Lintilla translation and execution

Ahmad Sikder. 44229828
COMP332 Programming Languages 2019

Overview

The report outlines the results of a project which worked to complete the extension of the Lintilla Programming language. In this project, translations of the following operators were added:

Logical Operators - &&(AND), ||(OR), ~(NOT)

Logical operators && and || compare two boolean values to produce a boolean output. Logical operator ~ Not returns the changed value of a boolean. Logical operators in Lintilla must follow the contract of short circuit evaluation.

For example, in the logical && Operator, the right side of the operation will only be checked if and only if left side of the expression is true. Short circuit terminates the evaluation if the left side of the expression is false, without checking Q

In the logical || operator, the right side of the operation will only be check if and only if the left side of the expression is false. Short circuit terminates the evaluation if left side of the expression is true.

Array Operators

Arrays in Lintilla are mutable and can contain any type including functions and arrays. Arrays are initialized using the array t expression, updated using the ≔ operator, appended using using the += operators and dereferenced using the ! operator. The arrays in Lintilla can be dereferenced using ! operator, which is done using the expression: array!index. The same can be used alongside the ≔ operator to update the ith entry of the array, which is done using the expression array!index ≔ expression, where expression is the value to be added to indexth entry of the array.

Loop Operators

Loop operators include for and while loops. In this project the translation for for-loops were implemented. In Lintilla follows a for-do loop convention. This follows the syntax:

"for" idndef "=" logexp "to" logexp ("step" logexp)? "do" block [1]

The semantic rules for optional step expression is that it should be 1 when not present else a constant integer. An example of a for loop would be:

for i=0 to 3 do {print 3}

This will produce the output [3333]

Lintilla language and Translation

The Lintilla compiler uses an abstract machine called the SECD machine, to translate the scala translation codes to lintilla language.

The Lintilla language is a functional programming language that uses only expressions in place of statement. Variables in this language are immutable and are declared using the Let expression. The language can declare functions like any other programming language.

Design and Implementation

The Design and translation of this report was needed to be done using the built in SECD in machine in the sbt module. The module also contained the semantic analyzer and the syntax analyzer for the Lintilla language. Translation scheme of each expression was made by following the rules of and analyzers and using instructions of the SECD machine.

Logical operators:

As described above, logical operators compare two boolean values to return a boolean output. To outline the working of these operators truth tables can be used. The truth tables of the &&, ||, ~ operators are shown below.

P	Q	P&&Q
True	True	True
True	False	False
False	True	False
False	False	False

P	Q	P || Q
True	True	True
True	False	True
False	True	True
False	False	False

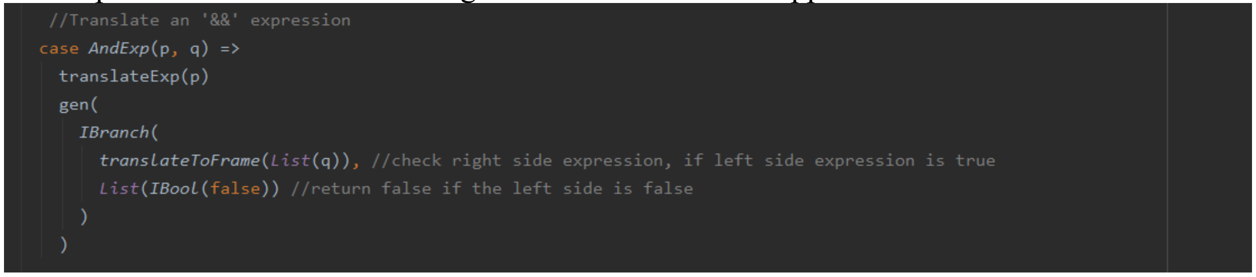
P	~P
True	False
False	True

The short circuit evaluation terminates the evaluation of the logical operation as soon as first condition that satisfies or negates the operation is found.

The AND expression '&&'- As shown above the and expression returns true if and only if both sides of the expression is true.
This knowledge can be used for the short circuit evaluation of the && operator: For the operation to return true. both sides of the expression need to be true. So, Q shall be checked if and only if P is true. If P is false, short circuit logic must be used to end the termination and return false.
Hence the translation scheme to be passed to SECD machine for the above will be

```
<SECD code translation of p>
IBranch(
    <SECD code translation of q>,
    List(IBool(false))
)
```

The IBranch function pops a boolean value from the operand stack of the machine. If the popped value is true then the execution of the code continues with the first code instruction in the IBranch. Or else it continues with the second instruction. The translation scheme above ensures short circuit by having the second instruction of the IBranch to be false.
The implementation of the following is shown in the code snippet below:

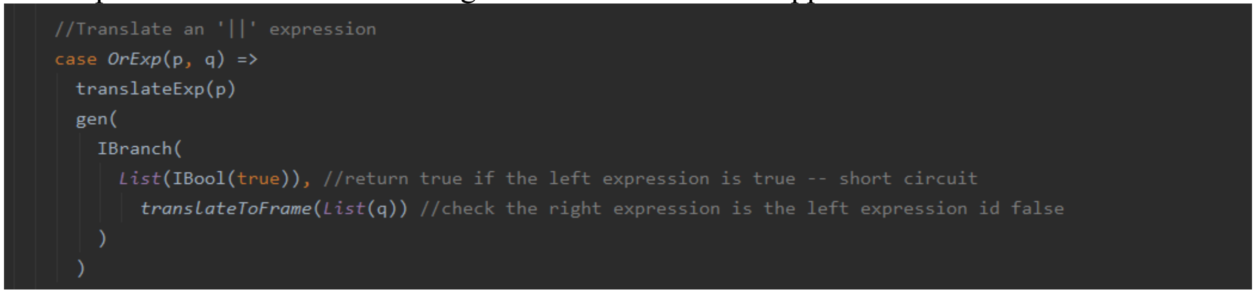


```
  //Translate an '&&' expression
 case AndExp(p, q) =>
   translateExp(p)
   gen(
     IBranch(
       translateToFrame(List(q)), //check right side expression, if left side expression is true
       List(IBool(false)) //return false if the left side is false
     )
   )
```

The OR expression '||'- As shown above the OR expression returns false if and only if both sides of the expression is false.
This knowledge can be used for the short circuit evaluation of the || operator: For the operation to return false, both sides of the expression need to be false. So, Q shall be checked if and only if P is false. If P is true, short circuit logic must be used to end the termination and return true.
Hence the translation scheme to be passed to SECD machine for the above will be

```
<SECD code translation of p>
IBranch(
    List(IBool(true))
    <SECD code translation of q>,
)
```

