كليـــــة ....

كونترول الفرقة ....

العــام الجامعــى 2019 / 2020 – دور مايـــو

الغلاف الخارجى للبحث

أولاً: البيانات الخاصة بالطالب

الفرقة الدراسية	الثانية منقول	التخصص	عام
اسم القسم	عام		
اسم المقرر	لغات الحاسب – 3 | 3- programming languages		
استاذ المقرر	د/مروة عبد الفتاح		

ثانياً: البيانات الخاصة بالبحث

عنوان البحث	Comparing between Haskell and Scala			
طبيعة المشاركة	بحث فردى	✓	بحث جماعى	
ارسال البحث	بواسطة البريد الالكتروني			
اسماء الطلاب المشاركين فى البحث (يكتب الاسم رباعيا)	م	الاسم رباعى	رقم الجلوس	الرقم القومى
	1	سيف هشام سالم سيد	20180284	30008042100434
	2			
	3			
	4			
	5			
تاريخ الإرسال	1 / 6 / 2020			

ثالثاً: البيانات الخاصة بالكونترول

النتيجة	ناجح		راسب	
أعضاء لجنة تقييم البحث		الاسماء	التوقيع	
	1			
	2			
	3			

فى حالة عدم قبول البحث يرجى ذكر الأسباب	- ....
	- ....
	- ....
	- ....

Haskell and Scala, A brief comparison!

Introduction:

- Haskell is a statically typed programming language with strictly functional programming functionality with lazy evaluation and inference styles.

  - This was created and developed by Lennart Augustsson, John Hughes, Paul Hudak, John Launchbury, Philip Wadler, Simon Peyton Jones, and Erik Meijer.
  - The typing discipline is solid, static, assumed, and does not have strict Semantics.
  - It had been licensed under the BSD certificate of Clause 3. Its characteristics are lazy programming and is non-strict and flexible. The first release was in 2010.

- Scala is also a general-purpose programming language, provides the features of functional programming and had a strong static type system which had object-oriented features.
  - Scala was written and created using Java, and it uses Java virtual machine (JVM) to compile its source codes and it runs on the JVM.
  - So that it gives it a great advantage to run anywhere. It was licensed under Clause 3 BSD license.
  - It was designed by Martin Odersky, Lex Spoon, and Bill Venners. It was developed in "EPFL" in Switzerland.

So, we are going to discuss differences between Haskell and Scala to show up the differences between them.

In Haskell:

- Variables, Datatypes, and Functions: while the syntax is strong and static so every expression and function in Haskell has a *type*. For example, the value `True` has the type `Bool`, while the value "`foo`" has the type string.

  Static type system means that the compiler knows the type of every value and expression at compile time before any code is executed.

  Type inference means that compiler can automatically deduce the types of almost all expressions in a program.

  Some of basic datatypes in Haskell: `Char, Bool, Int, Integer, Double` and there is another useful composite data types like `lists` and `Tuples`

```
Prelude> :type “a”
"a" :: [Char]
```

```
Prelude> :type [1,2,3,4]
[5,2,4] :: Num a => [a]
Prelude> head [1,2,3,4]
1
```

```
Prelude> :type (99, 88)
(99, 88) :: (Num a, Num b) => (a, b)
Prelude> fst (99, 88)
99
```

To Define functions in Haskell it easy, we type first the type of the function and its arguments and body of function is written in that way:

```
square :: Integer -> Integer
square x = x * x
```

- Pattern matching: consists of specifying patterns to which some data should comply and checking to see whether they do, deconstructing the data according to these patterns and defining separate function bodies for different patterns. You can design any form of data starting with the numbers, characters, lists, tuples and so on, one of most useful examples in how to use pattern matching.

```
-- To sum vectors, it very easy to perform using patter matching
AddVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
AddVectors (a1, b1) (a2, b2) = (a1 + a2, b1 + b2)

-- Another example in how to use pattern matching
SumList :: [Int] -> [Int]
SumList (x:xs) = x + sumList xs
SumList []     = 0
```

```
Note: Haskell programmers use the (x:xs) pattern often, especially with
recursive functions. However, patterns that include the (:) character will match
only against lists of length one or more.
```

- **List comprehension:** means in general creating a list from another one or more other lists, list comprehension is a solution for the problem of no-loops in Haskell -because it is pure functional- so it's the only way to loop in the data structure of lists, example:

```
Prelude> [(a, b) | a <- [1, 2], b <- "abc"]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]
```

- **Higher Order Functions:** Haskell functions may take functions as return values as parameters and functionality. The higher order function is a function that does one of these. More organized functions are not only a part of Haskell 's life, they are almost the reality of Haskell. If you choose to make equations by means on what stuff were, it turns out that higher order functions are invaluable, rather than specifying steps to alter any state and probably loop them. They are a very powerful way to solve problems and reflect on programs, examples on higher order functions:

```
multiplyThree :: Int -> (Int -> (Int -> Int))
multiplyThree x y z = x * y * z

Prelude> map (+3) [1,2,3,4]
[4,5,6,7]
```

- **Functions as values:** In an imperative language, appending two lists is cheap and easy, a simple C structure in which we maintain a pointer to the head and tail of a list.

```
struct list {
      struct node *head, *tail;
};
// This is example C structure to clarify the idea.
```

When we have one list and want to append another list onto its end, we modify the last node of the existing list to point to its head node, and then update its tail pointer to point to its tail node.

