Lecture 2: The Main Programming Paradigms

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/02-main-programming-paradigms.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we briefly discuss the main programming paradigms, example of languages implementing them, and what are the kind of software engineering problems they are good at solving.

Main Programming Paradigms and Sub-Paradigms

Below are some of the main programming paradigms. They generally belong to one of the two high-level paradigms: - **Imperative**: - Structured (or procedural) - Object-Oriented - Concurrent - **Declarative**: - Functional - Concurrent These are the main ones, there are many more.

Imperative Programming Paradigm

As we saw in the JavaScript example from the previous slide deck, the **imperative** paradigm requires the programmer to describe the computation step by step, as a sequence of statements, a bit like a cooking recipe. Here is an example in x86-64 assembly language:

```
        .global _start
        .text
_start:
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi          # address of string to output
        mov     $14, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write
message:
        .ascii  "Hello, world!\n"
```

It's a hello world program in assembly No need to go into the details but in this program instructions are executed one after the other to print a message on the console. Assembly is purely imperative is very low level, so it has some nice benefits such as very good performance and low memory footprint. It is also the only way to realise some low level operations that are needed when one writes things like an OS or a virtual machine monitor.

Unstructured Languages

The problem with assembly is that it's not structured. In terms of control flow there is no clear notion of functions, loops, and so on. We get what is called spaghetti code. So even a medium-sized code becomes very hard to understand and reason about. Other examples of purely imperative languages are early versions of FORTRAN, COBOL, basic.

Imperative Structured Programming Paradigm

To be able to write bigger programs we need to use **structured imperative** programming. It includes control flow operations and the notion of procedures or functions. It makes it much easier to reason about large programs. Here is an example in C:

```c
int is_prime_number(int number) { /* Check if a number is prime */
    if(number < 2)
        return 0;
    for(int j=2; j<number; j++)
        if(number % j == 0)
            return 0;
    return 1;
}

int main(void) { /* Check which of the first 10 natural integers are prime */
    int total_iterations = 10;
    for(int i=0; i<total_iterations; i++)
        if(is_prime_number(i))
            printf("%d is a prime number\n", i);
        else
            printf("%d is not a prime number\n", i);
    return 0;
}
```

Once again the instructions within a function are executed one after the other. But we can see some control flow operations: loops, conditionals, and function calls. This program prints the list of the first 10 integers and which ones are prime or not. Structured imperative programming originally appeared in languages such as ALGOL, Pascal, ADA, etc. Today most programming languages are structured for obvious reasons, compared to unstructured programming:

- Their modularity makes it easier to write code for medium/large programs.
- There is Less complexity and the code is easier to understand.
- It also eases testing and debugging.

Modern low-level imperative structural languages without too many additional layers include C and FORTRAN. We will see that they are very efficient for HPC and systems software development. We will also see that they have memory safety issues.

Imperative Object-Oriented Programming Paradigm

With the **object-oriented** paradigm, the programmer links together the data and the code manipulating it into objects. While in non-object oriented languages one calls a function out of nowhere passing data as parameters. With object-oriented this function can be called upon a given data, for example here with the dot operator. Object-oriented also defines concepts such as inheritance and polymorphism that facilitates code reuse and organisation. Here is a C# program implementing these object types, that are actually named classes: shape, square and circle:

```c
// Example in C#
abstract class Shape {
    public abstract void printMe();
}

class Square : Shape {
    override void printMe() {Console.WriteLine ("Square id: {0}, side: {1}", _id, _side);}
}

class Cicle : Shape {
    override void printMe() {Console.WriteLine ("Circle id: {0}, radius: {1}", _id, _radius);}
}
```

```
public class MainClass {
    public static void Main(string[] args) {
            Square mySquare = new Square(42, 10);
            Circle myCircle = new Circle(242, 12);
            mySquare.printMe();
            myCircle.printMe();
    }
}
```

We create two objects and call the `printme` function on them. A few examples of object-oriented languages are C++, C#, or Java. Object-oriented languages are good to represent problems with a lot of state and operations, for example when we have to process a lot of data that can be described as attributes. For example simulators, video games, and so on. This paradigm also helps in organising and understanding large code bases but for small programs it can sometimes be overkill.

