Fall, 2005 Name: _____

Com S 541 — Programming Languages 1

Test on Object-Oriented Languages, Scala, λ-calculus and Related Language Design Topics

Special Directions for this Test

This test has 5 questions and pages numbered 1 through 11.

When you write Scala code on this test, you may use anything in the standard imports (`java.lang`, `scala`, and `scala.Predef` without writing it in your test. Anything else used from the Java or Scala libraries must be imported. You are encouraged to define methods not specifically asked for if they are useful to your programming.

This test is open book and notes.

If you need more space, use the back of a page. Note when you do that on the front.

This exam is timed. We will not grade your exam if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire exam so that you can budget your time.

Clarity is important; if your diagrams or programs are sloppy and hard to read, you will lose points. Correct syntax also makes a difference for programming questions.

For Grading

Problem	Points	Score
1	15	
2	10	
3(a)	6	
3(b)	14	
4(a)	10	
4(b)	20	
5	25	
total	100	

1. (15 points) Using the type checking rules for the simply-typed lambda calculus, formally derive and prove the type of the following term, starting from the empty type environment.

$$\lambda f : \mathbf{o} \to \mathbf{o} \,.\, \lambda g : \mathbf{o} \to \mathbf{o} \to \mathbf{o} \,.\, \lambda x : \mathbf{o} \,.\, g\ x\ (f\ x)$$

   If this term does not have type in the empty type environment, then carry out the proof to the point where the proof fails, and say why it fails.

2. (10 points) Using the reduction rules for the untyped lambda calculus, show how to derive the normal form (if any) for the following.

$$(\lambda a\,.\,\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,x)\ ((\lambda f\,.\,f\ f\ f)\ (\lambda h\,.\,h\ h\ h))\ (\lambda g\,.\,g\ b)\ b\ y$$

If this term does not have a normal form, show using the reduction rules why it does not.

3. This is a problem about method overriding and dynamic dispatch.

   (a) (6 points) Give the output of the following Scala program, or, if it has no output, say why it does not.

```
package exam;
object MessagesPartA {

  trait Top {
    def and = " and ";
    def f = "Top.f calls " + g + and + h;
    def g: String;
    def h: String;
  }

  class High extends Top {
    def g = "High.g";
    def h = "High.h";
  }

  def main(args: Array[String]) = {
    val top = new High();
    Console.println(top.f);
  }
}
```

(b) (14 points) Give the output of the following Scala program, or, if it has no output, say why it does not.

```
package exam;
object MessagesPartB with Application {
  import MessagesPartA.Top;

  trait NeedsF extends Top {
    override def f: String;
    def h = "NeedsF.h calls " + g;
  }

  trait NeedsG extends Top {
    override def g: String = "NeedsG.g";
    def h: String;
  }

  class Bottom extends Top with NeedsF with NeedsG {
    override def and = "\n    and ";
    def bc = "Bottom calls ";
    override def f = bc + super.f;
    override def h = bc + super[NeedsF].h;
  }

  Console.println(new Bottom().f);
}
```

4. In this problem you will write two functions that work on combinator terms, formed from `S`, `K`, and `I`, variables, and applications. The abstract syntax of these combinator terms is given by the following.

```
package exam;

/** Abstract syntax of combinator terms. */
trait CombinatorTerm;
case class S extends CombinatorTerm {
  override def toString() = "S"
}
case class K extends CombinatorTerm {
  override def toString() = "K"
}
case class I extends CombinatorTerm {
  override def toString() = "I"
}
case class Var(name: String) extends CombinatorTerm {
  override def toString() = name;
}
case class App(rator: CombinatorTerm, rand: CombinatorTerm)
              extends CombinatorTerm {
  override def toString() =
    "(" + rator.toString() + " " + rand.toString() + ")";
}
```

Note that the Scala code `App(App(S(), K()), K()).toString()` returns the string `((S K) K)`. This kind of string is used in displaying tests.

(a) (10 points) A combinator term *contains a redex* if and only if it contains a subterm of one the following forms (where x, y, f, and g are arbitrary combinator terms):

`App(I(),` x`)`
`App(App(K(),` x`),` y`)`
`App(App(App(S(),` f`),` g`),` x`)`

In this problem you will write a definition to implement the following:

```
def containsRedex(t: CombinatorTerm): Boolean;
```

which returns true just when the argument, `t`, contains a redex as defined above.