Obviously, this method is kind of not available in Haskell. As, the data is immutable, so, we can not modify the lists declaration like this, so we will use (++) sign which concatenate two lists together.
Lambda expression is important appliance on functions as values and it could be defined as, examples:

```
Prelude> :type (++)
(++) :: [a] -> [a] -> [a]

-- concatenation sign redeclaration.
(++) :: [a] -> [a] -> [a]
(x:xs) ++ ys = x : xs ++ ys
_ ++ ys = ys

Prelude> map (\(a,b) -> a * b) [(1,2), (3,4), (5,6), (3, 8)]
[2,12,30,24]
```

- Tail Recursion: In an imperative language, a loop executes in constant space. Lacking loops, we use tail recursive functions in Haskell instead. Normally, a recursive function allocates some space each time it applies itself, so it knows where to return to.

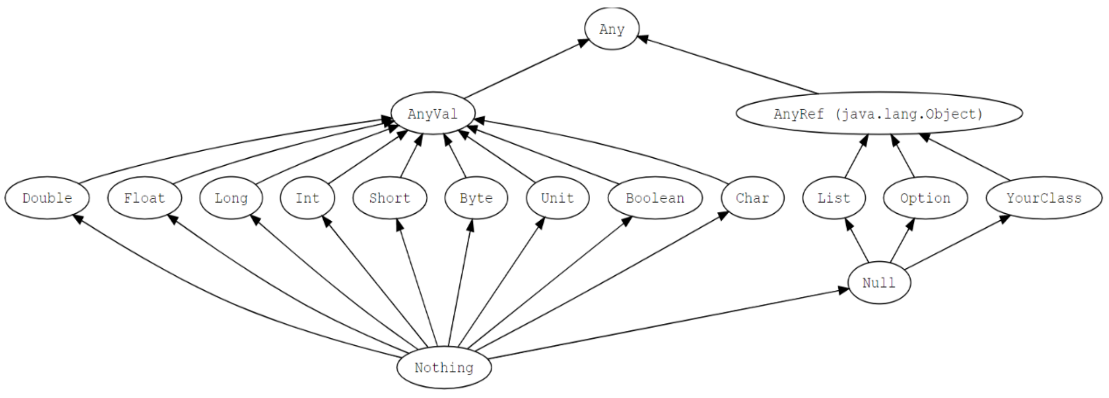
```
factTR :: (Integral a) => a -> a
factTR x = fact' x 1 where
  fact' 0 y = y
  fact' x y = fact' (x-1) $! (x * y)
```

- IO in Haskell:

In Scala:

- Variables, Datatypes, and Functions: While Scala is a blend of object-oriented and functional programming concepts and a statically typed language, so that is making Scala's functional programming construct make it easy to build interesting things quickly from simple parts, and makes it easy to structure larger systems and to adapt the new demands.

  Scala datatypes are the same with Haskell like:
  `Nothing, Double, Float, Long, Int, Short, Byte, Unit, Boolean, Char, Null, Lists, Options, Classes, Any`



Scala-lang.org

Variables are made in Scala into that way:

```
scala> var sayhi = “Hello World!!”
val sayhi:String = “Hello World!!”
scala> val PI:Double = 22/7.0
val PI: Double = 3.142857142857143
```

Note: the number 7 is typed 7.0 to cast it into Double datatype and all datatypes in both languages are capitalized.

And to define functions in Scala we will type, in example to find maximum number between two numbers:

```
scala> def max(x: Int, y: Int): Int = {
      | if (x > y) x
      | else y }
def max(x: Int, y: Int): Int
scala> max(5, 7)
val res0: Int = 7
```

- Pattern matching: is like that one in Haskell, but we will show that by examples to see what differs from one to another and we will use the same examples in Haskell section.

```
// To find sum of using pattern matching
scala> val vectors = List(((1,2), (3,4)))
val obj: List[((Int, Int), (Int, Int))] = List(((1,2),(3,4)))
scala> vectors.map { case ((a, b),(c, d)) => (a+c, b+d)}
val res1: List[(Int, Int)] = List((4,6))

// To find sum of list using pattern matching
scala> def sumList(xs: List[Int]): Int = {
     | if (xs.isEmpty) 0
     | else xs.head + sumList(xs.tail) }
def sumList(xs: List[Int]): Int
```

- List comprehension: in Scala, there is no list comprehensions like Haskell, unlike Haskell, Scala is bind between Functional programming and imperative Programming so there is loops as known: While, For, and Do While loops, That's makes a good point to Scala, because the time complexity and dealing with a lot of problems, GCD example:

```
scala> def gcd(x: Long, y: Long): Long = {
     | var a = x
     | var b = y
     | while (a != 0)
     | {
     |      val temp = a
     |      a = b % a
     |      b = temp
     | }
     | b}
def gcd(x: Long, y: Long): Long
```

- Higher Order Functions: All functions are separated into common parts, which are the same in every invocation of the function, and non-common parts, which may vary from

```
scala> val salaries = Seq(Functions as values:
20000, 70000, 40000)
val salaries: Seq[Int] = List(20000, 70000, 40000)

scala> val doubleSalary = (x: Int) => x * 2
val doubleSalary: Int => Int = $Lambda$1075/1329589315@38dbeb39

scala> val newSalaries = salaries.map(doubleSalary)
val newSalaries: Seq[Int] = List(40000, 140000, 80000)
```

- Functions as value: has the meaning on the Haskell the different is the syntax of the code because it is a concept of Functional programming:

```
scala> val l = List(1, 1, 2, 3, 5, 8)
val l: List[Int] = List(1, 1, 2, 3, 5, 8)

scala> val res = l.map( (x:Int) => x * x )
val res: List[Int] = List(1, 1, 4, 9, 25, 64)
```

- Tail Recursion:
- Power of 'OO' in Scala:
- cluster computing in Scala: