# Practical Work of Languages, Technologies, and Paradigms of Programming 2018-19 Part II Functional Programming

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Practice 6 - Prior-reading Material

Modules and Polymorphism in Haskell

## Contents

# 1 Modules

A Haskell program consists of a collection of modules. Each Haskell module can contain function definitions, data type definitions and type classes.

As previously explained, it is possible to import a module from another one by using the following syntax:

```
import ModuleName
```

that must appear before any function definition, usually at the beginning of the new module.

Module names are alphanumeric and start with uppercase. The contents of a module start with the reserved word `module`.

## 1.1 Importing modules

In order to import a module, it is required that the module name matches with the file name containing it (excepting the suffix `.hs`) when the file is in the same directory.

When the module is not in the same directory, we have to include the relative path (the sequence of directories) to find it.

For instance, in order to import module `ImportExample` from file in path `A/B/C/ImportExample.hs`, the import directive should be as follows:

```
import A.B.C.ImportExample
```

## 1.2 Exportation list

When a module is declared, we can optionally list what we want to export, which is the list of things that others can use from this module when importing it. This feature has the following syntax:

```
module Name ( list_of_things_to_be_exported ) where
```

For example, if we write the following module and we write it in the file `Sphere.hs`:

```
module Sphere (area, volume) where

  -- area of a sphere
  area :: Float -> Float
  area radius =  4 * pi * radius**2

  -- volume of a sphere
```

```
  volume :: Float -> Float
  volume radius = (4/3) * pi * radius**3

  -- area of a spherical lune (huso esferico, in Spanish)
  areaHuso :: Float -> Float -> Float
  areaHuso radius angle = (area radius) * angle / 360

  -- volume of a spherical wedge (cuña esferica, in Spanish)
  volumeCunya :: Float -> Float -> Float
  volumeCunya radius angle = (volume radius) * angle / 360
```

Programs importing this module (by means of `import Sphere`) will be able to use the exported functions (`area`, `volume`), but not the other functions defined in this module (`areaHuso`, `volumeCunya`).

### 1.3 Qualified Import

What happens when there are two modules with definitions that use the same identifiers? Let us consider the following example:

```
module NormalizeSpaces where
  normalize :: String -> String
  normalize = unwords . words
```

that uses the function `words` to break a string into a list of words (ignoring spaces, tabs, and return symbols) and the function `unwords` to put the string back. Now, let us suppose that there is another module `NormalizeCase` with another function with the same name:

```
module NormalizeCase where
  import Data.Char (toLower) -- import only function toLower
  normalize :: String -> String
  normalize = map toLower
```

If we import both modules at the same time, we will produce a name clash. To solve this problem, Haskell allows module importations using the reserved word *qualified* that makes identifiers imported from both modules to have a prefix with the name of the module:

```
module NormalizeAll where
  import qualified NormalizeSpaces
  import qualified NormalizeCase
  normalizeAll :: String -> String
  normalizeAll = NormalizeSpaces.normalize . NormalizeCase.normalize
```

# 2 Polymorphism in Haskell

## 2.1 Parametric Polymorphism

In previous sessions, we have used lists. The type of a list is `[a]` and it is an algebraic data type (with constructor symbols `[]` and `:`). A list is also polymorphic, since its data type `[a]` contains a type variable `a`.

A function is generic if its type contains type variales. For example, the function to calculate the length of a list, already described in previous sessions, has an implementation given for all possible data types for `a` (this kind of polymorphism is known as *parametric polymorphism*):

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

On the other hand, a universal definition of a function for all possible data types may be too generic in some cases. For example, the function for comparing two lists `==` requires that values stored in lists can also be compared using `==` (due to the expression `x==y` given below that, as we will see, will require to add a type constraint):

```
(==) :: [a] -> [a] -> Bool
[]     == []     = True
[]     == (x:xs) = False
(x:xs) == []     = False
(x:xs) == (y:ys) = x==y && xs==ys
```

To provide the constraint that actual types of the type variable `a` must admit comparison using `==`, Haskell uses *type classes*.

**Note:** Despite the use of the word *class* in this denomination, Haskell classes should not be confused with the notion of class from object-oriented programming languages. Types are not objects. Type classes group together or capture common sets of operations. In this way, when a type is an instance of a type class, we can guarantee that this type has the set of operations defined in the type class. As you can observe, this resembles more to the concept of Java interfaces than to the Java classes themselves. A type can be an instance of several type classes.

The Haskell type system allows parameterization for defining overloaded functions, imposing that types belong to type classes. For example, the class `Eq` represents types that have functions `==` and `/=` defined. The fact that `a` belongs to the type class `Eq` is denoted by `Eq a` and appears as a constraint "`(Eq a) =>`" when we define the type of the function `(==)`:

```
(==) :: (Eq a) => [a] -> [a] -> Bool
```

The restriction "`(Eq a) =>`" in the previous definition means: "for each type `a` that is an instance of the type class `Eq`, the function `(==)` has type `[a] -> [a] -> Bool`". That is, a type being an instance of a type class guarantees that the specified operations are included in the type.

Several examples and exercises with algebraic data types and type classes are shown in this practical exercise. We will see examples with parametric polymorphism and *ad-hoc* polymorphism (also known as *overloaded* polymorphism). For more information, you can consult several sources, including the web page:

`http://www.haskell.org/tutorial/classes.html`

In the definition of type classes in Haskell, there are no distinction between the methods of a class, as in Java with `public`, `private`, etc. Instead, the module system, with its capability to hide implementation details by means of exportation lists, is used for this purpose.