CHAPTER 4

Introducing Imperative Programming

In Chapter 3, you saw some of the constructs that make up F# functional programming. At the core of the functional-programming paradigm is "programming without side effects," called pure functional programming. In this chapter you learn about programming *with* side effects, called imperative programming.

About Functional and Imperative Programming

In the functional programming paradigm, programs compute the result of a mathematical expression and don't cause any side effects, simply returning the result of the computation. The formulas used in spreadsheets are often pure, as is the core of functional-programming languages such as Haskell.

However, F# is not a pure functional language – functions and expressions *can* have side-effects. It is very common to write pure functions and expressions in F#, but you can also write function and expression which mutate data, perform input/output, communicate over the network, start new parallel threads, and raise exceptions. These side-effects are collectively called imperative programming. The F#-type system doesn't enforce a strict distinction between expressions that perform these actions and expressions that don't.

If your primary programming experience has been with an imperative language such as C, C#, or Java, you may initially find yourself using imperative constructs fairly frequently in F#. Over time, however, F# programmers learn to perform a surprisingly large proportion of routine programming tasks within the side-effect-free subset of the language – functional programming can express an astounding range of computations succinctly and accurately. However, when you reach the limits of what functional programming can do, you will need to turn to imperative programming. In particular, F# programmers tend to use imperative programming in the following situations:

- When scripting and prototyping using F# Interactive
- When working with .NET library components that use side effects heavily, such as GUI libraries and I/O libraries
- When initializing complex data structures
- When using inherently imperative, efficient data structures, such as hash tables, hash sets and matrices
- When locally optimizing functions in a way that improves performance

D. Syme et al., *Expert F# 3.0*

- When working with very large data structures or in scenarios in which the allocation of data structures must be minimized for performance reasons

Some F# programs don't use any imperative techniques except as part of the outermost layer of their program architecture. Adopting this form of pure functional programming for a time is an excellent way to hone your functional programming techniques.

Programming with fewer side effects is attractive for many reasons. For example, eliminating unnecessary side effects nearly always reduces the complexity of your code, so it leads to fewer bugs and more rapid delivery of high-quality components. Another thing experienced functional programmers appreciate is that the programmer or compiler can easily adjust the order in which expressions are computed. If your programs don't have side effects, then it's easier to think clearly about your code: you can visually check when two programs are equivalent, and it's easier to make radical adjustments to your code without introducing new, subtle bugs. Programs that are free from side effects can often be computed on demand or in parallel, often by making very small, local changes to your code to introduce the use of delayed data structures or parallelism. Finally, mutation and other side-effects introduce complex time-dependent interactions into your code, making your code difficult to test and debug, especially when data is accessed concurrently from multiple threads, discussed further in Chapter 11.

Imperative Looping and Iterating

The first imperative constructs you look at are those associated with looping and iteration. Three available looping constructs help simplify writing iterative code with side effects:

- Simple `for` loops: `for` *`val = start-expr`* `to` *`end-expr`* `do` *`work-expr`*
- Simple `while` loops: `while` *`condition-expr`*`do` *`work-expr`*
- Sequence loops: `for` *`pattern`* `in` *`collection-expr`* `do` *`work-expr`*

All three constructs are for writing imperative programs, indicated partly by the fact that in all cases the body of the loop must have a return type of `unit`. Note that `unit` is the F# type that corresponds to `void` in imperative languages, such as C, and it has the single value `()`. This means the loops don't produce a useful value, so the only use a loop can have is to perform some kind of side-effect. The following sections cover these three constructs in more detail.

Simple for Loops

Simple `for` loops are used to iterate over integer ranges. This is illustrated here by a replacement implementation of the `repeatFetch` function from Chapter 2:

The first looping construct is simple for loops, which iterate over a range of integers. For example, the following construct fetched the same web page `n` times, printing the result each time:

```
let repeatFetch url n =
    for i = 1 to n do
        let html = http url
        printf "fetched <<< %s >>>\n" html
    printf "Done!\n"
```

This loop is executed for successive values of `i` over the given range, including both start and end indexes.

Simple While Loops

The second looping construct is a `while` loop, which repeats until a given guard is false. For example, here is a way to keep your computer busy until the weekend:

```
open System

let loopUntilSaturday() =
    while (DateTime.Now.DayOfWeek <> DayOfWeek.Saturday) do
        printf "Still working!\n"

    printf "Saturday at last!\n"
```

When executing this code in F# Interactive, you can interrupt its execution by choosing the "Cancel Interactive Evaluation" command in Visual Studiom or by using Ctrl+C when running `fsi.exe` from the command line.

More Iteration Loops over Sequences

As you will learn in depth in Chapter 9, any values compatible with the type `seq<type>` can be iterated using the `for` *`pattern`* `in` *`seq`* `do ...` construct. The input `seq` may be an F# list value, any `seq<type>`, or a value of any type supporting a `GetEnumerator` method. Here are some simple examples:

```
> for (b, pj) in [("Banana 1", false); ("Banana 2", true)] do
      if pj then
          printfn "%s is in pyjamas today!" b;;

Banana 2 is in pyjamas today!
```

The following example iterates the results of a regular expression match. The type returned by the .NET method `System.Text.RegularExpressions.Regex.Matches` is a `MatchCollection`, which, for reasons known best to the .NET designers, doesn't directly support the `seq<Match>` interface. It does, however, support a `GetEnumerator` method that permits iteration over the individual results of the operation, each of which is of type `Match`; the F# compiler inserts the conversions necessary to view the collection as a `seq<Match>` and perform the iteration. You will learn more about using the .NET Regular Expression library in Chapter 8:

```
> open System.Text.RegularExpressions;;
> for m in Regex.Matches("All the Pretty Horses","[a-zA-Z]+") do
      printf "res = %s\n" m.Value;;

res = All
res = the
res = Pretty
res = Horses
```

Using Mutable Records

One of the most common kinds of imperative programming is to use mutation to adjust the contents of values. Values whose contents can be adjusted are called *mutable* values. The simplest mutable values in F# are mutable records. Record types and their use in functional programming are discussed in more detail in Chapter 5. A record is mutable if one or more of its fields is labeled `mutable`. This means that record fields can be updated using the `<-` operator:—that is, the same syntax used to set a property. Mutable fields are generally used for records that implement the internal state of objects, discussed in Chapters 6 and 7.

For example, the following code defines a record used to count the number of times an event occurs and the number of times the event satisfies a particular criterion:

```
type DiscreteEventCounter =
    { mutable Total : int;
      mutable Positive : int;
      Name : string }

let recordEvent (s : DiscreteEventCounter) isPositive =
    s.Total <- s.Total+1
    if isPositive then s.Positive <- s.Positive + 1

let reportStatus (s : DiscreteEventCounter) =
    printfn "We have %d %s out of %d" s.Positive s.Name s.Total

let newCounter nm =
    { Total = 0;
      Positive = 0;
      Name = nm }
```

You can use this type as follows (this example uses the `http` function from Chapter 2):

```
let longPageCounter = newCounter "long page(s)"

let fetch url =
    let page = http url
    recordEvent longPageCounter (page.Length > 10000)
    page
```

Every call to the function `fetch` mutates the mutable-record fields in the global variable `longPageCounter`. For example:

```
> fetch "http://www.smh.com.au" |> ignore;;
> fetch "http://www.theage.com.au" |> ignore;;
> reportStatus longPageCounter;;

We have 2 long page(s) out of 2
```

Record types can also support members (for example, properties and methods) and give explicit implementations of interfaces, discussed in Chapter 6. When compiled, records become .NET classes and can be used fromC# and other .NET languages. Practically speaking, this means you can use them as one way to implement object-oriented abstractions.

