CHAPTER 1

INTRODUCTION

For the first time in history, and thanks to the exponential growth rate of computing power, an increasing number of scientists are finding that more time is spent creating, rather than executing, working programs. Indeed, much effort is spent writing small programs to automate otherwise tedious forms of analysis. In the future, this imbalance will doubtless be addressed by the adoption and teaching of more efficient programming techniques. An important step in this direction is the use of higher-level programming languages, such as F#, in place of more conventional languages for scientific programming such as Fortran, C, C++ and even Java and C#.

In this chapter, we shall begin by laying down some guidelines for good programming which are applicable in any language before briefly reviewing the history of the F# language and outlining some of the features of the language which enforce some of these guidelines and other features which allow the remaining guidelines to be met. As we shall see, these aspects of the design of F# greatly improve reliability and development speed. Coupled with the fact that a freely available, efficient compiler already exists for this language, no wonder F# is already being adopted by scientists of all disciplines.

1.1 PROGRAMMING GUIDELINES

Some generic guidelines can be productively adhered to when programming in any language:

Correctness over performance Programs should be written correctly first and optimized last.

Factor programs Complicated or common operations should be factored out into separate functions or objects.

Interfaces Abstract interfaces should be designed and concrete implementations should be coded to these interfaces.

Avoid magic numbers Numeric constants should be defined once and referred back to, rather than explicitly "hard-coding" their value multiple times at different places in a program.

Following these guidelines is the first step towards reusable programs.

1.2 A BRIEF HISTORY OF F#

The first version of ML (Meta Language) was developed at Edinburgh University in the 1970's as a language designed to efficiently represent and manipulate other languages. The original ML language was pioneered by Robin Milner for the *Logic of Computable Functions* (**LCF**) theorem prover. The original ML, and its derivatives, were designed to stretch theoretical computer science to the limit, yielding remarkably robust and concise programming languages without sacrificing the performance of low-level languages.

The *Categorical Abstract Machine Language* (**CAML**) was the acronym originally used to describe what is now known as the Caml family of languages, a dialect of ML that was designed and implemented by Gérard Huet at the *Institut National de Recherche en Informatique et en Automatique* (**INRIA**) in France, until 1994. Since then, development has continued as part of *projet Cristal*, now led by Xavier Leroy. *Objective Caml* (**OCaml**) is the current flagship language of projet Cristal. The OCaml programming language is one of the foremost high-performance and high-level programming languages used by scientists on the Linux and Mac OS X platforms [11].

Don Syme at Microsoft Research Cambridge has meticulously engineered the F# language for .NET, drawing heavily upon the success of the CAML family of languages. The F# language combines the remarkable brevity and robustness of the Caml family of languages with .NET interoperability, facilitating seamless integration of F# programs with any other programs written in .NET languages. Moreover, F# is the first mainstream language to implement some important features such as active patterns and asynchronous programming constructs.

1.3 BENEFITS OF F#

Before delving into the syntax of the language itself, we shall list the main, advantageous features offered by the F# language:

Safety F# programs are thoroughly checked prior to execution such that they are proven to be entirely safe to run, e.g. a compiled F# program cannot cause an access violation.

Functional Functions may be nested, passed as arguments to other functions and stored in data structures as values.

Strongly typed The types of all values are checked during compilation to ensure that they are well defined and validly used.

Statically typed Any typing errors in a program are picked up at compile-time by the compiler, instead of at run-time as in many other languages.

Type inference The types of values are automatically inferred during compilation by the context in which they occur. Therefore, the types of variables and functions in F# code rarely need to be specified explicitly, dramatically reducing source code size. Clarity is regaining by displaying inferred type information in the integrated development environment (**IDE**).

Generics Functions are automatically generalized by the F# compiler, greatly simplifying the writing of reusable functions.

Pattern matching Values, particularly the contents of data structures, can be matched against arbitrarily-complicated patterns in order to determine the appropriate course of action.

Modules and objects Programs can be structured by grouping their data structures and related functions into modules and objects.

Separate compilation Source files can be compiled separately into object files that are then linked together to form an executable or library. When linking, object files are automatically type checked and optimized before the final executable is created.

Interoperability F# programs can call and be called from programs written in other Microsoft .NET languages (e.g. C#), native code libraries and over the internet.

1.4 INTRODUCING F#

F# programs are typically written in Microsoft Visual Studio and can be executed either following a complete build or incrementally from the F# interactive mode. Throughout this book we shall present code snippets in the form seen using the F# interactive mode, with code input following the prompt:

```
>
```

Setup and use of the interactive mode is covered in more detail in chapter 2. Throughout this book, we assume the use of the #light syntax option, which requires the following command to be evaluated before any of the code examples:

```
> #light;;
```

Before we consider the features offered by F#, a brief overview of the syntax of the language is instructive, so that we can provide actual code examples later. Other books give more systematic, thorough and formal introductions to the whole of the F# language [25, 22].

1.4.1 Language overview

In this section we shall evolve the notions of values, types, variables, functions, simple containers (lists and arrays) and program flow control. These notions will then be used to introduce more advanced features in the later sections of this chapter.

When presented with a block of code, even the most seasoned and fluent programmer will not be able to infer the purpose of the code. Consequently, programs should contain additional descriptions written in plain English, known as *comments*. In F#, comments are enclosed between `(*` and `*)` or after `//` or `///` on a single line. Comments appearing after a `///` are known as *autodoc comments* and Visual Studio interprets them as official documentation according to standard .NET coding guidelines.

Comments may be nested, i.e. `(* (* ... *) *)` is a valid comment and comments are treated as whitespace, i.e. `a(* ... *)b` is understood to mean `a b` rather than `ab`.

Just as numbers are the members of sets such as the integers ($\in \mathbb{Z}$), reals ($\in \mathbb{R}$), complexes ($\in \mathbb{C}$) and so on, so *values* in programs are members of sets. These sets are known as *types*.

1.4.1.1 Basic types Fundamentally, languages provide basic types and, often, allow more sophisticated types to be defined in terms of the basic types. F# provides a number of built-in types, such as `unit`, `int`, `float`, `char`, `string` and `bool`. We shall examine these built-in types before discussing the compound *tuple*, *record* and *variant* (also known as *discriminated union*) types.

Only one value is of type `unit` and this value is written `()` and, therefore, conveys no information. This is used to implement functions that require no input or expressions that return no value. For example, a new line can be printed by calling the `print_newline` function:

```
> print_newline ();;

val it : unit = ()
```

This function requires no input, so it accepts a single argument `()` of the type `unit`, and returns the value `()` of type `unit`.

Integers are written `-2`, `-1`, `0`, `1` and `2`. Floating-point numbers are written `-2.0`, `-1.0`, `-0.5`, `0.0`, `0.5`, `1.0` and `2.0`. Note that a zero fractional part may be omitted, so `3.0` may be written `3.`, but we choose the more verbose format for purely esthetic reasons. For example:

```
> 3;;
val it : int = 3
```

```
> 5.0;;
val it : float = 5.0
```

Arithmetic can be performed using the conventional `+`, `-`, `*`, `/` and `%` binary infix[1] operators over many arithmetic types including `int` and `float`.

For example, the following expression is evaluated according to usual mathematical convention regarding operator precedence, with multiplication taking precedence over addition:

```
> 1 * 2 + 2 * 3;;
val it : int = 8
```

The same operators can be used for floating point arithmetic:

```
> 1.0 * 2.0 + 2.0 * 3.0;;
val it : float = 8.0
```

Defining new operators and overloading existing operators is discussed later, in section 2.4.1.3. Conversion functions or *type casts* are used to perform arithmetic with mixed types, e.g. the `float` function converts numeric types to the `float` type.

