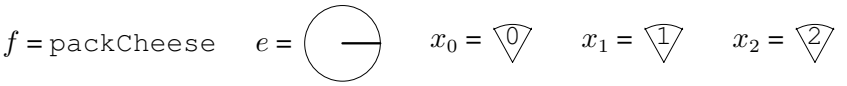
The function `List.fold` can be applied to the function `packCheese`, a start package and a list of cheeses. It uses `packCheese` to pack the elements of the list (the cheeses) into the package one after the other – starting with the given start package:



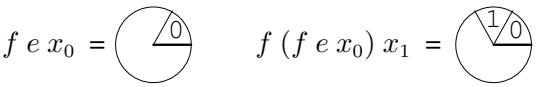
This is a special case of the general formula:

$$\texttt{List.fold}\ f\ e\ \texttt{[}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}\ =\ f\ (\ldots(f\ (f\ e\ x_0)\ x_1)\ldots)\ x_{n-1}$$

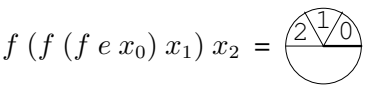
with



because we can identify the sub-expressions on the right-hand side of the general formula in our special case:



and



The function of `List.fold` can be expressed in words as follows:

> The evaluation of `List.fold` f e `[`x_0`;`x_1`; ...;`x_{n-1}`]` accumulates the list elements $x_0, x_1, \ldots, x_{n-1}$ using the accumulation function f and the start value e

One also says that the function f is folded over the list `[`x_0`;`x_1`; ...;`x_{n-1}`]` starting with the value e.

The type of `List.fold` is:

```
List.fold: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

with list element type `'b` and accumulator type `'a`.

When applying `List.fold` one has to look for the following entities:

List element type `'b`	corresponding to `cheese` in the example
Accumulator type `'a`	corresponding to `package` in the example
Accumulator function f	corresponding to `packCheese` in the example
Start value e	corresponding to the empty package in the example

and we have to arrange the parameters in the accumulator function to suit the type of `List.fold`.

As an example we consider a list $vs = [v_0; \ldots; v_{n-1}]$ of geometric vectors in the plane (see Section 3.3), where v_i is a pair (x_i, y_i) of floats, for $0 \leq i < n$. We want to compute the sum of the norms of the vectors in vs using the norm function declared as follows in Section 3.3:

```
let norm(x:float,y:float) = sqrt(x*x+y*y);;
val norm : float * float -> float
```

This is a case for applying `List.fold` with:

List element type:	`float * float`
Accumulator type:	`float`
Accumulator function:	`fun s (x,y) -> s + norm(x,y)`
Start value:	`0.0`

This leads to the declaration:

```
let sumOfNorms vs =
  List.fold (fun s (x,y) -> s + norm(x,y)) 0.0 vs;;
val sumOfNorms : (float * float) list -> float

let vs = [(1.0,2.0); (2.0,1.0); (2.0, 5.5)];;
val vs : (float * float) list =
            [(1.0, 2.0); (2.0, 1.0); (2.0, 5.5)]

sumOfNorms vs;;
val it : float = 10.32448591
```

The `length` function on lists can be defined using `List.fold` with

List element type:	`'a`
Accumulator type:	`int`
Accumulator function:	`fun e _ -> e + 1`
Start value:	`0`

This leads to the declaration:

```
let length lst = List.fold (fun e _ -> e+1) 0 lst;;
val length : 'a list -> int

length [[1;2];[];[3;5;8];[-2]];;
val it : int = 4
```

Applying `fold` to the following version of "cons":

```
fun rs x -> x::rs
```

where the parameters are interchanged, gives a declaration of the reverse function for lists:

```
let rev xs = List.fold (fun rs x -> x::rs) [] xs;;
val rev : 'a list -> 'a list

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

The function `List.foldBack` is similar to `List.fold` but the list elements are accumulated in the opposite order. The type of `List.foldBack` is:

```
List.foldBack: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

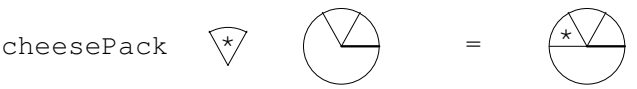
and the general formula is:

$$\texttt{List.foldBack}\ g\ [x_0;x_1;\ldots;x_{n-1}]\ e = g\,x_0\,(g\,x_1\,(\ldots(g\,x_{n-1}\,e)\ldots))$$

We may use our "cheese" example also in this case with a modified accumulation function:

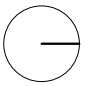
```
cheesePack: cheese -> package -> package
```

where



The function `List.foldBack` can be applied to the function `cheesePack`, a list of cheeses and a start package. It uses `cheesePack` to pack the elements of the list (the cheeses) taken in reverse order into the package:

This is a special case of the general formula with:

g = `cheesePack` x_0 =  x_1 =  x_2 =  e = 

because we can identify the sub-expressions in the right-hand side of the general formula in our special case:

$g\ x_2\ e\ =$  $g\ x_1\ (g\ x_2\ e)\ =$ 

and

$g\ x_0\ (g\ x_1\ (g\ x_2\ e))\ =$ 

The function of `List.foldBack` can be expressed in words as follows:

> The evaluation of `List.foldBack` g `[`x_0`;`x_1`; ...;`x_{n-1}`]` e accumulates the list elements in reverse order $x_{n-1}, \ldots, x_1, x_0$ using the accumulation function g and the start value e

When applying `List.foldBack` one has to look for the following entities:

List element type `'a`	corresponding to `cheese` in the example
Accumulator type `'b`	corresponding to `package` in the example
Accumulator function g	corresponding to `cheesePack` in the example
Start value e	corresponding to the empty package in the example

and we have to arrange the parameters in the accumulator function to suit the type of `List.foldBack`.

Using `List.foldBack` we may define an alternative "sum of norms" function. Element and accumulator types can be used unchanged, but the parameters in the accumulator function must be interchanged. This gives the following declaration:

```
let backSumOfNorms vs =
  List.foldBack (fun (x,y) s -> s + norm(x,y)) vs 0.0;;
val backSumOfNorms : (float * float) list -> float
```

This function will work like the previous `sumOfNorms` but the norms are added in the opposite order, starting with the norm of the last vector in the list.

Applying `List.foldBack` on the "cons" operator:

```
fun x xs -> x::xs
```

gives the append function:

```
let app ys zs = List.foldBack (fun x xs -> x::xs) ys zs;;
val app : 'a list -> 'a list -> 'a list
app [1;2;3] [4;5;6];;
val it : int list = [1; 2; 3; 4; 5; 6]
```

The `unzip` function on Page 77 can be obtained using `foldBack` with the following data:

List element type	`'a * 'b`
Accumulator type	`'a list * 'b list`
Accumulator function	`fun (x,y) (xs,ys) -> (x::xs,y::ys)`
Start value	`([],[])`

This gives the declaration