Mutable Reference Cells

One particularly useful mutable record is the general-purpose type of mutable reference cells, or ref cells for short. These often play much the same role as pointers in other imperative programming languages. You can see how to use mutable reference cells in this example:

```
> let cell1 = ref 1;;

val cell1 : int ref = {contents = 1;}

> cell1.Value;;

val it : int = 1

> cell1 := 3;;

val it : unit = ()

> cell1;;

val it : int ref = {contents = 3;}

> cell1.Value;;

val it : int = 3
```

The type is `'T ref`, and three associated functions are `ref`, `!`, and `:=`. The types of these are:

```
val ref : 'T -> 'T ref
val (:=) : 'T ref -> 'T -> unit
val (!) : 'T ref -> 'T
```

These allocate a reference cell, mutate the cell, and read the cell, respectively. The operation `cell1 := 3` is the key one; after this operation, the value returned by evaluating the expression `!cell1` is changed. You can also use either the `contents` field or the `Value` property to access the value of a reference cell.

Both the `'T ref` type and its operations are defined in the F# library as simple record-data structures with a single mutable field:

```
type 'T ref =
    {mutable contents : 'T}
    member cell.Value = cell.contents

let (!) r = r.contents

let (:=) r v = r.contents <- v

let ref v = {contents = v}
```

The type `'T ref` is a synonym for a type `Microsoft.FSharp.Core.Ref<'T>` defined in this way.

WHICH DATA STRUCTURES ARE MUTABLE?

It's useful to know which data structures are mutable and which aren't. If a data structure can be mutated, this is typically evident in the types of operations you can perform on that structure. For example, if a data structure `Table<'Key,'Value>` has an operation such as the following, in practice you can be sure that updates to the data structure modify the data structure itself:

```
val add : Table<'Key,'Value> -> 'Key -> 'Value -> unit
```

That is, the updates to the data structure are destructive, and no value is returned from the operation; the result is the type `unit`, which is akin to `void` in C and many other languages. Likewise, the following member indicates that the data structure is almost certainly mutable:

```
member Add : 'Key * 'Value -> unit
```

In both cases, the presence of `unit` as a return type is a sure sign that an operation performs some side effects. In contrast, operations on immutable data structures typically return a new instance of the data structure when an operation such as `add` is performed. For example:

```
val add : 'Key -> 'Value -> Table<'Key,'Value> -> Table<'Key,'Value>
```

Or for example:

```
member Add : 'Key * 'Value -> Table<'Key,'Value>
```

As discussed in Chapter 3, immutable data structures are also called functional or persistent. The latter name is used because the original table isn't modified when adding an element. Well-crafted persistent data structures don't duplicate the actual memory used to store the data structure every time an addition is made; instead, internal nodes can be shared between the old and new data structures. Example persistent data structures in the F# library are F# lists, options, tuples, and the types `Microsoft.FSharp.Collections.Map<'Key,'Value>` and `Microsoft.FSharp.Collections.Set<'Key>`. Most data structures in the .NET libraries aren't persistent, although if you're careful, you can use them as persistent data structures by accessing them in read-only mode and copying them if necessary.

Avoiding Aliasing

Like all mutable data structures, two mutable record values or two values of type `'T ref` may refer to the same reference cell—this is called *aliasing*. Aliasing of immutable data structures isn't a problem; no client consuming or inspecting the data values can detect that the values have been aliased. Aliasing of mutable data can lead to problems in understanding code, however. In general, it's good practice to ensure that no two values currently in scope directly alias the same mutable data structures. The following example continues from earlier and shows how an update to `cell1` can affect the value returned by `!cell2`:

```
> let cell2 = cell1;;

val cell2 : int ref = {contents = 3;}

> !cell2;;

val it : int = 3

> cell1 := 7;;

val it : unit = ()
```

```
> !cell2;;

val it : int = 7
```

Hiding Mutable Data

Mutable data is often hidden behind an encapsulation boundary. Chapter 7 looks at encapsulation in more detail, but one easy way to do this is to make data private to a function. For example, the following shows how to hide a mutable reference within the inner closure of values referenced by a function value:

```
let generateStamp =
    let count = ref 0
    (fun () -> count := !count + 1; !count)
```

```
val generateStamp: unit -> int
```

The line `let count = ref 0` is executed once, when the `generateStamp` function is defined. Here is an example of the use of this function:

```
> generateStamp();;

val it : int = 1

> generateStamp();;

val it : int = 2
```

This is a powerful technique for hiding and encapsulating mutable state without resorting to writing new type and class definitions. It's good programming practice in polished code to ensure that all related items of mutable state are collected under some named data structure or other entity, such as a function.

UNDERSTANDING MUTATION AND IDENTITY

F# encourages the use of objects whose logical identity (if any) is based purely on the characteristics (for example, fields and properties) of the object. For example, the identity of a pair of integers (1,2) is determined by the two integers themselves; two tuple values that each contain these two integers are, for practical purposes, identical. This is because tuples are immutable and support structural equality, hashing, and comparison, discussed further in Chapters 5 and 9.

Mutable reference cells are different; they can reveal their identities through aliasing and mutation. Not all mutable values necessarily reveal their identity through mutation, however. For example, sometimes mutation is used just to bootstrap a value into its initial configuration, such as when connecting the nodes of a graph. These are relatively benign uses of mutation.

Ultimately, you can detect whether two mutable values are the same object by using the function `System.Object.ReferenceEquals`. You can also use this function on immutable values to detect whether two values are represented by the physically same value. In this circumstance, however, the results returned by the function may change according to the optimization settings you apply to your F# code.

Using Mutable Locals

You saw in the previous section that mutable references must be explicitly dereferenced. F# also supports mutable locals that are implicitly dereferenced. These must either be top-level definitions or be local variables in a function:

```
> let mutable cell1 = 1;;

val mutable cell1 : int = 1

> cell1;;

val it : int = 1

> cell1 <- 3;;
> cell1;;

val it : int = 3
```

The following shows how to use a mutable local:

```
let sum n m =
    let mutable res = 0
    for i = n to m do
        res <- res + i
    res
```

```
> sum 3 6;;

val it : int = 18
```

F# places strong restrictions on the use of mutable locals. In particular, unlike mutable references, mutable locals are guaranteed to be stack-allocated values, which is important in some situations because the .NET garbage collector won't move stack values. As a result, mutable locals may not be used in any inner anonymous functions or other closure constructs, with the exception of top-level mutable values, which can be used anywhere, and mutable fields of records and objects, which are associated with the heap-allocated objects themselves. You will learn more about mutable object types in Chapter 6. Reference cells and types containing mutable fields can be used instead to make the existence of heap-allocated imperative state obvious.

Working with Arrays

Mutable arrays are a key data structure used as a building block in many high-performance computing scenarios. This example illustrates how to use a one-dimensional array of `float` values:

```
> let arr = [|1.0; 1.0; 1.0|];;

val arr : float [] = [|1.0; 1.0; 1.0|]

> arr.[1];;

val it : float = 1.0
```

```
> arr.[1] <- 3.0;;
> arr;;

val it : float [] = [|1.0; 3.0; 1.0|]
```

F# array values are usually manipulated using functions from the `Array` module; its full path is `Microsoft.FSharp.Collections.Array`, but you can access it with the short name `Array`. Arrays are created either by using the creation functions in that module (such as `Array.init`, `Array.create`, and `Array.zeroCreate`) or by using sequence expressions, as discussed in Chapter 9. Some useful methods are also contained in the `System.Array` class. Table 4-1 shows some common functions from the `Array` module.

Table 4-1. *Some Important Expressions, Functions, and Aggregate Operators from the* `Array` *Module*

Operator	Type	Explanation
`Array.append`	`'T[] -> 'T[] -> 'T[]`	Returns a new array containing elements of the first array followed by elements of the second array
`Array.sub`	`'T[] -> int -> int -> 'T[]`	Returns a new array containing a portion of elements of the input array
`Array.copy`	`'T[] -> 'T[]`	Returns a copy of the input array
`Array.iter`	`('T -> unit) -> 'T[] -> unit`	Applies a function to all elements of the input array
`Array.filter`	`('T -> bool) -> 'T[] -> 'T[]`	Returns a new array containing a selection of elements of the input array
`Array.length`	`'T[] -> int`	Returns the length of the input array
`Array.map`	`('T -> 'U) -> 'T[] -> 'U[]`	Returns a new array containing the results of applying the function to each element of the input array
`Array.fold`	`('T -> 'U -> 'T) -> 'T -> 'U[] -> 'T`	Accumulates left to right over the input array
`Array.foldBack`	`('T -> 'U -> 'U) -> 'T[] -> 'U -> 'U`	Accumulates right to left over the input array

F# arrays can be very large, up to the memory limitations of the machine (a 3GB limit applies on 32-bit systems). For example, the following creates an array of 100 million elements (of total size approximately 400MB for a 32-bit machine):

```
> let bigArray = Array.zeroCreate<int> 100000000;;

val bigArray : int [] = ...
```

The following attempt to create an array more than 4GB in size causes an `OutOfMemoryException` on one of our machines:

```
> let tooBig = Array.zeroCreate<int> 1000000000;;

System.OutOfMemoryException: Exception of type 'System.OutOfMemoryException'
was thrown.
```

■ **Note** Arrays of value types (such as `int`, `float32`, `float`, `int64`) are stored flat, so only one object is allocated for the entire array. Arrays of other types are stored as an array of object references. Primitive types, such as integers and floating-point numbers, are all value types; many other .NET types are also value types. The .NET documentation indicates whether each type is a value type. Often, the word struct is used for value types. You can also define new struct types directly in F# code, as discussed in Chapter 6. All other types in F# are reference types, such as all record, tuple, discriminated union, and class and interface values.

Generating and Slicing Arrays

As you will explore in more depth in Chapter 9, you can use "sequence expressions" as a way to generate interesting array values. For example:

```
> let arr = [|for i in 0 .. 5 -> (i, i * i)|];;
val arr : (int * int) [] = [|(0, 0); (1, 1); (2, 4); (3, 9); (4, 16); (5, 25)|]
```

You can also use a convenient syntax for extracting subarrays from existing arrays; this is called *slice notation*. A slice expression for a single-dimensional array has the form `arr.[start..finish]`, where one of `start` and `finish` may optionally be omitted, and index zero or the index of the last element of the array is assumed instead. For example:

```
> let arr = [|for i in 0 .. 5 -> (i, i * i)|];;
val arr : (int * int) [] = [|(0, 0); (1, 1); (2, 4); (3, 9); (4, 16); (5, 25)|]
> arr.[1..3];;
val it : (int * int) [] = [|(1, 1); (2, 4); (3, 9);|]
> arr.[..2];;
val it : (int * int) [] = [|(0, 0); (1, 1); (2, 4);|]
> arr.[3..];;
val it : (int * int) [] = [|(3, 9); (4, 16); (5, 25)|]
```

Slicing syntax is used extensively in the example "Verifying Circuits with Propositional Logic" in Chapter 12. You can also use slicing syntax with strings and several other F# types, such as vectors and matrices, and the operator can be overloaded to work with your own type definitions. The F# library definitions of vectors and matrices can be used as a guide.

■ **Note** Slices on arrays generate fresh arrays. Sometimes it's more efficient to use other techniques, such as accessing the array via an accessor function or object that performs one or more internal index adjustments before looking up the underlying array. If you add support for the slicing operators to your own types, you can choose whether they return copies of data structures or an accessor object.

Two-Dimensional Arrays

Like other .NET languages, F# directly supports two-dimensional array values that are stored flat—that is, where an array of dimensions (N, M) is stored using a contiguous array of N * M elements. The types for these values are written using `[,]`, such as in `int[,]` and `float[,]`, and these types also support slicing syntax. Values of these types are created and manipulated using the values in the `Array2D` module. Likewise, there is a module for manipulating three-dimensional array values whose types are written `int[,,]`. You can also use the code in those modules as a template for defining code to manipulate arrays of higher dimension.

Introducing the Imperative .NET Collections

The .NET Framework comes equipped with an excellent set of imperative collections under the namespace `System.Collections.Generic`. You've seen some of these already. The following sections look at some simple uses of these collections.

Using Resizeable Arrays

As mentioned in Chapter 3, the .NET Framework comes with a type `System.Collections.Generic.List<'T>`, which, although named `List`, is better described as a resizeable array, and is like `std::vector<T>` in C++. The F# library includes the following type abbreviation for this purpose:

```
type ResizeArray<'T> = System.Collections.Generic.List<'T>
```

Here is a simple example of using this data structure:

```
> let names = new ResizeArray<string>();;

val names : ResizeArray<string>

> for name in ["Claire"; "Sophie"; "Jane"] do
      names.Add(name);;

val it : unit = ()

> names.Count;;

val it : int = 3

> names.[0];;

val it : string = "Claire"

> names.[1];;

val it : string = "Sophie"

> names.[2];;

val it : string = "Jane"
```

Resizable arrays use an underlying array for storage and support constant-time random-access lookup. In many situations, this makes a resizable array more efficient than an F# list, which supports efficient access only from the head (left) of the list. You can find the full set of members supported by this

type in the .NET documentation. Commonly used properties and members include `Add`, `Count`, `ConvertAll`, `Insert`, `BinarySearch`, and `ToArray`. A module `ResizeArray` is included in the F# library; it provides operations over this type in the style of the other F# collections.

