COMP 332
Translation For Lintilla Language

Elie Ajaka
44929498

Introduction

The purpose of this assignment was to finalise the implementation for the translator by implementing **short circuit evaluation** (**And [&&], Or [||], Not [~]**) as well as **array operations** and **for loops, Loops and breaks**, for the Lintilla programming language.

I will explain in detail how I had completed these tasks, what challenges I had struggled with and how I had overcome these challenges.

Throughout the documentation, I will also be providing snippets of code with a description explaining the code's functionality and how I had gone about Implementing it.

There will be three sections to this documentation, the first being the understanding and choice of implementation for the assignment, the second being the construction of the **translator** code which will be split into another 3 minor sections explaining the implementation of the **short circuit evaluation**, **array operations** and **for loops, loops and breaks** and finally the third which will entail the tests I have implemented and a short description as to why I had chosen those tests.

Understanding the assignment

To comprehend the assignment I examined the documentation which was provided (**README.md** and **assignment3.md**) I had found a sufficient amount of relevant information as well as some snippets of code which was provided by Dominic. This gave me great insight on how to start and complete the assignment.
I had also examined the code in **Translator.scala, SECDTree.scala, SECDMachine.scala, LintillaTree.scala** and **ExecTests.scala**, I had found the provided comments allowed me to clarify what the code itself does without having to try and fully understand the implementation of it.

Translator.scala helped me understand the following

- **translateToFrame**
- **translateExp**
- **gen**

SECDTree.scala assisted me in understanding what all of the **"I"** classes do and allowed me to understand what arguments needed to be passed to their formal parameters.

SECDMachine.scala allowed me to understand all of the **tree structures representing the SEC program** as well as how they function.

LintillaTree.scala showed me how the expression classes operate Eg **OrExp** allowed me to examine what was being passed as the arguments to the class.

Eg **OrExp(left : Expression, right : Expression)**
It has also given me insight as to how they function.

ExecTests.scala had assisted me in understanding how to write my own test cases.

Translator Implementation

Short Circuit Evaluation

After analysing the code provided, I had found that the previously constructed **IfExp** was useful in showcasing how to construct my code.

Firstly I implemented a new case **AndExp** which has a **left** and **right** formal parameter I had found this information from **LintillaTree.scala.** After doing so I had translated the left and **generated** an **IBranch.**
After scanning the comments above **IBranch** it had explained in detail its functionality which pops a bool value from the top of the stack and if it's true executes the left otherwise it executes the right.

```
case AndExp(l, r) =>
 translateExp(l)
 gen(IBranch(translateToFrame(List(r)),
   List(IBool(false))))
```

After some time I realized that passing the **r** parameter as the first argument then passing **IBool(false)** as the second for **IBranch** was correct for constructing this short circuit evaluation.
If implemented the other way around will not correctly execute.

I carried on to implement the following for the **Or** and **Not** cases with some minor adjustments from the previously explained **AndExp**.

```
case OrExp(l, r) =>
 translateExp(l)
 gen(IBranch(List(IBool(true)),
   translateToFrame(List(r))))

case NotExp(exp) =>
 translateExp(exp)
 gen(IBranch(List(IBool(false)),
   List(IBool(true))))
```

OrExp - If the left is true we return true otherwise evaluate the right.
NotExp - If exp is true return false otherwise return true.

Notice that we translate the right expression/argument to a **frame** and pass this as a **list,** this is because **IBranch** takes 2 arguments **left and right frame**, a frame is constructed by a list of **instr.**
We also create a **List(IBool(...))** as **IBool** extends **instr** and must be passed as a list to properly integrate itself with **IBranch.**

Array

Once I had completed the ShortCircuit Evaluation I went back to view the code and found **IArray**, I noticed that it took no parameters and tried to implement it in **Translator.scala.** After seeing **arrayExp** needed a type I was bewildered on how to integrate this with **IArray** as it had no parameters. Upon reviewing I noticed it was as simple as **generating IArray** without passing it an argument.

```
case ArrayExp(t) =>
 gen(IArray())
```

After implementing **arrayExp** I perpetuated through the rest of my work and noticed that creating the array cases was as simple as translating the parameters and generating the **SECDTree** classes.