Here, if the popped value is true then the execution of the code continues with the first code instruction in the IBranch. Hence short circuit terminates the operation when p is true. Or else it continues with the second instruction to check if Q is true. The translation scheme above ensures short circuit by having the first instruction of the IBranch to be true.
The implementation of the following is shown in the code snippet below:

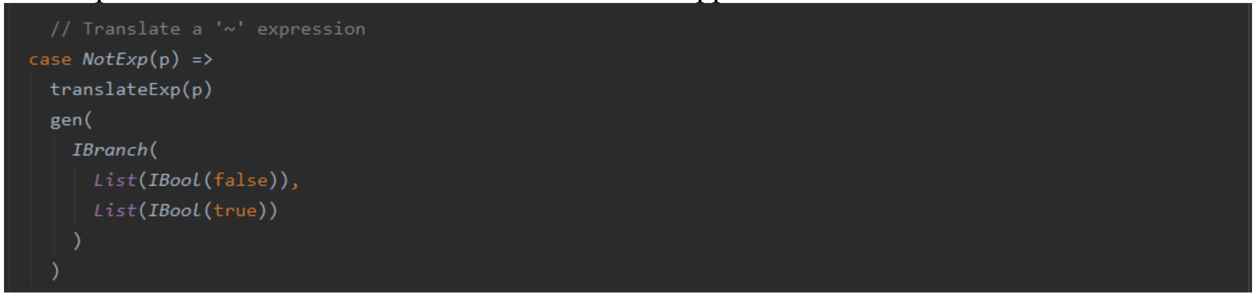


```
//Translate an '||' expression
case OrExp(p, q) =>
  translateExp(p)
  gen(
    IBranch(
      List(IBool(true)), //return true if the left expression is true -- short circuit
        translateToFrame(List(q)) //check the right expression is the left expression id false
    )
  )
```

The NOT expression '~' is used to return the opposite boolean value of the expression.
IBranch can also be used for the translation scheme in this case, if the expression popped from the operand stack is true, then return the first instruction false. Or else return the second instruction true. The translation scheme of this fact is show below

```
<SECD code translation of p>
IBranch(
    List(IBool(true))
    List(IBool(false)),
)
```

The implementation of above is shown in the code snippet below:



```
  // Translate a '~' expression
case NotExp(p) =>
  translateExp(p)
  gen(
    IBranch(
      List(IBool(false)),
      List(IBool(true))
    )
  )
```

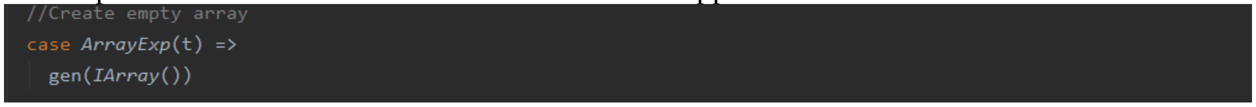
Array operators:

An array in lintilla is initialized using the *array t* expression. Where t stands for the type of array being created. So, the first operator of the array would be to initialize an empty array that has the type t. Once an array is made, it must be able to populate the array. In Lintilla arrays are mutable and must have a functionality to append values of type *t* into them. The concept is the same as the *list.add* functionality in Java or Python. This can be done using the expression: *arr += v,* where arr is the array and v is the value to be added at the end of the array.
Then, the array must have a functionality to dereference a value from an index of an array. In Java this functionality can be done using arr[i] method (where arr is the array and i is the index). However, in Lintilla since all statements are expressions, the method can be done using *arr!i* expression, where arr is the array, i is the index.
Furthermore, the above expression can also be used to add a value to a index. This is done using an ≔ expression next to the above expression. So, to add a value to an index the *arr!i ≔ v* expression must be used.
Finally, arrays must have the functionality to retrieve the length of the array. In Lintilla the functionality is done using the length(arr) method.

Initialisation of an empty array: array t expression – This expression allows to create an empty array of the type t in the program. The type t of the array is defined in the semantic analyser of the skeleton. The SECD machine tree contains an *IArray()* instruction, which creates an empty array and pushes onto the operand stack. So the translation scheme of this expression will be

```
Generate IArray()
```

The implementation of above is shown in the code snippet below

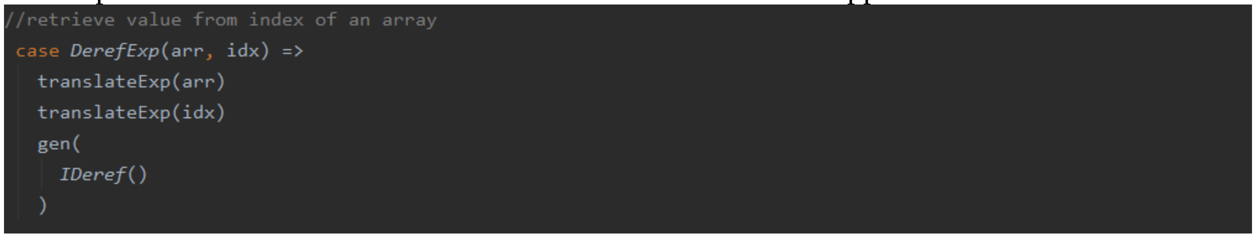


```
//Create empty array
case ArrayExp(t) =>
  gen(IArray())
```

Dereferencing an Index of an array: arr!i – The expression gets the value from index i of an array. The SECD machine tree contains an IDeref instruction, which pops an Integer index *idx* and an array *arr* from the top of the operand stack of the machine. And then it pushes the value back into the specified index of that array.
Hence, the translation scheme for the above can be written as:

```
<SECD code translation of array arr>
<SECD code translation of index idx>
IDeref()
```

The implementation of the above scheme is shown in the code snippet below

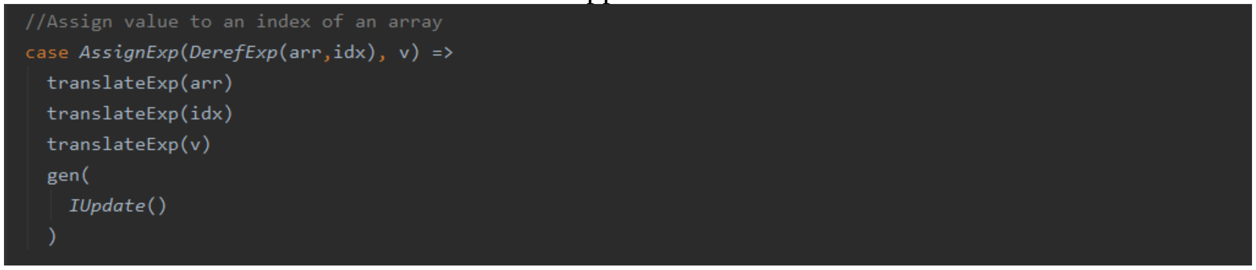


```
//retrieve value from index of an array
 case DerefExp(arr, idx) =>
   translateExp(arr)
   translateExp(idx)
   gen(
     IDeref()
   )
```