Like other .NET collections, values of type `ResizeArray<'T>` support the `seq<'T>` interface. There is also an overload of the `new` constructor for this collection type that lets you specify initial values via a `seq<'T>`. This means you can create and consume instances of this collection type using sequence expressions:

```
> let squares = new ResizeArray<int>(seq {for i in 0 .. 100 -> i * i});;

val squares : ResizeArray<int>

> for x in squares do

      printfn "square: %d" x;;

square: 0
square: 1
square: 4
square: 9
...
square: 9801
square: 10000
```

Using Dictionaries

The type `System.Collections.Generic.Dictionary<'Key,'Value>` is an efficient hash-table structure that is excellent for storing associations between keys and values. Using this collection from F# code requires a little care, because it must be able to correctly hash the key type. For simple key types such as integers, strings, and tuples, the default hashing behavior is adequate. Here is a simple example:

```
> open System.Collections.Generic;;
> let capitals = new Dictionary<string, string>(HashIdentity.Structural);;

val capitals : Dictionary<string,string> = dict []

> capitals.["USA"] <- "Washington";;
> capitals.["Bangladesh"] <- "Dhaka";;
> capitals.ContainsKey("USA");;

val it : bool = true

> capitals.ContainsKey("Australia");;

val it : bool = false

> capitals.Keys;;

val it : Dictionary'2.KeyCollection<string,string> = seq ["USA"; "Bangladesh"]

> capitals.["USA"];;

val it : string = "Washington"
```

Dictionaries are compatible with the type `seq<KeyValuePair<'key,'value>>`, where `KeyValuePair` is a type from the `System.Collections.Generic` namespace and simply supports the properties `Key` and `Value`. Armed with this knowledge, you can use iteration to perform an operation for each element of the collection:

```
> for kvp in capitals do
      printf "%s has capital %s\n" kvp.Key kvp.Value;;

USA has capital Washington
Bangladesh has capital Dhaka
```

Using Dictionary's TryGetValue

The `Dictionary` method `TryGetValue` is of special interest, because its use from F# is a little nonstandard. This method takes an input value of type `'Key` and looks it up in the table. It returns a `bool` indicating whether the lookup succeeded: `true` if the given key is in the dictionary and `false` otherwise. The value itself is returned via a .NET idiom called an *out parameter*. From F# code, three ways of using .NET methods rely on out parameters:

- You may use a local mutable in combination with the address-of operator `&`.
- You may use a reference cell.
- You may simply not give a parameter, and the result is returned as part of a tuple.

Here's how you do it using a mutable local:

```
open System.Collections.Generic

let lookupName nm (dict : Dictionary<string, string>) =
    let mutable res = ""
    let foundIt = dict.TryGetValue(nm, &res)
    if foundIt then res
    else failwithf "Didn't find %s" nm
```

The use of a reference cell can be cleaner. For example:

```
> let res = ref "";;

val res : string ref = {contents = "";}

> capitals.TryGetValue("Australia", res);;

val it : bool = false

> capitals.TryGetValue("USA", res);;

val it : bool = true

> res;;

val it : string ref = {contents = "Washington"}
```

Finally, with this technique you don't pass the final parameter, and instead the result is returned as part of a tuple:

```
> capitals.TryGetValue("Australia");;

val it : bool * string = (false, null)

> capitals.TryGetValue("USA");;

val it : bool * string = (true, "Washington")
```

Note that the value returned in the second element of the tuple may be `null` if the lookup fails when this technique is used; `null` values are discussed in the section "Working with null Values" in Chapter 6.

Using Dictionaries with Compound Keys

You can use dictionaries with compound keys, such as tuple keys of type `(int * int)`. If necessary, you can specify the hash function used for these values when creating the instance of the dictionary. The default is to use generic hashing, also called *structural hashing*, a topic covered in more detail in Chapter 9. To indicate this explicitly, specify `Microsoft.FSharp.Collections.HashIdentity.Structural` when creating the collection instance. In some cases, this can also lead to performance improvements, because the F# compiler often generates a hashing function appropriate for the compound type.

This example uses a dictionary with a compound key type to represent sparse maps:

```
> open System.Collections.Generic;;
> open Microsoft.FSharp.Collections;;
> let sparseMap = new Dictionary<(int * int), float>();;

val sparseMap : Dictionary<(int * int),float> = dict []

> sparseMap.[(0,2)] <- 4.0;;
> sparseMap.[(1021,1847)] <- 9.0;;
> sparseMap.Keys;;

val it : Dictionary'2.KeyCollection<(int * int),float> =
  seq [(0, 2); (1021, 1847)]
```

Some Other Mutable Data Structures

Some other important mutable data structures in the F# and .NET libraries are:

- `System.Collections.Generic.SortedList<'Key,'Value>`: A collection of sorted values. Searches are done by a binary search. The underlying data structure is a single array.
- `System.Collections.Generic.SortedDictionary<'Key,'Value>`: A collection of key/value pairs sorted by the key, rather than hashed. Searches are done by a binary search The underlying data structure is a single array.
- `System.Collections.Generic.Stack<'T>`: A variable-sized last-in/first-out (LIFO) collection.

- `System.Collections.Generic.Queue<'T>`: A variable-sized first-in/first-out (FIFO) collection.
- `System.Text.StringBuilder`: A mutable structure for building `string` values.
- `Microsoft.FSharp.Collections.HashSet<'Key>`: A hash table structure holding only keys and no values. From .NET 3.5, a `HashSet<'T>` type is available in the `System.Collections.Generic` namespace.

Exceptions and Controlling Them

When evaluating an expression encounters a problem, it may respond in several ways: by recovering internally, emitting a warning, returning a marker value or incomplete result, or throwing an exception. The following code indicates how an exception can be thrown by some of the code you've been using:

```
> let req = System.Net.WebRequest.Create("not a URL");;

System.UriFormatException: Invalid URI: The format of the URI could not be
determined.
```

Similarly, the `GetResponse` method also used in the `http` function may raise a `System.Net.WebException` exception. The exceptions that may be raised by routines are typically recorded in the documentation for those routines. Exception values may also be raised explicitly by F# code:

```
> (raise (System.InvalidOperationException("not today thank you")) : unit);;

System.InvalidOperationException: not today thank you
```

In F#, exceptions are commonly raised using the F# `failwith` function:

```
> if false then 3 else failwith "hit the wall";;

System.Exception: hit the wall
```

Types of some common functions used to raise exceptions are:

```
val failwith : string -> 'T
val raise : System.Exception -> 'T
val failwithf : Printf.StringFormat<'T,'U> -> 'T
val invalidArg : string -> string -> 'T
```

Note that the return types of all these are generic type variables: the functions never return normally and instead return by raising an exception. This means they can be used to form an expression of any particular type and can be handy when you're drafting your code. For example, in the following example, we've left part of the program incomplete:

```
if (System.DateTime.Now > failwith "not yet decided") then
    printfn "you've run out of time!"
```

Table 4-2 shows some common exceptions raised by `failwith` and other operations.