The **AssignExp** within this implementation is incorrect see the next snippet for the correct implementation.

```
case ArrayExp(t) =>
 gen(IArray())

case LengthExp(exp) =>
 translateExp(exp)
 gen(ILength())

case DerefExp(array, idx) =>
 translateExp(array)
 translateExp(idx)
 gen(IDeref())

//Incorrect Implementation
case AssignExp(left, right) =>
 translateExp(left)
 translateExp(right)
 gen(IUpdate())

case AppendExp(array, exp) =>
 translateExp(array)
 translateExp(exp)
 gen(IAppend())
```

One of the cases that had me puzzled was **AssignExp** as I hadn't noticed that it had been explained within the documentation. I realised that it required me to pass not just a **left expression** as shown above but required you to pass **DerefExp** as a parameter.

```
case AssignExp(DerefExp(arr,idx), right) =>
 translateExp(arr)
 translateExp(idx)
 translateExp(right)
 gen(IUpdate())
```

For Loop

After completing the short circuit evaluation and arrays I carried on reading through the documentation and grasped that the majority of the code for the **for loops**, **loop** and **break** had been provided by Dom.
[Dom Assignment 3] [Under for loops, loop and break operations]

This provided the entirety of the loop body, with some minor adjustments being needed to have it implemented correctly.

The code of which I had implemented after following the notes that were provided are as follows:

The following will ensure that it will default to one step if no step has been provided.

```
step_value = 1
// if no provided step get options value
if (step.nonEmpty){
 step_value = evalIntConst(step.get)
}
```

I also modified the case arguments to pass **body** as a **block**.

```
case ForExp(IdnDef(id),from,to,step,Block(body)) =>
```

I then translated **from** and **to**

```
translateExp(from)
translateExp(to)
```

Elie Ajaka
44929498
Macquarie University

On step 8 I simply created an **if statement** and followed the instructions from the documentation and used the supplied code as the body of the **if-else** statement

```
def testLoopTermination(a : Int): List[Instr] = {
 if (a > 0) {
  List(
   IVar("_to"),
   IVar(control_var),
   ILess()
  )
 }
 else {
  List(
   IVar(control_var),
   IVar("_to"),
   ILess()
  )
 }
}
```

I then implemented the for loop, modifying segments to allow for proper functionality I had done so by concatenating the lists together with the **testLoopTermination** and the **body.** Whilst doing so I noticed some minor bugs and spelling mistakes which I believe were intentional to throw us off. Eg **“cont” was spelled as “count”**

```
gen(IClosure(
  None,
  List("_from", "_to", "_break_cont"),
  List(
   IClosure(
    None,
    List("_loop_cont"),
    List(
      IVar("_from"),
      IVar("_loop_cont")
    )
  ),
  ICallCC(),
  IClosure(
    None,
    List(control_var, "_loop_cont"),
    testLoopTermination(step_value) ++
     List(IBranch(
       List(
         IVar("_break_cont"),
         IResume()
       ),
       List()
```

Elie Ajaka
44929498
Macquarie University

```
        )) ++
        translateToFrame(body) ++
        List(
          IVar(control_var),
          IInt(step_value),
          IAdd(),
          IVar("_loop_cont"),
          IVar("_loop_cont"),
          IResume()
        )
      ),
    ICall()
  )
))
  gen(ICallCC())
```

Finally I incorporated **break** and **loop** cases which was as simple as using the provided code within the documentation.

```
case BreakExp() =>
 gen(IDropAll()).
 gen(IVar("_break_cont"))
 gen(IResume())

case LoopExp() =>
 gen(IDropAll())
 gen(IVar(control_var))
 gen(IInt(step_value))
 gen(IAdd())
 gen(IVar("_loop_cont"))
 gen(IVar("_loop_cont"))
 gen(IResume())
```

Tests

I have conducted an estimate of 25 tests for the program I produced these tests after the completion of each stage of the assignment.

This was done to ensure I had constructed the program correctly and was able to make fixes sooner rather than later.

Short Circuit Evaluation Tests

And - &&

I constructed various tests to ensure that **And** had been executing correctly, I found how to test these cases for short circuit evaluation from the documentation.

This tests the short circuiting of **And** by simply checking the print output.
If we only see a print result of **false** and not **false true true** we will know that it short circuited.

```
test("Test true && true") {
 execTestInline("""print(true && true)""".stripMargin, "true\n")
}
test("Test false && true") {
 execTestInline("""print(false && true)""".stripMargin, "false\n")
}
test("Test true && false") {
 execTestInline("""print(true && false)""".stripMargin, "false\n")
}
test("Test Short Circuit && ") {
 execTestInline("""print({false} && {print true; true})""".stripMargin, "false\n")
}
```

Or - ||

Similar to the tests above I have tested for short circuiting for **Or**.
I constructed similar tests as above but modifying it slightly ensuring that it short circuits if the left side of the operator is true.

```
test("Test true || true") {
 execTestInline("""print(true || true)""".stripMargin, "true\n")
}
test("Test false || true") {
 execTestInline("""print(false || true)""".stripMargin, "true\n")
}
test("Test false || false") {
 execTestInline("""print(false || false)""".stripMargin, "false\n")
}
test("Test Short Circuit true || true ") {
 execTestInline("""print({true} || {print true; true})""".stripMargin, "true\n")
}
test("Test Short Circuit true || false ") {
 execTestInline("""print({true} || {print false; false})""".stripMargin, "true\n")
}
```

Not - ~

This was a simple test which ensured that the opposite of the input was printed.
Eg ~false should return true

```
test("Test ~false") {
 execTestInline("""print({~false})""".stripMargin, "true\n")
}
test("Test ~true") {
```

```
 execTestInline("""print({~true})""".stripMargin, "false\n")
}
```

Array Tests

I implemented a test for every constructed array case which ensured that they were functioning correctly, starting from the **empty array**, **Length**, **Deref** and finally **Assign** case.

This test prints an empty array.

```
// Empty Array
test("Test arrays") {
 execTestInline(
   """print(array int)""".stripMargin, "empty array\n")
}
```

Creates an array of ints and assign some values to the array then print its length.

```
//Array Length
test("Test arrays length") {
 execTestInline(
   """let a = array int;
     |a += 4;
     |a += 5;
     |print(length(a))""".stripMargin, "2\n")
}
```

Assign values to the created array and dereference the value at index 1.

```
//Array Deref
test("Test arrays deref") {
 execTestInline(
   """let a = array int;
     |a += 4;
     |a += 5;
     |print(a ! 1)""".stripMargin, "5\n")
}
```

Replaces the value at index 1 of the array with 10 and print the value at index 1.

```
//Array Assign
test("Test arrays assign") {
 execTestInline(
   """let a = array int;
     |a += 4;
     |a += 5;
     |a ! 1 := 10;
     |print(a ! 1)""".stripMargin, "10\n")
}
```

For Loop, Loop And Break Tests

I implemented a various amount of tests for **for loops.** I firstly created a test which was a simple for loop and had a step of one.

```
// for loop step 1
test("Test For Loops") {
 execTestInline(
   """for j = 0 to 4 step 1 do {
     | print j
     |}""".stripMargin, "0\n1\n2\n3\n4\n")
}
```

Then I created a test with a step of 2 and another test with no step to ensure that the steps were modifiable and will default to 1 if there was no step integrated within the **for loop** creation.

```
//For loop step 2
test("Test For Loops step 2") {
 execTestInline(
   """for j = 0 to 4 step 2 do {
     | print j
     |}""".stripMargin, "0\n2\n4\n")
}


//For loop no step
test("Test For Loops no step") {
 execTestInline(
   """for j = 0 to 4 do {
     | print j
     |}""".stripMargin, "0\n1\n2\n3\n4\n")
}
```

When I saw these tests ran correctly I then further wanted to tests to see if I was able to conduct **nested for loops** and created two tests. One printing the outer loop iterations and the other printing the inner loop iterations.

```
//Nested for loop Outer
test("Test For test nested loop (Outer)") {
 execTestInline(
   """for j = 0 to 2 step 1 do {
     | for i = 0 to 2 step 1 do {
     |   print j
     | }
     |}
     |""".stripMargin, "0\n0\n0\n1\n1\n1\n2\n2\n2\n")
```

```
}
//Nested for loop Inner
test("Test For test nested loop (Inner)") {
 execTestInline(
   """for j = 0 to 2 step 1 do {
     | for i = 0 to 2 step 1 do {
     |   print i
     | }
     |}
     |""".stripMargin, "0\n1\n2\n0\n1\n2\n0\n1\n2\n")
}
```

I noticed that **loop** was just like a **continue,** where it would skip an iteration of the **for loop.** I had incorporated an **if statement** into the **for loop,** when the iteration was 2 I had called loop to skip over that iteration.

```
//Loop test
test("Test For LOOP") {
 execTestInline(
   """for j = 0 to 3 step 1 do {
     | if (j = 2) {loop} else {};
     | print j
     |}""".stripMargin, "0\n1\n3\n")
}
```

Similarly, with the break statements I incorporated an if statement and broke out of the loop when the iteration had reached 7. I further tested to see if it would **break** without an **if statement.**

```
//Break test
test("Test For Loops Break") {
 execTestInline(
   """for j = 0 to 10 step 2 do {
     | if (j = 7) {break} else {}
     |}""".stripMargin, "")
}
//Break loop to bool
test("Test For Loops break to bool") {
 execTestInline(
   """for j = 0 to 10 step 2 do {
     | print 1;
     | break;
     | print 45
     |};
     |print (true)""".stripMargin, "1\ntrue\n")
}
```