Assigning value to an index of an array: arr!i ≔ v – The expression first evaluates expression v, which must be of type t of the array arr. It then assigns the resulting value to ith entry of the array arr. The SECD machine tree contains an IUpdate() instruction, which pops a value v, an Integer index idx and an array arr from the top of the operand stack. It then puts value v into the index i of the array. Using this fact the translation scheme can be written as:

```
<SECD code translation of array arr>
<SECD code translation of integer idx>
<SECD code translation of value v>
IUpdate()
```

The semantic analyser only allows a dereference expression to appear on the left side of the symbol '≔'. Hence the expression arr!i ≔ v was treated as a single ternary operator and translated in the form shown in the code snippet below

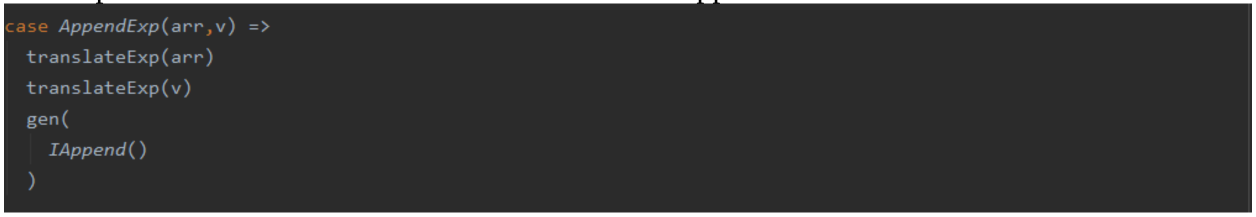


```
//Assign value to an index of an array
case AssignExp(DerefExp(arr,idx), v) =>
  translateExp(arr)
  translateExp(idx)
  translateExp(v)
  gen(
    IUpdate()
  )
```

Appending value to an array: arr += v – The expression appends a value to the tail of the array, the value must be of the type t. The SECD machine tree contains an IAppend() instruction, which pops a value and an array from the top of the operand stack, and then extends the translated array by adding the value to the end of the array.
Hence, the translation scheme for the above can be written as:

```
<SECD code translation of array arr>
<SECD code translation of value v>
IAppend()
```

The implementation of above is shown in the code snippet below

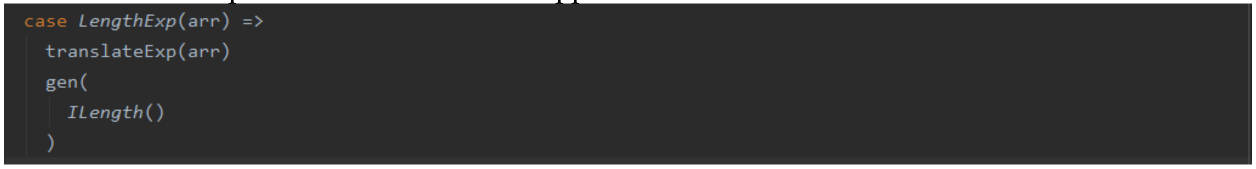


```
case AppendExp(arr,v) =>
  translateExp(arr)
  translateExp(v)
  gen(
    IAppend()
  )
```

Retrieving the length of an arry: length(arr) – The expression returns the number of elements in the array. The SECD machine tree contains an ILength() instruction, which pops an array from the top of the operand stack, retrieves the number of entries in the array and pushes it onto the top of the operand stack.
Hence, the translation scheme for the above can be written as:

```
<SECD code translation of array arr>
ILength()
```

The scheme is implemented in the code snippet below



```
case LengthExp(arr) =>
  translateExp(arr)
  gen(
    ILength()
  )
```

Loop operators

Loop operators contain for expression, break expression and loop expression. Here the for loop follows for-to-step-do method. Where the loop starts from defined integer till an integer defined after do. Step method is used to apply the do body of the loop only at a particular integer. There exisists a body of Lintilla code after the do expression which will execute for the for loop

For exp:

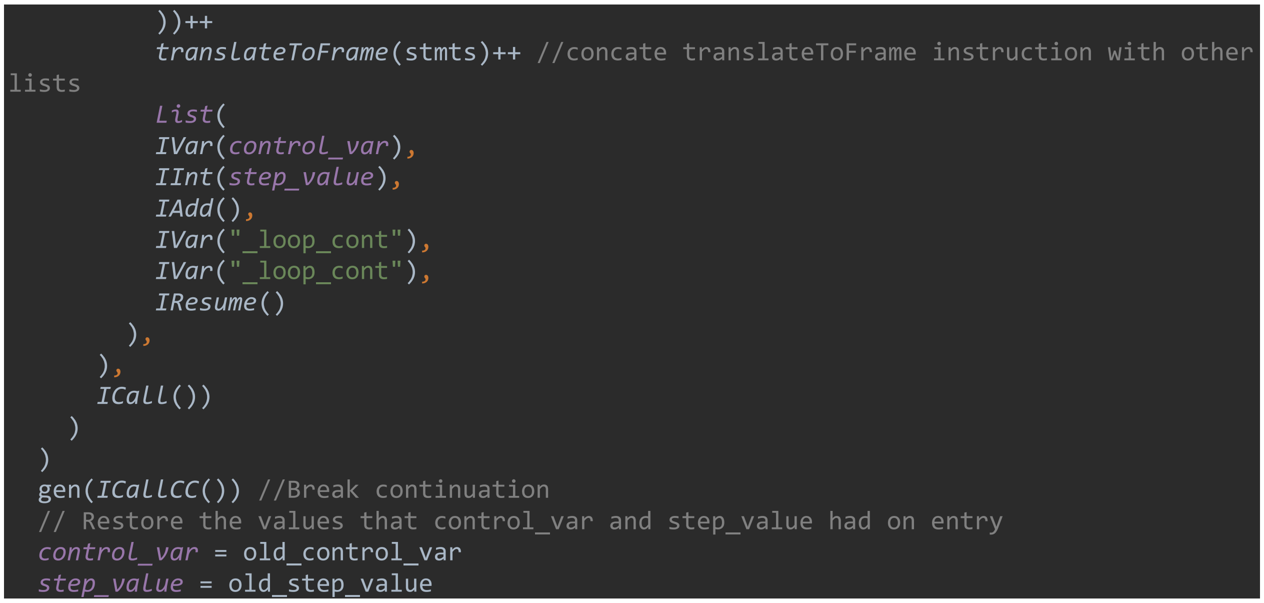
The implementation of the for expression was first taken from the provided translation scheme in the assignment.md file. Hence the scheme below is copied from the file

```
<SECD code translation of from>
<SECD code translation of to>
IClosure(
    None,
    List("_from", "_to", "_break_cont"),
    List(
        IClosure(
            None,
            List("_loop_cont"),
            List(
                IVar("_from"),
                Ivar("_loop_count")
            )
        ),
        ICallCC(),
        IClosure(
            None,
            List(control_var, "_loop_cont"),
            List(
           Ibool(step_value > 0)
           IBranch(
           List(
           IVar("_to"),
           IVar(control_var),
           ILess()),

          List(IVar(control_var),
           IVar("_to"),
           ILess())

                IBranch(
                    List(
                        IVar("_break_cont"),
                        IResume()
                    ),
26:                     List()
27:                 ),
28:                 <SECD code translation of body>,
29:                 IVar(control_var),
30:                 IInt(step_value),
31:                 IAdd(),
32:                 IVar("_loop_cont"),
33:                 IVar("_loop_cont"),
34:                 IResume()
35:             )
36:         ),
37:         ICall()
38:     )
39: ),
40: ICallCC()
41: ...
```

