CSCI 3155 Spring 2015 Final Exam

1. Some programming languages are typeless. What are the obvious advantages and disadvantages of having no types in a language?

Typeless languages are much more flexible and allows for any variable to be used for any type value. However, this typeless nature also allows for poor reliability and ambiguity. Type errors can more easily occur and there is no sort of type checking to detect these errors.

2. What are the arguments for and against representing Boolean values as single bits in memory?

Boolean values represented as single bits in memory allows for greater memory efficiency, conservation of memory, and can save a lot more space. However, representing Booleans in a single bit requires processors to have addresses for the single bits, and thus more operators are involved to use that Boolean. This means we must use bit-wise arithmetic to separate Boolean value from the byte, and although this conserves memory, it will take more processing time.

3. How does a decimal value waste memory space?

Binary coded decimal values store numeric information as digits encoded using the four bit binary equivalents, 0 (0000) to 9 (1001). That means a single byte can hold values between 0 and 99. However, simply using the same byte to hold a binary value will yield values between 0 and 255, or –128 and +127.

4. Dynamic type binding is closely related to implicit heap-dynamic variables. Explain this relationship.

Implicit heap dynamic variables are bound to a type at runtime when a value is assigned to a variable.

Dynamic type binding is the binding of a type to a variable at runtime or changing the type of a variable during runtime.

Implicit heap dynamic variables use dynamic type binding.

5. What are the primitive data types in Scala?

There are **no** primitive data types in Scala. All data types, such as a Boolean, byte, short, char, int, long, float, double, string, unit, null, and others that are primitives in Java are objects in Scala. The means that all the data types in Scala possess callable methods.

6. You are given the following Scala code:

```
class C(x: Int) {}
class D(x: Int) {}
c = d
```

What is the problem with the code?

Both lowercase c and d are undefined. However, if c was an instance of the class C, for example if c was first set to c = new C(1), and d was an instance of class D, then setting c to equal d will result in a type mismatch error. Both are objects of separate types, since in most languages class declarations create new separate types.

7. You are given the following Scala code:

```
val numbers = List(1, 2, 3, 4)
numbers.map((i: Int) => i * 2)
numbers.foreach((i: Int) => i * 2)
```

Why do the map and foreach calls produce different results?

> The method call map evaluates a function over each element in the list, returning a list with the same number of elements. The map method iterates over a list, transforms each member of that list, and returns another list of the same size with the transformed members (such as converting a list of strings to uppercase)
>
> The method call foreach behaves similarly to map, but is intended for side-effects only and will return nothing. The foreach method iterates over a list and applies some operation with side effects to each list member (such as saving each one to the database for example)

8. You are given the following Scala code:

```
import scala.language.reflectiveCalls
def foo(x: { def get: Int }) = 123 + x.get
foo(new { def get = 10 })
```

What is the output of these commands?

> 133

What kind of **type compatibility** is implicit in this?

> These are structural types that need reflection. A structural type is a type with Parents where the declarations contains new members that do not override any member in the Parents. To access one of these members, a reflective call is needed. Structural types provide great flexibility because they avoid the need to define inheritance hierarchies theoretically. Their definition falls out quite naturally from Scala's concept of type refinement. Reflection is not available on all platforms. Even where reflection is available, reflective dispatch can lead to surprising performance degradations.

9. You are given the following Scala code:

```
class someDS[A] {
      private class Node[A] (elem: A) {
            var next: Node[A] = _
            override def toString = elem.toString
      }
      private var head: Node[A] = _
      def add(elem: A) {
            val n = new Node(elem)
            n.next = head
            head = n
      }
      private def printNodes(n: Node[A]) {
            if (n != null) {
                  println(n)
                  printNodes(n.next)
            }
      }
```

```
    def printAll() { printNodes(head) }
}
```

a) What type of data structure does this code implement?

This is a linked list data structure.

b) What data type(s) does this class accept?

This class accepts the generic type A, an element. Meaning if I set a value to new LinkedList[String](), it will be a linked list of strings. The class can accept any types, such as ints, floats, chars, or even itself.

c) Give an example of how you would use it.

```
val strLinkedList = new LinkedList[String]()
ints.add("donuts")
ints.add("waffles")
strLinkedList.printAll()
```

10. What type of operator overloading does Scala permit?

Scala doesn't technically permit operator overloading, since there is technically no operators within the Scala language. Every operator, along with what are believed to be primitive data types, are actually objects. So unary and binary operators like not and plus are objects, and this means they possess methods. Thus, to 'overload' an operator in Scala, one must simply define a class object method for the binary operator or a function that will handle operations in multiple manners depending on the pass parameters.

