Appendix A

User Guide

A.1 Standard run instructions

To run the software and test it, you can follow these instructions:

1. Install the Scala[7] environment to compile and run the code. You can do so following this link: Download Scala and following the download instructions based on your operating system. The code should run on older version too, but be sure to install the latest version, 3.1.1, as that is the one the program has been tested on. Also, as written in that download page, if you don't have any version of the JDK installed, the installer will download that for you as well, as Scala uses the Java Virtual Machine.

2. After installing Scala, open the command line (or terminal) and go to the folder where the *RIMP-interpreter.sc* file is kept. You can do so by typing the command *cd "example/path"* replacing *example/path* by the path where the interpreter file is.

3. Once the terminal is looking in the right folder (you can usually see the path it is looking into just left of the the typing space) you can run the file, by typing *scala RIMP-interpreter.sc*. This will run the interpreter with the tree I already prepared, and the output will be visible in the same terminal you just used.

A.2 Custom run and test

To modify the abstract syntax tree used as input, and run the tree with custom trees to test the program functionality, follow the instructions below:

1. Make sure you have already done everything in section A.1, to know that the standard program runs and that the terminal is in the right folder.

2. Open the file *RIMP-interpreter.sc* using any text editor, for example Visual Studio Code by Microsoft. You will then find, just at the beginning of the file (specifically starting at lines 3-4) the declaration of the abstract syntax tree that will be used as input, and it's called *PROG*. A screenshot is given below to help identify it.

```
//For the user: modify here the abstract syntax tree to input
in the interpreter
val PROG = Seq(Assign("f1", Num(0)), Seq(Assign("f2", Num(1))
, Seq(Assign("f3", Num(0)), For(Assign("i", Num(3)), BOp("<",
Var("i"), Num(11)), Assign("i", AOp("+", Var("i"), Num(1))),
Seq(Assign("f3", AOp("+", Var("f1"), Var("f2"))), Seq(Assign
("f1", Var("f2")), Assign("f2", Var("f3"))))))))
```

3. Everything after the = sign is the tree, so you can delete it and rewrite it to be any tree that follows the RIMP specification. That means, they will be either an Expression *BExp/AExp* (which is the useless type as there is no computation), or a command, which of course includes the sequence *Seq*, that is the concatenation of two commands. The full specification of the abstract syntax trees in the interpreter will be found just below the declaration of PROG, starting at line 12-13, or in the screenshot below. You should look at the tree that PROG is already assigned to, to get an idea of what kind of combinations can be made. Anything interesting will include sequences. At the same time, because the standard program I used to test is the fibonacci sequence, you could just try changing some of the values of the assignments, for example the length of the sequence, to see the different results. Remember of course that the interpreter will only work for well written trees, and will throw an exception if there is some syntax error, with a more-or-less useful message to help you correct it.

```scala
// the abstract syntax trees for RIMP
abstract class Prog

abstract class Cmd extends Prog
abstract class AExp extends Prog
abstract class BExp extends Prog

case object Skip extends Cmd
case class Assign(l: String, e: AExp) extends Cmd
case class UnAssign(l: String, e: AExp) extends Cmd
case class Seq(c1: Cmd, c2: Cmd) extends Cmd
case class If(e: BExp, c1: Cmd, c2: Cmd) extends Cmd
case class While(e: BExp, c: Cmd) extends Cmd
case class For(c1: Cmd, e: BExp, c2: Cmd, c3: Cmd) extends Cmd
case class Def(n: String, c: Cmd) extends Cmd
case class UnDef(n: String, c: Cmd) extends Cmd
case class Call(n: String) extends Cmd
case class UnCall(n: String) extends Cmd

case class Var(l: String) extends AExp
case class Num(i: Int) extends AExp
case class AOp(op: String, l: AExp, r: AExp) extends AExp

case object True extends BExp
case object False extends BExp
case class BOp(op: String, l: AExp, r: AExp) extends BExp
case class And(l: BExp, r: BExp) extends BExp
case class Not(e: BExp) extends BExp
```

4. After creating your tree and writing at the right side of the = next to PROG, save the changes to the file, and run the interpreter in the same exact way as explained in point 3 of the Standard run instructions (A.1). If the tree is correctly designed, the 3 results should be printed as output to the terminal.