Table 4-2. *Common Categories of Exceptions and F# Functions That Raise Them*

Exception Type	F# Abbreviation	Description	Example
`Exception`	`Failure`	General failure	`failwith "fail"`
`ArgumentException`	`InvalidArgument`	Bad input	`invalidArg "x" "y"`
`DivideByZeroException`		Integer divide by 0	`1 / 0`
`NullReferenceException`		Unexpected `null`	`(null : string).Length`

Catching Exceptions

You can catch exceptions using the `try ... with ...` language construct and `:?` type-test patterns, which filter any exception value caught by the `with` clause. For example:

```
try
   raise (System.InvalidOperationException ("it's just not my day"))
with
   :? System.InvalidOperationException -> printfn "caught!"
```

Giving:

```
caught!
```

Chapter 5 covers these patterns more closely. The following code sample shows how to use `try ... with ...` to catch two kinds of exceptions that may arise from the operations that make up the `http` method, in both cases returning the empty string `""` as the incomplete result. Note that `try ... with ...` is just an expression, and it may return a result in both branches:

```
open System.IO

let http (url : string) =
    try
        let req = System.Net.WebRequest.Create(url)
        let resp = req.GetResponse()
        let stream = resp.GetResponseStream()
        let reader = new StreamReader(stream)
        let html = reader.ReadToEnd()
        html
    with
        | :? System.UriFormatException -> ""
        | :? System.Net.WebException -> ""
```

When an exception is thrown, a value is created that records information about the exception. This value is matched against the earlier type-test patterns. It may also be bound directly and manipulated in the `with` clause of the `try ... with` constructs. For example, all exception values support the `Message` property:

```
try
    raise (new System.InvalidOperationException ("invalid operation"))
with
    err -> printfn "oops, msg = '%s'" err.Message
```

Giving:

```
oops, msg = 'invalid operation'
```

Using try . . . finally

Exceptions may also be processed using the `try ... finally ...` construct. This guarantees to run the `finally` clause both when an exception is thrown and when the expression evaluates normally. This allows you to ensure that resources are disposed after the completion of an operation. For example, you can ensure that the web response from the previous example is closed as follows:

```
let httpViaTryFinally(url : string) =
    let req = System.Net.WebRequest.Create(url)
    let resp = req.GetResponse()
    try
        let stream = resp.GetResponseStream()
        let reader = new StreamReader(stream)
        let html = reader.ReadToEnd()
        html
    finally
        resp.Close()
```

In practice, you can use a shorter form to close and dispose of resources simply by using a `use` binding instead of a `let` binding if the resource implements IDisposable, a technique covered in Chapter 6. This closes the response at the end of the scope of the `resp` variable. Here is how the previous function looks using this form:

```
let httpViaUseBinding(url: string) =
    let req = System.Net.WebRequest.Create(url)
    use resp = req.GetResponse()
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    let html = reader.ReadToEnd()
    html
```

Defining New Exception Types

F# lets you define new kinds of exception objects that carry data in a conveniently accessible form. For example, here is a declaration of a new class of exceptions and a function that wraps `http` with a filter that catches particular cases:

```
exception BlockedURL of string

let http2 url =
    if url = "http://www.kaos.org"
    then raise(BlockedURL(url))
    else http url
```

You can extract the information from F# exception values, again using pattern matching:

```
  try
      raise(BlockedURL("http://www.kaos.org"))
```

```
  with
      BlockedURL(url) -> printfn "blocked! url = '%s'" url
```

Giving:

```
blocked! url = 'http://www.kaos.org'
```

Exception values are always subtypes of the F# type `exn`, an abbreviation for the .NET type `System.Exception`. The declaration `exception BlockedURL of string` is shorthand for defining a new F# class type `BlockedURLException`, which is a subtype of `System.Exception`. Exception types can also be defined explicitly by defining new object types. Chapters 5 and 6 look more closely at object types and subtyping.

Table 4-3 summarizes the exception-related language and library constructs.

Table 4-3. *Exception-Related Language and Library Constructs*

Example Code	Kind	Notes
`raise expr`	F# library function	Raises the given exception
`failwith expr`	F# library function	Raises an `Exception` exception
`try expr with rules`	F# expression	Catches expressions matching the pattern rules
`try expr finally expr`	F# expression	Executes the `finally` expression both when the computation is successful and when an exception is raised
`| :? ArgumentException ->`	F# pattern rule	A rule matching the given .NET exception type
`| :? ArgumentException as e ->`	F# pattern rule	A rule matching the given .NET exception type and naming it as its stronger type
`| Failure(msg) -> expr`	F# pattern rule	A rule matching the given data-carrying F# exception
`| exn -> expr`	F# pattern rule	A rule matching any exception, binding the name `exn` to the exception object value
`| exn when expr -> expr`	F# pattern rule	A rule matching the exception under the given condition, binding the name `exn` to the exception object value

Having an Effect: Basic I/O

Imperative programming and input/output are closely related topics. The following sections show some very simple I/O techniques using F# and .NET libraries.

The .NET types `System.IO.File` and `System.IO.Directory` contain a number of simple functions to make working with files easy. For example, here's a way to output lines of text to a file:

```
> open System.IO;;
> File.WriteAllLines("test.txt", [|"This is a test file.";
                                   "It is easy to read."|]);;
```

Many simple file-processing tasks require reading all the lines of a file. You can do this by reading all the lines in one action as an array using `System.IO.File.ReadAllLines`:

```
> open System.IO;;
> File.ReadAllLines "test.txt";;

val it : string [] = [|"This is a test file."; "It is easy to read."|]
```

If necessary, the entire file can be read as a single string using `System.IO.File.ReadAllText`:

```
> File.ReadAllText "test.txt";;

val it : string = "This is a test file.
It is easy to read.
```

You can also use the method `System.IO.File.ReadLines`, which reads lines on-demand, as a sequence. This is often used as the input to a sequence expression or to a pipeline on sequence operators. For example, this;

```
seq { for line in File.ReadLines("test.txt") do
          let words = line.Split [|' '|]
          if words.Length > 3 && words.[2] = "easy" then
              yield line}
```

give results:

```
val it : seq<string> = seq ["It is easy to read."]
```

.NET I/O via Streams

The .NET namespace `System.IO` contains the primary .NET types for reading/writing bytes and text from/to data sources. The primary output constructs in this namespace are:

- `System.IO.BinaryWriter`: Writes primitive data types as binary values. Create using `new BinaryWriter(stream)`. You can create output streams using `File.Create(filename)`.
- `System.IO.StreamWriter`: Writes textual strings and characters to a stream. The text is encoded according to a particular Unicode encoding. Create by using `new StreamWriter(stream)` and its variants or by using `File.CreateText(filename)`.
- `System.IO.StringWriter`: Writes textual strings to a `StringBuilder`, which eventually can be used to generate a string.

Here is a simple example of using `System.IO.File.CreateText` to create a `StreamWriter` and write two strings:

```
> let outp = File.CreateText "playlist.txt";;

val outp : StreamWriter

> outp.WriteLine "Enchanted";;
```

```
> outp.WriteLine "Put your records on";;
> outp.Close();
```

The primary input constructs in the `System.IO` namespace are:

- `System.IO.BinaryReader`: Reads primitive data types as binary values. When reading the binary data as a string, it interprets the bytes according to a particular Unicode encoding. Create using `new BinaryReader(stream)`.
- `System.IO.StreamReader`: Reads a stream as textual strings and characters. The bytes are decoded to strings according to a particular Unicode encoding. Create by using `new StreamReader(stream)` and its variants or by using `File.OpenText(filename)`.
- `System.IO.StringReader`: Reads a string as textual strings and characters.