From line 1 and 2 the integers after for and the integer after to is translated and pushed onto the operand stack. After the values are bounded to the variable _from and _to by the instruction IClosure call.
ICallCC at line 40 bounds a break cont parameter to the above closure call. That means when break is called this instruction is triggered and stops the loop.
ICallCC at line 7 continues normal functionality of the loop.
At each iteration the body of the closure pushes the starting of the loop and continuation value of the loop to the top of the operand stack which is assigned to _from and _loop_cont statements of the operand stack
At each iteration the closure at line 16 is called by the ICall instruction at line 37 this binds the control variable of the loop and loop_cont_ value to the top two elements of the operand stack. That means at the first iteration of the loop, control variable will be starting value of the loop and the loop_cont_ is binded to the loop continuation value.
Line 20 checks if a step value is present. If it is line 29-31 adds the step value to the control variable. Or else 1 is added to control variable as default.
Finally, two copies of the loop continuation is bounded to _loop_cont. The second is resumed which causes the closure at line 16 to be executed again, which causes the next iteration of the loop. The current loop continuation is then bounded to _loop_cont and the new value of the control variable is updated

The implementation of the above is shown in the next page

```
/* Acknowledgement: translation scheme and help for the code below was found in
assignment.md file and help from general discussion forums */
case ForExp(IdnDef(id), from, to, step, Block(stmts)) =>

  //check stepValue
  var checkStep = 1
  var loop_termination = false
  if(step.isDefined)
    checkStep = evalIntConst(step.get) // Check if step is defined else leave it at
default

  //loop termination initalisation
  if(checkStep>0)
    loop_termination = true

  // Update control_var and step_value for the loop we are currently translating
  control_var = id
  step_value = checkStep

  // Save original values of control_var and step_value in local variables
  val old_control_var = control_var
  val old_step_value = step_value

  translateExp(from)
  translateExp(to)
  gen(
  IClosure(
    None,
    List("_from", "_to", "_break_cont"),
    List(
      IClosure(
        None,
        List("_loop_cont"),
        List(
          IVar("_from"),
          IVar("_loop_cont")
        )
      ),
      ICallCC(), // Loop continuation
      IClosure(
        None,
        List(control_var,"_loop_cont"),
        List(
        //Translation code for loop termination
          IBool(loop_termination),
          IBranch(
            List(IVar("_to"),IVar(control_var),ILess()),
            List(IVar(control_var),IVar("_to"),ILess())
          ),
          IBranch(
            List(
              IVar("_break_cont"),
              IResume()
            ),
            List()
```

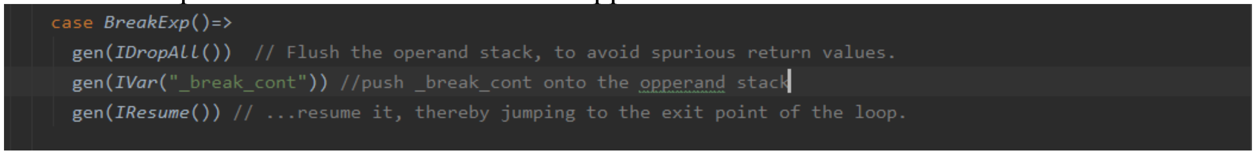


```
        ))++
        translateToFrame(stmts)++ //concate translateToFrame instruction with other
lists
        List(
        IVar(control_var),
        IInt(step_value),
        IAdd(),
        IVar("_loop_cont"),
        IVar("_loop_cont"),
        IResume()
      ),
    ),
    ICall())
  )
)
gen(ICallCC()) //Break continuation
// Restore the values that control_var and step_value had on entry
control_var = old_control_var
step_value = old_step_value
```

Break Expression
The break expression causes the iteration of the loop to stop. The iteration i of the loop is bounded to value called _break_cont_ which triggers the break continuation call at above implementation.(ICallCC() at line 40)
Translation scheme below was used for the break expression

```
IDropAll(),
IVar("_break_cont"),
IResume()
```

The IDropAll() is used to empty the operand stack. This is to avoid unexpected return to the loop if there were such instruction in the operand stack. The second line pushes the pushes the break_cont_ value to the operand stack and triggers the break continuation instruction for the for loop. IResume() jumps to the exit point of the loop
The above implementation is shown in the snippet below