Appendix B

Source Code

I verify that I'm the sole owner of the source code printed below, as well as the programs submitted in the (separate) code submission zip, except where explicitly stated to the contrary.
Christian Impollonia, 08/04/2022

```scala
//import from the scala standard library to acess Try or Else functionality (DO NOT EDIT NEXT LINE)
import scala.util.Try

//For the user: modify here the abstract syntax tree to input in the interpreter
val PROG = Seq(Assign("f1", Num(0)), Seq(Assign("f2", Num(1)), Seq(Assign("f3", Num(0)), For(Assign("i", Num(3)), BOp("<", Var("i"), Num(11)), Assign("i", AOp("+", Var("i"), Num(1))), Seq(Assign("f3", AOp("+", Var("f1"), Var("f2"))), Seq(Assign("f1", Var("f2")), Assign("f2", Var("f3"))))))))





//Here begins the code. Do not edit after this unless a developer who wants to change the functionality.

// the abstract syntax trees for RIMP
abstract class Prog

abstract class Cmd extends Prog
abstract class AExp extends Prog
abstract class BExp extends Prog

case object Skip extends Cmd
case class Assign(l: String, e: AExp) extends Cmd
case class UnAssign(l: String, e: AExp) extends Cmd
case class Seq(c1: Cmd, c2: Cmd) extends Cmd
case class If(e: BExp, c1: Cmd, c2: Cmd) extends Cmd
case class While(e: BExp, c: Cmd) extends Cmd
case class For(c1: Cmd, e: BExp, c2: Cmd, c3: Cmd) extends Cmd
case class Def(n: String, c: Cmd) extends Cmd
case class UnDef(n: String, c: Cmd) extends Cmd
case class Call(n: String) extends Cmd
case class UnCall(n: String) extends Cmd

case class Var(l: String) extends AExp
case class Num(i: Int) extends AExp
case class AOp(op: String, l: AExp, r: AExp) extends AExp

case object True extends BExp
case object False extends BExp
case class BOp(op: String, l: AExp, r: AExp) extends BExp
case class And(l: BExp, r: BExp) extends BExp
case class Not(e: BExp) extends BExp

//these classes are necessary to implement reversibility, do not include them in an input tree
case class AugWhile(n: Int, e: BExp, c: Cmd) extends Cmd
case class AugFor(n: Int, c1: Cmd, e: BExp, c2: Cmd, c3: Cmd) extends Cmd



//the table containing function definitions
type DefT = Map[String, (Cmd, Cmd)]

//the table containing the loop conditions
type CondT = Map[Int, BExp]