Here is a simple example of using `System.IO.File.OpenText` to create a `StreamReader` and read two strings:

```
> let inp = File.OpenText("playlist.txt");;
val inp : StreamReader
> inp.ReadLine();;
val it : string = "Enchanted"
> inp.ReadLine();;
val it : string = "Put your records on"
> inp.Close();;
```

■ **Tip** In non-scripting production code, whenever you create objects such as a `StreamReader` that have a `Close` or `Dispose` operation or that implement the `IDisposable` interface, consider how to eventually close or otherwise dispose of the resource. We discuss this in Chapter 6.

Some Other I/O-Related Types

The `System.IO` namespace contains a number of other types, all of which are useful for corner cases of advanced I/O but that you won't need to use from day to day. For example, these abstractions appear in the .NET documentation:

- `System.IO.TextReader`: Reads textual strings and characters from an unspecified source. This is the common functionality implemented by the `StreamReader` and `StringReader` types and the `System.Console.In` object. The latter is used to access the `stdin` input.
- `System.IO.TextWriter`: Writes textual strings and characters to an unspecified output. This is the common functionality implemented by the `StreamWriter` and

`StringWriter` types and the `System.Console.Out` and `System.Console.Error` objects. The latter are used to access the `stdout` and `stderr` output streams.

- `System.IO.Stream`: Provides a generic view of a sequence of bytes.

Some functions that are generic over different kinds of output streams make use of these; for example, the formatting function `twprintf` discussed in "Using `printf` and Friends" writes to any `System.IO.TextWriter`.

Using System.Console

Some simple input/output routines are provided in the `System.Console` class. For example:

```
> System.Console.WriteLine "Hello World";;

Hello World

> System.Console.ReadLine();;

<enter "I'm still here" here>

val it : string = "I'm still here"
```

The `System.Console.Out` object can also be used as a `TextWriter`.

Combining Functional and Imperative: Efficient Precomputation and Caching

All experienced programmers are familiar with the concept of *precomputation*, by which computations are performed as soon as some of the inputs to a function are known. The following sections cover a number of manifestations of precomputation in F# programming and the related topics of memoization and caching. These represent one common pattern in which imperative programming is used safely and non-intrusively within a broader setting of functional programming.

Precomputation and Partial Application

Let's say you're given a large input list of words and you want to compute a function that checks whether a word is in this list. Do this:

```
let isWord (words : string list) =
    let wordTable = Set.ofList words
    fun w -> wordTable.Contains(w)
```

Here, `isWord` has the following type:

```
val isWord : words:string list -> (string -> bool)
```

The efficient use of this function depends crucially on the fact that useful intermediary results are computed after only one argument is applied and a function value is returned. For example:

```
> let isCapital = isWord ["London"; "Paris"; "Warsaw"; "Tokyo"];;

val isCapital : (string -> bool)

> isCapital "Paris";;

val it : bool = true

> isCapital "Manchester";;

val it : bool = false
```

Here, the internal table `wordTable` is computed as soon as `isCapital` is applied to one argument. It would be a mistake to write `isCapital` as:

```
let isCapitalSlow inp = isWord ["London"; "Paris"; "Warsaw"; "Tokyo"] inp
```

This function computes the same results as `isCapital`. It does so inefficiently, however, because `isWord` is applied to both its first argument and its second argument every time you use the function `isCapitalSlow`. This means the internal table is rebuilt every time the function `isCapitalSlow` is applied, somewhat defeating the point of having an internal table in the first place. In a similar vein, the definition of `isCapital` shown previously is more efficient than either `isCapitalSlow2` or `isCapitalSlow3` in the following:

```
let isWordSlow2 (words : string list) (word : string) =
    List.exists (fun word2 -> word = word2) words

let isCapitalSlow2 word = isWordSlow2 ["London"; "Paris"; "Warsaw"; "Tokyo"] word

let isWordSlow3 (words : string list) (word : string) =
    let wordTable = Set<string>(words)
    wordTable.Contains(word)

let isCapitalSlow3 word = isWordSlow3 ["London"; "Paris"; "Warsaw"; "Tokyo"] word
```

The first uses an inappropriate data structure for the lookup (an F# list, which has O(n) lookup time), and the second attempts to build a better intermediate data structure (an F# set, which has O(log n) lookup time) but does so on every invocation.

There are often trade-offs among different intermediate data structures or whether to use them at all. For example, in the previous example, you could just as well use a `HashSet` as the internal data structure. This approach, in general, gives better lookup times (constant time), but it requires slightly more care to use, because a `HashSet` is a mutable data structure. In this case, you don't mutate the data structure after you create it, and you don't reveal it to the outside world, so it's entirely safe:

```
let isWord (words : string list) =
    let wordTable = HashSet<string>(words)
    fun word -> wordTable.Contains word
```

Precomputation and Objects

The examples of precomputation given previously are variations on the theme of computing functions, introduced in Chapter 3. The functions computed capture the precomputed intermediate data structures. It's clear, however, that precomputing via partial applications and functions can be subtle, because it

matters when you apply the first argument of a function (triggering the construction of intermediate data structures) and when you apply the subsequent arguments (triggering the real computation that uses the intermediate data structures).

Luckily, functions don't just have to compute functions; they can also return more sophisticated values, such as objects. This can help make it clear when precomputation is being performed. It also allows you to build richer services based on precomputed results. For example, Listing 4-1 shows how to use precomputation as part of building a name-lookup service. The returned object includes both a `Contains` method and a `ClosestPrefixMatch` method.

Listing 4-1. *Precomputing a Word Table Before Creating an Object*

```
open System

type NameLookupService =
    abstract Contains : string -> bool

let buildSimpleNameLookup (words : string list) =
    let wordTable = HashSet<_>(words)
    {new NameLookupService with
         member t.Contains w = wordTable.Contains w}
```

The internal data structure used in Listing 4-1 is the same as before: an F# set of type `Microsoft.FSharp.Collections.Set<string>`. The service can now be instantiated and used as follows:

```
> let capitalLookup = buildSimpleNameLookup ["London"; "Paris"; "Warsaw"; "Tokyo"];;

val capitalLookup : NameLookupService

> capitalLookup.Contains "Paris";;

val it : bool = true
```

In passing, note the following about this implementation:

- You can extend the returned service to support a richer set of queries of the underlying information by adding further methods to the object returned.

Precomputation of the kind used previously is an essential technique for implementing many services and abstractions, from simple functions to sophisticated computation engines. You see further examples of these techniques in Chapter 9.

Memoizing Computations

Precomputation is one important way to amortize the costs of computation in F#. Another is called *memoization*. A memoizing function avoids recomputing its results by keeping an internal table, often called a *lookaside table*. For example, consider the well-known Fibonacci function, whose naive, unmemoized version is:

```
let rec fib n = if n <= 2 then 1 else fib (n - 1) + fib (n - 2)
```

Not surprisingly, a version keeping a lookaside table is much faster:

```
let fibFast =
    let t = new System.Collections.Generic.Dictionary<int, int>()
```

```
    let rec fibCached n =
        if t.ContainsKey n then t.[n]
        elif n <= 2 then 1
        else let res = fibCached (n - 1) + fibCached (n - 2)
             t.Add (n, res)
             res
    fun n -> fibCached n

// From Chapter 2, but modified to use stop watch.
let time f =
    let sw = System.Diagnostics.Stopwatch.StartNew()
    let res = f()
    let finish = sw.Stop()
    (res, sw.Elapsed.TotalMilliseconds |> sprintf "%f ms")

time(fun () -> fibFast 30)
time(fun () -> fibFast 30)
time(fun () -> fibFast 30)
```

```
val fibFast : (int -> int)
val time : f:(unit -> 'a) -> 'a * string
val it : int * string = (832040, "0.727200 ms")
val it : int * string = (832040, "0.066100 ms")
val it : int * string = (832040, "0.077400 ms")
```

On one of our laptops, with n = 30, there's an order of magnitude speed up from the first to second run. Listing 4-2 shows how to write a generic function that encapsulates the memoization technique.

