Before you turn this problem in, make sure everything runs as expected.

1. First, **restart the kernel** (in the menubar, select Kernel→Restart) and
2. Then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name below.

CSCI 3155: Assignment 6

Topics Covered:

Name: Ishika Patel

In [1]:

```
// TEST HELPER
def passed(points: Int) {
    require(points >=0)
    if (points == 1) print(s"\n*** Tests Passed (1 point) ***\n")
    else print(s"\n*** Tests Passed ($points points) ***\n")
}
```

Out[1]:

```
defined function passed
```

Problem 1 (30 Points): Mutual Recursion in Lettuce

In class, we have explored recursive functions in lettuce using the *let rec* syntax. In this problem, we will explore, mutually recursive function, specifically two mutually recursive functions.

Consider:

```
let rec
        pos = function (x)
                 if (x >= 0)
                 then x
                 else neg(x)
        neg = function (y)
                 if (y <= 0)
                 then pos (1 + y * y)
                 else pos(1 - y)
   in
     neg(10.0)
```

The two functions are *mutually recursive* since `pos` calls `neg` and vice-versa. Convince yourself that thhe program must return `82`.

1A (5 points): Extending the Abstract Syntax

Consider the grammar specification we have seen thus far.

$$
\begin{array}{rcll}
\mathbf{Program} & \rightarrow & \mathit{TopLevel}(\mathbf{Expr}) & \\
\mathbf{Expr} & \rightarrow & \mathit{Const}(\mathbf{Number}) & \\
 & | & \mathit{Ident}(\mathbf{Identifier}) & \\
 & | & \mathit{Plus}(\mathbf{Expr}, \mathbf{Expr}) & \\
 & | & \mathit{Mult}(\mathbf{Expr}, \mathbf{Expr}) & \\
 & | & \mathit{Eq}(\mathbf{Expr}, \mathbf{Expr}) & \\
 & | & \mathit{Geq}(\mathbf{Expr}, \mathbf{Expr}) & \\
 & | & \mathit{IfThenElse}(\mathbf{Expr}, \mathbf{Expr}, \mathbf{Expr}) & \text{if (expr) then expr} \\
 & | & \mathit{Let}(\mathbf{Identifier}, \mathbf{Expr}, \mathbf{Expr}) & \text{let identifier = exp} \\
 & | & \mathit{FunDef}(\mathbf{Identifier}, \mathbf{Expr}) & \text{function (identifier} \\
 & | & \mathit{FunCall}(\mathbf{Expr}, \mathbf{Expr}) & \text{function call - expr} \\
 & | & \mathit{LetRec}(\mathbf{Identifier}, \mathbf{Identifier}, \mathbf{Expr}, \mathbf{Expr}) &
\end{array}
$$

We wish to add a new rule for two mutually recursive functions

$$\mathbf{Expr} \rightarrow \mathit{LetRec2}(\mathbf{Identifier}, \mathbf{Identifier}, \mathbf{Expr}, \mathbf{Identifier}, \mathbf{Identifier}, \mathbf{Expr}, \mathbf{Ex}$$

Such that a mutual call such as

```
let rec
         f1 = function (x1) e1
         f2 = function (x2) e2
     in
         e3
```

is represented in the AST as

```
LetRec2(f1, x1, e1, f2, x2, e2, e3)
```

Extend the existing AST specification to add support for `LetRec2` .

In [2]:
```
sealed trait Program
sealed trait Expr
case class Const(f: Double) extends Expr
case class Ident(s: String) extends Expr
case class Minus(e1: Expr, e2: Expr) extends Expr
case class Plus(e1: Expr, e2: Expr) extends Expr
case class Mult(e1: Expr, e2: Expr) extends Expr
case class Eq(e1: Expr, e2: Expr) extends Expr
case class Geq(e1: Expr, e2: Expr) extends Expr
case class IfThenElse(e1: Expr, e2: Expr, e3: Expr) extends Expr
case class Let(x: String, e1: Expr, e2: Expr) extends Expr
case class FunDef(id: String, e: Expr) extends Expr
case class FunCall(calledFun: Expr, argExpr: Expr) extends Expr
case class LetRec(funName: String, param: String, funExpr: Expr, bodyE

// YOUR CODE HERE
case class LetRec2(f1: String, x1: String, e1: Expr, f2: String, x2: S
case class TopLevel(e: Expr) extends Program
```

