

In [ ]: 1

## CSCI 3155: Assignment 8

**Name:** WRITE YOUR NAME HERE

```
In [ ]: 1 // TEST HELPER
2 def passed(points: Int) {
3     require(points >= 0)
4     if (points == 1) print(s"\n*** Tests Passed (1 point) ***\n")
5     else print(s"\n*** Tests Passed ($points points) ***\n")
6 }
```

### Problem 1 (20 points): CPS Style Transformation

Reimplement the programs below in the CPS style.

#### A (10 points)

Convert `helloUp`, `doubleUp` and `mainFun` into the CPS functions `helloUp_k`, `doubleUp_k`, `mainFun_k`. They should take continuations that are of type `String => String`.

```
def helloUp(x: String): String = {
    "Hello " + x
}

def doubleUp(x: String): String = {
    x + x
}

def mainFun(x: String): String = {
    doubleUp(helloUp(x) + "World")
}
```

In [ ]: 1 ??? // YOUR CODE HERE

```
In [ ]: 1 //BEGIN TEST
        2 assert(helloUp_k("World", x => x) == "HelloWorld", "Test 1, Set 1
        3 assert(doubleUp_k("World", x => x) == "WorldWorld", "Test 2 Set 1
        4 assert(mainFun_k("Donkey", x => x) == "HelloDonkeyWorldHelloDonkey
        5 passed(5)
        6 //END TEST
```

```
In [ ]: 1 //BEGIN TEST
        2 assert(helloUp_k("Hello", x => ("*"+x+"*")) == "*HelloHello*", "Te
        3 assert(doubleUp_k("HelloWorld", x => ("*"+x+"*")) == "*HelloWorldH
        4 assert(mainFun_k("Cruel", x => ("*"+x+"*")) == "*HelloCruelWorldHe
        5 passed(5)
        6 //END TEST
```

**B (10 points)**

Convert the following program into CPS.

```
def util(x: Int):String = {
  val v1 = 2 * x
  v1.toString
}

def fun1(x: String): Int = {
  val v1 = util(x.toInt)
  v1.toInt
}

def fun2(x: String): String = {
  util(x.toInt)
}

def mainFun(x: String): Int = {
  val v1 = fun2(x)
  val v2 = fun1(x)
  val v3 = v1.length
  v2 - v3
}
```

To enable this create a polymorphic versions of each function.

```
def util_k[T](x: Int, k: String => T): T
def fun1_k[T](x: String, k: Int => T):T
def fun2_k[T] (x: String, k: String => T):T
def mainFun_k[T](x: String, k: Int => T ):T
```

```
In [ ]: 1 def util_k[T] (x: Int, k: String => T):T = {
2         ??? // YOUR CODE HERE
3     }
4
5     def fun1_k[T](x: String, k: Int => T):T = {
6         ??? // YOUR CODE HERE
7     }
8     def fun2_k[T] (x: String, k: String => T):T = {
9         ??? // YOUR CODE HERE
10    }
11    def mainFun_k[T](x: String, k: Int => T ):T = {
12        ??? // YOUR CODE HERE
13    }
14
```

```
In [ ]: 1 assert(util_k(10, _ + "d") == "20d", "Test 1 failed")
2 assert(util_k(21, Some[String]) == Some("42"), "Test 2 failed")
3 passed(4)
```

```
In [ ]: 1 assert(fun1_k("10", Some[Int]) == Some(20), "Test 1 failed")
2 assert(fun2_k("-6", _ + "3sdf") == "-123sdf", "Test 2 failed")
3 passed(2)
```

```
In [ ]: 1 //BEGIN TEST
2 mainFun_k[Unit]("217", println)
3 mainFun_k[Unit]("5000", println)
4 mainFun_k[Unit]("0", println)
5 assert(mainFun_k[String]("217", x=>(x.toString)) == "431", "Test 2")
6 assert(mainFun_k[Int]("217", x=>x) == 431, "Test 3, Set 1 Failed")
7 assert(mainFun_k[Int]("5000", x=>x) == 9995, "Test 4, Set 1 Failed")
8 assert(mainFun_k[List[Int]]("5000", x => List(x)) == List(9995), '
9 passed(4)
10 //END TEST
```

## Problem 2 (25 points): Regular Expression Pattern Matching and Continuations

Consider the problem of pattern matching regular expressions. The grammar for regular expressions are given as

$$\begin{aligned} \mathbf{RegExpr} &\rightarrow \text{Atom}(\mathit{String}) \\ &| \text{Or}(\mathbf{RegExpr}, \mathbf{RegExpr}) \\ &| \text{Concat}(\mathbf{RegExpr}, \mathbf{RegExpr}) \\ &| \text{KleeneStar}(\mathbf{RegExpr}) \\ &| \text{And}(\mathbf{RegExpr}, \mathbf{RegExpr}) \end{aligned}$$

