#6 Assignment - CMPT 383

Luiz Fernando Peres de Oliveira - 301288301 - lperesde@sfu.ca

I. Types

A type is a classification of data that helps us to abstract the real world [2]. In computer science, this abstraction allows scientists to prove that a given algorithm will work for all items (or objects) in a given set. A group of types is often called a **data structure**, for example *lists* (a vector containing multiple elements of same type) and *records* (a vector containing multiple elements of different types).

Depending on the target paradigm, types **might** be considered functions. The Functional paradigm, for example, is based on *Lambda Calculus* [3], a formal system in mathematical logic that is proved to be 100% able to represent any real-world abstraction such as objects, numbers and even **functions** themselves, that can be represented by morphisms[3]. In the (pure) Functional paradigm, everything is a function *"under the hoods"* by definition, hence my affirmation that types might be functions. Although, the same might not be true for other paradigms.

As said before, types are used to create new data structures. Not only that, types themselves are data structures, as they form abstractions of data sets. Data structures can be subdivided in two categories: primitives and compounds. Primitive types are the types that are predefined by programming languages (each one of them have their own primitive data types), such as **int** and **byte**. Compound data types are types that have been *"merged"* into one new structure and they can be made out of other compound data types as well as primitive data. A point (x, y) for example would be a kind compound data. A Java object would fall in the same category.

Going back to the kinds of abstractions related to types, we also have type signatures. A type signature is defined by the input parameters and output types of a function. The **C family** of languages such as C, C++, Objective-C and others might (not obligatory) use **headers** in order to make some function type signatures available externally to the rest of the application without exposing their function implementation. A header is a separate file containing a lot information and may have not only type signatures, but also type definitions necessary to use the functions within the implementation file. The header file has the power of making functions and structures **public**-like. In Java, for example, the header would contain information (or type signatures) related to **public** functions (or the functions that one would like to expose externally), while the other functions and data structures would only be available locally (**private**).

Alike programming languages, there are many kinds of databases such as relational, document-based, object-based, graph-based and others. The reason why we have many styles of databases is the same reason why we have different styles of programming languages, so called different paradigms. The differences among them are that each approach has their own theory. Relational databases, for example, use *Set Theory* as its main abstraction in order to solve given problems, while Graph databases use *Graph Theory*. In its turn, Document-based ones are designed for storing, fetching and handling document-oriented information and object-based bases its structure on objects (as its name suggests).

Returning to primitive types, it is important to distinguish three different data structures: *list*, *dictionary* and *object*. In **most** programming languages, lists and dictionaries are homogeneous data structures, while objects are heterogeneous data structures. Homogeneous data structures are those that have similar data structures. An example of that can be a list of *Users* in a system. The list as an indexed (each element has a defined index position i, where $i = 0, 1, 2, ...(n-1)$) homogeneous structure, will only keep data of same type. The same holds true for dictionaries, however, they are not indexed. Dictionaries are homogeneous structures that have a *key-value* pair for each element. Unlike lists, dictionaries do not have a specific order. A dictionary also cannot have 2 different keys, and for this reason, dictionaries fall under the category of sets. Nonetheless, objects are heterogeneous data structures, meaning that a object is a structure with multiple types (dissimilar data). Imagine an abstraction of a house, for example. A house has a colour (*string*), size (*number*) and other different structures that form a heterogeneous structure.

A *key-value* structure so called Dictionary in Haskell must have its keys as **strings**, however, it is possible to recreate this data structure with composition of a list of tuples in the format $[(key_1, value_1), (key_2, value_2), (key_3, value_3), ...]$, as a dictionary is nothing more than a list (or set) of tuples.

Lastly, types and data structures might be implemented (purely) in functional programming languages, such as *Haskell* and *F#*. Purely functional data structures are strongly **immutable** structures and not all programming languages are able to implement them, as it requires the programming language to have immutable structures (not the case of C language, for example). In this way, it is not possible to write purely functional data structures in languages that do not have access to immutable data. One point to mention is that a language that is able to implement purely functional data structures has much higher assertive code (usually proven mathematically) than a programming language that is not able to implement them. Moreover, languages with this kind of structure are able to implement memoization, lazy evaluation and concurrency easier as well. This type of structure (purely functional data structure) is strictly necessary for functional programming, once functional programming has the constraint of not changing states.