Out[2]:
```
defined trait Program
defined trait Expr
defined class Const
defined class Ident
defined class Minus
defined class Plus
defined class Mult
defined class Eq
defined class Geq
defined class IfThenElse
defined class Let
defined class FunDef
defined class FunCall
defined class LetRec
defined class LetRec2
defined class TopLevel
```

In [3]:
```
//BEGIN TEST
val x = Ident("x")
val y = Ident("y")
val foo = Ident("foo")
val bar = Ident("bar")

val e1 = IfThenElse( Geq(x, Const(0.0)), x, FunCall(bar, Plus(x, Const
val e2 = IfThenElse( Geq(Const(1.0), x), Plus(Const(2.0), x), FunCall(
val e3 = FunCall(bar, Const(10))
val lr2 = LetRec2("foo", "x", e1, "bar", "x", e2, e3)
val p1 = TopLevel(lr2)
passed(3)
```

```
//END TEST
```

```
*** Tests Passed (3 points) ***
```

Out[3]:

```
x: Ident = Ident("x")
y: Ident = Ident("y")
foo: Ident = Ident("foo")
bar: Ident = Ident("bar")
e1: IfThenElse = IfThenElse(
  Geq(Ident("x"), Const(0.0)),
  Ident("x"),
  FunCall(Ident("bar"), Plus(Ident("x"), Const(1.0)))
)
e2: IfThenElse = IfThenElse(
  Geq(Const(1.0), Ident("x")),
  Plus(Const(2.0), Ident("x")),
  FunCall(Ident("foo"), Minus(Ident("x"), Const(2.0)))
)
e3: FunCall = FunCall(Ident("bar"), Const(10.0))
lr2: LetRec2 = LetRec2(
  "foo",
  "x",
  IfThenElse(
    Geq(Ident("x"), Const(0.0)),
    Ident("x"),
    FunCall(Ident("bar"), Plus(Ident("x"), Const(1.0)))
  ),
  "bar",
  "x",
  IfThenElse(
    Geq(Const(1.0), Ident("x")),
    Plus(Const(2.0), Ident("x")),
    FunCall(Ident("foo"), Minus(Ident("x"), Const(2.0)))
  ),
  FunCall(Ident("bar"), Const(10.0))
)
p1: TopLevel = TopLevel(
  LetRec2(
    "foo",
    "x",
    IfThenElse(
      Geq(Ident("x"), Const(0.0)),
      Ident("x"),
      FunCall(Ident("bar"), Plus(Ident("x"), Const(1.0)))
    ),
    "bar",
    "x",
    IfThenElse(
      Geq(Const(1.0), Ident("x")),
      Plus(Const(2.0), Ident("x")),
      FunCall(Ident("foo"), Minus(Ident("x"), Const(2.0)))
    ),
    FunCall(Ident("bar"), Const(10.0))
  )
)
```

In [4]:

```
//BEGIN TEST
val x = Ident("x")
val y = Ident("y")
val foo = Ident("foo")
val bar = Ident("bar")
val e11 = IfThenElse(Geq(x, Const(0.0)), FunCall(bar, Minus(Const(1.0)
val e1 = IfThenElse(Eq(x, Const(0.0)), Const(1.0), e11)
val e2 = IfThenElse(Geq(Const(0.0), y), Mult(Const(-0.5), y) , FunCall
val e3 = FunCall(foo, Const(10.5))
val lr3 = LetRec2("foo", "x", e1, "bar", "y", e2, e3)
passed(2)
//END TEST
```

```
*** Tests Passed (2 points) ***
```