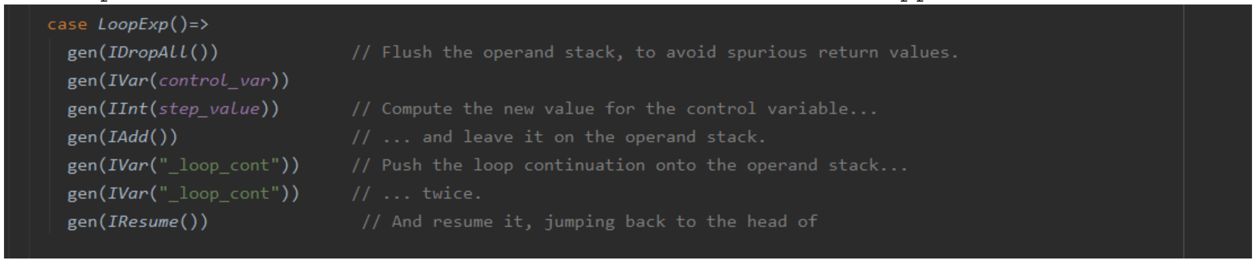
```
case BreakExp()=>
  gen(IDropAll())  // Flush the operand stack, to avoid spurious return values.
  gen(IVar("_break_cont")) //push _break_cont onto the opperand stack
  gen(IResume()) // ...resume it, thereby jumping to the exit point of the loop.
```

Loop Expression
The loop expression is used to continue the loop from the last break expression. When the expression executed the loop starts from the _loop_cont which was pushed to operand stack at line 33 of the for loop translation scheme.
 Translation scheme below was used for the loop expression

```
IDropAll()
IVar(control_var),
IInt(step_value),
IAdd(),
IVar("_loop_cont"),
IVar("_loop_cont"),
IResume()
```

The translation scheme first empties the operand scheme to avoid unexpected return to the loop. Then computes the new value for control variable and pushes it on the top of the operand stack twice. The resume instruction then jumps on head of for loop expression

The implementation of the above is shown in the code snippet below



```
case LoopExp()=>
  gen(IDropAll())             // Flush the operand stack, to avoid spurious return values.
  gen(IVar(control_var))
  gen(IInt(step_value))       // Compute the new value for the control variable...
  gen(IAdd())                 // ... and leave it on the operand stack.
  gen(IVar("_loop_cont"))     // Push the loop continuation onto the operand stack...
  gen(IVar("_loop_cont"))     // ... twice.
  gen(IResume())               // And resume it, jumping back to the head of
```

Testing Description

The aim of for testing for this project was to check the working of all combinations of the operators and to check if the Lintilla code was outputting the correct output according to the translation scheme. The skeleton of project contained tests for parsing syntax and semantic rule. The skeleton also contained tests for builtin translations. It was assumed that the provided tests for each module each were enough,
Hence in this project, only tests for testing the logical, array and loop operators were created.
Built in tests contained methods:
targetTestInLine – this method is used to run the compiler on a piece of Lintilla code, which created a tree and checked against an expected tree. This method was used to check if the output tree matched against the translation scheme tree.
execTestInLine – this method was used to run the compiler on a piece of Lintilla code, which executed the piece of code. The result of the executed code could be checked against an expected value.

Logical Operators

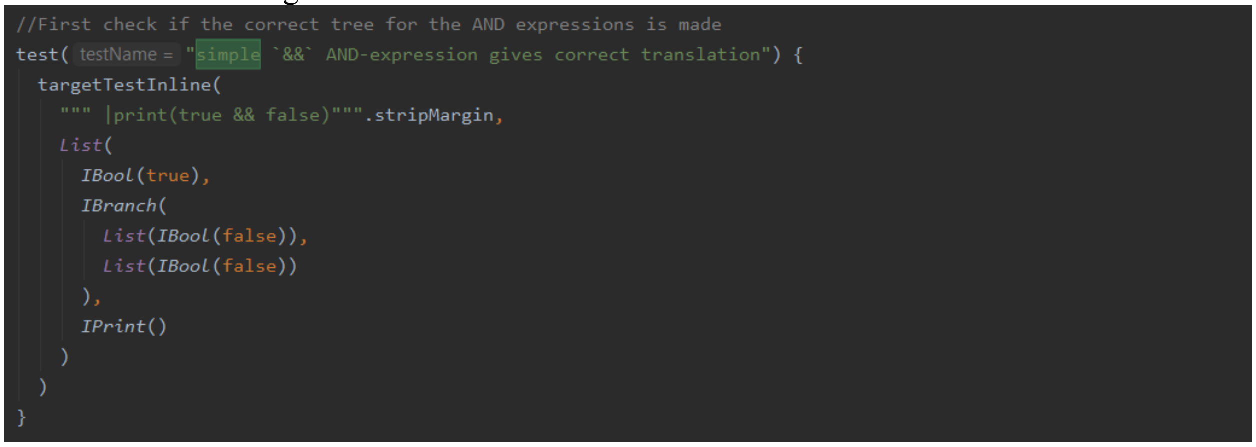
Tests for each Logical operator were tested in the following categories:

- Test if correct tree is generated. The tree generated must be checked against the translation scheme of the operator
- Test for values of the truth table
- Test short circuit evaluation for operators that contains it
- Black box testing/Integration testing: testing random values such as complex operations and combination of operators

The structure of this section will describe tests for each operator of the logical operators