11. In Scala operators are actually methods. Provide an example of a method using traditional dot notation and its equivalent as an operator.

Dot notation:
```
60.+(9)
```

Operator notation:
```
60 + 9
```

Both will call the + operator method and add 60 and 9 together. However the dot notation form allows you to properly see that + is a method within some class.

12. In Scala, match is used instead of switch. Why is this advantageous in comparison to how switch is implemented in C/C++? Provide at least two reasons.

Pattern matching allows you to match a value against several cases, sort of like a switch statement in C/C++. However, instead of just matching numbers, which is what switch statements are only able to do, you can match what are essentially the creation forms of objects. Thus, it is not restricted to just primitives and other types that C/C++ have. Pattern matching can also extract variable values the match statement is comparing.

13. Why does the following code fail in Scala and how would you fix it?

```
val list = 1::2::3
```

This list lacks an ending, which means it will output an error. To fix this issue, simple add another element to the list at the end, with a null (nil) value: `val list = 1::2::3::Nil`

14. Scala does not include the keywords **break** and **continue**.

a. Why not?

They are slightly imperative, which means that it is better to simply use many smaller functions or recursion to acquire the same results much more effectively than the two excluded key words. There are also issues with how they interact with closures, which can become problematic. Both of these two reasons means that break and continue In Scala are unneeded and obsolete.

b. How does Scala provide equivalent functionality?

Scala's versatility has an importable utility where you can implement break and continue. Simply add `import util.control.Breaks._` to the Scala project. This means equivalent functionality is provided purely through support libraries. However, one can also create objects with methods of break and continue with the same functionality.

15. Explain why it is difficult to eliminate functional side effects in C.

All functions in C are subprograms and each of them return only one value, although they can return an array. In order to return more than one single value, pointer actual parameters must be used. If the parameters are pointers, then there is high susceptibility in creating functional side effects. This problem can change the value of a variables and may affect the program.

16. Scala provides closure functionality.

a. Define what a closure is and provide a Scala example.

A closure is a persistent scope which holds on to local variables even after the code execution has moved out of that block. So, essentially, closure is a subprogram and the referencing environment where it was defined. It is a function whose return value depends on the value of one or more variables declared outside this function. A simple example in Scala would be:

```
var toAdd = 69
val doSomeAddition = (someNum:Int) => someNum + toAdd
```

b. What conditions are necessary for a language to allow for closures and why is this so?

If a static-scoped programming language does not allow nested subprograms, in addition to local and global variables, closures are not useful, so such languages do not support them. Therefore, nested subprograms and scope are necessary for closure.

17. Briefly describe the three methods used to provide synchronization in the context of concurrency.

Semaphore is used to provide cooperation synchronization and is a data structure that consists of an integer and a queue storing task descriptors. To provide limited access to a data structure, guards can be placed around the code that accesses the structure. A guard is a linguistic device that allows the guarded code to be executed only when a specified condition is true.

Monitors are to encapsulate shared data structures with their operations and hide their representations—that is, to make shared data structures abstract data types with some special restrictions. This solution can provide competition synchronization without semaphores by transferring responsibility for synchronization to the run-time system.

Message Passing can be either synchronous or asynchronous. The basic concept of synchronous message passing is that tasks are often busy, and when busy, they cannot be interrupted by other units. The alternative is to provide a linguistic mechanism that allows a task to specify to other tasks when it is ready to receive messages.

18. What types of synchronization do Java and Scala provide? What are the advantages to Scala's approach?

The concurrent units in Java and Scala are methods named run, whose code can be in concurrent execution with other such methods, of other objects, and with the main method. The process in which the run methods execute is called a thread. The priorities of threads need not all be the same. A thread's default priority initially is the same as the thread that created it.

There are packages that define the Semaphore class in Java and Scala. Objects of this class implement counting semaphores. A counting semaphore has a counter, but no queue for storing thread descriptors. The Semaphore class defines two methods, acquire and release, which correspond to the wait and release operations

Java and Scala methods, but not constructors, can be specified to be synchronized. A synchronized method called through a specific object must complete its execution before any other synchronized method can run on that object. Competition synchronization on an object is implemented by specifying that the methods that access shared data are synchronized.