However, the types of the two arguments to these operators must be the same, so `*` cannot be used to multiply an `int` by a `float`:

```
> 2 * 2.0;;
Error: FS0001: This expression has type float but is
here used with type int
```

Explicitly converting the value of type `float` to a value of type `int` using the built-in function `int` results in a valid expression that the interactive session will execute:

```
> 2 * int 2.0;;
val it : int = 4
```

In most programs, arithmetic is typically performed using a single number representation (e.g. either `int` or `float`) and conversions between representations are, therefore, comparatively rare. Thus, the overhead of having to apply functions to explicitly convert between types is a small price to pay for the added robustness that results from more thorough type checking.

[1]An infix function is a function that appears between its arguments rather than before them. For example, the arguments i and j of the conventional addition operator $+$ appear on either side: $i + j$.

Single characters (of type `char`) are written in single quotes, e.g. `'a'`, that may also be written using a 3-digit decimal code, e.g. `'\097'`.

Strings are written in double quotes, e.g. `"Hello World!"`. Characters in a string of length n may be extracted using the notation `s.[i]` for $i \in \{0 \ldots n-1\}$. For example, the fifth character in this string is "o":

```
> "Hello world!".[4];;
val it : char = 'o'
```

Strings are immutable in F#, i.e. the characters in a string cannot be altered once the string is created. The `char array` and `byte array` types may be used as mutable strings.

A pair of strings may be concatenated using the overloaded + operator:

```
> "Hello " + "world!";;
val it : string = "Hello world!"
```

Booleans are either `true` or `false`. Booleans are created by the usual comparison functions =, <> (not equal to), <, >, <=, >=. These functions are polymorphic, meaning they may be applied to pairs of values of the same type for any type. The usual, short-circuit-evaluated[2] logical comparisons `&&` and `||` are also present. For example, the following expression tests that one is less than three and 2.5 is less than 2.7:

```
> 1 < 3 && 2.5 < 2.7;;
val it : bool = true
```

Values may be assigned, or *bound*, to names. As F# is a functional language, these values may be expressions that map values to values — functions. We shall now examine the binding of values and expressions to variable and function names.

1.4.1.2 Variables and functions Variables and functions are both defined using the `let` construct. For example, the following defines a variable called `a` to have the value 2:

```
> let a = 2;;
val a : int
```

Note that the language automatically infers types. In this case, `a` has been inferred to be of type `int`.

Definitions using `let` can be defined locally using the syntax:

`let` *var* = $expr_1$ `in`
$expr_2$

This evaluates $expr_1$ and binds the result to the variable *var* before evaluating $expr_2$. For example, the following evaluates a^2 in the context $a = 3$, giving 9:

[2]Short-circuit evaluation refers to the premature escaping of a sequence of operations (in this case, boolean comparisons). For example, the expression `false && expr` need not evaluate *expr* as the result of the whole expression is necessarily `false` due to the preceding `false`.

```
> let a = 3 in
  a * a;;
val it : int = 9
```

Note that the value 3 bound to the variable a in this example was local to the expression a * a and, therefore, the global definition of a is still 2:

```
> a;;
val it : int = 2
```

More recent definitions shadow previous definitions. For example, the following supersedes a definition $a = 5$ with $a = a \times a$ in order to calculate $5^4 = 625$:

```
> let a = 5;;
val a : int
> let a = a * a;;
val a : int
> a * a;;
val it : int = 625
```

Note that many of the keywords at the ends of lines (such as the in keyword) may be omitted when using the #light syntax option. This simplifies F# code and makes it easier to read. More importantly, nested lines of code are written in the same style and may be evaluated directly in a running F# interactive session. This is discussed in chapter 2.

As F# is a functional language, values can be functions and variables can be bound to them in exactly the same way as we have just seen. Specifically, function definitions include a list of arguments between the name of the function and the = in the let construct. For example, a function called sqr that accepts an argument n and returns n * n may be defined as:

```
> let sqr n = n * n;;
val sqr : int -> int
```

Type inference for arithmetic operators defaults to int. In this case, the use of the overloaded multiply * results in F# inferring the type of sqr to be int -> int, i.e. the sqr function accepts a value of type int and returns a value of type int.

The function sqr may then be applied to an int as:

```
> sqr 5;;
val it : int = 25
```

In order to write a function to square a float, it is necessary to override this default type inference. This can be done by explicitly annotating the type. Types may be constrained by specifying types in a definition using the syntax (*expr* : *type*). For example, specifying the type of the argument alters the type of the whole function:

```
> let sqr (x : float) = x * x;;
val sqr : float -> float
```

The return type of a function can also be constrained using a similar syntax, having the same result in this case:

```
> let sqr x : float = x * x;;
val sqr : float -> float
```

A variation on the `let` binding called a `use` binding is used to automatically dispose a value at the end of the scope of the `use` binding. This is particularly useful when handling file streams (discussed in chapter 5) because the file is guaranteed to be closed.

Typically, more sophisticated computations require the use of more complicated types. We shall now examine the three simplest ways by which more complicated types may be constructed.

1.4.1.3 Product types: tuples and records Tuples are the simplest form of compound types, containing a fixed number of values which may be of different types. The type of a tuple is written analogously to conventional set-theoretic style, using `*` to denote the cartesian product between the sets of possible values for each type. For example, a tuple of three integers, conventionally denoted by the triple $(i, j, k) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, can be represented by values `(i, j, k)` of the type `int * int * int`. When written, tuple values are comma-separated and often enclosed in parentheses. For example, the following tuple contains three different values of type `int`:

```
> (1, 2, 3);;
val it : int * int * int = (1, 2, 3)
```

At this point, it is instructive to introduce some nomenclature: A tuple containing n values is described as an n-tuple, e.g. the tuple `(1, 2, 3)` is a 3-tuple. The value n is said to be the *arity* of the tuple.

Records are essentially tuples with named components, known as *fields*. Records and, in particular, the names of their fields must be defined using a `type` construct before they can be used. When defined, record fields are written *name* : *type* where *name* is the name of the field (which must start with a lower-case letter) and *type* is the type of values in that field, and are semicolon-separated and enclosed in curly braces. For example, a record containing the x and y components of a 2D vector could be defined as:

```
> type vec2 = { x : float; y : float };;
type vec2 = { x:float; y:float }
```

A value of this type representing the zero vector can then be defined using:

```
> let zero = { x = 0.0; y = 0.0 };;
val zero : vec2
```

Note that the use of a record with fields `x` and `y` allowed F# to infer the type of `zero` as `vec2`.

Whereas the tuples are order-dependent, i.e. $(1, 2) \neq (2, 1)$, the named fields of a record may appear in any order, i.e. $\{x = 1; y = 2\} \equiv \{y = 2; x = 1\}$. Thus, we could, equivalently, have provided the x and y fields in reverse order:

```
> let zero = { y = 0.0; x = 0.0 };;
val zero : vec2
```

The fields in this record can be extracted individually using the notation *record.field* where *record* is the name of the record and *field* is the name of the field within that record. For example, the x field in the variable `zero` is 0:

```
> zero.x;;
val it : float = 0.0
```

Also, a shorthand `with` notation exists for the creation of a new record from an existing record with some of the fields replaced. This is particularly useful when records contain many fields. For example, the record `{x=1.0; y=0.0}` may be obtained by replacing the field x in the variable `zero` with 1:

```
> let x_axis = { zero with x = 1.0 };;
val x_axis : vec2
> x_axis;;
val it : vec2 = {x = 1.0; y = 0.0}
```

Like many operations in F#, the `with` notation leaves the original record unaltered, creating a new record instead.