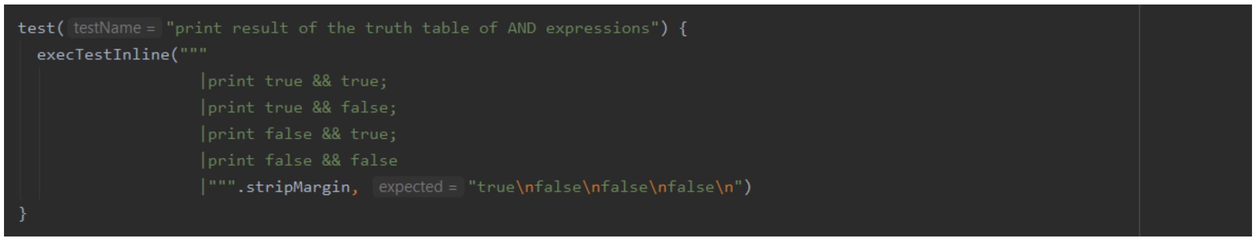
AND Operations

- Test if correct Tree is generated



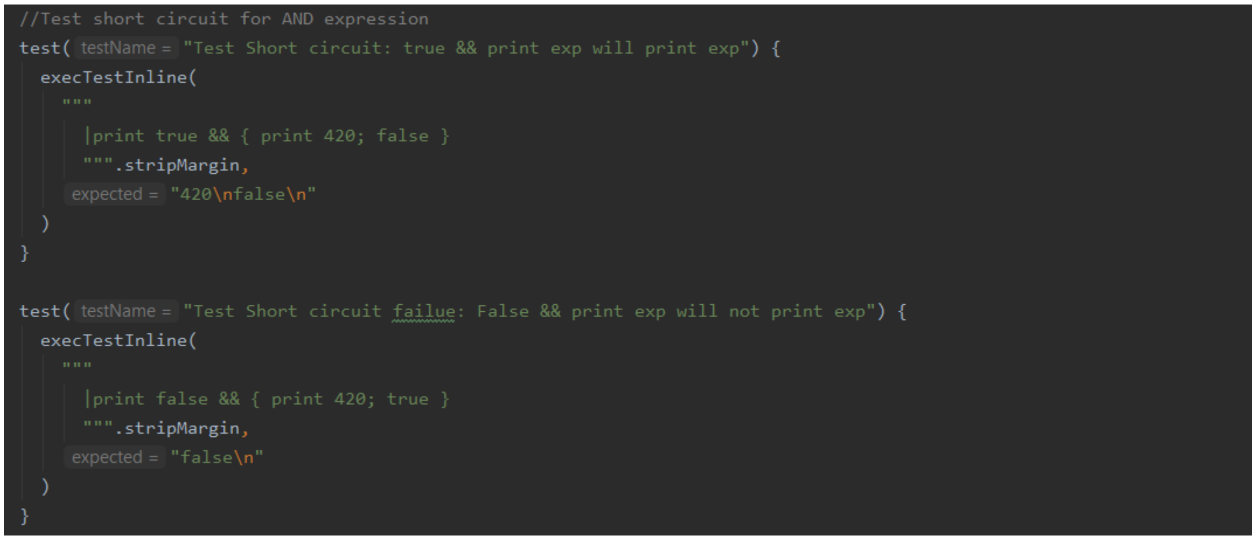
```
//First check if the correct tree for the AND expressions is made
test( testName = "simple `&&` AND-expression gives correct translation") {
  targetTestInline(
    """ |print(true && false)""".stripMargin,
    List(
      IBool(true),
      IBranch(
        List(IBool(false)),
        List(IBool(false))
      ),
      IPrint()
    )
  )
}
```

- Test for values of the AND truth table



```
test( testName = "print result of the truth table of AND expressions") {
  execTestInline("""
                 |print true && true;
                 |print true && false;
                 |print false && true;
                 |print false && false
                 |""".stripMargin,  expected = "true\nfalse\nfalse\nfalse\n")
}
```

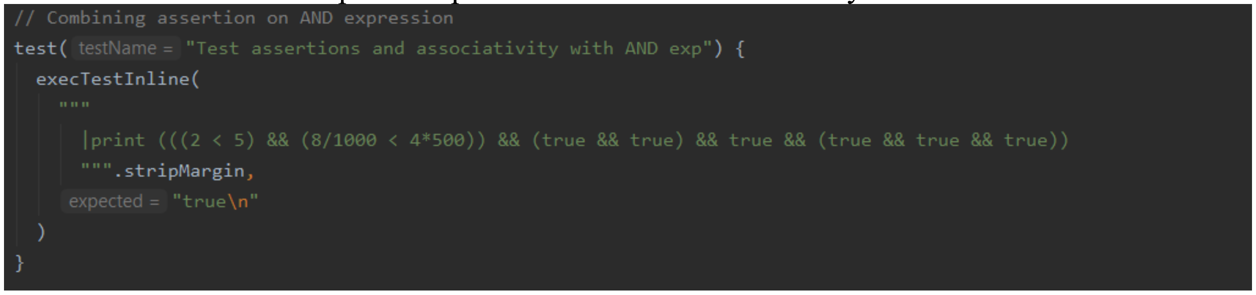
- Test for short circuit of the AND evaluation



```
//Test short circuit for AND expression
test( testName = "Test Short circuit: true && print exp will print exp") {
  execTestInline(
    """
      |print true && { print 420; false }
      """.stripMargin,
    expected = "420\nfalse\n"
  )
}

test( testName = "Test Short circuit failue: False && print exp will not print exp") {
  execTestInline(
    """
      |print false && { print 420; true }
      """.stripMargin,
    expected = "false\n"
  )
}
```

The above tests represents that the right hand side of the expression must only executed when the left hand side is true

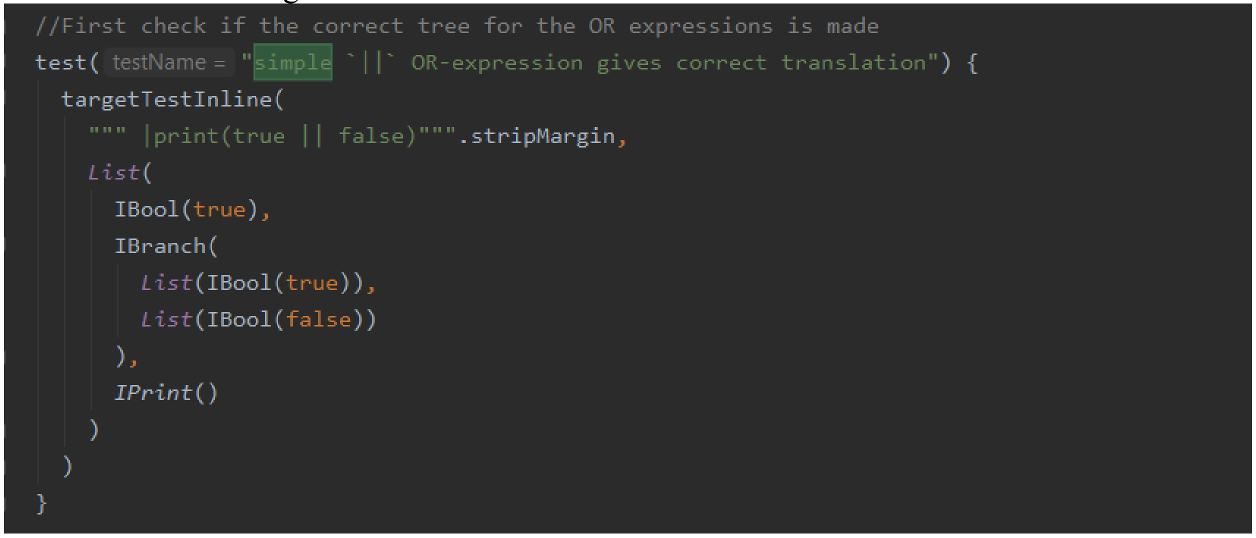
- Test random values: example- complex assertions and associativity



```
// Combining assertion on AND expression
test( testName = "Test assertions and associativity with AND exp") {
  execTestInline(
    """
      |print (((2 < 5) && (8/1000 < 4*500)) && (true && true) && true && (true && true && true))
      """.stripMargin,
    expected = "true\n"
  )
}
```