Cooperation synchronization in Java and Scala is implemented with the wait, notify, and notifyAll methods, all of which are defined in Object, the root class of all Java/Scala classes. All classes except Object inherit these methods. Every object has a wait list of all of the threads that have called wait on the object.

Java and Scala include some classes for controlling accesses to certain variables that do not include blocking or waiting. There are packages that define classes that allow certain nonblocking synchronized access to int, long, and boolean variables, as well as references and arrays.

19. Scala uses Actors to implement concurrency. (Note that the latest versions of Scala use the Akka actor library but the set-up is a bit more involved so I stuck with the Actors library because it is covered in the 2nd edition of the Odersky book.) Below is a simple version of a program that creates two actors and runs them concurrently.

```
package actors
import scala.actors._
object SillyActor extends Actor {
      def act() {
            for (i <- 1 to 5) {
            println("I'm acting!")
            Thread.sleep(1000)
            }
      }
}
object SeriousActor extends Actor {
      def act() {
            for (i <- 1 to 5) {
                  println("To be or not to be.")
                  Thread.sleep(1000)
            }
      }
}
object Actors {
      def main(args: Array[String]) {
            SillyActor.start()
            SeriousActor.start()
      }
}
```

a. Run this example and list its output.

```
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
```

b. Comment the code explaining what is happening.

```
// Part of a package, called actors
package actors
// Imports some Scala library called actors
import scala.actors._
// Creates an object that extends the object Actor
object SillyActor extends Actor {
      // Defines a function called act, or overrides one in Actors
      def act() {
            // For loop that loops 5 times...
            // Wait, this is Scala... for loops? What?
            for (i <- 1 to 5) {
            // Print statement
            println("I'm acting!")
```

```
            // Thread method that pauses the process for 1000 ms
            Thread.sleep(1000)
            }
      }
}
// See comments about SillyActor object that extends Actor
object SeriousActor extends Actor {
      def act() {
            for (i <- 1 to 5) {
                  println("To be or not to be.")
                  Thread.sleep(1000)
            }
      }
}
// New object called Actors, not Actor, contains main function
object Actors {
       // Main function method, sets up what to run
      def main(args: Array[String]) {
            // Calls the method start in Actor for the extended object
            SillyActor.start()
            // Calls the method start in Actor for the extended object
            SeriousActor.start()
            // Both start methods are running nearly concurrently!
            // Runtime result varies each time you run it,
            // due to the prints being concurrent.
      }
}
```

It appears that both print statements are running near concurrently, thus at run time, the printed output may vary.

c. Add a third actor with a longer slee p time and run the code listing the output.

```
// Part of a package, called actors
package actors
// Imports some Scala library called actors
import scala.actors._
// Creates an object that extends the object Actor
object SillyActor extends Actor {
       // Defines a function called act, or overrides one in Actors
      def act() {
            // For loop that loops 5 times...
            // Wait, this is Scala... for loops? What?
            for (i <- 1 to 5) {
            // Print statement
            println("I'm acting!")
            // Thread method that pauses the process for 1000 ms
            Thread.sleep(1000)
            }
      }
}
// See comments about SillyActor object that extend Actor
object SeriousActor extends Actor {
      def act() {
            for (i <- 1 to 5) {
                  println("To be or not to be.")
                  Thread.sleep(1000)
            }
      }
```

```
}
// I added my own extending object, except with a longer sleep
time.
object RoboActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("BEEEP")
      Thread.sleep(5000)
    }
  }
}
// New object called Actors, not Actor, contains main function to
run
object Actors {
      // Main function method, sets up what to run
     def main(args: Array[String]) {
           // Calls the method start in Actor for the extended
object
           SillyActor.start()
           // Calls the method start in Actor for the extended
object
           SeriousActor.start()
           // Both start methods are running nearly concurrently!
           // Runtime result varies each time you run it,
           // due to the prints being concurrent.
           // Added a new "actor" with a longer sleep time,
           // this actor outputs much slower than the other two
           // So most of its outputs will be near the end.
            RoboActor.start()
     }
}
```

The Output (May vary each run):

```
BEEEP
I'm acting!
To be or not to be.
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
I'm acting!
To be or not to be.
To be or not to be.
I'm acting!
BEEEP
BEEEP
BEEEP
BEEEP
```