Listing 4-2. *A Generic Memoization Function*

```
open System.Collections.Generic

let memoize (f : 'T -> 'U) =
    let t = new Dictionary<'T, 'U>(HashIdentity.Structural)
    fun n ->
        if t.ContainsKey n then t.[n]
        else let res = f n
             t.Add (n, res)
             res

let rec fibFast =
    memoize (fun n -> if n <= 2 then 1 else fibFast (n - 1) + fibFast (n - 2))
```

Here, the functions have the types:

```
val memoize : f:('T -> 'U) -> ('T -> 'U) when 'T : equality
val fibFast : (int -> int)
```

In the definition of fibFast, you use `let rec` because fibFast is self-referential—that is, used as part of its own definition. You can think of fibFast as a *computed, recursive* function. Such a function generates an

informational warning when used in F# code, because it's important to understand when this feature of F# is being used; you then suppress the warning with `#nowarn "40"`. As with the examples of computed functions from the previous section, omit the extra argument from the application of `memoize`, because including it would lead to a fresh memoization table being allocated each time the function `fibNotFast` was called:

```
let rec fibNotFast n =
    memoize (fun n -> if n <= 2 then 1 else fibNotFast (n - 1) + fibNotFast (n - 2)) n
```

Due to this subtlety, it's often a good idea to define your memoization strategies to generate objects other than functions (think of functions as very simple kinds of objects). For example, Listing 4-3 shows how to define a new variation on `memoize` that returns a `Table` object that supports both a lookup and a `Discard` method.

Listing 4-3. *A Generic Memoization Service*

```
open System.Collections.Generic

type Table<'T, 'U> =
    abstract Item : 'T -> 'U with get
    abstract Discard : unit -> unit

let memoizeAndPermitDiscard f =
    let lookasideTable = new Dictionary<_, _>(HashIdentity.Structural)
    {new Table<'T, 'U> with
          member t.Item
             with get(n) =
                 if lookasideTable.ContainsKey(n)
                 then lookasideTable.[n]
                 else let res = f n
                      lookasideTable.Add(n, res)
                      res

          member t.Discard() =
              lookasideTable.Clear()}


#nowarn "40" // do not warn on recursive computed objects and functions

let rec fibFast =
    memoizeAndPermitDiscard
        (fun n ->
            printfn "computing fibFast %d" n
            if n <= 2 then 1 else fibFast.[n - 1] + fibFast.[n - 2])
```

In Listing 4-3, lookup uses the `a.[b]` associative `Item` lookup property syntax, and the `Discard` method discards any internal partial results. The functions have these types:

```
val memoizeAndPermitDiscard : ('T -> 'U) -> Table<'T, 'U> when 'T : equality
val fibFast : Table<int,int>
```

This example shows how fibFast caches results but recomputes them after a Discard:

```
> fibFast.[3];;
computing fibFast 3
computing fibFast 2
computing fibFast 1
val it : int = 2

> fibFast.[5];;
computing fibFast 5
computing fibFast 4
val it : int = 5

> fibFast.Discard();;
> fibFast.[5];;
computing fibFast 5
computing fibFast 4
computing fibFast 3
computing fibFast 2
computing fibFast 1
val it : int = 5
```

■ **Note** Memoization relies on the memoized function being stable and idempotent. In other words, it always returns the same results, and no additional interesting side effects are caused by further invocations of the function. In addition, memoization strategies rely on mutable internal tables. The implementation of memoize shown in this chapter isn't thread safe, because it doesn't lock this table during reading or writing. This is fine if the computed function is used only from at most one thread at a time, but in a multithreaded application, use memoization strategies that use internal tables protected by locks, such as a .NET ReaderWriterLock. Chapter 11 further discusses thread synchronization and mutable state.

Lazy Values

Memoization is a form of *caching*. Another important variation on caching is a simple *lazy* value, a delayed computation of type Microsoft.FSharp.Control.Lazy<'T> for some type 'T. Lazy values are usually formed by using the special keyword lazy (you can also make them explicitly using the functions in the Microsoft.FSharp.Core.Lazy module). For example:

```
> let sixty = lazy (30 + 30);;

val sixty : Lazy<int> = Value is not created.

> sixty.Force();;

val it : int = 60
```

Lazy values of this kind are implemented as thunks holding either a function value that computes the result or the actual computed result. The lazy value is computed only once, and thus its effects are executed only once. For example, in the following code fragment, "Hello world" is printed only once:

```
> let sixtyWithSideEffect = lazy (printfn "Hello world"; 30 + 30);;

val sixtyWithSideEffect: Lazy<int> = Value is not created.

> sixtyWithSideEffect.Force();;

Hello world

val it : int = 60

> sixtyWithSideEffect.Force();;

val it : int = 60
```

Lazy values are implemented by a simple data structure containing a mutable reference cell. The definition of this data structure is in the F# library source code.

Other Variations on Caching and Memoization

You can apply many different caching and memoization techniques in advanced programming, and this chapter can't cover them all. Some common variations are:

- Using an internal data structure that records only the last invocation of a function and basing the lookup on a very cheap test on the input.
- Using an internal data structure that contains both a fixed-size queue of input keys and a dictionary of results. Entries are added to both the table and the queue as they're computed. When the queue is full, the input keys for the oldest computed results are dequeued, and the computed results are discarded from the dictionary.

Combining Functional and Imperative: Functional Programming with Side Effects

F# stems from a tradition in programming languages in which state is made explicit, largely by passing extra parameters. Many F# programmers use functional-programming techniques first before turning to their imperative alternatives, and we encourage you to do the same, for all the reasons listed at the start of this chapter.

F# also integrates imperative and functional programming together in a powerful way. F# is actually an extremely succinct imperative-programming language! Furthermore, in some cases, no good functional techniques exist to solve a problem, or those that do are too experimental for production use. This means that in practice, using imperative constructs and libraries is common in F#. For example, many examples you saw in Chapters 2 and 3 used side effects to report their results or to create GUI components.

Regardless, we still encourage you to think functionally, even about your imperative programming. In particular, it's always helpful to be aware of the potential side effects of your overall program and the characteristics of those side effects. The following sections describe five ways to help tame and reduce the use of side effects in your programs.

