Part I

Declarative Programming Paradigms

A programming problem may have many solutions, e.g., squaring a real value, x^2, can in F# be written as both `x*x` and `x**2.0`, and more complicated problems typically have many valid solutions. Particularly long programs can be complex and can have a high risk of programming errors. Different programming languages offer different structures to aid the programming in managing complex solutions, which is sometimes called a *programming paradigm*. Paradigms may be classified as either *declarative* or *imperative*. Some languages such as F# are multiparadigm, making the boundary between programming paradigms fuzzy, however, in their pure form, programs in declarative programming languages are a list of properties of the desired result without a specification on how to compute it, while programs in imperative programming languages is a specific set of instructions on how to change *states* on the computer in order to reach the desired result. This part will emphasize the *functional programming paradigm*, which is a declarative paradigm. In **??**, the *imperative* and the *object-oriented programming paradigms* will be emphasized.

Functional programming is a style of programming which performs computations by evaluating functions. Functional programming is declarative in nature, e.g., by the use of value- and function-bindings – `let`-bindings – and avoids statements – `do`-bindings. Thus, all values are constants, and the result of a function in functional programming depends only on its arguments. It is deterministic, i.e., repeated call to a function with the same arguments always gives the same result. In functional programming, data and functions are clearly separated, and hence data structures are dum as compared to objects in object-oriented programming paradigm, see **??**. Functional programs clearly separate behavior from data and subscribes to the view that *it is better to have 100 functions operate on one data structure than 10 functions on 10 data structures*. Simplifying the data structure has the advantage that it is much easier to communicate data than functions and procedures between programs and environments. The .Net, mono, and java's virtual machine are all examples of an attempt to rectify this, however, the argument still holds.

The functional programming paradigm can trace its roots to lambda calculus introduced by Alonzo Church in 1936 [**?**]. Church designed lambda calculus to discuss computability. Some of the forces of the functional programming paradigm are that it is often easier to prove the correctness of code, and since no states are involved, then functional programs are often also much easier to parallelize than other paradigms.

Functional programming has a number of features:

Pure functions
: Functional programming is performed with pure functions. A pure function always returns the same value, when given the same arguments, and it has no side-effects. A function in F# is an example of a pure function. Pure functions can be replaced by their result without changing the meaning of the program. This is known as *referential transparency*.

higher-order functions
: Functional programming makes use of higher-order functions, where functions may be given as arguments and returned as results of a function application. higher-order functions and *first-class citizenship* are related concepts, where higher-order functions are the mathematical description of functions that operator on functions, while a first-class citizen is the computer science term for functions as values. F# implements higher-order functions. The `List.map` is and example of a higher-order functions.

Recursion
: Functional programs use recursion instead of `for`- and `while`-loops. Recursion can make programs ineffective, but compilers are often designed to optimize *tail-recursion* calls. Common recursive programming structures are often available as optimized higher-order functions such as *iter*, *map*, *reduce*, *fold*, and *foldback*. F# has good support for all of these features.

Immutable states
: Functional programs operate on values, not on *mutable values* also known as *variables*. This implies *lexicographical scope* in contrast to mutable values, which implies *dynamic scope*.

Strongly typed
: Functional programs are often strongly typed, meaning that types are set no later than at *compile-time*. F# does have the ability to perform runtime type assertion, but for most parts it relies on explicit *type annotations* and *type inference* at compile-time. This means that type errors are caught at compile time instead of at runtime.

Lazy evaluation
: Due to referential transparency, values can be computed any time up until the point when it is needed. Hence, they need not be computed at compilation time, which allows for infinite data structures. F# has support for lazy evaluations using the `lazy`-keyword, sequences using the `seq`-type, and computation expressions, all of which are advanced topics and not treated in this book.

Immutable states imply that data structures in functional programming are different than in imperative programming. E.g., in F# lists are immutable, so if an element of a list is to be changed, a new list must be created by copying all old values except that which is to be changed. Such an operation is therefore linear in computational complexity. In contrast, arrays are mutable values, and changing a value is done by reference to the value's position and changing the value at that location. This has constant computational complexity. While fast, mutable values give dynamic scope and makes reasoning about the correctness of a program harder, since mutable states do not have referential transparency.

Functional programming may be considered a subset of *imperative programming*, in the sense that functional programming does not include the concept of a state, or one may think of functional programming as only having one unchanging state. Functional programming also has a bigger focus on declaring rules for *what* should be solved, and not explicitly listing statements describing *how* these rules should be combined and executed in order to solve a given problem. Functional programming is often found to be less error-prone at runtime, making more stable, safer programs that are less open for, e.g., hacking.