```
let unzip zs = List.foldBack
                 (fun (x,y) (xs,ys) -> (x::xs,y::ys))
                 zs
                 ([],[]);;
val unzip : ('a * 'b) list -> 'a list * 'b list

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

A similar construction using `List.fold` gives a `revUnzip` function where the resulting lists are reversed:

```
let revUnzip zs =
  List.fold (fun (xs,ys) (x,y) -> (x::xs,y::ys)) ([],[]) zs;;
val revUnzip : ('a * 'b) list -> 'a list * 'b list

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

The prefix version of an *infix operator* can be used as argument in `fold` and `foldBack`:

```
List.fold (+) 0 [1; 2; 3];;
val it : int = 6

List.foldBack (+) [1; 2; 3] 0;;
val it : int = 6
```

These expression compute `((0 + 1) + 2) + 3` and `1 + (2 + (3 + 0))`, but the results are equal because + is a *commutative* operator: $a + b = b + a$.

A difference in using `fold` or `foldBack` shows up when using a non-commutative operator, that is:

```
List.fold (-) 0 [1; 2; 3];;
val it : int = -6

List.foldBack (-) [1; 2; 3] 0;;
val it : int = 2
```

These expressions use the functions:

```
fun e x -> e - x
fun x e -> x - e
```

and we get

```
List.fold (-) 0 [1;2;3]       =  ((0 - 1) - 2) - 3  =  -6
List.foldBack (-) [1;2;3] 0   =  1 - (2 - (3 - 0))  =   2
```

The `map` function can be declared using `foldBack`:

```
let map f xs = List.foldBack (fun x rs -> f x :: rs) xs [];;
val map : ('a -> 'b) -> 'a list -> 'b list

map (fun x -> x+1) [0; 1; 2];;
val it : int list = [1; 2; 3]
```

Remark

A function declared by means of `fold` or `foldBack` will always scan the whole list. Thus, the following declaration for the `exists` function

```
let existsF p =
    List.fold (fun b -> (fun x -> p x || b)) false;;
val existsF : ('a -> bool) -> ('a list -> bool)
```

will not behave like the function `List.exists` with regard to non-termination: It will give a non-terminating evaluation if the list contains any element where the evaluation of the predicate `p` does not terminate, while the library function `List.exists` may terminate in this case as it does not scan the list further when an element satisfying the predicate has been found. So it is not considered a good idea to use `fold` or `foldBack` to declare functions like `exists` or `find` (cf. Page 95) as these functions need not scan the whole list in all cases.

Declarations of `fold` *and* `foldBack`

The list functions `fold` and `foldBack` are defined on Pages 97 and 99 by the formulas:

$$\begin{aligned}\texttt{fold}\ f\ e\ [x_0;x_1;\ldots;x_{n-1}] &= f\,(\ldots(f\,(f\ e\ x_0)\ x_1)\ldots)\ x_{n-1}\\ \texttt{foldBack}\ g\ [x_0;x_1;\ldots;x_{n-1}]\ e &= g\ x_0\ (g\ x_1\ (\ldots(g\ x_{n-1}\ e)\ldots))\end{aligned}$$

A recursion formula for `fold` is obtained by observing that:

$$f\,(\ldots(f\,(f\ e\ x_0)\ x_1)\ldots)\ x_{n-1} = f\,(\ldots(f\ e'\ x_1)\ldots)\ x_{n-1}$$

where $e' = f\ e\ x_0$. The expression on the right-hand side is equal to:

$$\texttt{fold}\ f\ e'\ [x_1;\ldots;x_{n-1}]$$

and we get the recursion formula:

$$\texttt{fold}\ f\ e\ [x_0;x_1;\ldots;x_{n-1}] = \texttt{fold}\ f\ (f\ e\ x_0)\ [x_1;\ldots;x_{n-1}]$$

A recursion formula for `foldBack` is obtained by observing that the subexpression:

$$(g\ x_1\ (\ldots(g\ x_{n-1}\ e)\ldots)$$

on the right-hand side of the formula for `foldBack` is equal to:

$$\texttt{foldBack}\ g\ \texttt{[}x_0\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}\ e$$

and we get the recursion formula:

$$\texttt{foldBack}\ g\ \texttt{[}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}\ e = g\ x_0\ (\texttt{foldBack}\ g\ \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}\ e)$$

These recursion formulas lead to the declarations:

```
let rec fold f e = function
    | x::xs -> fold f (f e x) xs
    | []    -> e;;