Consider Replacing Mutable Locals and Loops with Recursion

When imperative programmers begin to use F#, they frequently use mutable local variables or reference cells heavily as they translate code fragments from their favorite imperative language into F#. The resulting code often looks very bad. Over time, programmers learn to avoid many uses of mutable locals. For example, consider this (naive) implementation of factorization, transliterated from C code:

```
let factorizeImperative n =
    let mutable factor1 = 1
    let mutable factor2 = n
    let mutable i = 2
    let mutable fin = false
    while (i < n && not fin) do
        if (n % i = 0) then
            factor1 <- i
            factor2 <- n / i
            fin <- true
        i <- i + 1

    if (primefactor1 = 1) then None
    else Some (primefactor1, primefactor2)
```

This code can be replaced by use of an inner recursive function:

```
let factorizeRecursive n =
    let rec find i =
        if i >= n then None
        elif (n % i = 0) then Some(i, n / i)
        else find (i + 1)
    find 2
```

The second code is not only shorter but also uses no mutation, which makes it easier to reuse and maintain. You can also see that the loop terminates (`i` is increasing toward `n`) and see the two exit conditions for the function (`i >= n` and `n % i = 0`). Note that the state `i` has become an explicit parameter.

Separating Pure Computation from Side-Effecting Computations

Where possible, separate out as much of your computation as possible using side-effect-free functional programming. For example, sprinkling `printf` expressions throughout your code may make for a good debugging technique but, if not used wisely, it can lead to code that is difficult to understand and inherently imperative.

Separating Mutable Data Structures

A common technique of object-oriented programming is to ensure that mutable data structures are private, and where possible, fully separated, which means there is no way that distinct pieces of code can access one another's internal state in undesirable ways. Fully separated state can even be used inside the implementation of what, to the outside world, appears to be a purely functional piece of code.

For example, where necessary, you can use side effects on private data structures allocated at the start of an algorithm and then discard these data structures before returning a result; the overall result is then effectively a side-effect-free function. One example of separation from the F# library is the library's

implementation of `List.map`, which uses mutation internally; the writes occur on an internal, separated data structure that no other code can access. Thus, as far as callers are concerned, `List.map` is pure and functional.

This second example divides a sequence of inputs into equivalence classes (the F# library function `Seq.groupBy` does a similar thing):

```
open System.Collections.Generic

let divideIntoEquivalenceClasses keyf seq =

    // The dictionary to hold the equivalence classes
    let dict = new Dictionary<'key, ResizeArray<'T>>()

    // Build the groupings
    seq |> Seq.iter (fun v ->
        let key = keyf v
        let ok, prev = dict.TryGetValue(key)
        if ok then prev.Add(v)
        else let prev = new ResizeArray<'T>()
             dict.[key] <- prev
             prev.Add(v))

    // Return the sequence-of-sequences. Don't reveal the
    // internal collections: just reveal them as sequences
    dict |> Seq.map (fun group -> group.Key, Seq.readonly group.Value)
```

This uses the `Dictionary` and `ResizeArray` mutable data structures internally, but these mutable data structures aren't revealed externally. The inferred type of the overall function is:

```
val divideIntoEquivalenceClasses :
  keyf:('T -> 'key) -> seq:seq<'T> -> seq<'key * seq<'T>> when 'key : equality
```

Here is an example use:

```
> divideIntoEquivalenceClasses (fun n -> n % 3) [0 .. 10];;

val it : seq<int * seq<int>>
= seq [(0, seq [0; 3; 6; 9]); (1, seq [1; 4; 7; 10]); (2, seq [2; 5; 8])]
```

Not All Side Effects Are Equal

It's often helpful to use the weakest set of side effects necessary to achieve your programming task and at least be aware when you're using strong side effects:

- Weak side effects are effectively benign, given the assumptions you're making about your application. For example, writing to a log file is very useful and essentially benign (if the log file can't grow arbitrarily large and crash your machine!). Similarly, reading data from a stable, unchanging file store on a local disk is effectively treating

the disk as an extension of read-only memory, so reading these files is a weak form of side effect that isn't difficult to incorporate into your programs.

- Strong side effects have a much more corrosive effect on the correctness and operational properties of your program. For example, blocking network I/O is a relatively strong side effect by any measure. Performing blocking network I/O in the middle of a library routine can destroy the responsiveness of a GUI application, at least if the routine is invoked by the GUI thread of an application. Any constructs that perform synchronization between threads are also a major source of strong side effects.

Whether a particular side effect is stronger or weaker depends on your application and whether the consequences of the side effect are sufficiently isolated and separated from other entities. Strong side effects can and should be used freely in the outer shell of an application or when you're scripting with F# Interactive; otherwise, not much can be achieved.

When you're writing larger pieces of code, write your application and libraries in such a way that most of your code either doesn't use strong side effects or at least makes it obvious when these side effects are being used. Threads and concurrency are commonly used to mediate problems associated with strong side effects. Chapter 11 covers these issues in more depth.

Avoid Combining Imperative Programming and Laziness

It's generally thought to be bad style to combine delayed computations (that is, laziness) and side effects. This isn't entirely true; for example, it's reasonable to set up a read from a file system as a lazy computation using sequences. It's relatively easy to make mistakes in this sort of programming, however. For example, consider:

```
open System.IO
let reader1, reader2 =
    let reader = new StreamReader(File.OpenRead("test.txt"))
    let firstReader() = reader.ReadLine()
    let secondReader() = reader.ReadLine()

    // Note: we close the stream reader here!
    // But we are returning function values which use the reader
    // This is very bad!
    reader.Close()
    firstReader, secondReader

// Note: stream reader is now closed! The next line will fail!
let firstLine = reader1()
let secondLine = reader2()
firstLine, secondLine
```

This code is wrong, because the `StreamReader` object `reader` is used after the point indicated by the comment. The returned function values are then called, and they try to read from the captured variable `reader`. Function values are just one example of delayed computations; others are lazy values, sequences, and any objects that perform computations on demand. Avoid building delayed objects such as `reader` that represent handles to transient, disposable resources, unless those objects are used in a way that respects the lifetime of that resource.

The previous code can be corrected to avoid using laziness in combination with a transient resource:

```
open System.IO

let line1, line2 =
    let reader = new StreamReader(File.OpenRead("test.txt"))
    let firstLine = reader.ReadLine()
    let secondLine = reader.ReadLine()
    reader.Close()
    firstLine, secondLine
```

Another technique uses language and/or library constructs that tie the lifetime of an object to some larger object. For example, you can use a `use` binding within a sequence expression, which augments the sequence object with the code needed to clean up the resource when iteration is finished or terminates. This technique, discussed further in Chapter 9, is shown by example here (see also the `System.IO.File.ReadLines`, which serves a similar role):

```
let linesOfFile =
    seq {use reader = new StreamReader(File.OpenRead("test.txt"))
         while not reader.EndOfStream do
             yield reader.ReadLine()}
```

The general lesson is to minimize the side effects you use, and to use separated, isolated state where possible. Orchestrating compositional, separated objects using functional programming as the orchestration language is often a highly effective technique. Use both delayed computations (laziness) and imperative programming (side effects) where appropriate, but be careful about using them together.

Summary

In this chapter, you learned how to do imperative programming in F#, from some of the basic mutable data structures, such as reference cells, to working with side effects, such as exceptions and I/O. You also looked at some general principles for avoiding the need for imperative programming and isolating your uses of side effects. The next chapter continues to explore some building-blocks of typed functional programming in F#, with a deeper look at types, type inference, and generics.