1.4.1.4 Sum types: variants The types of values stored in tuples and records are known at compile-time. The F# compiler enforces the correct use of these types at compile-time. However, this is too restrictive in many circumstances. These requirements can be slightly relaxed by allowing a type to be defined which can acquire one of several possible types at run-time. These are known as *variant types*.

Variant types are defined using the `type` construct with the possible constituent types referred to by *constructors* (the names of which must begin with upper-case letters) separated by the | character. For example, a variant type named `button` that may adopt the values `On` or `Off` may be written:

```
> type button =
    | On
    | Off;;
type button = On | Off
```

The constructors `On` and `Off` may then be used as values of type `button`:

```
> On;;
val it : button = On
> Off;;
val it : button = Off
```

In this case, the constructors `On` and `Off` convey no information in themselves (i.e. like the type `unit`, `On` and `Off` do not carry data) but the choice of `On` or `Off` does convey information. Note that both expressions were correctly inferred to be of type `button`.

More usefully, constructors may take arguments, allowing them to convey information by carrying data. The arguments are defined using `of` and are written in the same form as that of a tuple. For example, a replacement `button` type which provides an `On` constructor accepting two arguments (and `int` and a `string`) may be written:

```
> type button =
    | On of int * string
    | Off;;
type button = On of int * string | Off
```

The `On` constructor may then be used to create values of type `button` by appending the argument in the style of a tuple:

```
> On (1, "mine");;
val it : button = On (1, "mine")
> On (2, "hers");;
val it : button = On (2, "hers")
> Off;;
val it : button = Off
```

Types can also be defined recursively, which is very useful when defining more sophisticated data structures, such as trees. For example, a binary tree contains either zero or two binary trees and can be defined as:

```
> type binary_tree =
    | Leaf
    | Node of binary_tree * binary_tree;;
type binary_tree =
  | Leaf
  | Node of binary_tree * binary_tree
```

A value of type `binary_tree` may be written in terms of these constructors:

```
> Node (Node (Leaf, Leaf), Leaf);;
val it : binary_tree = Node (Node (Leaf, Leaf), Leaf)
```

Of course, we could also place data in the nodes to make a more useful data structure. This line of thinking will be pursued in chapter 3. In the meantime, let us consider two special data structures which have notations built into the language.

1.4.1.5 Generics The automatic generalization of function definitions to their most generic form is one of the critical benefits offered by the F# language. In order to exploit such genericity it is essential to be able to parameterize tuple, record and variant types over type variables. A type variable is simply the type theory equivalent of a variable in mathematics. In any given type expression, a type variable denotes any type and a concrete type may be substituted accordingly. Type variables are written `'a`, `'b` and so on.

For example, the following function definition handles a 2-tuple (a pair) but the type of the two elements of the pair are not known and, consequently, the F# compiler automatically generalizes the function to apply to pairs of any two types denoted `'a` and `'b`, respectively:

```
> let swap(a, b) = b, a;;
val swap : 'a * 'b -> 'b * 'a
```

Note that the behaviour of this `swap` function, to swap the elements of a pair, is reflected in its type because the type variables appear in reverse order in the return value. As a programmer grows accustomed to the implications of inferred types, the types of expressions and function definitions come to convey a significant amount of information. Moreover, the type information printed explicitly following an interactive definition or expression in an F# interactive session (as shown here) is made available directly from the source code in an IDE such as Visual Studio. This is described in more detail in chapter 2.

So the type of a generic pair is written `'a * 'b`, the type of a generic record t is written `('a, 'b) t`. For example, the previous record type `vec` may be parameterized over a generic field type `'a`. This is defined and used as follows:

```
> type 'a vec = { x: 'a; y: 'a };;
type 'a vec = { x: 'a; y: 'a }
> { x = 3.0; y = 4.0 };;
val it : float vec = { x = 3.0; y = 4.0 }
```

Note that the parameterized record type is referred to generically as `'a vec` and specifically in this case as `float vec` because the elements are of the type `float`.

Generic variant types are written in an equivalent notation. For example, the following defines and uses a generic variant type called `'a option`:

```
> type 'a option =
    | None
    | Some of 'a;;
type 'a option = None | Some of 'a
> None;;
val it : 'a option = None
> Some 3;;
val it : int option = Some 3
```

This type is actually so useful that it is provided by the F# standard library. Many of the built in data structures, including lists and arrays, are parameterized over the type of elements they contain and generic functions and types are used extensively in the remainder of this book.

In the context of generic classes and generic .NET data structures, a generic type `'a t` is often written equivalently as `t<'a>`. This alternative syntax arises in section 1.4.4 in the context of sequence expressions.

1.4.1.6 Lists and arrays Lists are written `[a; b; c]` and arrays are written `[|a; b; c|]`. As we shall see in chapter 3, lists and arrays have different merits.

Following the notation for generic types, the types of lists and arrays of integers are written `int list` and `int array`, respectively:

```
> [1; 2; 3];;
val it : int list = [1; 2; 3]
> [|1; 2; 3|];;
val it : int array = [|1; 2; 3|]
```

In the case of lists, the infix cons operator `::` provides a simple way to prepend an element to the front of a list. For example, prepending `1` onto the list `[2; 3]` gives the list `[1; 2; 3]`:

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

In the case of arrays, the notation *array*`.[`i`]` may be used to extract the $i+1^{\text{th}}$ element. For example, `[|3; 5; 7|].[1]` gives the second element `5`:

```
> [|3; 5; 7|].[1];;
val it : int = 5
```

Also, a short-hand notation can be used to represent lists or arrays of tuples by omitting unnecessary parentheses. For example, `[(a, b); (c, d)]` may be written `[a, b; c, d]`.

The use and properties of lists, arrays and several other data structures will be discussed in chapter 3. In the mean time, we shall examine programming constructs which allow more interesting computations to be performed.

1.4.1.7 The `if` ***expression*** Like many other programming languages, F# provides an `if` construct which allows a boolean "predicate" expression to determine which of two expressions is evaluated and returned, as well as a special `if` construct which optionally evaluates an expression of type `unit`:

`if` $expr_1$ `then` $expr_2$
`if` $expr_1$ `then` $expr_2$ `else` $expr_3$

In both cases, $expr_1$ must evaluate to a value of type `bool`. In the former case, $expr_2$ is expected to evaluate to the value of type `unit`. In the latter case, both $expr_2$ and $expr_3$ must evaluate to values of the same type.

The former evaluates the boolean expression $expr_1$ and, only if the result is `true`, evaluates the expression $expr_2$. Thus, the former is equivalent to:

`if` $expr_1$ `then` $expr_2$ `else ()`

The latter similarly evaluates $expr_1$ but returning the result of either $expr_2$, if $expr_1$ evaluated to `true`, or of $expr_3$ otherwise.

For example, the following function prints "Less than three" if the given argument is less than three:

```
> let f x =
    if x < 3 then
      print_endline "Less than three";;
val f : int -> unit
> f 5;;
val it : unit = ()
> f 1;;
Less than three
val it : unit = ()
```

The following function returns the string "Less" if the argument is less than 3 and "Greater" otherwise:

```
> let f x =
    if x < 3 then
      "Less"
    else
      "Greater";;
val f : int -> string
> f 1;;
val it : string = "Less"
> f 5;;
val it : string = "Greater"
```

The if expression is significantly less common in F# than many other languages because a much more powerful form of run-time dispatch is provided by pattern matching, which will be introduced in section 1.4.2.