let rec foldBack g xlst e =
    match xlst with
    | x::xs -> g x (foldBack g xs e)
    | []    -> e;;
```

The evaluation of a function value `fold` $f\ e$ `[`x_0`;`x_1`;`$\ldots$`;`x_{n-1}`]` proceeds as follows applying f in each evaluation step without building any large expression:

$$\begin{array}{lll}
 & \texttt{fold}\ f\ e\ \texttt{[}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]} & \\
\rightsquigarrow & \texttt{fold}\ f\ e_1\ \texttt{[}x_1\texttt{;}x_2\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]} & e_1 = f\ e\ x_0 \\
\rightsquigarrow & \texttt{fold}\ f\ e_2\ \texttt{[}x_2\texttt{;}x_3\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]} & e_2 = f\ e_1\ x_1 \\
 & \ldots & \\
\rightsquigarrow & \texttt{fold}\ f\ e_{n-1}\ \texttt{[}x_{n-1}\texttt{]} & e_{n-1} = f\ e_{n-2}\ x_{n-2} \\
\rightsquigarrow & \texttt{fold}\ f\ e_n\ \texttt{[]} & e_n = f\ e_{n-1}\ x_{n-1} \\
\rightsquigarrow & e_n &
\end{array}$$

The evaluation of `foldBack` g `[`x_0`;`x_1`;`$\ldots$`;`x_{n-1}`]` e first builds a large expression:

$$\begin{array}{ll}
 & \texttt{foldBack}\ g\ \texttt{[}x_0\texttt{;}x_1\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}\ e \\
\rightsquigarrow & g\ x_0\ (\texttt{foldBack}\ g\ \texttt{[}x_1\texttt{;}x_2\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}\ e) \\
\rightsquigarrow & g\ x_0\ (g\ x_1\ (\texttt{foldBack}\ g\ \texttt{[}x_1\texttt{;}x_2\texttt{;}\ldots\texttt{;}x_{n-1}\texttt{]}\ e)) \\
 & \ldots \\
\rightsquigarrow & g\ x_0\ (g\ x_1\ (g\ x_2\ (\ldots(g\ x_{n-2}\ (\texttt{foldBack}\ g\ \texttt{[}x_{n-1}\texttt{]}\ e))\ldots))) \\
\rightsquigarrow & g\ x_0\ (g\ x_1\ (g\ x_2\ (\ldots(g\ x_{n-2}\ (g\ x_{n-1}\ (\texttt{foldBack}\ g\ \texttt{[]}\ e)\ldots))) \\
\rightsquigarrow & g\ x_0\ (g\ x_1\ (g\ x_2\ (\ldots(g\ x_{n-2}\ (g\ x_{n-1}\ e))\ldots)))
\end{array}$$

and this expression is then evaluated "inside-out" using repeated calls of g:

$$\begin{array}{lll}
 & g\ x_0\ (g\ x_1\ (g\ x_2\ (\ldots(g\ x_{n-2}\ (g\ x_{n-1}\ e))\ldots))) & \\
\rightsquigarrow & g\ x_0\ (g\ x_1\ (g\ x_2\ (\ldots(g\ x_{n-2}\ e'_1)))\ldots))) & e'_1 = g\ x_{n-1}\ e \\
 & \ldots & \\
\rightsquigarrow & g\ x_0\ (g\ x_1\ e'_{n-2}) & e'_{n-2} = g\ x_2\ e'_{n-3} \\
\rightsquigarrow & g\ x_0\ e'_{n-1} & e'_{n-1} = g\ x_1\ e'_{n-2} \\
\rightsquigarrow & e'_n & e'_n = g\ x_0\ e'_{n-1}
\end{array}$$

The evaluation of `fold` is obviously much more efficient than the evaluation of `foldBack`, so `fold` should be preferred whenever possible. The `List.foldBack` function in the library is more efficient than the above `foldBack` but `List.fold` is still more efficient.

5.2 Finite sets

In solving programming problems it is often convenient to use values that are *finite sets* of form $\{a_1, a_2, \ldots, a_n\}$ with elements $a_1, \ldots, a_n$ from some set A. The notion of a set provides a useful abstraction in cases where we have an unordered collection of elements where repetitions among the elements are of no concern.

This section introduces the set concept and operations on sets in F# on the basis of the library `Set`. The focus is on the principal issues so just a small part of the available operations will be covered. Please consult the on-line documentation (in [9]) for an overview of the complete `Set` library.

The mathematical set concept

A *set* (in mathematics) is a collection of elements like

$$\{\text{Bob}, \text{Bill}, \text{Ben}\} \qquad \text{and} \qquad \{1, 3, 5, 7, 9\}$$

where it is possible to decide whether a given value is in the set. For example, Alice is not in the set $\{\text{Bob}, \text{Bill}, \text{Ben}\}$ and 7 is in the set $\{1, 3, 5, 7, 9\}$, also written:

$$\text{Alice} \notin \{\text{Bob}, \text{Bill}, \text{Ben}\} \qquad \text{and} \qquad 7 \in \{1, 3, 5, 7, 9\}$$

The empty set containing no element is written $\{\}$ or $\emptyset$.

Since the order in which elements are enumerated in a set is of no concern, and repetitions among members of a set is of no concern either, the following expressions denote the same set:

$$\{\text{Bob}, \text{Bill}, \text{Ben}\} \qquad \{\text{Bob}, \text{Bill}, \text{Ben}, \text{Bill}\} \qquad \{\text{Bill}, \text{Ben}, \text{Bill}, \text{Bob}\}$$

The above examples are all finite sets; but sets may be infinite and examples are the set of all natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ and the set of all real numbers $\mathbb{R}$.

A set A is a *subset* of a set B, written $A \subseteq B$, if all the elements of A are also elements of B, for example

$$\{\text{Ben}, \text{Bob}\} \subseteq \{\text{Bob}, \text{Bill}, \text{Ben}\} \qquad \text{and} \qquad \{1, 3, 5, 7, 9\} \subseteq \mathbb{N}$$

Furthermore, two sets A and B are equal, if they are both subsets of each other:

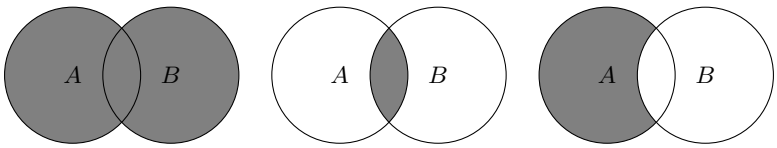
$$A = B \qquad \text{if and only if} \qquad A \subseteq B \text{ and } B \subseteq A$$

that is, two sets are equal if they contain exactly the same elements.

The subset of a set A that consists of those elements satisfying a predicate p can be expressed using a *set-comprehension* $\{x \in A \mid p(x)\}$. For example, the set $\{1, 3, 5, 7, 9\}$ consists of the odd natural numbers that are smaller than 11:

$$\{1, 3, 5, 7, 9\} = \{x \in \mathbb{N} \mid \text{odd}(x) \text{ and } x < 11\}$$

If it is clear from the context from which set A the elements of the set-comprehension originate, then we use the simplified notation: $\{x \mid p(x)\}$.



(a) $A \cup B$ (b) $A \cap B$ (c) $A \setminus B$

Figure 5.1 Venn diagrams for (a) union, (b) intersection and (c) difference

Some of the standard operations on sets are *union*: $A \cup B$, *intersection* $A \cap B$ and *difference* $A \setminus B$:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$
$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$
$$A \setminus B = \{x \in A \mid x \notin B\}$$

that is, $A \cup B$ is the set of elements that are in at least one of the sets A and B, $A \cap B$ is the set of elements that are in both A and B, and $A \setminus B$ is the subset of the elements from A that are not in B. These operations are illustrated using Venn diagrams in Figure 5.1. For example:

$$\{\text{Bob}, \text{Bill}, \text{Ben}\} \cup \{\text{Alice}, \text{Bill}, \text{Ann}\} = \{\text{Alice}, \text{Ann}, \text{Bob}, \text{Bill}, \text{Ben}\}$$
$$\{\text{Bob}, \text{Bill}, \text{Ben}\} \cap \{\text{Alice}, \text{Bill}, \text{Ann}\} = \{\text{Bill}\}$$
$$\{\text{Bob}, \text{Bill}, \text{Ben}\} \setminus \{\text{Alice}, \text{Bill}, \text{Ann}\} = \{\text{Bob}, \text{Ben}\}$$

Sets in F#

The `Set` library of F# supports finite sets of elements of a type where ordering is defined, and provides efficient implementations for a rich collection of set operations. The implementation is based on a balanced binary tree representation of a set and this is why an ordering of the elements is required (but we will not consider such implementation details in this section).

Consider the following example of a set in F#:

```
set ["Bob"; "Bill"; "Ben"];;
val it : Set<string> = set ["Ben"; "Bill"; "Bob"]
```

Hence, a set can be given in a manner similar to a list using the "set-builder" function `set`. The resulting value is of type `Set<string>`, that is, a set of strings, and we can see from the F# answer that the elements occur according to a lexicographical ordering. A standard number ordering is used for sets of integers, for example:

```
set [3; 1; 9; 5; 7; 9; 1];;
val it : Set<int> = set [1; 3; 5; 7; 9]
```

Equality of two sets is tested in the usual manner:

```
set ["Bob";"Bill";"Ben"] = set ["Bill";"Ben";"Bill";"Bob"];;
val it : bool = true
```

and sets are ordered on the basis of a similar kind of lexicographical ordering as used for lists. (See Section 4.1.) For example, $\{\text{Ann}, \text{Jane}\}$ is smaller than $\{\text{Bob}, \text{Bill}, \text{Ben}\}$ (in the F# representation) since Ann is smaller than every element in $\{\text{Bob}, \text{Bill}, \text{Ben}\}$ using the string representation:

```
compare (set ["Ann";"Jane"]) (set ["Bill";"Ben"; "Bob"]);;
val it : int = -1
```

Operation Meaning
`ofList: 'a list -> Set<'a>`, where `ofList` $[a_0; \ldots; a_{n-1}] =$ `set` $[a_0; \ldots; a_{n-1}]$
`toList: Set<'a> -> 'a list`, where `toList` $\{a_0, \ldots, a_{n-1}\} = [a_0; \ldots; a_{n-1}]$
`add: 'a -> Set<'a> -> Set<'a>`, where `add` $a\ A = \{a\} \cup A$
`remove: 'a -> Set<'a> -> Set<'a>`, where `remove` $a\ A = A \setminus \{a\}$
`contains: 'a -> Set<'a> -> bool`, where `contains` $a\ A = a \in A$
`isSubset: Set<'a> -> Set<'a> -> bool`, where `isSubset` $A\ B = A \subseteq B$
`minElement: Set<'a> -> 'a`, where `minElement` $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\} = a_0$ when $n > 0$
`maxElement: Set<'a> -> 'a`, where `maxElement` $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\} = a_{n-1}$ when $n > 0$
`count: Set<'a> -> int`, where `count` $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\} = n$

These operations are found under the names: `Set.add`, `Set.contains`, and so on.
It is assumed that the enumeration $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\}$ respects the ordering of elements.

Table 5.2 *A selection of basic operations from the* `Set` *library*

Basic properties and operations on sets

We shall now describe the basic properties of sets and the operations on sets in F# as shown in Table 5.2. The functions `Set.ofList` and `Set.toList` are conversion functions between lists and sets:

```
let males = Set.ofList ["Bob"; "Bill"; "Ben"; "Bill"];;
val males : Set<string> = set ["Ben"; "Bill"; "Bob"]

Set.toList males;;
val it : string list = ["Ben"; "Bill"; "Bob"]
```

Note that the resulting list is ordered and contains no repeated elements.

An element can be inserted in a set with the function `Set.add`:

```
Set.add "Barry" males;;
val it : Set<string> = set ["Barry"; "Ben"; "Bill"; "Bob"]
```

and removed from a set with the function `Set.remove`:

```
Set.remove "Bill" males;;
val it : Set<string> = set ["Ben"; "Bob"]
```

The `add` and `remove` operations do not change the original set, that is, they have no side effect. The same observation applies for all other operations in the `Set` library. For example, the add and remove operations above did not change the value of `males`:

```
males;;
val males : Set<string> = set ["Ben"; "Bill"; "Bob"]
```

Containment in a set is tested using `Set.contains` and a subset relationship is tested using `Set.isSubset`:

```
Set.contains "Barry" males;;
val it : bool = false

Set.isSubset males (set ["Bob"; "Bill"; "Ann"]);;
val it : bool = false

Set.isSubset males (Set.add "Ben" (set ["Bob";"Bill";"Ann"]));;
val it : bool = true
```

Due to the ordering required for set elements, every non-empty set has a minimal and a maximal element:

```
Set.minElement (set ["Bob"; "Bill"; "Ben"]);;
val it : string = "Ben"
Set.maxElement (set ["Bob"; "Bill"; "Ben"]);;
val it : string = "Bob"
```

Furthermore, the cardinality of a finite set is in F# given by the function `Set.count`:

```
Set.count (set ["Bob"; "Bill"; "Ben"]);;
val it : int = 3
Set.count (Set.empty);;
val it : int = 0
```

which also shows that the cardinality of the empty set (denoted by `Set.empty`) is 0.

Fundamental operations on sets

We shall now consider the selection of fundamental operations from the `Set` library in F# shown in Table 5.3.

We illustrate set operations for union, intersection and difference using an example where `males` are supposed to be all the males at a golf club, and `boardMembers` are the members of the board for that club:

```
let boardMembers = Set.ofList [ "Alice"; "Bill"; "Ann"];;
val boardMembers : Set<string> = set ["Alice"; "Ann"; "Bill"]

Set.union males boardMembers;;
val it : Set<string> = set ["Alice";"Ann";"Ben";"Bill";"Bob"]
```

Operation Meaning
`union: Set<'a> -> Set<'a> -> Set<'a>`, where `union` $A\ B = A \cup B$
`intersect: Set<'a> -> Set<'a> -> Set<'a>`, where `intersect` $A\ B = A \cap B$
`difference: Set<'a> -> Set<'a> -> Set<'a>`, where `difference` $A\ B = A \setminus B$
`filter: ('a -> bool) -> Set<'a> -> Set<'a>`, where `filter` $p\ A = \{x \in A \mid p(x)\}$
`exists: ('a -> bool) -> Set<'a> -> bool`, where `exists` $p\ A = \exists x \in A.p(x)$
`forall: ('a -> bool) -> Set<'a> -> bool`, where `forall` $p\ A = \forall x \in A.p(x)$
`map: ('a -> 'b) -> Set<'a> -> Set<'b>`, where `map` $f\ A = \{f(x) \mid x \in A\}$
`fold: ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a`, where `fold` $f\ a\ \{b_0, b_1, \ldots, b_{n-2}, b_{n-1}\} = f(f(f(\cdots f(f(a, b_0), b_1), \ldots), b_{n-2}), b_{n-1})$
`foldBack: ('a -> 'b -> 'b) -> Set<'a> -> 'b -> 'b`, where `foldBack` $f\ \{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\}\ b = f(a_0, f(a_1, f(\ldots, f(a_{n-2}, f(a_{n-1}, b)) \cdots)))$

It is assumed that the enumerations in the sets $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\}$ and $\{b_0, b_1, \ldots, b_{n-2}, b_{n-1}\}$ respect the ordering of the respective types.

Table 5.3 *A selection of operations from the* `Set` *library*