II. Shared State

In non-functional programming language is very common to have **global constants** (data structures which users can access at any time and anywhere in the program). Likewise, in functional programming languages, those global variables may be used without losing any aspect of its paradigm. This is only possible because functional languages are only functional when they use data structures that are not allowed to changed their content. Thus, it is possible to use global constants from a more "general" context in functional programming because constants do not change their state, holding true for all cases.

III. Paradigms

Clearly, there exists many different programming paradigms. Each one of the implement different computing theories (different ways of thinking). Functional programming, as said before, is completely based on *Lambda Calculus* and views every bit of a program as a function. Usually, functional programming languages implement all functions as **high order functions**, a mathematical function that takes a function as input and returns another function as output. It is usuallt necessary to implement pure high order functions in order for a language to be able to do **church enconding**. A **function** is a mathematical abstraction and often is referred as a *black box*. Please note that *functional programming functions* are not the same as *procedural programming function* (once these ones might have different formats as type functions are not the same as type signatures). Functional programming is also **declarative**, meaning that the code uses **declarations** that describes what the user wants rather than describing the actions step-by-step (imperative approach). Moreover, these actions do not need to happen sequentially. On the other hand, a procedural programming is derived from the **imperative paradigm**. Procedural programming uses routines (or **procedures**) based upon the concept of *procedural call*. Unlike functional programming functions, procedural programming functions (or procedures) do not need to take functions as parameters and do not necessarily need to return functions as their output. They have also use the concept of **type signature** explained earlier. Procedural programming uses imperative statements in order to solve a given problem, meaning that one needs to describe all the solution steps to the machine and they often happen sequentially.

Additionally, functional programming does not include a keyword **void**. These concept is unknown to the paradigm, because a *"functional"* function always return a value (which is also a function). However, it is possible to return a function that returns a function with no values, so called **unit**. An unit is a function in the format $a \to ()$, which is equivalent to the *void* type on imperative languages.

Speaking now of **concurrent programming** and **event-driven programming**, it is known that both can be outlined as many (at least two) processes running throughout the same period of time (processes do not need to run in parallel), however, on concurrent programming (not to confound parallel programming), the execution does not need to happen at the same instant, thus, it consists of processes that overlap their lifetime. The main objective of concurrent programming is to model processes that happen concurrently. Many clients fetching data in the same time span from a server would be a good example of how that works. On the other side, **event-driven programming** controls the program flow by using event listeners, the occurrence of these events are observed (or monitored) and executed by an, so called, event handler, which is typically a callback (delegate function) or method that determines the course of the program through user actions such as clicks or key presses or through messages sent by other threads and/or programs. It is very important to notice that both concurrent and event-driven programming can happen simultaneously and can add parallel programming.

Programming paradigms and **design patterns** are quite different. Programming paradigms are styles or approach of programming, as explained before, they are the proven mathematical theory behind the programming languages. Design patterns are not quite the same, they are usually best practices of how to write code. One can understand better the differences between programming paradigms and design pattern with the facts that the user will always be constrained to the programming paradigm(s) of a language in order to solve a given problem, but the same is not true for design patterns, which are up to the user to use a X, Y or No design patterns. Moreover, design patterns is related to quality of code, while paradigms is related to the way of solving a problem.

IV. Programming Questions

1. Create an Object data structure in **C**.

```c
typedef struct _Point {
  float x;
  float y;
} Point;
```

2. Create an Object data structure in **Haskell** using functions.

```haskell
class Point a where
  coord :: a -> (Float, Float)
```

3. Implement recursion in **C**. (recursive function that count # elements in an array)

```
int arr_count(int *arr, int c)
{
  if (!arr[c]) {
    return c;
  }

  return arr_count(arr, c+1);
}
```

4. For the following functional languages (Haskell, Scheme, Scala, Clojure, F#), implement a function's type signature and function definition.