1.4.1.8 More about functions Functions can also be defined anonymously, known as λ-abstraction in computer science. For example, the following defines a function $f(x) = x \times x$ which has a type representing[3] $f : \mathbb{Z} \to \mathbb{Z}$:

```
> fun x -> x * x;;
val it : int -> int = <fun:clo@0_3>
```

This is an anonymous equivalent to the `sqr` function defined earlier. The type of this expression is also inferred to be `int -> int`. This anonymous function may be applied as if it were the name of a conventional function. For example, applying the function f to the value 2 gives $2 \times 2 = 4$:

```
> (fun x -> x * x) 2;;
val : int = 4
```

Consequently, we could have defined the `sqr` function equivalently as:

```
> let sqr = fun x -> x * x;;
```

[3]We say "representing" because the F# type `int` is, in fact, a finite subset of $\mathbb{Z}$, as we shall see in chapter 4.

```
val sqr : int -> int
```

Once defined, this version of the sqr function is indistinguishable from the original.

The let ... in construct allows definitions to be nested, including function definitions. For example, the following function ipow3 raises a given int to the power three using a sqr function nested within the body of the ipow3 function:

```
> let ipow3 x =
    let sqr x = x * x
    x * sqr x;;
val ipow3 : int -> int
```

Note that the #light syntax option allowed us to omit the in keyword from the inner let binding, and that the function application sqr x takes precedence over the multiplication.

The let construct may also be used to define the elements of a tuple simultaneously. For example, the following defines two variables, a and b, simultaneously:

```
> let a, b = 3, 4;;
val a : int
val b : int
```

This is particularly useful when factoring code. For example, the following definition of the ipow4 function contains an implementation of the sqr function which is identical to that in our previous definition of the ipow3 function:

```
> let ipow4 x =
    let sqr x = x * x
    sqr(sqr x);;
val ipow4 : int -> int
```

Just as common subexpressions can be factored out of a mathematical expression, so the ipow3 and ipow4 functions can be factored by sharing a common sqr function and returning the ipow3 and ipow4 functions simultaneously in a 2-tuple:

```
> let ipow3, ipow4 =
    let sqr x = x * x
    (fun x -> x * sqr x), (fun x -> sqr(sqr x));;
val ipow3 : int -> int
val ipow4 : int -> int
```

Factoring code is an important way to keep programs manageable. In particular, programs can be factored much more aggressively through the use of higher-order functions (**HOFs**) — something that can be done in F# but not Java, C++ or Fortran. We shall discuss such factoring of F# programs as a means of code structuring in chapter 2. In the meantime, we shall examine *recursive* functions, which perform computations by applying themselves.

As we have already seen, variable names in let definitions refer to their previously defined values. This default behaviour can be overridden using the rec keyword, which allows a variable definition to refer to itself. This is necessary to define a recursive function[4]. For example, the following implementation of the `ipow` function, which computes n^m for $n, m \geq 0 \in \mathbb{Z}$, calls itself recursively with smaller m to build up the result until the base-case $n^0 = 1$ is reached:

```
> let rec ipow n m =
    if m = 0 then 1 else
      n * ipow n (m - 1);;
val ipow : int -> int -> int
```

For example, $2^{16} = 65,536$:

```
> ipow 2 16;;
val it : int = 65536
```

Recursion is an essential construct in functional programming and will be discussed in more detail in section 1.6.

The programming constructs described so far may already be used to write some interesting functions, using recursion to act upon values of non-trivial types. However, one important piece of functionality is still missing: the ability to dissect variant types, dispatching according to constructor and extracting any data contained in them. Pattern matching is an incredibly powerful core construct in F# that provides exactly this functionality.

1.4.2 Pattern matching

As a program is executed, it is quite often necessary to choose the future course of action based upon the value of a previously computed result. As we have already seen, a two-way choice can be implemented using the `if` construct. However, the ability to choose from several different possible actions is often desirable. Although such cases can be reduced to a series of `if` tests, languages typically provide a more general construct to compare a result with several different possibilities more succinctly, more clearly and sometimes more efficiently than manually-nested `if`s. In Fortran, this is the `SELECT CASE` construct. In C and C++, it is the `switch case` construct.

Unlike conventional languages, F# allows the value of a previous result to be compared against various patterns - *pattern matching*. As we shall see, this approach is considerably more powerful and even more efficient than the conventional approaches.

The most common pattern matching construct in F# is in the `match ... with ...` expression:

`match` *expr* `with`
| $pattern_1$ -> $expr_1$
| $pattern_2$ -> $expr_2$

[4]A recursive function is a function that calls itself, possibly via other functions.

| *pattern*$_3$ -> *expr*$_3$
| ...
| *pattern*$_n$ -> *expr*$_n$

This evaluates *expr* and compares the resulting value firstly with *pattern*$_1$ then with *pattern*$_2$ and so on, until a pattern is found to match the value of *expr*, in which case the corresponding expression *expr*$_n$ is evaluated and returned.

Patterns may reflect arbitrary data structures (tuples, records, variant types, lists and arrays) that are to be matched verbatim and, in particular, the cons operator `::` may be used in a pattern to decapitate a list. Also, the pattern _ matches any value without assigning a name to it. This is useful for clarifying that part of a pattern is not referred to in the corresponding expression.

For example, the following function `f` compares its argument `i` against three patterns, returning the expression of type `string` corresponding to the first pattern that matches:

```
> let f i =
    match i with
    | 0 -> "Zero"
    | 3 -> "Three"
    | _ -> "Neither zero nor three";;
val f : int -> string
```

Applying this function to some expressions of type `int` demonstrates the most basic functionality of the `match` construct:

```
> f 0;;
val it : string = "Zero"
> f 1;;
val it : string = "Neither zero nor three"
> f (1 + 2);;
val it : string = "Three"
```

As pattern matching is such a fundamental concept in F# programming, we shall provide several more examples using pattern matching in this section.

A function `is_empty_list` which examines a given list and returns `true` if the list is empty and `false` if the list contains any elements, may be written without pattern matching by simply testing equality with the empty list:

```
> let is_empty_list list =
    list = [];;
val is_empty_list : 'a list -> bool
```

Note that the clean design of the F# language allows the identifier `list` to be used to refer to both a variable name and a type without conflict.

Using pattern matching, this example may be written using the `match ... with` ... construct as:

```
> let is_empty_list list =
    match list with
    | [] -> true
    | _::_ -> false;;
val is_empty_list : 'a list -> bool
```

Note the use of the anonymous _ pattern to match any value, in this case accounting for all other possibilities.

The `is_empty_list` function can also be written using the `function ...` construct, used to create one-argument λ-functions which are pattern matched over their argument:

```
> let is_empty_list = function
    | [] -> true
    | _::_ -> false;;
val is_empty_list : 'a list -> bool
```

In general, functions that pattern match over their last argument may be rewritten more succinctly using `function`.

1.4.2.1 Variables in patterns Variables that are named in the pattern on the left hand side of a match case are bound to the corresponding parts of the value being matched when evaluating the corresponding expression on the right hand side of the match case. This allows parts of a data structure to be used in the resulting computation and, in particular, this is the only way to extract the values of the arguments of a variant type constructor.

The following function `f` tries to extract the argument of the `Some` constructor of the built in option type, returning a default value of `0` if the given value is `None`:

```
> let f = function
    | None -> 0
    | Some x -> x;;
val f : int option -> int
```

Note that the default value of `0` returned by the first match case of this pattern match led type inference to determine that the argument to the `f` function must be of the type `int option`.

For example, applying this function the values `None` and `Some 3` gives the results `0` and `3` as expected:

```
> f None;;
val it : int = 0
> f(Some 3);;
val it : int = 3
```

In the latter case, the second pattern is matched and the variable name `x` appearing in the pattern is bound to the corresponding value in the data structure, which is `3` in this case. The second match case simply returns the value bound to `x`, returning `3` in this case.

The ability to deconstruct the value of a variant type into its constituent parts is the single most important use of pattern matching.

1.4.2.2 Named subpatterns Part of a data structure can be bound to a variable by giving the variable name in the pattern. Occasionally, it is also useful to be able to bind part of a data structure matched by a sub pattern.