```
Set.intersect males boardMembers;;
val it : Set<string> = set ["Bill"]

Set.difference males boardMembers;;
val it : Set<string> = set ["Ben"; "Bob"]
```

where, for example, the set of males being board members is obtained using intersections and the set of those who are not board members is obtained using difference.

A function can be applied to every member of a set using `Set.map` in the same manner it can be applied to every element of a list using `List.map`. The following function, that transforms a set of sets $S = \{s_0, \ldots, s_{n-1}\}$ to the set $\{|s_0|, \ldots, |s_{n-1}|\}$ containing the cardinalities of the elements of S, is, for example, defined using `Set.map` in a natural manner:

```
let setOfCounts s = Set.map Set.count s;;
val setOfCounts: Set<Set<'a>> -> Set<int> when 'a: comparison
```

Consider the F# value for the set of sets $\{\{1, 3, 5\}, \{2, 4\}, \{7, 8, 9\}\}$:

```
let ss = set [set [1;3;5]; set [2;4]; set [7;8;9] ];;
val it : Set<Set<int>>
       = set [set [1; 3; 5]; set [2; 4]; set [7; 8; 9]]

setOfCounts ss;;
val it : Set<int> = set [2; 3]
```

The functions: `Set.exists`, `Set.forall` and `Set.filter`, work in a similar manner to their `List` siblings:

```
Set.exists (fun x -> x>=2) (set [1;3;1;4]);;
val it : bool = true
```

```
Set.forall (fun x -> x>=2) (set [1;3;1;4]);;
val it : bool = false

Set.filter (fun x -> x>3) (set [1;5;-2;8]);;
val it : Set<int> = set [5; 8]
```

The functions `Set.fold` and `Set.foldBack` also correspond to their list siblings. This is illustrated in the following evaluations:

$$\begin{array}{rcl} \texttt{Set.fold (-) 0 (set [1;2;3])} & = & ((0-1)-2)-3 \;=\; -6 \\ \texttt{Set.foldBack (-) (set [1;2;3]) 0} & = & 1-(2-(3-0)) \;=\; 2 \end{array}$$

where the ordering on the set elements is exploited.

The functions `sumSet` and `setOfCounts` can be succinctly declared using `foldBack`:

```
let sumSet s = Set.foldBack (+) s 0;;
val sumSet : Set<int> -> int

let setOfCounts s = Set.foldBack
                      (fun se sn -> Set.add (Set.count se) sn)
                      s
                      Set.empty;;
setOfCounts : Set<Set<'a>> -> Set<int> when 'a : comparison

sumSet (set [1 .. 5]);;
val it : int = 15

setOfCounts (set [set [1;3;5]; set [2;4]; set [7;8;9] ]);;
val it : Set<int> = set [2; 3]
```

Declarations of these functions could also be based on `Set.fold`:

```
let sumSet s = Set.fold (+) 0 s;;

let setOfCounts s = Set.fold
                      (fun sn se -> Set.add (Set.count se) sn)
                      Set.empty
                      s;;
```

Notice that is is more natural to base a declaration of `setOfCounts` on `Set.map` as done above, rather than basing it on one of the fold functions.

Recursive functions on sets

The functions `Set.map`, `Set.filter`, `Set.fold` and `Set.foldBack` will traverse the complete set before they terminate, unless the evaluation is aborted by raising an exception, and this may be undesirable in some situations. Consider, for example, the function that finds the least element in a set satisfying a given predicate:

```
tryFind: ('a -> bool) -> Set<'a> -> 'a option
         when 'a : comparison
```

This function can be declared by repeated extraction of the minimal element from a set until an element satisfying the predicate is found:

```
let rec tryFind p s =
   if Set.isEmpty s then None
   else let minE = Set.minElement s
        if p minE then Some minE
        else tryFind p (Set.remove minE s);;
```

For example, the least three-element set from a set of sets is extracted as follows:

```
let ss = set [set [1;3;5]; set [2;4]; set [7;8;9] ];;

tryFind (fun s -> Set.count s = 3) ss;;
val it : Set<int> option = Some (set [1; 3; 5])
```

A declaration of this function that is based on `Set.fold` will always traverse the entire set leading to a linear best-case running time, while the function declared above will terminate as soon as an element satisfying the predicate is found, and the best-case execution time is dominated by the time required for finding the minimal element in a set, and that execution time is logarithmic in the size of the set when it is represented by a balanced binary tree. Note however, that the worst-case execution time of traversing a set S using `Set.fold` or `Set.foldBack` is $O(|S|)$, that is linear in the size $|S|$ of the set, while it is $O(|S| \cdot \log(|S|))$ for a function based on a recursion schema like that for `tryFind`, due to the logarithmic operations for finding and removing the minimal element of a set.

A more efficient implementation of the function `tryFind` using an enumerator is given on Page 191, and the efficiency of different methods for traversal of collections is analyzed in Exercise 9.14. Enumerators for collections (to be introduced in Section 8.12) provide a far more efficient method than the above used recursion schema for `tryFind`.

Example: Map colouring

The solution of the map-colouring problem from Section 4.6 shall now be improved using sets. The basic algorithmic idea for the solution below using sets is basically the same as that for using lists. But the model using sets is a more natural one. Furthermore, we shall take advantage of the higher-order library functions.

A map is mathematically modelled as a binary relation of countries, that is, as a set of country pairs. Furthermore, since the order in which countries occur in a colour is not relevant and since repetition among the countries in a colour is of no concern, the natural model of a colour is a country set. A similar observation applies to a colouring:

```
type Country   = string;;
type Map       = Set<Country*Country>;;
type Colour    = Set<Country>;;
type Colouring = Set<Colour>;;
```

Two countries c_1, c_2 are neighbors in a map m, if either $(c_1, c_2) \in m$ or $(c_2, c_1) \in m$. In F# this is expressed as follows:

```
let areNb c1 c2 m =
    Set.contains (c1,c2) m || Set.contains (c2,c1) m;;
```

A colour col can be extended by a country c for a given map m, if for every country c' in col, we have that c and c' are not neighbours in m. This can be directly expressed using `Set.forall`:

```
let canBeExtBy m col c =
    Set.forall (fun c' -> not (areNb c' c m)) col;;
```

The function

```
extColouring: Map -> Colouring -> Country -> Colouring
```

is declared as a recursive function over the colouring:

```
let rec extColouring m cols c =
    if Set.isEmpty cols
    then Set.singleton (Set.singleton c)
    else let col = Set.minElement cols
         let cols' = Set.remove col cols
         if canBeExtBy m col c
         then Set.add (Set.add c col) cols'
         else Set.add col (extColouring m cols' c);;
```

This recursive declaration is preferred to using a declaration based on either `Set.fold` or `Set.foldBack`, since the recursive version terminates as soon as a colour that can be extended by the country is found, whereas a declaration based on one of the fold functions always will iterate through the entire colouring.

A set of countries is obtained from a map by the function:

```
countries: Map -> Set<Country>
```

The declaration of this function is based on repeated insertion (using `Set.fold`) of the countries in the map into a set:

```
let countries m =
    Set.fold
      (fun set (c1,c2) -> Set.add c1 (Set.add c2 set))
      Set.empty
      m;;
```

The function

```
colCntrs: Map -> Set<Country> -> Colouring
```

that creates a colouring for a set of countries in a given map, can be declared by repeated insertion of countries in colourings using the `extColouring` function:

```
let colCntrs m cs = Set.fold (extColouring m) Set.empty cs;;
```

The function that creates a colouring from a map is declared using function composition and used as follows:

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

let exMap = Set.ofList [("a","b"); ("c","d"); ("d","a")];;

colMap exMap;;
val it: Set<Set<string>> = set [set ["a";"c"]; set ["b";"d"]]
```

Comparing this set-based solution with the list-based one in Section 4.6 we can first observe that the set-based model is more natural, due to the facts that a map is a binary relation of countries and a colouring is a partitioning of the set of countries in a map. For most of the functions there is even an efficiency advantage with the set-based functions. This advantage is due to the following

- the worst-case execution time for testing for membership of a set (represented by a balanced binary tree) is logarithmic in the size of the set, while this operation is linear when the set is represented by a list, and
- the worst-case execution time for inserting an element into a set (represented by a balanced binary tree) is logarithmic in the size of the set, while this operation is linear when the set is represented by a list without duplicated elements.

The use of lists has an advantage in the case of the recursive function `extColouring` since the pattern matching for lists yields a more readable declaration and since the worst-case execution time of this list-based version is linear in the size $|S|$ of the colouring S, while it is $O(|S| \cdot \log(|S|))$ for the set-based one. (See remark on Page 110.)

An improved version is therefore based on the following type declaration:

```
type Country   = string;;
type Map       = Set<Country*Country>;;
type Colour    = Set<Country>;;
type Colouring = Colour list;;
```

Just two functions `extColouring` and `colCntrs` are affected by this change of the type for `colouring` while the remaining functions are as above. The new declarations are:

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

let colCntrs m cs = Set.fold (extColouring m) [] cs;;

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

5.3 Maps

In the modelling and solution for many problems it is often convenient to use finite functions to uniquely associate *values* with *keys*. Such finite functions from keys to values are called *maps*. This section introduces the map concept and some of the main operations on maps in the F# `Map` library. Please consult the on-line documentation in [9] for an overview of the complete `Map` library.

The mathematical concept of a map

A *map* from a set A to a set B is a *finite* subset A' of A together with a *function* m defined on A':

$$m : A' \to B$$

The set A' is called the *domain* of m and we write $\mathrm{dom}\, m = A'$.

A map m can be described in a tabular form as shown below. The left column contains the elements $a_0, a_1, \ldots, a_{n-1}$ of the set A', while the right column contains the corresponding values $m(a_0) = b_0, m(a_1) = b_1, \ldots, m(a_{n-1}) = b_{n-1}$:

a_0	b_0
a_1	b_1
$\vdots$	
a_{n-1}	b_{n-1}

An element a_i in the set A' is called a *key* for the map m. A pair (a_i, b_i) is called an *entry*, and b_i is called the *value* for the key a_i. Note that the order of the entries is of no significance, as the map only expresses an association of values to keys. Note also that any two keys a_i and a_j in different entries are different, as there is only one value for each key. Thus, a map may be represented as a finite set of its entries. We use

$$\mathrm{entriesOf}(m) = \{(a_0, b_0), \ldots, (a_{n-1}, b_{n-1})\}$$

to denote the sets of entries of a map.

The cash register example in Chapter 4.6 comprises an article register associating name and price to article codes, and this register can be viewed as a map. A key in the map is an article code and the corresponding value is the pair with the name and price of the article.

A particular article register is given by the following map:

reg_1 :

a1	(cheese, 25)
a2	(herring, 4)
a3	(soft drink, 5)

It associates the value (cheese, 25) with the key a1, the value (herring, 4) with the key a2, and the value (soft drink, 5) with the key a3. Hence, it has the domain $\{\mathrm{a1}, \mathrm{a2}, \mathrm{a3}\}$.

Operation Meaning
`ofList: ('a*'b) list -> Map<'a,'b>`, where `ofList` $[(a_0, b_0); \ldots; (a_{n-1}, b_{n-1})] = m$
`toList: Map<'a,'b> -> ('a*'b) list`, where `toList` $m = [(a_0, b_0); \ldots; (a_{n-1}, b_{n-1})]$
`add: 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>`, where `add` $a\ b\ m = m'$, where m' is obtained by overriding m with the entry (a, b)
`containsKey: 'a -> Map<'a,'b> -> bool`, where `containsKey` $a\ m = a \in \mathrm{dom}\ m$
`find: 'a -> Map<'a,'b> -> 'b`, where `find` $a\ m = m(a)$, if $a \in \mathrm{dom}\ m$; otherwise an exception is raised
`tryFind: 'a -> Map<'a,'b> -> 'b option`, where `tryFind` $a\ m =$ `Some` $(m(a))$, if $a \in \mathrm{dom}\ m$; `None` otherwise
`filter: ('a -> 'b -> bool) -> Map<'a,'b> -> Map<'a,'b>`, where `filter` $p\ m$ is obtained from m by deletion of entries (a_i, b_i) where $p\ a_i\ b_i =$ `false`
`exists: ('a -> 'b -> bool) -> Map<'a,'b> -> bool`, where `exists` $p\ m = \exists (a, b) \in \mathrm{entriesOf}(m).p\ a\ b$
`forall: ('a -> 'b -> bool) -> Map<'a,'b> -> bool`, where `forall` $p\ A = \forall (a, b) \in \mathrm{entriesOf}(m).p\ a\ b$
`map: ('a -> 'b -> 'c) -> Map<'a,'b> -> Map<'a,'c>`, where `map` $f\ m =$ `ofList` $[(a_0, f\ a_0\ b_0); \ldots; (a_{n-1}, f\ a_{n-1}\ b_{n-1})]$
`fold: ('a -> 'b -> 'c -> 'a) -> 'a -> Map<'b,'c> -> 'a`, where `fold` $f\ a\ m_{bc} = f(\cdots(f(f\ a\ b_0\ c_0)\ b_1\ c_1)\ldots)\ b_{n-1}\ c_{n-1}$
`foldBack: ('a -> 'b -> 'c -> 'c) -> Map<'a,'b> -> 'c -> 'c`, where `foldBack` $f\ m\ c = f\ a_0\ b_0\ (f\ a_1\ b_1\ (f \ldots (f\ a_{n-1}\ b_{n-1}\ c)\cdots))$

It is assumed that m and m_{bc} are maps with types `Map<'a,'b>` and `Map<'b,'c>`, that

$$\begin{array}{lcl} \mathrm{entriesOf}(m) & = & \{(a_0, b_0), \ldots, (a_{n-1}, b_{n-1})\} \\ \mathrm{entriesOf}(m_{bc}) & = & \{(b_0, c_0), \ldots, (b_{n-1}, c_{n-1})\} \end{array}$$

and that the enumerations $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\}$ and $\{b_0, b_1, \ldots, b_{n-2}, b_{n-1}\}$ respect the ordering of the respective types.