Given a string,  $s$  and regular expression  $r$ , we wish to define a function  $\mathbf{Matches}(r, s)$  which returns a tuple  $(v_1, j)$

- Wherein  $v_1$  is either *true* if some *prefix* of the string  $s$  matches the expression  $r$  or *false* otherwise.
- If  $v_1 = \text{true}$ , then  $j$  denotes the position where the match ends, such that  $j \geq 0$  and  $j < \text{length}(s)$ . In other words, the substring  $s(0), \dots, s(j)$  matches the regular expression  $r$ .
- If *false*, we will simply set  $j = -1$ .

We will use operational semantics rules to define  $\mathbf{Matches}$ .

## Atom

$$\frac{s(0, \dots, j) = t}{\mathbf{Matches}(\text{Atom}(t), s) = (\text{true}, j)} \text{ (atom-match)}$$

$$\frac{t \text{ is not a prefix of } s}{\mathbf{Matches}(\text{Atom}(t), s) = (\text{false}, -1)} \text{ (atom-no-match)}$$

## Or

$$\frac{\mathbf{Matches}(r_1, s) = (v_1, j_1), \mathbf{Matches}(r_2, s) = (v_2, j_2)}{\mathbf{Matches}(\text{Or}(r_1, r_2), s) = (v_1 \text{ or } v_2, \max(j_1, j_2))} \text{ (or-match)}$$

## Concat

$$\frac{\mathbf{Matches}(r_1, s) = (\text{true}, j_1), \mathbf{Matches}(r_2, s(j_1 + 1, \dots, n)) = (\text{true}, j_2)}{\mathbf{Matches}(\text{Concat}(r_1, r_2), s) = (\text{true}, j_1 + j_2 + 1)} \text{ (concat-match)}$$

$$\frac{\mathbf{Matches}(r_1, s) = (\text{false}, -1)}{\mathbf{Matches}(\text{Concat}(r_1, r_2), s) = (\text{false}, -1)} \text{ (concat-no-match-1)}$$

$$\frac{\mathbf{Matches}(r1, s) = (true, j_1), \mathbf{Matches}(r2, s) = (false, -1)}{\mathbf{Matches}(\text{Concat}(r1, r2), s) = (false, -1)} \text{ (concat-no-match-2)}$$

## Kleene Star

$$\frac{\mathbf{Matches}(r, s) = (true, j_1), n = \text{length}(s), \mathbf{Matches}(\text{KleeneStar}(r), s(j + 1, \dots, n))}{\mathbf{Matches}(\text{KleeneStar}(r), s) = (true, j_1 + j_2 + 1)}$$

$$\frac{\mathbf{Matches}(r, s) = (false, -1)}{\mathbf{Matches}(\text{KleeneStar}(r), s) = (true, -1)} \text{ (kleene-star-no-match)}$$

The Kleene-star-no-match rule looks very strange on first sight. But really, it is trying to say that any string matches the Kleene star of a regular expression. However, if the inner regular expression does not match, the Kleene star matches trivially with a match of length 0.

## A (10 points)

Complete the code below for evaluating the semantics above.

```

In [ ]: 1 sealed trait RegExp
2 case class Atom(s: String) extends RegExp
3 case class Or(r1: RegExp, r2: RegExp) extends RegExp
4 case class Concat(r1: RegExp, r2: RegExp) extends RegExp
5 case class KleeneStar(r1: RegExp) extends RegExp
6
7 def isPrefixOf(t: String, s: String) = {
8   /* If string t is a prefix of s, then return true,
9     else return false
10    */
11   s.startsWith(t)
12 }
13
14 def evalRegexp(r: RegExp, s: String ): (Boolean, Int) = r match {
15   case Atom(t) => {
16     ??? // YOUR CODE HERE
17   }
18
19   case Or(r1, r2) => {
20     val (v1, j1) = evalRegexp(r1, s)
21     val (v2, j2) = evalRegexp(r2, s)
22     ((v1||v2), math.max(j1, j2) )
23   }
24
25   case Concat(r1, r2) => {
26     ??? // YOUR CODE HERE
27   }
28
29   case KleeneStar(rHat) => {
30     val (v1, j1) = evalRegexp(rHat, s)
31     if (v1) {
32       val (v2, j2) = evalRegexp(KleeneStar(rHat), s.substring(
33         (true, j1+1+j2)
34       ) else {
35         (true, -1)
36       }
37   }
38
39 }
40 }