The `as` construct provides this functionality, allowing the value corresponding to a matched subpattern to be bound to a variable.

For example, the following recursive function returns a list of all adjacent pairs from the given list:

```
> let rec pairs = function
    | h1::(h2::_ as t) -> (h1, h2) :: pairs t
    | _ -> [];;
val pairs : 'a list -> ('a * 'a) list
```

In this case, the first two elements from the input are named `h1` and `h2` and the tail list after `h1` is named `t`, i.e. `h2` is the head of the tail list `t`.

Applying the `pairs` function to an example list returns the pairs of adjacent elements as a list of 2-tuples:

```
> pairs [1; 2; 3; 4; 5];;
val it : (int * int) list =
  [(1, 2); (2, 3); (3, 4); (4, 5)]
```

Named subpatterns are used in some of the later examples in this book.

1.4.2.3 Guarded patterns Patterns may also have arbitrary tests associated with them, written using the `when` construct. Such patterns are referred to as *guarded patterns* and are only allowed to match when the associated boolean expression (the *guard*) evaluates to true.

For example, the following recursive function filters out only the non-negative numbers from a list:

```
> let rec positive = function
    | [] -> []
    | h::t when h < 0 -> positive t
    | h::t -> h::positive t;;
val positive : int list -> int list
```

Applying this function to a list containing positive and negative numbers results in a list with the negative numbers removed

```
> positive [-3; 1; -1; 4];;
val it : int list = [1; 4]
```

Although guarded patterns undermine some of the static checking of pattern matches that the F# compiler can perform, they can be used to good effect in a variety of circumstances.

1.4.2.4 Or patterns In many cases it is useful for several different patterns to be combined into a single pattern that is matched when any of the alternatives matches. Such patterns are known as *or-patterns* and use the syntax:

pattern$_1$ | *pattern*$_2$

Or patterns must bind the same sets of variables.

For example, the following function returns `true` when its argument is in $\{-1, 0, 1\}$ and `false` otherwise:

```
> let is_sign = function
    | -1 | 0 | 1 -> true
    | _ -> false;;
val is_sign : int -> bool
```

The sophistication provided by pattern matching may be misused. Fortunately, the F# compilers go to great lengths to enforce correct use, even brashly criticising the programmers style when appropriate.

1.4.2.5 *Erroneous patterns* Alternative patterns in a match case must share the same set of variable bindings. For example, although the following function makes sense to a human, the F# compilers complain about the patterns `(a, 0)` and `(0, b)` binding different sets of variables ($\{a\}$ and $\{b\}$, respectively):

```
> let product a b =
    match a, b with
    | a, 0 | 0, b -> 0
    | a, b -> a * b;;
Error: FS0018: The two sides of this 'or' pattern bind
different sets of variables
```

In this case, this function can be corrected by using the anonymous _ pattern as neither a nor b is used in the first case:

```
> let product a b =
    match a, b with
    | _, 0 | 0, _ -> 0
    | a, b -> a * b;;
val product : int -> int -> int
```

This actually conveys useful information about the code. Specifically, that the values matched by _ are not used in the corresponding expression.

F# uses type information to determine the possible values of expression being matched. If the patterns fail to cover all of the possible values of the input then, at compile-time, the compiler emits:

```
Warning: FS0025: Incomplete pattern match.
```

If a program containing such pattern matches is executed and no matching pattern is found at run-time then `MatchFailureException` is raised. Exceptions will be discussed in section 1.4.5.

For example, in the context of the built-in option type, the F# compiler will warn of a function matching only the `Some` type constructor and neglecting `None`:

```
> let extract = function
```

```
    | Some x -> x;;
Warning: FS0025: Incomplete pattern match.
The value 'None' will not be matched.
val extract : 'a option -> int
```

This `extract` function then works as expected when given that value `Some 3`:

```
> extract (Some 3);;
val it : int = 3
```

but causes `MatchFailureException` to be raised at run-time if a `None` value is given, as none of the patterns in the pattern match of the `extract` function match this value:

```
> extract None;;
Exception of type
'Microsoft.FSharp.MatchFailureException' was thrown.
```

As some approaches to pattern matching lead to more robust programs, some notions of good and bad programming styles arise in the context of pattern matching.

1.4.2.6 Good style The compiler cannot prove that any given pattern match covers all eventualities in the general case. Thus, some style guidelines may be productively adhered to when writing pattern matches, to aid the compiler in its proofs:

- Guarded patterns should be used only when necessary. In particular, in any given pattern matching, the last pattern should not be guarded.
- In the case of user-defined variant types, all eventualities should be covered explicitly (such as `[]` and `h::t` which, between them, match any list).

As proof generation cannot be automated in general, the F# compilers do not try to prove that a sequence of guarded patterns will match all possible inputs. Instead, the programmer is expected to adhere to a good programming style, making the breadth of the final match explicit by removing the guard. For example, the F# compilers do not prove that the following pattern match covers all possibilities:

```
> let sign = function
    | i when i < 0.0 -> -1
    | 0.0 -> 0
    | i when i > 0.0 -> 1;;
Warning: FS0025: Incomplete pattern match.
val sign : float -> int
```

In this case, the function should have been written without the guard on the last pattern:

```
> let sign = function
    | i when i < 0.0 -> -1
    | 0.0 -> 0
    | _ -> 1;;
val sign : float -> int
```

Also, the F# compilers will try to determine any patterns which can never be matched. If such a pattern is found, the compiler will emit a warning. For example, in this case the first match accounts for all possible input values and, therefore, the second match will never be used:

```
> let product a b =
    match a, b with
    | a, b -> a * b
    | _, 0.0 | 0.0, _ -> 0.0;;
Warning: FS0026: This rule will never be matched.
val product : int -> int -> int
```

When matching over the constructors of a type, all eventualities should be caught explicitly, i.e. the final pattern should not be made completely general. For example, in the context of a type which can represent different number representations:

```
> type number =
    | Integer of int
    | Real of float;;
type number = Integer of int | Real of float
```

A function to test for equality with zero could be written in the following, poor style:

```
> let bad_is_zero = function
    | Integer 0 | Real 0.0 -> true
    | _ -> false;;
val bad_is_zero : number -> bool
```

When applied to various values of type `number`, this function correctly acts a predicate to test for equality with zero:

```
> bad_is_zero (Integer (-1));;
val it : bool = false
> bad_is_zero (Integer 0);;
val it : bool = true
> bad_is_zero (Real 0.0);;
val it : bool = true
> bad_is_zero (Real 2.6);;
val it : bool = false
```

Although the `bad_is_zero` function works in this case, this formulation is fragile when the variant type is extended during later development of the program. Instead, the constructors of the variant type should be matched against explicitly, to ensure that later extensions to the variant type yield compile-time warnings for this function (which could then be fixed):

```
> let good_is_zero = function
    | Integer 0 | Real 0.0 -> true
    | Integer _ | Real _ -> false;;
val good_is_zero : number -> bool
```

The style used in the good_is_zero function is more robust. For example, if whilst developing our program, we were to supplement the definition of our number type with a new representation, say of the complex numbers $z = x + iy \in \mathbb{C}$:

```
> type number =
    | Integer of int
    | Real of float
    | Complex of float * float;;
type number =
  | Integer of int
  | Real of float
  | Complex of float * float
```

the bad_is_zero function, which is written in the poor style, would compile without warning despite being incorrect:

```
> let bad_is_zero = function
    | Integer 0 | Real 0.0 -> true
    | _ -> false;;
val bad_is_zero : number -> bool
```

Specifically, this function treats all values which are not zero-integers or zero-reals as being non-zero. Thus, zero-complex $z = 0 + 0i$ is incorrectly deemed to be non-zero:

```
> bad_is_zero (Complex (0.0, 0.0));;
val it : bool = false
```

In contrast, the good_is_zero function, which was written using the good style, would allow the compiler to spot that part of the number type was not being accounted for in the pattern match:

```
> let good_is_zero = function
    | Integer 0 | Real 0.0 -> true
    | Integer _ | Real _ -> false;;
Warning: FS0025: Incomplete pattern match.
val good_is_zero : number -> bool
```

The programmer could then supplement this function with a case for complex numbers:

```
> let good_is_zero = function
    | Integer 0 | Real 0.0 | Complex(0.0, 0.0) -> true
    | Integer _ | Real _ | Complex _ -> false;;
val good_is_zero : number -> bool
```

The resulting function would then provide the correct functionality:

```
> good_is_zero (Complex (0.0, 0.0));;
val it : bool = true
```

Clearly, the ability have such safety checks performed at compile-time can be very valuable during development. This is another important aspect of safety provided by the F# language, which results in considerably more robust programs.