Table 5.4 *A selection of operations from the* `Map` *library*

Maps in F#

The `Map` library of F# supports maps of polymorphic types `Map<'a,'b>`, where `'a` and `'b` are the types of the keys and values, respectively, of the map. The `Map` is implemented using balanced binary trees, and requires therefore that an ordering is defined on the type `'a` of keys. Some of the functions of the `Map` library are specified in Table 5.4.

A map in F# can be generated from a list of its entries. For example:

```
let reg1 = Map.ofList [("a1",("cheese",25));
                       ("a2",("herring",4));
                       ("a3",("soft drink",5))];;
val reg1 : Map<string,(string * int)> =
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));
       ("a3", ("soft drink", 5))]
```

is an F# map for the register reg_1, where keys are strings and values are pairs of the type

`string*int`. If the list contains multiple entries for the same key, then the last occurring entry is the significant one:

```
Map.ofList [(1,"a"); (2,"b"); (2,"c"); (1,"d")];;
val it : Map<int,string> = map [(1, "d"); (2, "c")]
```

The list of entries of a map is achieved using the `Map.toList` function:

```
Map.toList reg1;;
val it : (string * (string * int)) list =
  [("a1", ("cheese", 25)); ("a2", ("herring", 4));
   ("a3", ("soft drink", 5))]
```

An entry can be added to a map using `add` while the value for a key in a map is retrieved using either `find` or `tryFind`:

```
let reg2 = Map.add "a4" ("bread", 6) reg1;;
val reg2 : Map<string,(string * int)> =
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));
       ("a3", ("soft drink", 5)); ("a4", ("bread", 6))]

Map.find "a2" reg1;;
val it : string * int = ("herring", 4)

Map.tryFind "a2" reg1;;
val it : (string * int) option = Some ("herring", 4)

Map.containsKey "a4" reg1;;
val it : bool = false

Map.find "a4" reg1;;
System.Collections.Generic.KeyNotFoundException: The given key
was not present in the dictionary.
...
Stopped due to error

Map.tryFind "a4" reg1;;
val it : (string * int) option = None
```

where `find` raises an exception if the key is not in the domain of the map and `tryFind` returns `None` in that case.

The old entry is overridden if you add an entry for an already existing key. The entry for a given key can be deleted using the `remove` function:

```
let reg3 = Map.add "a4" ("bread", 8) reg1;;
val reg3 : Map<string,(string * int)> =
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));
       ("a3", ("soft drink", 5)); ("a4", ("bread", 8))]
```

```
let reg4 = Map.remove "a2" reg3;;
val reg4 : Map<string,(string * int)> =
  map [("a1", ("cheese", 25)); ("a3", ("soft drink", 5));
       ("a4", ("bread", 8))]
```

The `Map` functions `exists`, `forall`, `map`, `fold` and `foldBack` are similar to their `List` and `Set` siblings. These functions are specified with type and meaning in Table 5.4, so we just give some illustrative examples below.

The following expression tests whether there are expensive articles, for which the price exceeds 100, in a register:

```
Map.exists (fun _ (_,p) -> p > 100) reg1;;
val it : bool = false
```

The natural requirement that every price occurring in a register must be positive is expressed by:

```
Map.forall (fun _ (_,p) -> p > 0) reg1;;
val it : bool = true
```

The part of a register with articles having a price smaller than 7 is extracted as follows:

```
Map.filter (fun _ (_,p) -> p < 7) reg3;;
val it : Map<string,(string * int)> =
  map [("a2", ("herring", 4)); ("a3", ("soft drink", 5))]
```

A new register, where a 15% discount is given on all articles, can be computed as follows:

```
Map.map
  (fun ac (an,p) -> (an,int(round(0.85*(float p)))))
  reg3;;
val it : Map<string,(string * int)> =
  map [("a1", ("cheese", 21)); ("a2", ("herring", 3));
       ("a3", ("soft drink", 4)); ("a4", ("bread", 7))]
```

We can extract the list of article codes and prices for a given register using the fold functions for maps:

```
Map.foldBack (fun ac (_,p) cps -> (ac,p)::cps) reg1 [];;
val it: (string*int) list = [("a1",25); ("a2",4); ("a3",5)]

Map.fold (fun cps ac (_,p) -> (ac,p)::cps) [] reg1;;
val it: (string*int) list = [("a3",5); ("a2",4); ("a1",25)]
```

where these two examples show that the entries of a map are ordered according to the keys.

Example: Cash register

We give a solution to the cash register example discussed in Section 4.6. Article codes and names, number of pieces and prices are modelled just like in Section 4.6:

```
type ArticleCode = string;;
type ArticleName = string;;
type NoPieces    = int;;
type Price       = int;;
```

The natural model of a register, associating article name and price with each article code, is using a map:

```
type Register = Map<ArticleCode, ArticleName*Price>;;
```

since an article code is a unique identification of an article.

The information concerning a bill is also modelled as in Section 4.6:

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

For the remaining parts we give three versions.

Version 1

In the first version we model a purchase just as in Section 4.6:

```
type Item     = NoPieces * ArticleCode;;
type Purchase = Item list;;
```

The function `makebill1: Register -> Purchase -> Bill` makes the bill for a given register and purchase and it can be defined by a recursion following the structure of a purchase:

```
let rec makeBill1 reg = function
    | []           -> ([],0)
    | (np,ac)::pur ->
        match Map.tryFind ac reg with
        | Some(aname,aprice) ->
            let tprice          = np*aprice
            let (infos,sumbill) = makeBill1 reg pur
            ((np,aname,tprice)::infos, tprice+sumbill)
        | None               ->
            failwith(ac + " is an unknown article code");;
```

where an exception signals an undefined article code in a register. We use the function `Map.tryFind` in order to detect when this exception should be raised. A simple application of the program is:

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

makeBill1 reg1 pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

where `reg1` is declared on Page 114.