```

```

In [ ]: 1 //BEGIN TEST
2 assert( evalRegexp(Atom("hello"), "helloworld") == (true, 4) , "Test 1 failed")
3 assert(evalRegexp(Atom("hello"), "Helloworld") == (false, -1), "Test 2 failed")
4 assert(evalRegexp(Or(Atom("hello"), Atom("Hellow")), "Helloworld") == (true, 4), "Test 3 failed")
5 passed(4)
6 //END TEST

```

```
In [ ]: 1 //BEGIN TEST
        2 assert(evalRegexp(KleeneStar(Atom("H")), "Helloworld") == (true, 0))
        3 assert(evalRegexp(Concat(Atom("hell"), Atom("owor")), "helloworld") == (true, 0))
        4 passed(3)
        5 //END TEST
```

```
In [ ]: 1 //BEGIN TEST
        2 assert(evalRegexp(Concat( Or( Atom("what"), Or(Atom("why"), Atom('
        3 passed(3)
        4 //END TEST
```

## B (15 points)

Reimplement the interpreter in the CPS so that all recursion happens at the tail position.

```

In [ ]: 1 def isPrefixOf_k[T](t: String, s: String, k: Boolean => T): T = {
2         /* If string t is a prefix of s, then return true,
3            else return false
4            */
5         ??? // YOUR CODE HERE
6     }
7
8     def evalRegex(r: Regex, s: String): (Boolean, Int) = {
9         /*--
10          This has been deliberately placed so that you
11          should not be using evalRegex for
12          this problem.
13          --*/
14
15         ???
16     }
17
18     def evalRegex_k[T](r: Regex, s: String, k: (Boolean, Int) => T) = {
19         case Atom(t) => {
20             isPrefixOf_k(t, s, v => {
21                 ??? // YOUR CODE HERE
22             })
23         }
24
25         case Or(r1, r2) => {
26             evalRegex_k(r1, s, (v1:Boolean, j1:Int)=> {
27                 evalRegex_k(r2, s.substring(j1+1, s.length), (v2:Boo
28                 k((v1 || v2), math.max(j1,j2))
29             })
30         })
31     }
32
33     case Concat(r1, r2) => {
34         ??? // YOUR CODE HERE
35     }
36
37     case KleeneStar(rHat) => {
38         evalRegex_k(rHat, s, (v1: Boolean, j1: Int) => {
39             ??? // YOUR CODE HERE
40         })
41     }
42 }
43
44
45 }
46

```

```
In [ ]: 1 //BEGIN TEST
        2 assert( evalRegexp_k[Int](Atom("hello"), "helloworld", (x: Boolean) => x == 5), "helloworld", (x: Boolean) => x == 5)
        3 assert( evalRegexp_k[Boolean](Atom("hello"), "Helloworld", (x: Boolean) => x == 5), "Helloworld", (x: Boolean) => x == 5)
        4 passed(5)
        5 //END TEST
```

```
In [ ]: 1 //BEGIN TEST
        2 assert( evalRegexp_k[String](Or(Atom("hello"), Atom("Hellow")), "Helloworld", (x: String) => x == "Helloworld"), "Helloworld", (x: String) => x == "Helloworld")
        3 passed(5)
        4 //END TEST
```

```
In [ ]: 1 //BEGIN TEST
        2 assert(evalRegexp_k(KleeneStar(Atom("H")), "HHHHHelloworld", (x: String) => x == "HHHHHelloworld"), "HHHHHelloworld", (x: String) => x == "HHHHHelloworld")
        3 assert(evalRegexp_k(Concat(Atom("hell"), Atom("owor")), "helloworld", (x: String) => x == "helloworld"), "helloworld", (x: String) => x == "helloworld")
        4 assert(evalRegexp_k(Concat( Or( Atom("what"), Or(Atom("why"), Atom("how")), "helloworld", (x: String) => x == "helloworld"), "helloworld", (x: String) => x == "helloworld")
        5 passed(5)
        6 //END TEST
```