Out[4]:

```
x: Ident = Ident("x")
y: Ident = Ident("y")
foo: Ident = Ident("foo")
bar: Ident = Ident("bar")
e11: IfThenElse = IfThenElse(
  Geq(Ident("x"), Const(0.0)),
  FunCall(Ident("bar"), Minus(Const(1.0), Ident("x"))),
  Mult(Ident("x"), FunCall(Ident("foo"), Minus(Ident("x"), Const(1.0)
)))
)
e1: IfThenElse = IfThenElse(
  Eq(Ident("x"), Const(0.0)),
  Const(1.0),
  IfThenElse(
    Geq(Ident("x"), Const(0.0)),
    FunCall(Ident("bar"), Minus(Const(1.0), Ident("x"))),
    Mult(Ident("x"), FunCall(Ident("foo"), Minus(Ident("x"), Const(1.
0))))
  )
)
e2: IfThenElse = IfThenElse(
  Geq(Const(0.0), Ident("y")),
  Mult(Const(-0.5), Ident("y")),
  FunCall(Ident("foo"), Ident("y"))
)
e3: FunCall = FunCall(Ident("foo"), Const(10.5))
```

```
lr3: LetRec2 = LetRec2(
  "foo",
  "x",
  IfThenElse(
    Eq(Ident("x"), Const(0.0)),
    Const(1.0),
    IfThenElse(
      Geq(Ident("x"), Const(0.0)),
      FunCall(Ident("bar"), Minus(Const(1.0), Ident("x"))),
      Mult(Ident("x"), FunCall(Ident("foo"), Minus(Ident("x"), Const(
1.0))))
    )
  ),
  "bar",
  "y",
  IfThenElse(
    Geq(Const(0.0), Ident("y")),
    Mult(Const(-0.5), Ident("y")),
    FunCall(Ident("foo"), Ident("y"))
  ),
  FunCall(Ident("foo"), Const(10.5))
)
```

1B (10 points): Build an Environment to Handle Mutual Recursion

In class we saw how to build an environment that handles recursive calls using the `ExtendRec` construct.

Now we propose an `ExtendMutualRec2` construct such that

$$\frac{}{\text{eval}(\texttt{LetRec2}(\texttt{f1},\ \texttt{x1},\ \texttt{e1},\ \texttt{f2},\ \texttt{x2},\ \texttt{e2},\ \texttt{e}),\sigma) = \text{eval}(e, \texttt{ExtendMutualRec2}(\texttt{f1},\ \texttt{x1},\ \texttt{e1},\ \texttt{f2},\ \texttt{x2},\ \texttt{e2},\sigma))} \text{((Mutual-Rec-(}$$

Complete the mathematical specification of $\texttt{ExtendMutualRec2}$ Let $\pi = \texttt{ExtendMutualRec2}(\texttt{f1},\ \texttt{x1},\ \texttt{e1},\ \texttt{f2},\ \texttt{x2},\ \texttt{e2},\sigma))$

$$\pi(y) = \begin{cases} 1 & \text{if } y = \texttt{f1} \\ 2 & \text{if } y = \texttt{f2} \\ 3 & \text{otherwise} \end{cases}$$

Fill in the appropriate values for $1, 2, 3$.

Write your answer in the cell bellow. You can make a numbered list in markdown to represent your answers as follows:

1. First
2. Second
3. And so on...

1. Closure(x1, e1, σ)
2. Closure(x2, e2, σ)
3. π(y)

1C (9 points): Code up the Environment Spec

Implement the environment for `ExtendMutualRec`.