Version 2

The recursion pattern of `makeBill1` is the same as that of `List.foldBack`, and the explicit recursion can be replaced by application of that function. Furthermore, it may be acceptable to use the exception from the `Map` library instead of using `failwith`. This leads to the following declaration:

```
let makeBill2 reg pur =
    let f (np,ac) (infos,billprice) =
        let (aname, aprice) = Map.find ac reg
        let tprice          = np*aprice
        ((np,aname,tprice)::infos, tprice+billprice)
    List.foldBack f pur ([],0);;

makeBill2 reg1 pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

where `Map.find` will raise an exception if an article code is not found in a register.

Version 3

A purchase is so far just modelled as a list of items, each item consisting of a count and an article code. The order of appearance in the list may represent the sequence in which items are placed on the counter in the shop. One may, however, argue that a purchase of the following three items: three herrings, one piece of cheese, and two herrings, is the same as a purchase of one piece of cheese and five herrings. Furthermore, the latter form is more convenient if we have to model a discount on five herrings, as the discount applies independently of the order in which the items are placed on the counter. Thus one could model a purchase as a map, where article codes are keys and number of pieces are values of a map.

```
type Purchase = Map<ArticleCode,NoPieces>;;
```

With this model, the `makeBill3: Register -> Purchase -> Bill` function is declared and used as follows:

```
let makeBill3 reg pur =
    let f ac np (infos,billprice) =
        let (aname, aprice) = Map.find ac reg
        let tprice          = np*aprice
        ((np,aname,tprice)::infos, tprice+billprice)
    Map.foldBack f pur ([],0);;
```

where we use `Map.foldBack` to fold the function `f` over a purchase.

An example showing the use of this function is:

```
let purMap = Map.ofList [("a2",3); ("a1",1)];;
val purMap : Map<string,int> = map [("a1", 1); ("a2", 3)]

makeBill3 reg1 purMap;;
val it : (int * string * int) list * int =
  ([(1, "cheese", 25); (3, "herring", 12)], 37)
```

We leave the generation of a map for a purchase on the basis of a list of items for Exercise 5.9. Furthermore, it is left for Exercise 5.10 to take discounts for certain articles into account.

Summary

In this chapter we have introduced the list, set and map parts from the collection library of F#. These three libraries are efficient implementations of such finite, immutable collections. Notice that this chapter just covers a small part of the libraries. Furthermore, in many applications these collections provide a natural data model and we strongly encourage to use these libraries whenever it is appropriate.

In Chapter 11 we introduce sequences, which is another part of the collection library. Sequences are (possibly infinite) list-like structures, where just a finite part of the sequence is computed at any stage of a computation.

Exercises

5.1 Give a declaration for `List.filter` using `List.foldBack`.

5.2 Solve Exercise 4.15 using `List.fold` or `List.foldBack`.

5.3 Solve Exercise 4.12 using `List.fold` or `List.foldBack`.

5.4 Declare a function `downto1` such that:

$$\begin{array}{lcll} \texttt{downto1}\ f\ n\ e & = & f(1, f(2, \ldots, f(n-1, f(n, e))\ldots)) & \text{for } n > 0 \\ \texttt{downto1}\ f\ n\ e & = & e & \text{for } n \leq 0 \end{array}$$

Declare the factorial function by use of `downto1`.
Use `downto1` to declare a function that builds the list $[g(1), g(2), \ldots, g(n)]$ for a function g and an integer n.

5.5 Consider the map colouring example in Section 4.6. Give declarations for the functions `areNb canBeExtBy`, `extColouring`, `countries` and `colCntrs` using higher-order list functions. Are there cases where the old declaration from Section 4.6 is preferable?

5.6 We define a *relation* from a set A to a set B as a subset of $A \times B$. A relation r' is said to be *smaller* than r, if r' is a subset of r, that is, if $r' \subseteq r$. A relation r is called *finite* if it is a finite subset of $A \times B$. Assuming that the sets A and B are represented by F# types `'a` and `'b` allowing comparison we can represent a finite relation r by a value of type `set<'a * 'b>`.

1. The domain $\mathrm{dom}\ r$ of a relation r is the set of elements a in A where there exists an element b in B such that $(a, b) \in r$. Write an F# declaration expressing the domain function.
2. The range $\mathrm{rng}\ r$ of a relation r is the set of elements b in B where there exists an element a in A such that $(a, b) \in r$. Write an F# declaration expressing the range function.
3. If r is a finite relation from A to B and a is an element of A, then the application of r to a, $\mathrm{apply}\ r\ a$, is the set of elements b in B such that $(a, b) \in r$. Write an F# declaration expressing the apply function.
4. A relation r from a set A to the same set is said to be *symmetric* if $(a_1, a_2) \in r$ implies $(a_2, a_1) \in r$ for any elements a_1 and a_2 in A. The symmetric closure of a relation r is the smallest symmetric relation containing r. Declare an F# function to compute the symmetric closure.

5. The relation composition $r \circ\circ s$ of a relation r from a set A to a set B and a relation s from B to a set C is a relation from A to C. It is defined as the set of pairs (a, c) where there exist an element b in B such that $(a, b) \in r$ and $(b, c) \in s$. Declare an F# function to compute the relational composition.
6. A relation r from a set A to the same set A is said to be *transitive* if $(a_1, a_2) \in r$ and $(a_2, a_3) \in r$ implies $(a_1, a_3) \in r$ for any elements a_1, a_2 and a_3 in A. The transitive closure of a relation r is the smallest transitive relation containing r. If r contains n elements, then the transitive closure can be computed as the union of the following n relations:

$$r \cup (r \circ\circ r) \cup (r \circ\circ r \circ\circ r) \cup \cdots \cup (r \circ\circ r \circ\circ \cdots \circ\circ r)$$

Declare an F# function to compute the transitive closure.

5.7 Declare a function `allSubsets` such that `allSubsets n k` is the set of all subsets of $\{1, 2, \ldots, n\}$ containing exactly k elements. Hint: use ideas from Exercise 2.8. For example, $\binom{n}{k}$ is the number of subsets of $\{1, 2, \ldots, n\}$ containing exactly k elements.

5.8 Give declarations for `makeBill3` using `map.fold` rather than `map.foldBack`.

5.9 Declare a function to give a purchase map (see Version 3 on Page 118) on the basis of a list of items (from the Versions 1 and 2).

5.10 Extend the cash register example to take discounts for certain articles into account. For example, find a suitable representation of discounts and revise the function to make a bill accordingly.

5.11 Give a solution for Exercise 4.23 using the `Set` and `Map` libraries.