Due to the ubiquity of pattern matching in F# programs, the number and structure of pattern matches can be non-trivial. In particular, patterns may be nested and may be performed in parallel.

1.4.2.7 Parallel pattern matching Pattern matching is often applied to several different values in a single function. The most obvious way to pattern match over several values is to nest pattern matches. However, nested patterns are rather ugly and confusing.

For example, the following function tries to unbox three option types, returning None if any of the inputs is None:

```
> let unbox3 a b c =
    match a with
    | Some a ->
        match b with
        | Some b ->
            match c with
            | Some c -> Some(a, b, c)
            | None -> None
        | None -> None
    | _ -> None;;
val unbox3 :
  'a option -> 'b option -> 'c option ->
    ('a * 'b * 'c) option
```

Applying this function to three option values gives an option value in response:

```
> unbox3 (Some 1) (Some 2) (Some 3);;
val it : (int * int * int) option = Some (1, 2, 3)
```

Fortunately, parallel pattern matching can be used to perform the same task more concisely. This refers to the act of pattern matching over a tuple of values rather than nesting different pattern matches for each value.

For example, a function to unbox three option values simultaneously may be written more concisely using a parallel pattern match:

```
> let unbox3 a b c =
    match a, b, c with
    | Some a, Some b, Some c -> Some (a, b, c)
    | _ -> None;;
val unbox3 :
  'a option -> 'b option -> 'c option ->
    ('a * 'b * 'c) option
```

As a core feature of the F# language, pattern matching will be used extensively in the remainder of this book, particularly when dissecting data structures in chapter 3.

1.4.2.8 Active patterns ML-style pattern matching provides a simple and efficient way to dissect concrete data structures such as trees and, consequently, is ubiquitous in this family of programming languages. However, ML-style pattern matching has the disadvantage that it ties a function to a particular concrete data structure. A new feature in the F# programming language called *active patterns* is designed to alleviate this problem by allowing patterns to perform computations to dissect a concrete data structure and present it in a different form, known as a *view* of the underlying structure.

As a simple example, active patterns can be used to sanitize the strange total ordering function `compare` that F# inherited from OCaml by viewing the `int` result as the sum type that it really represents:

```
> let (|Less|Equal|Greater|) = function
    | c when c<0 -> Less
    | c when c>0 -> Greater
    | _ -> Equal;;
val (|Less|Equal|Greater|) :
  int -> Choice<unit, unit, unit>
```

Pattern matches over `int` values can now use the active patterns `Less`, `Equal` and `Greater`. Moreover, the pattern matcher is now aware that these three patterns form a complete set of alternatives.

A more useful example of active patterns is the dissecting of object oriented data structures carried over from the .NET world. The use of active patterns to simplify the dissection of XML trees is described in chapter 10.

1.4.3 Equality

The F# programming language includes a notion of structural equality that automatically traverses values of compound types such as tuples, records, variant types, lists and arrays as well as handling primitive types. The equality operator = calls the `Equals` method of the .NET object, allowing the equality operation to be overridden for specific types where applicable.

For example, the following checks that $3 - 1 = 2$:

```
> 3 - 1 = 2;;
val it : bool = true
```

The following tests the contents of two pairs for equality:

```
> (2, 3) = (3, 4);;
val it : bool = false
```

In some cases, the built-in structural equality is not the appropriate notion of equality. For example, the set data structure (described in detail in chapter 3) is represented internally as a balanced binary tree. However, some sets have degenerate representations, e.g. they may be balanced differently but the contents are the same. So structural equality is not the correct notion of equality for a set. Consequently, the `Set` module overrides the default `Equals` member to give the = operator an appropriate notion of set equality, where sets are compared by the elements they contain regardless of how they happen to be balanced.

Occasionally, the ability to test if two values refer to identical representations (e.g. the same memory location) may be useful. This is known as *reference* equality in the context of .NET and is provided by the built-in == operator.

For example, pairs defined in different places will reside in different memory locations. So, in the following example, the pair `a` is referentially equal to itself but `a` and `b` are logically but not referentially equal:

```
> let a = 1, 2;;
val a : int * int
> let b = 1, 2;;
val b : int * int
> a == a;;
val it : bool = true
stdin(26,1): warning: FS0062: This construct is for
compatibility with OCaml. The use of the physical
equality operator '==' is not recommended except in
cross-compiled code. You probably want to use generic
structural equality '='. Disable this warning using
--no-warn 632 or #nowarn "62"
```

This warning is designed for programmers used to languages where == denotes ordinary equality. Safe in the knowledge that == denotes referential equality in F#, we can disable this warning before using it:

```
> a = b;;
val it : bool = true
> #nowarn "62";;
> a == b;;
val it : bool = false
```

Referential equality may be considered a probabilistic alternative to logical equality. If two values are referentially equal then they must also be logically equal, otherwise they may or may not be logically equal. The notion of referential equality can be used to implement productive optimizations by avoiding unnecessary copying and is discussed in chapter 8.

1.4.4 Sequence expressions

The F# programming language provides an elegant syntax called *sequence expressions* for generating lists, arrays and `Seq`[5]. A contiguous sequence of integers can be specified using the syntax:

`seq` {*first* `..` *last*}

For example, the the integers $i \in \{1 \ldots 5\}$ may be created using:

```
> seq {1 .. 5};;
val it : seq<int> = seq [1; 2; 3; ...]
```

Note that the generic sequence type is written using the syntax `seq<'a>` by default rather than `'a seq`.

All comprehension syntaxes can be used with different brackets to generate lists and arrays. For example:

```
> [1 .. 5];;
val it : int list = [1; 2; 3; 4; 5]
> [|1 .. 5|];;
val it : int array = [|1; 2; 3; 4; 5|]
```

Non-contiguous sequences can also be created by specifying a step size using the syntax:

`seq` {*first* `..` *step* `..` *last*}

For example, the integers $[0 \ldots 9]$ in steps of 3 may be created using:

```
> seq {0 .. 3 .. 9};;
val it : seq<int> = seq [0; 3; 6; ...]
```

Lists, arrays and sequences can be filtered into a sequence using the syntax:

`seq` {`for` *pattern* `in` *container* `->`
expr}

For example, the squares of the `Some` values in an option list may be filtered out using:

```
> seq {for Some i in [Some 1; Some 3; None ; Some 2] ->
         i * i};;
val it : seq<int> = seq [1; 9; 4]
```

The pattern used for filtering can be guarded using the syntax:

`seq` {`for` *pattern* `in` *container* `when` *guard* `->`
expr}

For example, extracting only results for which $i < 3$ in the previous example:

[5] `Seq` is discussed in detail in section 3.8.

```
> let xs = [Some 1; Some 3; None; Some 2] in
  seq {for Some i in xs when i < 3 ->
         i * i};;
val it : seq<int> = seq [1; 4]
```

These examples all generate a data structure called `Seq`. This data structure is discussed in detail in chapter 3.

