Part I

Imperative Programming Paradigms

In this part, we will primarily consider the *imperative* and *object-oriented programming paradigms*. Unfortunately, the imperative paradigm is used in the literature both to mean the overarching term of imperative paradigms and, as we do here, imperative programming without using object-oriented programming or other features. Thus it is ok, to say that object-oriented programming follows the imperative paradigm, but imperative programming does not necessarily follow the object-oriented paradigm.

Imperative programming is a paradigm for programming *states*. In imperative programming, the focus is on how a problem is to be solved, as a list of *statements* that affect states. In F#, states are *mutable values*, and they are affected by functions. In imperative programming, functions are sometimes called *procedures*, to emphasize that they may have *side-effects*. A side-effect is the result of the change of a state on the computer not related to the list of return parameters from the procedure. An imperative program is typically identified as using:

Mutable values
: Mutable values are holders of states, they may change over time, and thus have dynamic scope.

Procedures
: Procedures are functions that return "`()`", as opposed to functions that transform data. They are the embodiment of side-effects.

Side-effects
: Side-effects are changes of state that are not reflected in the arguments and return values of a function. The `printf` is an example of a procedure that writes text on the terminal but returns "`()`".

Loops
: The `for`- and `while`-loops typically use an iteration value to update some state, e.g., `for`-loops are often used to iterate through a list and summarize its contents.

Functional programming, can be seen as a subset of imperative programming and is discussed in **??**. *Object-oriented programming* is an extension of imperative programming, where statements and states are grouped into classes. For a discussion on object-oriented programming, see **??**–**??**. Almost all computer hardware is designed for *machine code*, which is a common term used for many low-level computer programming languages, and almost all machine languages follow the imperative programming paradigm.

A prototypical example of an imperative program is a baking recipe, e.g., to make a loaf of bread, do the following:

1. Mix yeast with water.

2. Stir in salt, oil, and flour.
3. Knead until the dough has a smooth surface.
4. Let the dough rise until it has doubled its size.
5. Shape dough into a loaf.
6. Let the loaf rise until it is approximately double in size.
7. Bake in the oven until the bread is golden brown.

Each line in this example consists of one or more statements that are to be executed, and while executing them, states such as the size of the dough and the color of the bread change. Some execution will halt execution until certain conditions of these states are fulfilled, e.g., the bread will not be put into the oven for baking before it has risen sufficiently.

Object-oriented programming is a paradigm for encapsulating data and methods into cohesive units. E.g., a car can be modeled as an object, where data about the car including the amount of fuel, position, velocity, passengers, etc., can be stored together with functions for manipulating this data, e.g., move the car, add or extract passengers, etc. Key features of object-oriented programming are:

Encapsulation
: Data and methods are collected into a cohesive unit, and an application program need only focus on how to use the object, not on its implementation details.

Inheritance
: Objects are organized in a hierarchy of gradually increased specialties. This promotes a design of code that is of general use and code reuse.

Polymorphism
: By overriding methods from a base class, derived classes define new data types while their methods still produce results compatible with the base class definitions.

Object-oriented programming has a well-developed methodology for analysis and design. The analysis serves as input to the design phase, where the analysis reveals *what* a program is supposed to do, and the design *how* it is supposed to be doing it. The analysis should be expressed in general terms irrespective of the technological constraints, while the design should include technological constraints such as defined by the targeted language and hardware.