In [5]:

```
sealed trait Environment
sealed trait Value

case object EmptyEnv extends Environment
case class Extend(x: String, v: Value, sigma: Environment) extends Env
case class ExtendRec(f: String, x: String, e: Expr, sigma: Environment
case class ExtendMutualRec2(f1: String, x1: String, e1: Expr, f2: Stri
/* -- we need to redefine values to accomodate the new representation
case class NumValue(d: Double) extends Value
case class BoolValue(b: Boolean) extends Value
case class Closure(x: String, e: Expr, pi: Environment) extends Value
case object ErrorValue extends Value


/*2. Operators on values */

def valueToNumber(v: Value): Double = v match {
    case NumValue(d) => d
    case _ => throw new IllegalArgumentException(s"Error: Asking me to
}

def valueToBoolean(v: Value): Boolean = v match {
    case BoolValue(b) => b
    case _ => throw new IllegalArgumentException(s"Error: Asking me to
}

def valueToClosure(v: Value): Closure = v match {
    case Closure(x, e, pi) => Closure(x, e, pi)
    case _ =>  throw new IllegalArgumentException(s"Error: Asking me t
}


/*-- Operations on environments --*/

def lookupEnv(sigma: Environment, x: String): Value = sigma match {
    case EmptyEnv => throw new IllegalArgumentException(s"Error could
    case Extend(y, v, _) if y == x => v
    case Extend(_, _, pi) => lookupEnv(pi, x)
    case ExtendRec(f, y, e, pi) => if (x == f)
                                              Closure(y, e, sigma)
                                   else
                                              lookupEnv(pi, x)
    case ExtendMutualRec2(f1, x1, e1, f2, x2, e2, pi) =>
    {
        if(x == f1)
        {
            Closure(x1, e1, sigma)
        }
        else if(x == f2)
        {
            Closure(x2, e2, sigma)
        }
        else
        {
            lookupEnv(pi, x)
        }
    }
}
```

Out[5]:

```
defined trait Environment
defined trait Value
defined object EmptyEnv
defined class Extend
defined class ExtendRec
defined class ExtendMutualRec2
defined class NumValue
defined class BoolValue
defined class Closure
defined object ErrorValue
defined function valueToNumber
defined function valueToBoolean
defined function valueToClosure
defined function lookupEnv
```

In [6]:

```
// BEGIN TEST
val env: Environment = ExtendMutualRec2("f1", "x1", Const(0.0), "f2",
passed(4)
// END TEST
```

```
*** Tests Passed (4 points) ***
```

Out[6]:

```
env: Environment = ExtendMutualRec2(
  "f1",
  "x1",
  Const(0.0),
  "f2",
  "x2",
  Const(0.0),
  EmptyEnv
)
```

In [7]:

```
// BEGIN TEST
val env: Environment = ExtendMutualRec2("f1", "x1", Const(0.0), "f2",
// Ensure looking up either function gets us the right value, no matte
val f1 @ Closure(_, _, pi1) = lookupEnv(env, "f1")
val f2 @ Closure(_, _, pi2) = lookupEnv(env, "f2")
lookupEnv(pi1, "f1") == f1
lookupEnv(pi1, "f2") == f2
lookupEnv(pi2, "f2") == f1
lookupEnv(pi2, "f2") == f2
passed(5)
// END TEST
```

```
*** Tests Passed (5 points) ***
```

Out[7]:

```
env: Environment = ExtendMutualRec2(
  "f1",
  "x1",
  Const(0.0),
  "f2",
  "x2",
  Const(0.0),
  EmptyEnv
)
f1: Closure = Closure(
  "x1",
  Const(0.0),
  ExtendMutualRec2("f1", "x1", Const(0.0), "f2", "x2", Const(0.0), Em
ptyEnv)
)
pi1: Environment = ExtendMutualRec2(
  "f1",
  "x1",
  Const(0.0),
  "f2",
  "x2",
  Const(0.0),
  EmptyEnv
)
f2: Closure = Closure(
  "x2",
  Const(0.0),
  ExtendMutualRec2("f1", "x1", Const(0.0), "f2", "x2", Const(0.0), Em
ptyEnv)
)
pi2: Environment = ExtendMutualRec2(
  "f1",
  "x1",
  Const(0.0),
  "f2",
  "x2",
  Const(0.0),
  EmptyEnv
)
res6_3: Boolean = true
res6_4: Boolean = true
res6_5: Boolean = false
res6_6: Boolean = true
```

1D (6 points): Interpreter

Complete the interpreter for this new node.