### Problem 3 (20 points)

In this problem, you are asked to write the eval function for Lettuce using CPS style. Here is the stripped down grammar for Lettuce expressions.

```
In [ ]: 1 sealed trait Expr
        2 case class Const(v: Double) extends Expr // Expr -> Const(v)
        3 case class Ident(s: String) extends Expr // Expr -> Ident(s)
        4 // Arithmetic Expressions
        5 case class Plus(e1: Expr, e2: Expr) extends Expr // Expr -> Plus(e1, e2)
        6 // Boolean Expression
        7 case class Geq(e1: Expr, e2: Expr) extends Expr // Expr -> Bool(Expr, Expr)
        8 //If then else
        9 case class IfThenElse(e: Expr, eIf: Expr, eElse: Expr) extends Expr
        10 //Let bindings
        11 case class Let(s: String, defExpr: Expr, bodyExpr: Expr) extends Expr
        12 //Function definition
        13 case class FunDef(param: String, bodyExpr: Expr) extends Expr
        14 // Function call
        15 case class FunCall(funCalled: Expr, argExpr: Expr) extends Expr
```

```
In [ ]: 1 sealed trait Value
        2 type Environment = Map[String, Value]
        3 case class NumValue(d: Double) extends Value
        4 case class BoolValue(b: Boolean) extends Value
        5 case class Closure(x: String, body: Expr, env: Environment) extends Value
```

You are asked to implement an eval function using CPS transformation.

```
In [ ]: 1 case class Exception(msg: String) extends Exception
2
3 def evalExpr_cps[T](e: Expr, env: Environment, k: Value => T): T =
4   case Const(d) => {
5     ??? // YOUR CODE HERE
6   }
7   case Ident(x) => {
8     if (env contains x) {
9       ??? // YOUR CODE HERE
10    } else {
11      throw new Exception(s"Unknown identifier $x")
12    }
13  }
14
15  case Plus(e1, e2) => {
16    evalExpr_cps(e1, env, (v1) => {
17      evalExpr_cps(e2, env, (v2) => {
18        (v1, v2) match {
19          case (NumValue(d1), NumValue(d2)) => k(NumValue(d1 + d2))
20          case _ => throw new Exception(s"Trying to add non-numeric values")
21        }
22      })
23    })
24  }
25
26  case Geq(e1, e2) => {
27    ??? // YOUR CODE HERE
28  }
29
30  case IfThenElse(e, e1, e2) => {
31    evalExpr_cps(e, env, (v) => {
32      v match {
33        case BoolValue(true) => {
34          ??? // YOUR CODE HERE
35        }
36        case BoolValue(false) => {
37          ??? // YOUR CODE HERE
38        }
39        case _ => {
40          throw new Exception("Condition is not a boolean")
41        }
42      }
43    })
44  }
45
46  case Let(x, e1, e2) => {
```

```

48     evalExpr_cps(e1, env, (v) => {
49         ??? // YOUR CODE HERE
50     })
51 }
52
53 case FunDef(x, e) => {
54     ??? // YOUR CODE HERE
55 }
56
57 case FunCall(e1, e2) => {
58     evalExpr_cps(e1, env, (v) => {
59         v match {
60             case Closure(param, body, closureEnv) => {
61                 ??? // YOUR CODE HERE
62             }
63             case _ => {
64                 throw new ErrorException("function call on some")
65             }
66         }
67     })
68 }
69
70 }

```

```

In [ ]: 1 val p1 =
2         Let("square", // let square =
3             FunDef("x", Plus(Ident("x"), Ident("x"))), // function
4             FunCall(Ident("square"), Const(10)) // in square(10)
5         )
6
7 assert( evalExpr_cps(p1, Map.empty, x => x) == NumValue(20.0), "Test
8 passed(6)

```

```

In [ ]: 1 val x = Ident("x")
2 val y = Ident("y")
3 val fdef_inner = FunDef("y", Plus(x, Plus(y, y)))
4 val fdef_outer = FunDef("x", fdef_inner)
5 val call_expr = FunCall(FunCall(Ident("sq1"), x), y)
6 val sq1_call = Let("sq1", fdef_outer, call_expr)
7 val lety = Let("y", Const(15), sq1_call)
8 val letx = Let("x", Const(10), lety)
9 val p2 = letx
10
11 assert(evalExpr_cps(p2, Map.empty, x => x) == NumValue(40.0))
12 passed(7)

```

```
In [ ]: 1 val x = Ident("x")
        2 val y = Ident("y")
        3 val iteExpr = IfThenElse(Geq(x, y), Plus(x, Plus(x,y)), Plus(y, P
        4 val fdef_inner = FunDef("y", iteExpr)
        5 val fdef_outer = FunDef("x", fdef_inner)
        6 val call_expr = FunCall(FunCall(Ident("sq1"), x), y)
        7 val sq1_call = Let("sq1", fdef_outer, call_expr)
        8 val lety = Let("y", Const(15), sq1_call)
        9 val letx = Let("x", Const(10), lety)
       10 val p3 = letx
       11
       12 assert(evalExpr_cps(p3, Map.empty, x => x) == NumValue(35.0))
       13 passed(7)
       14
```