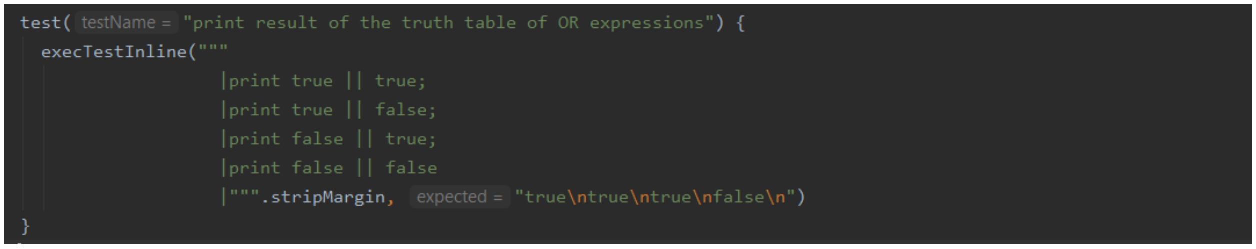
OR Operations

- Test if correct Tree is generated



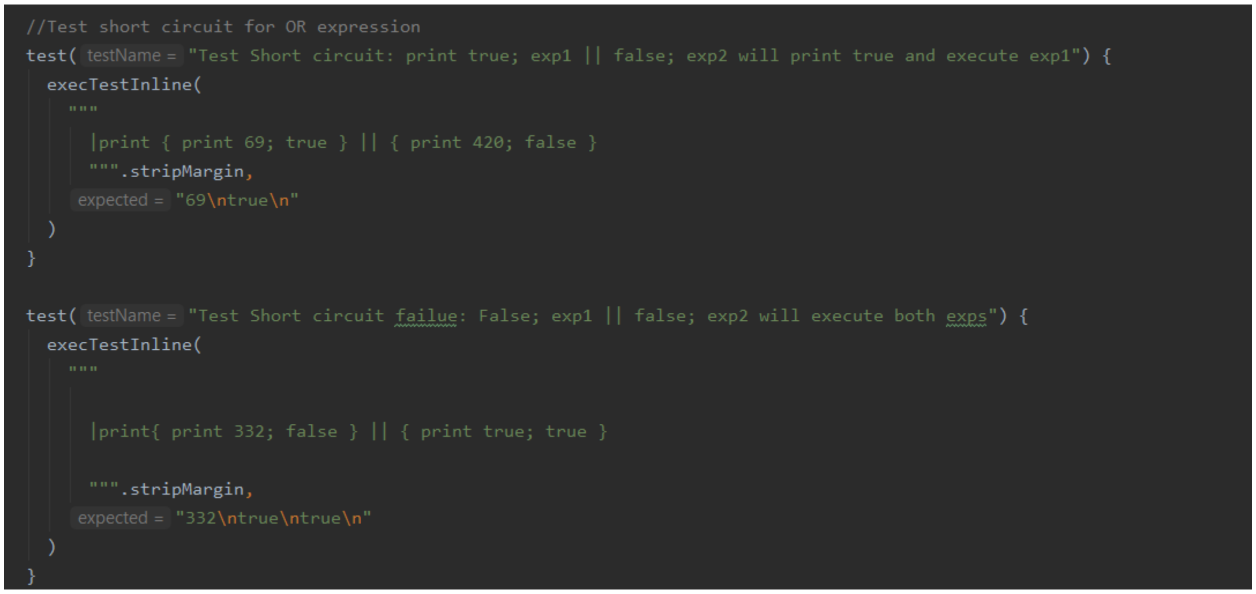
```
//First check if the correct tree for the OR expressions is made
test( testName = "simple `||` OR-expression gives correct translation") {
  targetTestInline(
    """ |print(true || false)""".stripMargin,
    List(
      IBool(true),
      IBranch(
        List(IBool(true)),
        List(IBool(false))
      ),
      IPrint()
    )
  )
}
```

- Test for values of the OR truth table



```
test( testName = "print result of the truth table of OR expressions") {
  execTestInline("""
                 |print true || true;
                 |print true || false;
                 |print false || true;
                 |print false || false
                 |""".stripMargin,  expected = "true\ntrue\ntrue\nfalse\n")
}
```

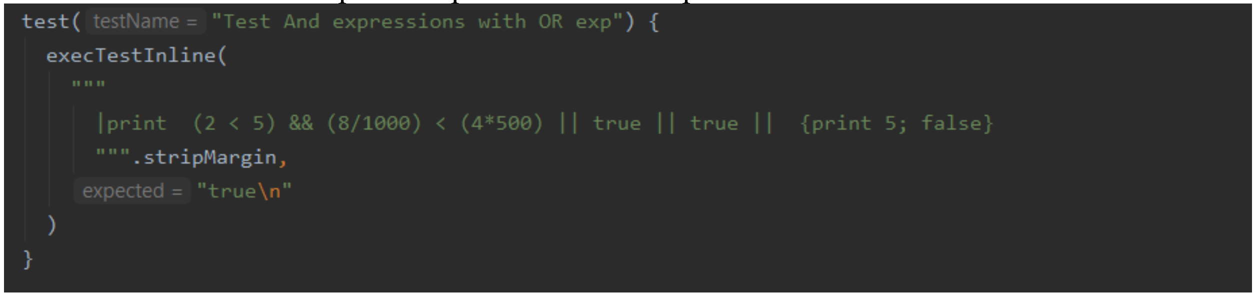
- Test for short circuit of the OR evaluation



```
//Test short circuit for OR expression
test( testName = "Test Short circuit: print true; exp1 || false; exp2 will print true and execute exp1") {
  execTestInline(
    """
      |print { print 69; true } || { print 420; false }
      """.stripMargin,
    expected = "69\ntrue\n"
  )
}

test( testName = "Test Short circuit failue: False; exp1 || false; exp2 will execute both exps") {
  execTestInline(
    """

      |print{ print 332; false } || { print true; true }

      """.stripMargin,
    expected = "332\ntrue\ntrue\n"
  )
}
```

The above tests represent that the right hand side of the expression is only executed when the left hand side is false