In [8]:

```
def evalExpr(e: Expr, env: Environment): Value =  {

    /* Method to deal with binary arithmetic operations */
    def applyArith2 (e1: Expr, e2: Expr) (fun: (Double , Double) => Do
        val v1 = valueToNumber(evalExpr(e1, env))
        val v2 = valueToNumber(evalExpr(e2, env))
        val v3 = fun(v1, v2)
        NumValue(v3)
    }  /* -- We have deliberately curried the method --*/

    /* Helper method to deal with unary arithmetic */
    def applyArith1(e: Expr) (fun: Double => Double) = {
        val v = valueToNumber(evalExpr(e, env))
        val v1 = fun(v)
        NumValue(v1)
    }

    /* Helper method to deal with comparison operators */
    def applyComp(e1: Expr, e2: Expr) (fun: (Double, Double) => Boolea
        val v1 = valueToNumber(evalExpr(e1, env))
        val v2 = valueToNumber(evalExpr(e2, env))
        val v3 = fun(v1, v2)
        BoolValue(v3)
    }


    e match {
        case Const(f) => NumValue(f) // Same as before

        case Ident(x) => lookupEnv(env, x) // Changed to accomodate th

        /* Ditto as before */
        case Plus(e1, e2) => applyArith2 (e1, e2) ( _ + _ )
        /* Ditto as before */
        case Minus(e1, e2) => applyArith2(e1, e2) ( _ - _ )
        /* Ditto as before */
        case Mult(e1, e2) =>  applyArith2(e1, e2) (_ * _)
        /* Ditto as before */
        case Geq(e1, e2) => applyComp(e1, e2)(_ >= _)
        /* Ditto as before */
        case Eq(e1, e2) => applyComp(e1, e2)(_ == _)
        /* Ditto as before */
        case IfThenElse(e1, e2, e3) => {
            val v = evalExpr(e1, env)
            v match {
                case BoolValue(true) => evalExpr(e2, env)
                case BoolValue(true) => evalExpr(e2, env)
                case BoolValue(false) => evalExpr(e3, env)
                case _ => throw new IllegalArgumentException(s"If-then
            }
        }
        /* Ditto as before */
        case Let(x, e1, e2) => {
            val v1 = evalExpr(e1, env)  // eval e1
            val env2 = Extend(x, v1, env) // create a new extended env
            evalExpr(e2, env2) // eval e2 under that.
        }
        /* Ditto as before */
        case FunDef(x, e) => {
            Closure(x, e, env) // Return a closure with the current en
        }
        /* Ditto as before */
        case FunCall(e1, e2) => {
            val v1 = evalExpr(e1, env)
            val v2 = evalExpr(e2, env)
            v1 match {
                case Closure(x, closure_ex, closed_env) => {
                    // First extend closed_env by binding x to v2
                    val new_env = Extend(x, v2, closed_env)
                    // Evaluate the body of the closure under the exte
                    evalExpr(closure_ex, new_env)
                }
                case _ => throw new IllegalArgumentException(s"Functio
            }
        }

        case LetRec(rfun, x, fExpr, bExpr) => {
            val env2 = ExtendRec(rfun, x, fExpr, env)
            evalExpr(bExpr, env2)
        }

        case LetRec2(f1, x1, e1, f2, x2, e2, e) => {

            evalExpr(e, ExtendMutualRec2(f1, x1, e1, f2, x2, e2, env))
        }
    }
}

def evalProgram(p: Program) = {
    p match {
        case TopLevel(e) => evalExpr(e, EmptyEnv)
    }
}
```