Comprehensions may also be nested to produce a flat data structure. For example, nesting loops over x and y coordinates is an easy way to obtain a sequence of grid coordinates:

```
> [ for x in 1 .. 3
      for y in 1 .. 3 ->
        x, y ];;
val it : (int * int) list =
  [1, 1; 1, 2; 1, 3; 2, 1; 2, 2; 2, 3; 3, 1; 3, 2; 3, 3]
```

Sequence expressions have a wide variety of uses, from random number generators to file IO.

1.4.5 Exceptions

In many programming languages, program execution can be interrupted by the raising[6] of an exception. This is a useful facility, typically used to handle problems such as failing to open a file or an unexpected flow of execution (e.g. due to a program being given invalid input).

Like a variant constructor in F#, the name of an exception must begin with a capital letter and an exception may or may not carry an associated value. Before an exception can be used, it must declared. An exception which does not carry associated data may be declared as:

`exception` *Name*

An exception which carries associated data of type *type* may be declared:

`exception` *Name* `of` *type*

Exceptions are raised using the built-in `raise` function. For example, the following raises a built-in exception called `Failure` which carries a string:

```
raise (Failure "My problem")
```

F# exceptions may be caught using the syntax:

`try`
  expr
`with`
| *pattern*$_1$ `->` *expr*$_1$

[6]Sometimes known as *throwing* an exception, e.g. in the context of the C++ language.

| *pattern*$_2$ -> *expr*$_2$
| *pattern*$_3$ -> *expr*$_3$
| ...
| *pattern*$_n$ -> *expr*$_n$

where *expr* is evaluated and its result returned if no exception was raised. If an exception was raised then the exception is matched against the patterns and the value of the corresponding expression (if any) is returned instead.

For example, the following raises and catches the `Failure` exception and returns the string that was carried by the exception:

```
> try
    raise (Failure "My problem")
  with
  | Failure s ->
      s;;
val it : string = "My problem"
```

Note that, unlike other pattern matching constructs, patterns matching over exceptions need not account for all eventualities — any uncaught exceptions simply continue to propagate.

For example, an exception called `ZeroLength` that does not carry associated data may be declared with:

```
> exception ZeroLength;;
exception ZeroLength
```

A function to normalize a 2D vector $\mathbf{r} = (x, y)$ to create a unit-length 2D vector:

$$\hat{\mathbf{r}} = \frac{\mathbf{r}}{|\mathbf{r}|}$$

Catching the erroneous case of a zero-length vector, this may be written:

```
> let norm (x, y) =
    match sqrt(x * x + y * y) with
    | 0.0 -> raise ZeroLength
    | s -> x / s, y / s;;
val norm : float * float -> float * float
```

Applying the `norm` function to a non-zero-length vector produces the correct result to within numerical error (a subject discussed in chapter 4):

```
> norm (3.0, 4.0);;
val it : float * float = (0.6, 0.8)
```

Applying the `norm` function to the zero vector raises the `ZeroLength` exception:

```
> norm (0.0, 0.0);;
Exception of type 'FSI_0159+ZeroLength' was thrown.
```

A "safe" version of the norm function might catch this exception and return some reasonable result in the case of a zero-length vector:

```
> let safe_norm r =
    try
      norm r
    with
    | ZeroLength ->
        0.0, 0.0;;
val safe_norm : float * float -> float * float
```

Applying the safe_norm function to a non-zero-length vector causes the result of the expression norm r to be returned:

```
> safe_norm (3.0, 4.0);;
val it : float * float = (0.6, 0.8)
```

However, applying the safe_norm function to the zero vector causes the norm function to raise the ZeroLength exception which is then caught within the safe_norm function which then returns the zero vector:

```
> safe_norm (0.0, 0.0);;
val it : float * float = (0.0, 0.0)
```

The use of exceptions to handle unusual occurrences, such as in the safe_norm function, is one important application of exceptions. This functionality is exploited by many of the functions provided by the core F# library, such as those for handling files (discussed in chapter 5). The safe_norm function is a simple example of using exceptions that could have been written using an if expression. However, exceptions are much more useful in more complicated circumstances, where an exception might propagate through several functions before being caught.

Another important application is the use of exceptions to escape computations. The usefulness of this way of exploiting exceptions cannot be fully understood without first understanding data structures and algorithms and, therefore, this topic will be discussed in much more detail in chapter 3 and again, in the context of performance, in chapter 8.

The Exit, Invalid_argument and Failure exceptions are built-in, as well as two functions to simplify the raising of these exceptions. Specifically, the invalid_arg and failwith functions raise the Invalid_argument and Failure exceptions, respectively, using the given string.

F# also provides a try ... finally ... construct that executes a final expression whether or not an exception is raised. This can be used to ensure that state changes are correctly undone even under exceptional circumstances.

1.5 IMPERATIVE PROGRAMMING

Just like conventional programming languages, F# supports mutable variables and side effects: imperative programming. Record fields can be marked as mutable, in which case their value may be changed. For example, the type of a mutable, two-dimensional vector called `vec2` may be defined as:

```
> type vec2 = { mutable x: float; mutable y: float };;
type vec2 = { mutable x : float; mutable y : float; }
```

A value `r` of this type may be defined:

```
> let r = { x = 1.0; y = 2.0 };;
val r : vec2
```

The x-coordinate of the vector `r` may be altered in-place using an imperative style:

```
> r.x <- 3.0;;
val it : unit = ()
```

The side-effect of this expression has mutated the value of the variable `r`, the x-coordinate of which is now 3 instead of 1:

```
> r;;
val it : vec = {x = 3.0; y = 2.0}
```

A record with a single, mutable field can often be useful. This data structure, called a *reference*, is already provided by the type `ref`. For example, the following defines a variable named `a` that is a reference to the integer 2:

```
> let a = ref 2;;
val a : int ref = {contents = 2}
```

The type of `a` is then `int ref`. The value referred to by `a` may be obtained using `!a`:

```
> !a;;
val it : int = 2
```

The value of `a` may be set using `:=`:

```
> a := 3;;
val it : unit = ()
> !a;;
val it : int = 3
```

In the case of references to integers, two additional functions are provided, `incr` and `decr`, which increment and decrement references to integers, respectively:

```
> incr a;;
val it : unit = ()
> !a;;
val a : int = 4
```

In addition to mutable data structures, the F# language provides looping constructs for imperative programming. The `while` loop executes its body repeatedly while the condition is `true`, returning the value of type `unit` upon completion. For example, this `while` loop repeatedly decrements the mutable variable `x`, until it reaches zero:

```
> let x = ref 5;;
val x : int ref
> while !x > 0 do
    decr x;;
val it : unit = ()
> !x;;
val it : int = 0
```

The `for` loop introduces a new loop variable explicitly, giving the initial and final values of the loop variable. For example, this `for` loop runs a loop variable called `i` from one to five, incrementing the mutable value `x` five times in total:

```
> for i = 1 to 5 do
    incr x;;
val it : unit = ()
> !x;;
val it : int = 5
```

Thus, `while` and `for` loops in F# are analogous to those found in most imperative languages.

1.6 FUNCTIONAL PROGRAMMING

Unlike the *imperative* programming languages C, C++, C#, Java and Fortran, F# is a *functional* programming language. Functional programming is a higher-level and mathematically more elegant approach to programming that is ideally suited to scientific computing. Indeed, most scientists do not realise that they naturally compose programs in a functional style even if they are using an imperative language. We shall now examine the various aspects of functional programming and their implications in more detail.