Examples are given on the next page.

The test code below

```
package exam;
import testing._;
object CombinatorTestA with ConsoleTestRunner {

  class RedexTest(term: CombinatorTerm, expected: => Boolean)
     extends Expect[Boolean]("containsRedex(" + term.toString() + ")",
                 CombinatorPartA.containsRedex(term), expected)
     { override val arrow = " ==> "; }

  val suite = new TestSuite(
    new RedexTest(App(App(App(S(), Var("q")), Var("r")), Var("z")), true),
    new RedexTest(App(App(K(), Var("y")), (App(K(), I()))), true),
    new RedexTest(App(I(), Var("x")), true),
    new RedexTest(App(S(), App(I(), Var("x"))), true),
    new RedexTest(App(App(I(), I()), S()), true),
    new RedexTest(App(App(I(), I()), App(I(), Var("x"))), true),
    new RedexTest(App(S(), App(App(K(), App(I(), Var("x"))), S())), true),
    new RedexTest(App(App(S(), S()), App(App(K(), App(I(), Var("x"))), S())), true),
    new RedexTest(App(App(App(App(S(), K()), K()), App(I(), I())),
                    App(App(K(), App(I(), Var("x"))), S())), true),
    new RedexTest(I(), false),
    new RedexTest(K(), false),
    new RedexTest(S(), false),
    new RedexTest(Var("x"), false),
    new RedexTest(App(K(), Var("x")), false),
    new RedexTest(App(S(), Var("f1")), false),
    new RedexTest(App(App(S(), Var("f1")), Var("h")), false),
    new RedexTest(App(App(S(), S()), App(K(), I())), false)  );
}
```

should produce the following output.

```
containsRedex((((S q) r) z)) ==> true
containsRedex(((K y) (K I))) ==> true
containsRedex((I x)) ==> true
containsRedex((S (I x))) ==> true
containsRedex(((I I) S)) ==> true
containsRedex(((I I) (I x))) ==> true
containsRedex((S ((K (I x)) S))) ==> true
containsRedex(((S S) ((K (I x)) S))) ==> true
containsRedex(((((S K) K) (I I)) ((K (I x)) S))) ==> true
containsRedex(I) ==> false
containsRedex(K) ==> false
containsRedex(S) ==> false
containsRedex(x) ==> false
containsRedex((K x)) ==> false
containsRedex((S f1)) ==> false
containsRedex(((S f1) h)) ==> false
containsRedex(((S S) (K I))) ==> false
OK
```

(b) (20 points) The reduction rules for combinator terms are based on the following axioms that act on the kinds of redexes described above (where x, y, f, and g are arbitrary combinator terms):

`App(I(), `x`)` $\Rightarrow x$
`App(App(K(), `x`), `y`)` $\Rightarrow x$
`App(App(App(S(), `f`), `g`), `x`)` $\Rightarrow$ `App(App(`f`, `x`), App(`g`, `x`))`

In this problem you will write a definition to implement the following:

```
def normalForm(t: CombinatorTerm): CombinatorTerm;
```

which returns a combinator term that does not contain redexes, but which is derived from the argument, `t`, by repeatedly applying the rules given above until the result contains no redexes. (Note that your code can't be guaranteed to terminate on all input terms, since some don't have a normal form, and since there is no algorithm that is guaranteed to terminate that can decide if a term has a normal form.)

You can use the `containsRedex` function from part (a) in your solution.

Examples are given on the next page.