```

```
// the code augmentation function for a RIMP parse tree

var counter = -1

def Fresh(x: String) = {
  counter += 1
  x ++ "_" ++ counter.toString()
}

def aug(c: Cmd, t: CondT) : (Cmd, CondT) = c match {
    case While(e, c1) => {
        val str = Fresh("counter")
        val t_new = t + (counter -> e)
        val c1_new = aug(c1, t_new)
        (Seq(Assign(str, Num(0)), AugWhile(counter, e, Seq(c1_new._1, Assign(str, AOp("+", Var(str), Num(1)))))), c1_new._2)
    }
    case For(c1, e, c2, c3) => {
        val str = Fresh("counter")
        if (!c1.isInstanceOf[Assign]) throw new Exception("Initial statement in for loop " + counter + " must be assignment, instead it is: " + c1)
        val t_new = t + (counter -> e)
        val c2_new = aug(c2, t_new)
        val c3_new = aug(c3, c2_new._2)
        (Seq(Assign(str, Num(0)), AugFor(counter, c1, e, Seq(c2_new._1, Assign(str, AOp("+", Var(str), Num(1)))), c3_new._1)), c3_new._2)
    }
    case If(b, c1, c2) => {
        val e_prime = replace_exp(b)
        val assignments = create_assignments(b)
        (Seq(assignments, If(e_prime, c1, c2)), t)
    }
    case Seq(c1, c2) => {
        val c1_new = aug(c1, t)
        val c2_new = aug(c2, c1_new._2)
        (Seq(c1_new._1, c2_new._1), c2_new._2)
    }
    case _ => (c, t)
}

// helper function to replace the old variables with the duplicates for ifs in aug
def replace_exp(b: BExp) : BExp = b match {
    case True => True
    case False => False
    case BOp(s, Var(l), Var(t)) => BOp(s, Var(l + "_prime"), Var(t + "_prime"))
    case BOp(s, Var(l), t) => BOp(s, Var(l + "_prime"), t)
    case BOp(s, l, Var(t)) => BOp(s, l, Var(t + "_prime"))
    case BOp(s, l, t) => BOp(s, l, t)
    case And(b1, b2) => And(replace_exp(b1), replace_exp(b2))
    case Not(b) => Not(replace_exp(b))
}

// helper function to create the duplicate variables for ifs in aug
def create_assignments(b: BExp) : Cmd = b match {
    case True => Skip
    case False => Skip
    case BOp(s, Var(l), Var(t)) if l == t => Assign(l + "_prime", Var(l))
    case BOp(s, Var(l), Var(t)) => Seq(Assign(l + "_prime", Var(l)), Assign(t + "_prime", Var(t)))
    case BOp(s, Var(l), t) => Assign(l + "_prime", Var(l))
```

```
    case BOp(s, l, Var(t)) => Assign(t + "_prime", Var(t))
    case BOp(s, l, t) => Skip
    case And(b1, b2) => Seq(create_assignments(b1), create_assignments(b2))
    case Not(b) => create_assignments(b)
}


//helper function for rev, to check if counter is appropriate one
def check_counter(n: Int, b: BExp) : Boolean = b match {
    case BOp(">", Var(l), Num(0)) if (l == "counter_" ++ n.toString()) => true
    case _ => false
}

//the inversion function for RIMP parse trees

def rev(c: Cmd, t: CondT) : Cmd = c match {
    case Assign(l, e) => UnAssign(l, e)
    case UnAssign(l, e) => Assign(l, e)
    case Skip => Skip
    case Seq(AugFor(i, Skip, b, c1, c2), c3) if (check_counter(i, b)) => AugFor(i,
rev(c3, t), t(i), rev(c2, t), rev(c1, t))
    case Seq(c1, c2) => Seq(rev(c2, t), rev(c1, t))
    case If(e, c1, c2) => If(e, rev(c1, t), rev(c2, t))
    case AugWhile(i, b, c) if (check_counter(i, b)) => AugWhile(i, t(i), rev(c, t))
    case AugWhile(i, e, c) => AugWhile(i, BOp(">", Var("counter_" ++ i.toString()),
Num(0)), rev(c, t))
    case AugFor(i, c1, e, c2, c3) => Seq(AugFor(i, Skip, BOp(">", Var("counter_" ++
i.toString()), Num(0)), rev(c3, t), rev(c2, t)), rev(c1, t))
    case Def(n, c) => UnDef(n, c)
    case UnDef(n, c) => Def(n, c)
    case Call(n) => UnCall(n)
    case UnCall(n) => Call(n)
    case _ => c

}



// the evaluator for a RIMP program


abstract class V

case object ZERO extends V
case class PLUS(n: Int, v: V) extends V

//the main store of variables in RIMP
type Store = Map[String, (Int, V)]

//helper function to get n part of store
def get_n(v: V) : Int = v match {
    case ZERO => 0
    case PLUS(n, v) => n
}

//helper function to get v part of store
def get_v(v: V) : V = v match {
    case ZERO => ZERO
    case PLUS(n, v) => v
}
```

```scala

// evaluator for arithmetic expressions
def ev_aexp(a: AExp, s: Store) : Int = a match {
