***Hamdi Mohamed Amine Formal Languages And Compilers Politecnico di Torino s301247***

***Swift Language:***

Swift is a powerful language used mainly in the Apple ecosystem, Swift is a compiled language but has the simplicity and readability that is typical of scripted languages.

Swift is easy to learn and provides tons of optimizations during the compiling phase, it guarantees a lot of error checking in the semantic analysis such as initialization before use of variables or overflow checking, and also through a lot of checks Swift guarantees a high level of memory safety,so it is easier for the programmer to manage memory.

Differently from JavaScript and other scripting languages Swift is ***type-safe*** and doesn’t let the programmer use a variable or a constant before its initialization.

To assure the type safety a variable or constant is declared (and also initialized in some case) in the following two ways:

**TYPE ANNOTATION WAY:**

**let/var id1,id2,…,idN : type (= value) ->value is optional**

**DIRECT INITILIZATION WAY:**

**let/var id1=value1,id2=value2,…,idN=valueN**

In this way the compiler can infer the type if we directly initialize a variable or a constant or if we use the type annotation way get directly by us if the type was not clear maybe, so the next time if we try to initialize or change the value of a variable we control that the new type is conformant to the old one.

To declare a constant we use the let keyword, this means initializing or declaring an immutable, in case of objects, if we declare it with the let keyword the object can change but the reference must remain the same, this can become also a good practice for programmers(it is similar to the concept of const in Javascript).

Otherwise to declare a variable we use the var keyword.

The main types are *Int, Double, Float, Bool, String,* on top of these we can also obviously build arrays, sets and dictionaries collections.

Optionals are an important part of the language, they are heavily used, to define them we use the ? and to access properties of optionals we use the ! unwrapping operator, or the ?? operator that does the ternary check x!=nil ? x : defaultValue.

Arrays are a collection of a series of values of the same type(or also of heterogenous types if we allow type inference with [Any], in some cases if there are multiple types that can be directly casted there is a cast to the parent type):

**let/var id=[13,14.5,15] -> if we do print(id[0]) we obtain 13.0 because the Int has been casted to Float in this case**

**let/var id : [Any] = [1,4.4,false,”Hello World!”,35] -> in this case if we do print(id[0]) we obtain 1, this is because we generated a heterogenous array.**

Other types can be defined with classes, structures and enumerations.

Classes are reference types, while structures and enumerations and all the other primitives are value types, that means that a class instance can be pointed to by many variables, while a structure is copied every time it is called in a function or used.

In fact if we do:

**class C{**

**var x=0;**

**}**

**struct S{**

**var y=1;**

**}**

**var cl=C();**

**var cl2=cl;**

**cl2.x=140;**

**print(cl.x) -> we get 140 because cl and cl2 refers to the same instance**

**var str=S();**

**var str2=str; -> a copy of str is created and assigned to str2**

**str2.y=13;**

**print(str.y) -> we get 1 because str and str2 refer to two different structures**

Another interesting part of the Swift language is the memory management system, for reference types, so for classes, the memory management is pretty much automatic, an instance is deallocated only when no strong reference to that instance remains, a strong reference to an instance is generated every time we assign a variable to a class.

Because of this we could generate memory leaks whenever we have a cycle of strong references, and so causing two instances to never release memory.

**class Course{**

**var prof=Professor?;**

**}**

**class Professor{**

**var course=Course?;**

**}**

**var prof=Professor();**

**var course=Course();**

**prof.course=course;**

**course.prof=prof;**

In this example we generate a strong reference cycle and these instances will create a memory leak, to resolve these types of issues we have the definition of weak and unowned references, weak references are used when the other instance has a shorter lifecycle, while we use unowned references when the other instance has the same or higher lifecycle.

A weak reference must always be set to an optional variable, because when the instance it refers to is deallocated the variable is set to ***nil***, instead unowned references always expect to have a value.

Swift as other modern programming languages provides us with the possibility to use functional programming, in fact functions can be assigned to variables and are treated just as another type.

Function in Swift are also interesting because by default the passed parameters can’t be changed because they are treated as a constant, otherwise the compiler will generate an error, to have the possibility to modify parameters of a function there is the concept of ***inout*** parameters, these parameters are mutable, passing a constant as an inout parameter is wrong.

**func swap(\_ a: inout Int, \_ b: inout Int){**

**let temp=a;**

**a=b;**

**b=temp;**

**}**

To use inout parameters we have to use the ***&*** operator, so we can call swap in this way:

**var a=10,b=99**

**print("A equals \(a), while B equals \(b)"); -> A equals 10, while B equals 99**

**swap(&a,&b)**

**print("A equals \(a), while B equals \(b)"); -> A equals 99, while B equals 10**

Functions can have the possibility to have arguments labels or not, an argument label is the name used for a parameter while calling the function, an argument label is assigned to the parameter name if it is not assigned, while if the ***\_*** operator is put before the parameter name we don’t use any argument label.

So a function like the following:

**func tryArgLabel(\_ par1: String,argLabel2 par2: String, par3: String) -> String {**

**return par1+par2+par3;**

**}**

**print("Example about argument labels returned \(tryArgLabel("Hello ",argLabel2:"World!",par3:" ;)"))")**

contains every type of argument label usable, none, one, default by parameter name.

The use of argument label is another example of how Swift is assertive and readable.

Additionally Swift provides a complete access level management, with various types of access level, from open to private: ***open, public, internal, fileprivate, private***.

Open and public are quite similar, but open applies only to classes and its members, and it used to allow code outside that module to subclass and override.

Internal is the default one and it is similar to the protected access level in Java, so an internal property is visible inside the whole module.

While as the name suggests fileprivate means that a property is visible only inside a source code file, and finally private is associated to properties visible only inside that scope, so inside that class, structure or enumeration.

For tuples the access level is given to the most restrictive type of that tuple, so between public and private the latter one is chosen, the same concept is applied to functions access level.

Lastly another interesting feature in Swift are the opaque types, they are used when the function doesn’t want the caller to know the return type, to do that we use the some keyword like in this example:

**func launchOpaqueFighter() -> some Fighter {**

**return Obj;**

**}**

Summing up Swift has a lot of other features from modern languages such as C++ Java and Javascript, such as Generics, inheritance, interfaces, lambda functions, nested functions with closures etc., and it combines them with simplicity and security.

Here is an example of the bubble sort algorithm implemented in Swift:

**var data\_set = [1,3,6,2,4,5]**

**var last\_position = data\_set.count - 1**

**var swap = true**

**while swap == true {**

**swap = false**

**for i in 0..<last\_position {**

**if data\_set[i] > data\_set[i + 1] {**

**let temp = data\_set [i + 1]**

**data\_set [i + 1] = data\_set[i]**

**data\_set[i] = temp**

**swap = true**

**}**

**}**

**}**

**print (data\_set)**

**Implemented functionalities:**

* basic operators
  + addition
  + subtraction
  + multiplication
  + division
  + and
  + or
  + xor
  + not
  + unwrap of optional(!):
* functions
  + argument labels
  + inout parameters
* types
  + Basic data types
    - Int
    - Double
    - String
  + Collections
    - Arrays
    - Dictionaries
  + Optionals
* Variables
* Constants
* Classes
* Generics
* control statements
  + for in loop
  + while
  + if statement
* output
  + print()