Imperative Concurrent Programming Paradigm

With the **imperative concurrent** paradigm the programmer uses various features provided by the language to describe parallel and interleaving operations. Here we have an example with a C library named the POSIX threads:

```
static void *thread_function(void *argument) {
    int id = *(int *)argument;

    for(int i=0; i<10; i++)
        printf("Thread %d running on core %d\n", id, sched_getcpu());
}

int main(void) {
    pthread_t threads[NUMBER_OF_THREADS];
    int thread_ids[NUMBER_OF_THREADS];

    for(int i=0; i<NUMBER_OF_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, &thread_function, &thread_ids[i]);
    }
    /* ... */
}
```

The important part is within the for loop where we create a certain number of threads. These threads are parallel execution flows and each will execute the function named thread function. A concurrent programming language describes in particular how the execution flows can communicate, can they share data structures, and if yes what is the consistency of that sharing. There are many methods for concurrent programming in imperative languages. They include shared memory threads and processes, message passing and automatic parallelization libraries, GPU programming languages, and so on. Important use cases for concurrent programming are of course high performance and distributed computing, graphic processing, etc.

Declarative Programming Paradigm

Contrary to imperative programming where the programmer describes a sequence of instructions, in other words the way to obtain a certain result, with **declarative** programming the programmer describes how the result should look like. A good example of a pure declarative language is HTML. It is used to construct web pages, declaring elements such as headings and paragraphs with tags. Declarative programming is at a much higher level of abstraction compared to imperative programming that is closer to the machine model. One can describe complex programs with few lines of code and the low level implementation details are abstracted. A downside is that the code can easily become convoluted and hard to understand in large programs. Some examples of pure declarative languages include: HTML, SQL, XML, CSS, Latex. Note that these are not Turing complete. Pure declarative programming is useful for document rendering, structured data storage and manipulation.

Declarative Functional Programming Paradigm

A very important declarative subparadigm is **functional** programming. With this paradigm, the programmer calls and composes functions to describe the program. Here is an example in Haskell:

```
add_10 x = x + 10
twice f = f . f
main = do
    print $ twice (add_10) 7
```

We define a function `add_10` that takes one parameter and adds ten to it. We also define another function that takes a function as parameter and applies it twice. The last line will print 27 as it first adds 10 to 7 and passes the result as parameter to a second invocation of `add_10`.

Functional Programming Features

With functional programming we have first-class/higher-order functions can be bound to identifier and passed as parameters or returned. Loops are implemented with recursion, which is well suited for some optimisations, as well as operations such as tree traversal problems, used for example when parsing. Pure functions have no side effects on global state. It is easier to understand and reason about what they do, it is also easier to proof, test and debug, because having less state means less chances for mistakes. Some example of functional languages are Haskell, Scala, F#, etc.

Declarative Concurrent Programming Paradigm

Some declarative languages also provide ways to describe concurrent operations. For example Erlang provides what is called the actor model. One of the advantages of this model is a reduction in synchronisation operations, for example there is no lock. Declarative concurrent languages are useful when designing distributed applications, web services, and so on.

Other Paradigms

There are many other programming paradigms: Logic, Dataflow, Metaprogramming/Reflexive, Constraint, Aspect-oriented, Quantum, etc. A lot more information can be found here: https://en.wikipedia.org/wiki/Programming_paradigm.

Multi-Paradigm Languages

A Language X is said to support paradigm Y if the language offers to possibility to program in the style defined by the paradigm. Many languages are multi-paradigm. For example Haskell is purely functional and does not allow OO style. OCaml is mainly functional but allow OO and imperative constructs. C, C++ and many others are imperative but allow some functional idioms. For example this is a recursive factorial implementation using C:

```
int fact(int x) {
    if(x == 0) return 1;
    return x * fact(x-1);
}
```

It all Boils down to Machine Code

A last thing to remember is that independently of the programming language used to build a program, the only language that the CPU understands is **machine code**, which is a binary representation of assembly. So in the end all programming languages compile or translate to machine code.

Summary

We have the two principal categories of programming paradigms, declarative and imperative. On the declarative side, a major subparadigm is functional programming, defined by several features including first class and higher order functions. Regarding imperative paradigms, we have procedural or structured programming enabled by control flow

structures such as loops and functions or procedures. Adding to this the notion of object we get the object-oriented paradigm. And adding the notion of concurrent execution flows we get the imperative concurrent paradigm.

COMP26020 Coverage of Programming Paradigms

Now let's see how the different parts of COMP26020 map to these paradigms:

- In Semester one we will start with imperative procedural programming in C.
- Then we will cover object-oriented programming with C++.
- Next we will see functional programming, with Haskell.
- After that, in Semester two, we will first study compilation, the process of transforming the textual sources of programs written in various languages into the machine code that is eventually executed on the CPU.
- Finally, we will see concurrent programming with a language named solidity.