- Haskell

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

- Scheme

```
(define applyTwice
  (lambda (f x)
    (f (f x))))
```

- Scala

```
def applyTwice[A](f:A => A, x:A): A
    = { f(f(x)) }
```

- Clojure

```
let [applyTwice f(f x)]
```

- F#

```
let applyTwice (f:a->a) (x:a) =
   f(f x)
```

5. Repeat #4 for the following non-functional languages (without the requirement that the parameters must be a function) Python, C, Java, Go, Objective C.

- Python

```
def add(x, y)
  return x + y
```

- C

```
int add(int x, int y) {
  return x + y;
}
```

- Java

```
static class Math {
  int add(int x, int y) {
    return x + y;
  }
}
```

- Go

```
func add(x int, y int) int {
        return x + y
}
```

- Objective-C

```
- (int) addX:(int)x andY:(int)y {
  return x + y;
}
```

References

[1] "Function Types", Bartosz Milewski's Programming Cafe, 2017. [Online]. Available: https://bartoszmilewski.com/2015/03/13/function-types/. [Accessed: 29- Nov- 2017].

[2] "Data type", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Data_type. [Accessed: 29- Nov- 2017].

[3] "Lambda calculus", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Lambda_calculus. [Accessed: 29- Nov- 2017].

[4] "What are primitive and non-primitive data types?", Pc.net, 2017. [Online]. Available: https://pc.net/helpcenter/answers/primitive_and_non_primitive_data. [Accessed: 29- Nov- 2017].

[5] "Type signature", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Type_signature. [Accessed: 29- Nov- 2017].

[6] 2017. [Online]. Available: https://www.quora.com/Why-are-there-so-many-types-of-SQL-What-is-the-difference-between-all-of-them. [Accessed: 29- Nov- 2017].

[7] "difference between list and dictionary? - CodeProject", Codeproject.com, 2017. [Online]. Available: https://www.codeproject.com/Questions/365134/difference-between-list-and-dictionary. [Accessed: 29- Nov- 2017].

[8] "Whats the equivalent of a Python Dictionary in Haskell /haskellquestions", reddit, 2017. [Online]. Available: https://www.reddit.com/r/haskellquestions/comments/2xn9y2/whats_the_equivalent_of_a_python_dictionary_in/. [Accessed: 29- Nov- 2017].

[9] "Purely functional data structure", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Purely_functional_data_structure. [Accessed: 29- Nov- 2017].

[10] "Comparison of programming paradigms", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_programming_paradigms. [Accessed: 01- Dec- 2017].

[11] "Procedural programming", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Procedural_programming. [Accessed: 01- Dec- 2017].

[12] W. [closed], "What is the difference between declarative and imperative programming", Stackoverflow.com, 2017. [Online]. Available: https://stackoverflow.com/questions/1784664/what-is-the-difference-between-declarative-and-imperative-programming. [Accessed: 01- Dec- 2017].

[13] W. [closed], "What is the difference between declarative and imperative programming", Stackoverflow.com, 2017. [Online]. Available: https://stackoverflow.com/questions/1784664/what-is-the-difference-between-declarative-and-imperative-programming. [Accessed: 01- Dec- 2017].

[14] "Threads VS Events", Courses.cs.vt.edu, 2017. [Online]. Available: http://courses.cs.vt.edu/cs5204/fall09-kafura/Presentations/Threads-VS-Events.pdf. [Accessed: 01- Dec- 2017].

[15] V. Hancock, "What is the Difference Between a Programming Paradigm and a Design Pattern? All Things JavaScript", Allthingsjavascript.com, 2017. [Online]. Available: http://allthingsjavascript.com/blog/index.php/2017/04/20/what-is-the-difference-between-a-programming-paradigm-and-a-design-pattern/. [Accessed: 01- Dec- 2017].

[16] "Learn X in Y Minutes: Scenic Programming Language Tours", Learnxinyminutes.com, 2017. [Online]. Available: https://learnxinyminutes.com/. [Accessed: 01- Dec- 2017].