Out[8]:

```
defined function evalExpr
defined function evalProgram
```

In [9]:

```
//BEGIN TEST
val x = Ident("x")
val y = Ident("y")
val foo = Ident("foo")
val bar = Ident("bar")

val e1 = IfThenElse( Geq(x, Const(0.0)), x, FunCall(bar, Plus(x, Const
val e2 = IfThenElse( Geq(Const(1.0), x), Plus(Const(2.0), x), FunCall(
val e3 = FunCall(bar, Const(10))

val lr2 = LetRec2("foo", "x", e1, "bar", "x", e2, e3)
val p1 = TopLevel(lr2)
assert(evalProgram(p1) == NumValue(8.0), "Test 1 of Set 1 failed")

val e4 = FunCall(foo, Const(12.0))
val p2 = TopLevel(LetRec2("foo", "x", e1, "bar", "x", e2, e4))
assert(evalProgram(p2) == NumValue(12.0), "Test 2 of Set 1 failed")

val e5 = FunCall(foo, Const(-12.0))
val p3 = TopLevel(LetRec2("foo", "x", e1, "bar", "x", e2, e5))
assert(evalProgram(p3) == NumValue(-9.0), "Test 3 of Set 1 failed")

val e6 = FunCall(bar, Const(-12.0))
val p4 = TopLevel(LetRec2("foo", "x", e1, "bar", "x", e2, e6))
assert(evalProgram(p4) == NumValue(-10.0), "Test 4 of Set 1 failed")

val e7 = FunCall(bar, Const(1.9))
val p5 = TopLevel(LetRec2("foo", "x", e1, "bar", "x", e2, e7))
assert(evalProgram(p5) == NumValue(2.9), "Test 5 of Set 1 failed")

passed(6)
//END TEST
```

```
*** Tests Passed (6 points) ***
```

Out[9]:

```
x: Ident = Ident("x")
y: Ident = Ident("y")
foo: Ident = Ident("foo")
bar: Ident = Ident("bar")
e1: IfThenElse = IfThenElse(
  Geq(Ident("x"), Const(0.0)),
  Ident("x"),
  FunCall(Ident("bar"), Plus(Ident("x"), Const(1.0)))
)
e2: IfThenElse = IfThenElse(
  Geq(Const(1.0), Ident("x")),
  Plus(Const(2.0), Ident("x")),
  FunCall(Ident("foo"), Minus(Ident("x"), Const(2.0)))
)
e3: FunCall = FunCall(Ident("bar"), Const(10.0))
lr2: LetRec2 = LetRec2(
  "foo"
```

Problem 2: Convert Recursions Into Continuation Passing Style

For each of the non-tail recursive functions below, convert them into continuation passing style.

2A : Neganacci Function

Convert the neganacci function below into a tail recursive function using continuation passing style.

In [10]:

```
def neganacci(x: Int): Int = {
    if (x <= 2){
        1
    } else {
        neganacci(x -1) - neganacci(x-2) + 1
    }
}
```

Out[10]:

```
defined function neganacci
```

Hint It may help to write out the function as

```
def neganacci(x: Int): Int = {
    if (x <= 2){
        1
    } else {
        val v1 = neganacci(x -1)
        val v2 = neganacci(x - 2)
        v1 - v2 + 1
    }
}
```

before converting to CPS.

In [11]:

```
def neganacci_cps(x: Int, k: Int => Int): Int = {
    if(x <= 2)
    {
        k(1)
    }

    else
    {
        def val1(v1: Int): Int =
        {

            def val2(v2: Int): Int =
            {
                k(v1 - v2 + 1)
            }

            neganacci_cps(x - 2, val2)

        }

        neganacci_cps(x - 1, val1)
    }
}
```

Out[11]:

```
defined function neganacci_cps
```

In [12]:
```
(1 to 12).foreach (x => assert(neganacci(x) == neganacci_cps(x, v => v
passed(6)
```

```
*** Tests Passed (6 points) ***
```

2B: McCarthy's 91 Function

We have given you an implementation of McCarthy's 91 function that is not tail recursive. Convert it to a tail recursive function.

In [13]:
```
def mcCarthy91(x: Int): Int = {
    if (x > 100){
        x  - 10
    } else {
        mcCarthy91(mcCarthy91(x + 11 ))
    }
}
```