1.6.1 Immutability

In mathematics, once a variable is defined to have a particular value, it keeps that value indefinitely. Thus, variables in mathematics are *immutable*. Similarly, most variables in F# are immutable.

In practice, the ability to choose between imperative and functional styles when programming in F# is very productive. Many programming tasks are naturally suited to either an imperative or a functional style. For example, portions of a program dealing with user input, such as mouse movements and key-presses, are likely to benefit from an imperative style where the program maintains a state and user input may result in a change of state. In contrast, functions dealing with the manipulation of complex data structures, such as trees and graphs, are likely to benefit from being written in a functional style, using recursive functions and immutable data, as this greatly simplifies the task of writing such functions correctly. In both cases, functions can refer to themselves — *recursive* functions. However, recursive functions are pivotal in functional programming, where they are used to implement functionality equivalent to the `while` and `for` looping constructs we have just examined.

One of the simplest differences between conventional imperative languages and functional programming languages like F# is the ubiquitous use of immutable data structures in functional programming. Indeed, the F# standard library provides a wealth of efficiently-implemented immutable data structures. The use of immutable data structures has some subtle implications and important benefits.

When a function acts upon an immutable data structure to produce a similar immutable data structure there is no need to copy the parts of the input that are reused in the output because the input data structure can never be changed. This ability to refer back to old values is known as *referential transparency*. So functions that act over immutable data structures typically compose an output that refers back to parts of the input.

For example, creating a list `b` as an element prepended onto a list `a` does not alter `a`:

```
> let a = [1; 2; 3];;
val a : int list
> let b = 0::a;;
val b : int list
> a, b;;
val it : int list * int list = ([1; 2; 3], [0; 1; 2; 3])
```

Note that the original list a is still `[1; 2; 3]`. Imperative programming would either require that a is altered (losing its original value) or that a is copied. Essentially, the former is confusing and the latter is slow.

Immutable data structures are beneficial for two main reasons:

- Simplicity: Mathematical expressions can often be translated into efficient functional programs much more easily than into efficient imperative programs.

- Concurrent: Immutable data structures are inherently thread safe so they are ideal for parallel programming.

When a programmer is introduced to the concept of functional programming for the first time, the way to implement simple programming constructs such as loops does not appear obvious. If the loop variable cannot be changed because it is immutable then how can the loop proceed?

1.6.2 Recursion

Looping constructs can be converted into recursive constructs, such as recurrence relations. For example, the factorial function is typically considered to be a product with the special case $0! = 1$:

$$n! = \prod_{i=1}^{n} i$$

This may be translated into an imperative function that loops over $i \in \{1 \dots n\}$, altering a mutable accumulator called `accu`:

```
> let factorial n =
    let accu = ref 1
    for i = 1 to n do
      accu := i * !accu
    !accu;;
val factorial : int -> int
```

For example, $5! = 120$:

```
> factorial 5;;
val it : int = 120
```

However, the factorial may be expressed equivalently as a recurrence relation:

$$0! = 1$$
$$n! = n \times (n-1)!$$

This may be translated into an recursive function that calls itself until the base case $0! = 1$ is reached:

```
> let rec factorial = function
    | 0 -> 1
    | n -> n * factorial (n - 1);;
val factorial : int -> int
> factorial 5;;
val it : int = 120
```

In this case, the functional style is significantly simpler than the imperative style. As we shall see in the remainder of this book, functional programming is often more concise and simpler than imperative programming. This is particularly true in the context of mathematical programs.

The remaining aspects of functional programming are concerned with passing functions to functions and returning functions from functions.

1.6.3 Curried functions

A curried function is a function that returns a function as its result. Curried functions are best introduced as a more flexible alternative to the conventional (non-curried) functions provided by imperative programming languages.

Effectively, imperative languages only allow functions to accept a single value (often a tuple) as an argument. For example, a raise-to-the-power function for integers would have to accept a single tuple as an argument which contained the two values used by the function:

```
> let rec ipow_1(x, n) =
    match n with
    | 0 -> 1.0
    | n -> x * ipow_1(x, n - 1);;
val ipow_1 : float * int -> float
```

But, as we have seen, F# also allows:

```
> let rec ipow_2 n x =
    match n with
    | 0 -> 1.0
    | n -> x * ipow_2 (n - 1) x;;
val ipow_2 : int -> float -> float
```

This latter approach is actually a powerful generalization of the former, only available in functional programming languages.

The difference between these two styles is subtle but important. In the latter case, the type can be understood to mean:

```
val ipow_2 : int -> (float -> float)
```

i.e. this `ipow_2` function accepts an exponent n and returns a function that raises a float x to the power of n, i.e. this is a curried function.

The utility of curried functions lies in their ability to have their arguments *partially applied*.

In this case, the curried `ipow_2` function can have the power n partially applied to obtain a more specialized function for raising a `float` to a particular power. For example, functions to square and cube a `float` may now be written very succinctly in terms of `ipow_2`:

```
> let square = ipow_2 2;;
val square : float -> float

> square 5.0;;
val it : float = 25.0

> let cube = ipow_2 3;;
val cube : float -> float

> cube 3.0;;
val it : float = 27.0
```

Thus, the use of currying has allowed an expression of the form:

```
fun x -> ipow_1(x, 2)
```

to be replaced with the more succinct alternative:

```
ipow_2 2
```

This technique actually scales very well to more complicated situations with several curried arguments being partially applied one after another. As we shall see in the next chapter, currying is particularly useful when used in combination with higher-order functions.

1.6.4 Higher-order functions

Conventional languages vehemently separate functions from data. In contrast, F# allows the seamless treatment of functions as data. Specifically, F# allows functions to be stored as values in data structures, passed as arguments to other functions and returned as the results of expressions, including the return-values of functions.

A *higher-order function* is a function that accepts another function as an argument. As we shall now demonstrate, this ability can be of direct relevance to scientific applications.

Many numerical algorithms are most obviously expressed as one function parameterized over another function. For example, consider a function called d that calculates a numerical approximation to the derivative of a given one-argument function. The function d accepts a function $f : \mathbb{R} \to \mathbb{R}$ and a value x and returns a function to compute an approximation to the derivative $\frac{df}{dx}$ given by:

$$d[f](x) = \frac{f(x+\delta) - f(x-\delta)}{2\delta} \simeq \frac{df}{dx}$$

where $d : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$.

This is easily written in F# as the higher-order function d that accepts the function f as an argument[7]:

```
> let d (f : float -> float) x =
    let dx = sqrt epsilon_float
    (f (x + dx) - f (x - dx)) / (2.0 * dx);;
val d : (float -> float) -> float -> float
```

For example, consider the function $f(x) = x^3 - x - 1$:

```
> let f x = x ** 3.0 - x - 1.0;;
val f : float -> float
```

The higher-order function d can be used to approximate $\left.\frac{df}{dx}\right|_{x=2} = 11$:

```
> d f 2.0;;
val it : float = 11.0
```

More importantly, as d is a curried function, we can use d to create derivative functions. For example, the derivative $f'(x) = \frac{\partial f}{\partial x}$ can be obtained by partially applying the curried higher-order function d to f:

[7]The built-in value epsilon_float is the smallest floating-point number that, when added to 1, does not give 1. The square root of this value can be shown to give optimal properties when used in this way.

```
> let f' = d f;;
val f' : (float -> float)
```

The function f' can now be used to calculate a numerical approximation to the derivative of f for any x. For example, $f'(2) = 11$:

```
> f' 2.0;;
val it : float = 11.0
```

Higher-order functions are invaluable for representing many operators found in mathematics, science and engineering.

Now that the foundations of F# have been introduced, the next chapter describes how these building blocks can be structured into working programs.