1 The Imperative Programming 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 affects *states*. In F#, states are mutable and immutable values, and they are affected by functions and procedures. An imperative program is typically identified as using:

- imperative programming
- statement
- states
- mutable value

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

- procedure

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

- side-effect

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 uses side-effects to communicate with the terminal.

- `for`
- `while`

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.

Mono state or stateless programs, as *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 **??**.

- functional programming
- object-oriented programming

An imperative program is like a Turing machine, a theoretical machine introduced by Alan Turing in 1936 [**?**]. 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.

- machine code

A prototypical example 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 double 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 changes. 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.

1.1 Imperative Design

Programming is the act of solving a problem by writing a program to be executed on a computer. The imperative programming paradigm focuses on states. To solve a problem, you could work through the following list of actions:

1. Understand the problem. As Pólya described it, see **??**, the first step in any solution is to understand the problem. A good trick to check whether you understand the problem, is to briefly describe it in your own words.

2. Identify the main values, variables, functions, and procedures needed. If the list of procedures is large, then you most likely should organize them in modules.

3. For each function and procedure, write a precise description of what it should do. This can conveniently be performed as an in-code comment for the procedure, using the F# XML documentation standard.

4. Make mockup functions and procedures using the intended types, but do not necessarily compute anything sensible. Run through examples in your mind, using this mockup program to identify any obvious oversights.

5. Write a suite of unit tests that tests the basic requirements for your code. The unit tests should be runnable with your mockup code. Writing unit tests will also allow you to evaluate the usefulness of the code pieces as seen from an application point of view.

6. Replace the mockup functions in a prioritized order, i.e., write the must-have code before you write the nice-to-have code, while regularly running your unit tests to keep track of your progress.

7. Evaluate the code in relation to the desired goal, and reiterate earlier actions as needed until the task has been sufficiently completed.

8. Complete your documentation both in-code and outside to ensure that the intended user has sufficient knowledge to effectively use your program and to ensure that you or a fellow programmer will be able to maintain and extend the program in the future.