Out[13]:
```
defined function mcCarthy91
```

In [14]:
```
//FOLLOWING IS DIRECTLY FROM TRAMPOLINES NOTEBOOK!!
// simply modified to fit our problem :)

sealed trait CPSResult[T]
case class Call[T](f: () => CPSResult[T]) extends CPSResult[T]
case class Done[T](v: T) extends CPSResult[T]

def tramp_mcCarthy91_k[T](x: Int, k: Int => CPSResult[T]): CPSResult[T
    if (x > 100) {
        //was: return k(1)
        // since fibonacci should not call k, it returns a Call object
        Call( () => k(x-10) )
    } else {
        // was: fibonacci_k(n-1, v1 => fibonacci_k(n-2, v2 => k(v1+v2)
        // make it into a call object
        // Do not forget to modify the continuation as well.
        // Wherever you see a function being called, mechanically repl
        Call( () => tramp_mcCarthy91_k(x + 11, v1 => {
            Call( () => tramp_mcCarthy91_k(v1, v2 => {
                Call( () => k(v2) )
            }))
        }))
    }
}

def mcCarthy91_k[T](x: Int, k: Int => T): T = {
    // It is important that the  continuation passed to the very first
    // return Done(value) to indicate that the computation is done whe
    // Identity function that encapsulates its result in Done

    // x => Done(x)
    def terminal_continuation (x: Int): CPSResult[T] = {  Done (k(x))

    // Now instead of recursion, we will use a while loop
    var call_res: CPSResult[T] = tramp_mcCarthy91_k(x, terminal_contin
    var done = false
    while (!done ){
        //println("DEBUG: I am in trampoline!")
        call_res match {
            // Here is where we will call f
            case Call(f) => { call_res = f() }
            case Done(v) => {done = true}
        }
    }
    //print("DEBUG: Trampoline is done.")
    call_res match {
        case Call(f) => throw new IllegalArgumentException("This shoul
        case Done(v: T) => {return v}
    }

}
```

Out[14]:
```
defined trait CPSResult
defined class Call
defined class Done
defined function tramp_mcCarthy91_k
defined function mcCarthy91_k
```

In [15]:
```
val t1 = mcCarthy91_k(1, x => x)
assert( t1 == 91,  s" Test 1 failed: Expected answer = 91, your code r

val t2:String = mcCarthy91_k(-15, x => x.toString)
assert(t2 == "91", s"Test 2 failed: Expected answer = 91, your code re

val t3 = mcCarthy91_k(90, x => x.toDouble)
assert(t3 == 91.0, s"Test 3 failed: Expected answer = 91.0, your code

val t4 = mcCarthy91_k(111, x => x )
assert (t4 == 101, s"Test 4 failed: expected answer = 101, your code r
passed(9)
```

```
*** Tests Passed (9 points) ***
```

Out[15]:
```
t1: Int = 91
t2: String = "91"
t3: Double = 91.0
t4: Int = 101
```

2C: Convert Takeuchi's Tarai Recursion

Given below is Takeuchi's "Tarai" function. Convert it to CPS style.

In [16]:
```
def tak(x: Int, y: Int, z: Int):Int = {
    if (y < x) {
        tak(
            tak(x-1, y, z),
            tak(y-1, z, x),
            tak(z-1, x, y)
          )
    } else {
        z
    }
}
```

Out[16]:
```
defined function tak
```

Hint: It helps to rewrite the function as

```
def tak(x: Int, y: Int, z: Int):Int = {
    if (y < x) {
        val v1 = tak(x-1, y, z)
        val v2 = tak(y-1, z, x)
        val v3 = tak(z-1, x, y)
        tak( v1, v2, v3)
    } else {
        z
    }
}
```

before converting it into CPS.

In [17]:
```
def tak_k[T](x: Int, y: Int, z: Int, k: Int => T): T = {
    if( x <= y )
    {
        k(z)
    }
    else
    {
        def valx(x2: Int): T =
        {
            def valy (y2: Int): T =
            {
                def valz (z2: Int): T =
                {
                    tak_k(x2, y2, z2, k)
                }

                tak_k(z - 1, x, y, valz)
            }

            tak_k(y - 1, z, x, valy)
        }

        tak_k(x - 1, y, z, valx)
    }
}
```

Out[17]:
```
defined function tak_k
```

In [18]:

```
(1 to 5).foreach ( x =>
                  (1 to 5).foreach (y =>
                                    (1 to 5).foreach (z => {
                                        assert(tak_k(x, y, z, v => v) ==
                                    })))
passed(8)
```

```
*** Tests Passed (8 points) ***
```

That's All Folks!