Unit 1e: Declarative Approach

1 Introduction

The declarative approach to solving a problem involves a focus on results rather than processing; we are not concerned with how a result is achieved just the result. Often this transfers control of how results are achieved to a processor i.e. the processor decides how to move from input to output.

A variety of different ways of implementing this approach exist; some quite general e.g. functional programming, some domain specific e.g. logic programming.

The module will mainly focus on functional thinking and programming introducing a number of new programming concepts; large general-purpose languages now commonly provide functional programming support, we are going to concentrate on the use of Scala in the main.

Towards the end of the module we will explore other declarative approaches.

Declarative programming encompasses a number of other specific programming models, just as imperative programming does:

2 Declarative Paradigms

2.1 Logic

A program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain.

2.2 Domain Specific

A domain-specific language (*DSL*) is a computer language specialized to a particular application domain, usually declarative. These are “small languages”, focused on a particular aspect of a software system. Examples include markup languages such as HTML, software build

configuration languages, CSS, regular expressions – they typically rely on a general purpose language to complete a task.

2.3 Functional Approach

The functional approach involves composing the problem as a set of functions to be executed; you define carefully the input to each function, and what each function returns. There are (in principle) no changes of state (side-effects) or control flow.

The equivalent example to the imperative example (this is also in C# which supports functional programming to some extent):

```
result = mylist.Select(i => i + 1).Sum();
```

This takes a list, selects and applies a transformation to each element in the list (add 1) and applies a sum function to aggregate the results of applying that transformation. No variables change their values (*i => i + 1* defines a transformation, not an assignment statement).

An imperative approach focuses on how to perform tasks (algorithms) and how to track changes in state; a functional approach focuses on what information is desired and what transformations are required.

3 Functional Programming

The core of functional programming is the concept of a mathematical function i.e. a first class function

3.1 Function

In programming, a 'function' is a self-contained module (procedure) of code that accomplishes a specific task; functions provide a way to make code modular/procedural. In OO programming these belong to classes and are called methods.

In mathematics, the word 'function' has a more specific meaning:

> *A function is a relation, or mapping between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output*

For example:

$$\mathbf{f(x) = x^2}$$

maps any number (x) to the square of that number, e.g. **f(3) = 9**, **f(4) = 16**.

For given input x you always get the same result; you don't change x in process.

Functional programming is based on the mathematical definition.

3.2 Composition

Composition is a combination of individual units to perform a more complex task. In OO programming we compose objects to model a complex scenario, so classes are the building blocks of OO programming.

In functional programming, we can compose functions rather like the way mathematical functions can be composed. Functions are the building blocks of functional programming.

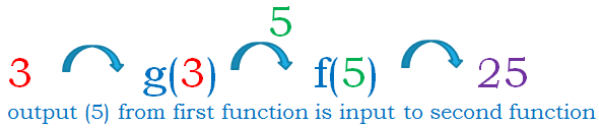
For example, mathematical functions **f**, **g** and **h** can be defined as

f(x) = x^2

g(x) = x + 2

h(x) = f(g(x)) - h is a composition of f and g

So h(3) = f(g(3)) = f(3+2) = f(5) = 25



What about (g(f(x))?

3.3 (Im)mutability and (no) shared state

Compare the following, more imperative, approach to the same computation

x = 3

x = x^2

x = x + 2

Here the computation is a series of steps which produces the same result as before, but:

- the value of the variable x changes, so x must be *mutable* (can change);
- the value of x is shared between the steps, both access the same variable and assume it will have the correct value for that stage of the computation - x is *shared state*

It is harder to be sure about the result - value of x could potentially be changed by other code in between these steps executing.

The functional approach favours *immutability* (*no side effects*) and *no shared state*. Each function does its evaluation based on the input it is given and doesn't care about what has happened previously - generates new value which it outputs and doesn't change the input.

3.4 Side effects

Side effects are operations that change the global state of a computation. Assignments and all input/output operations are considered side-effects. "Pure" functional programming is completely free from side effects.

However, it is impossible to write any real-world applications without any side effects as you could never write anything to the user, and you could never save anything to disk. We can get most of the benefits of functional approach while still allowing some kinds of side effects, e.g.

- input/output that does not affect the global state of the system, such as output to the screen;
- assignments to local variables that only exist within the scope of the function in which they are located and cease to exist when the function returns.

3.5 First class functions

The core value of functional programming is that functions should be *first-class*. They can be declared and invoked & can be used in every segment of the language as just another data type.