    case Num(i) => i
    case Var(l) => Try(s(l)._1).getOrElse(throw new Exception("Tried to access undeclared variable called " + l))
    case AOp("+", l, r) => ev_aexp(l, s) + ev_aexp(r, s)
    case AOp("-", l, r) => ev_aexp(l, s) - ev_aexp(r, s)
    case AOp("*", l, r) => ev_aexp(l, s) * ev_aexp(r, s)
    case AOp("/", l, r) => ev_aexp(l, s) / ev_aexp(r, s)
    case _ => throw new Exception("Poorly written arithmetic expression: somewhere in " + a)
}

// evaluator for boolean expressions
def ev_bexp(b: BExp, s: Store) : Boolean = b match {
    case True => true
    case False => false
    case BOp(">", l, r) => ev_aexp(l, s) > ev_aexp(r, s)
    case BOp("<", l, r) => ev_aexp(l, s) < ev_aexp(r, s)
    case BOp("=", l, r) => ev_aexp(l, s) == ev_aexp(r, s)
    case And(l, r) => ev_bexp(l, s) && ev_bexp(r, s)
    case Not(e) => !(ev_bexp(e, s))
    case _ => throw new Exception("Poorly written boolean expression: somewhere in " + b)
}

// evaluator for commands
def ev_cmd(c: Cmd, s: Store, d: DefT, t: CondT) : (Store, DefT, CondT) = c match {
    case Skip => (s, d, t)
    case Assign(l, e) =>
        if (s.contains(l)) {
            (s + (l -> (ev_aexp(e, s), PLUS(ev_aexp(e, s) - s(l)._1, s(l)._2))), d, t)
        }
        else {
            (s + (l -> (ev_aexp(e, s), PLUS(ev_aexp(e, s), ZERO))), d, t)
        }
    case UnAssign(l, e) => (s + (l -> (s(l)._1 - get_n(s(l)._2), get_v(s(l)._2))), d, t)
    case If(e, c1, c2) => if (ev_bexp(e, s)) ev_cmd(c1, s, d, t)  else ev_cmd(c2, s, d, t)
    case AugWhile(i, e, c) => {
        val state_new = ev_cmd(c, s, d, t)
        if (ev_bexp(e, s)) ev_cmd(AugWhile(i, e, c), state_new._1, state_new._2, state_new._3) else (s, d, t)
    }
    case AugFor(i, c1, e, c2, c3) => ev_cmd(Seq(c1, AugWhile(i, e, Seq(c3, c2))), s, d, t)
    case Def(n, c) => ev_cmd(Skip, s, (d + (n -> ((c, rev(c, t))))), t)
    case UnDef(n, c) => ev_cmd(Skip, s, d - (n), t)
    case Call(n) => Try(ev_cmd(d(n)._1, s, d, t)).getOrElse(throw new Exception("Could not find procedure called " + n))
    case UnCall(n) => ev_cmd(d(n)._2, s, d, t)
    case Seq(c1, c2) => {
        val state_new = ev_cmd(c1, s, d, t)
        ev_cmd(c2, state_new._1, state_new._2, state_new._3)
    }
    case _ => throw new Exception("Poorly written command: somewhere in " + c)
}
```

```
//main interpreter function: creates the store, definitions table and conditions table, then runs augmentation function and evaluation
//on the input tree as well as building the back stack and then evaluating that too. Returns the state after
//evaluation of the tree, then the inverse tree, and then the state after the evaluation of the inverse tree

def interpreter(p: Prog) = {
    val store = Map[String, (Int, V)]()
    val defT = Map[String, (Cmd, Cmd)]()
    val condT = Map[Int, BExp]()
    if(p.isInstanceOf[AExp]) {
        ev_aexp(p.asInstanceOf[AExp], store)
        println("The store contents after running the tree are: " + store + "\n")
        println("The reversed tree is: " + p + "\n")
        println("The store contents after running the reversed tree are: " + store + "\n")
    }
    else if(p.isInstanceOf[BExp]) {
        ev_bexp(p.asInstanceOf[BExp], store)
        println("The store contents after running the tree are: " + store + "\n")
        println("The reversed tree is: " + p + "\n")
        println("The store contents after running the reversed tree are: " + store + "\n")
    }

    else {
        val c = p.asInstanceOf[Cmd]
        val c_aug = aug(c, condT)
        val c_new = ev_cmd(c_aug._1, store, defT, c_aug._2)
        println("The store contents after running the tree are: " + c_new._1 + "\n")
        val rev_c = rev(c_aug._1, c_new._3).asInstanceOf[Cmd]
        println("The reversed tree is: " + rev_c + "\n")
        println("The store contents after running the reversed tree are " + ev_cmd(rev_c, c_new._1, c_new._2, c_new._3)._1 + "\n")
    }

}

//main function of the whole program: runs the interpreter using as input the tree prog, created in the first line of this program
@main
    def main() : Unit = {
        interpreter(PROG)
    }
```