The test code below

```
package exam;
import testing._;
object CombinatorTestB with ConsoleTestRunner {

  class TermTest(term: CombinatorTerm, expected: => CombinatorTerm)
     extends Expect[CombinatorTerm]("normalForm(" + term.toString() + ")",
                 CombinatorPartB.normalForm(term), expected)
     { override val arrow = " ==> "; }

  val suite = new TestSuite(
    new TermTest(App(App(App(S(), Var("q")), Var("r")), Var("z")),
                 App(App(Var("q"), Var("z")), App(Var("r"), Var("z")))),
    new TermTest(App(App(K(), Var("y")), (App(K(), I()))), Var("y")),
    new TermTest(App(I(), Var("x")), Var("x")),
    new TermTest(App(App(I(), I()), App(I(), Var("x"))), Var("x")),
    new TermTest(App(App(I(), App(I(), I())), App(App(K(), App(I(), Var("x"))), S())),
                 Var("x")),
    new TermTest(App(App(App(App(S(), K()), K()), App(I(), I())),
                     App(App(K(), App(I(), Var("x"))), S())),
                 Var("x")),
    new TermTest(I(), I()),
    new TermTest(K(), K()),
    new TermTest(S(), S()),
    new TermTest(App(K(), Var("x")), App(K(), Var("x"))),
    new TermTest(App(S(), Var("f1")), App(S(), Var("f1"))),
    new TermTest(App(App(S(), Var("f1")), Var("h")),
                 App(App(S(), Var("f1")), Var("h"))),
    new TermTest(Var("haveAGreatHoliday"), Var("haveAGreatHoliday"))
  );
}
```

should produce the following output.

```
normalForm((((S q) r) z)) ==> ((q z) (r z))
normalForm(((K y) (K I))) ==> y
normalForm((I x)) ==> x
normalForm(((I I) (I x))) ==> x
normalForm(((I (I I)) ((K (I x)) S))) ==> x
normalForm(((((S K) K) (I I)) ((K (I x)) S))) ==> x
normalForm(I) ==> I
normalForm(K) ==> K
normalForm(S) ==> S
normalForm((K x)) ==> (K x)
normalForm((S f1)) ==> (S f1)
normalForm(((S f1) h)) ==> ((S f1) h)
normalForm(haveAGreatHoliday) ==> haveAGreatHoliday
OK
```

5. (25 points) This problem is about manipulating XML objects in Scala.

   In this problem you will write a definition to implement the following.

```
def gradeReport(input: scala.xml.Elem, student: String): scala.xml.Elem;
```

   You can assume that `input` is an XML element that looks something like the following.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<grades>
  <title style="bold">Grade Sample (imaginary)</title>
  <generatedOn>Wed, 16 Nov 2005 09:25:32 PM, CST</generatedOn>
  <table>
    <tr><th>Last 4 of</th> <th>HW1</th> <th>HW2</th> <th>Exam 1</th></tr>
    <tr><th>ID Number</th> <th>120</th> <th>236</th> <th>100</th></tr>
    <tr><td>0000</td> <td>  0</td> <td>  0</td> <td>  0</td></tr>
    <tr><td>1234</td> <td>120</td> <td>235</td> <td>100</td></tr>
    <tr><td>4321</td> <td>100</td> <td>200</td> <td> 89</td></tr>
    <tr><td>5000</td> <td> 60</td> <td>118</td> <td> 50</td></tr>
    <tr><td>5786</td> <td>102</td> <td>230</td> <td> 91</td></tr>
    <tr><td>6975</td> <td>  0</td> <td>  0</td> <td>100</td></tr>
    <tr><td>9999</td> <td>119</td> <td>234</td> <td>  0</td></tr>
  </table>
</grades>
```

   That is, the `input` XML element contains a single `table` node. This `table` node has several `tr` nodes, some of which are header nodes (identified by having children that are `th` nodes), and some of which are data nodes (identified by having children that are `td` nodes).

   Let us call the text in the 0^{th} `td` child node in a data row the *key* for that row. For example, the string `1234` is the key for the second non-header row in the above table.

   Assuming that `student` is equal to the key of some data row, `gradeReport(input, student)` must return an XML element that is the same as `input`, but whose only data row is the one whose key is equal to `student`.

   For example, if `parsed` is the result of running `XML.loadFile` on a file containing the above data, then the result of calling `gradeReport(parsed, "5000")` is an object that represents the following XML.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<grades>
  <title style="bold">Grade Sample (imaginary)</title>
  <generatedOn>Wed, 16 Nov 2005 09:25:32 PM, CST</generatedOn>
  <table>
    <tr><th>Last 4 of</th> <th>HW1</th> <th>HW2</th> <th>Exam 1</th></tr>
    <tr><th>ID Number</th> <th>120</th> <th>236</th> <th>100</th></tr>
    <tr><td>5000</td> <td> 60</td> <td>118</td> <td> 50</td></tr>
  </table>
</grades>
```

   Notice how this output only contains the data row for the key `"5000"`.

   There is space for your answer on the next page.