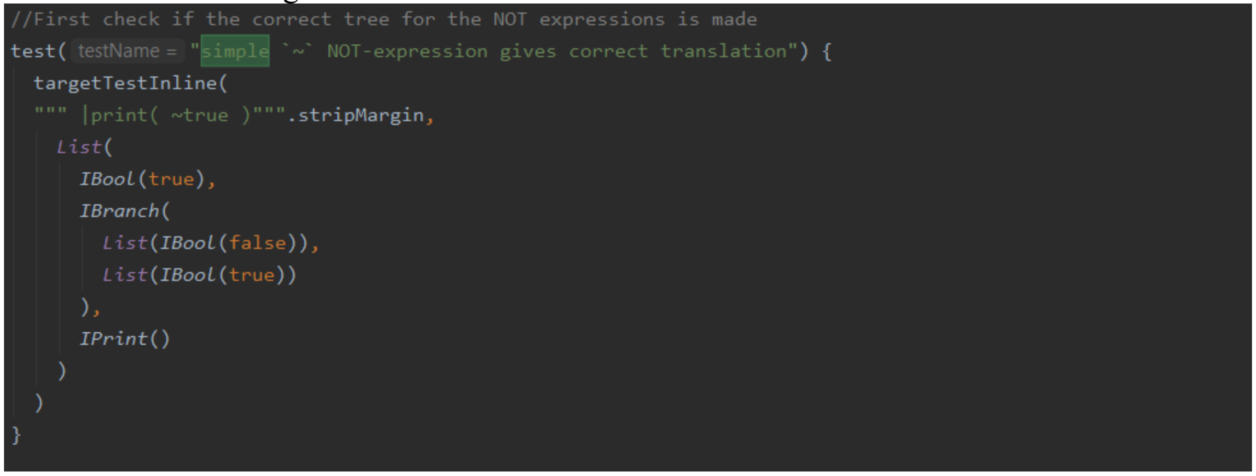
- Test random values: example- complex And or Or expression with short circuit



```
test( testName = "Test And expressions with OR exp") {
  execTestInline(
    """
      |print  (2 < 5) && (8/1000) < (4*500) || true || true ||  {print 5; false}
      """.stripMargin,
    expected = "true\n"
  )
}
```

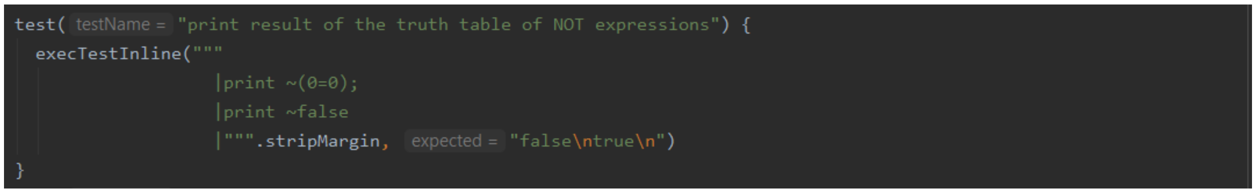
Not Expression

- Test if correct Tree is generated



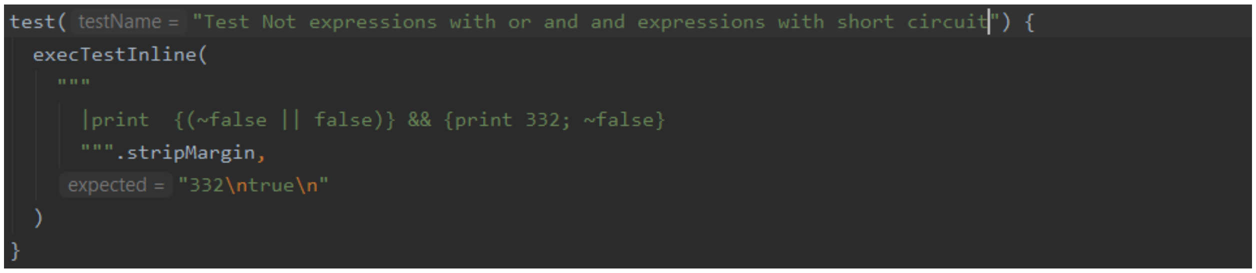
```
//First check if the correct tree for the NOT expressions is made
test( testName = "simple `~` NOT-expression gives correct translation") {
  targetTestInline(
  """ |print( ~true )""".stripMargin,
    List(
      IBool(true),
      IBranch(
        List(IBool(false)),
        List(IBool(true))
      ),
      IPrint()
    )
  )
}
```

-
- Test truth tables for not



```
test( testName = "print result of the truth table of NOT expressions") {
  execTestInline("""
                 |print ~(0=0);
                 |print ~false
                 |""".stripMargin,  expected = "false\ntrue\n")
}
```

- Test random values eg: test combination And or Or expressions with Not using short circuit



```
test( testName = "Test Not expressions with or and and expressions with short circuit") {
  execTestInline(
    """
      |print  {(~false || false)} && {print 332; ~false}
      """.stripMargin,
    expected = "332\ntrue\n"
  )
}
```

-

Array Operators

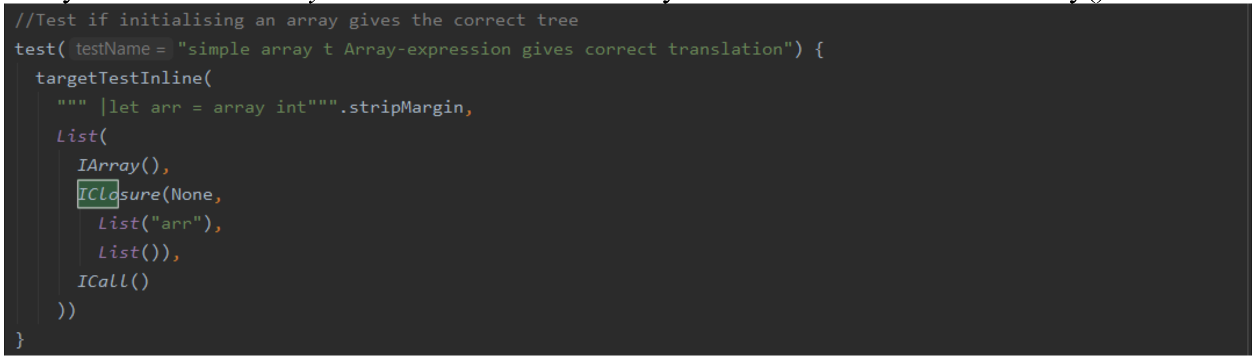
As logical opperators, tests for each array operator were tested in the following categories:

- Test if correct tree is generated. The tree generated must be checked against the translation scheme of the operator
- Testing to check array functionalities are producing correct outputs
- Smoke testing: test to reveal simple failures

The structure of this section will describe and represent the tests each category

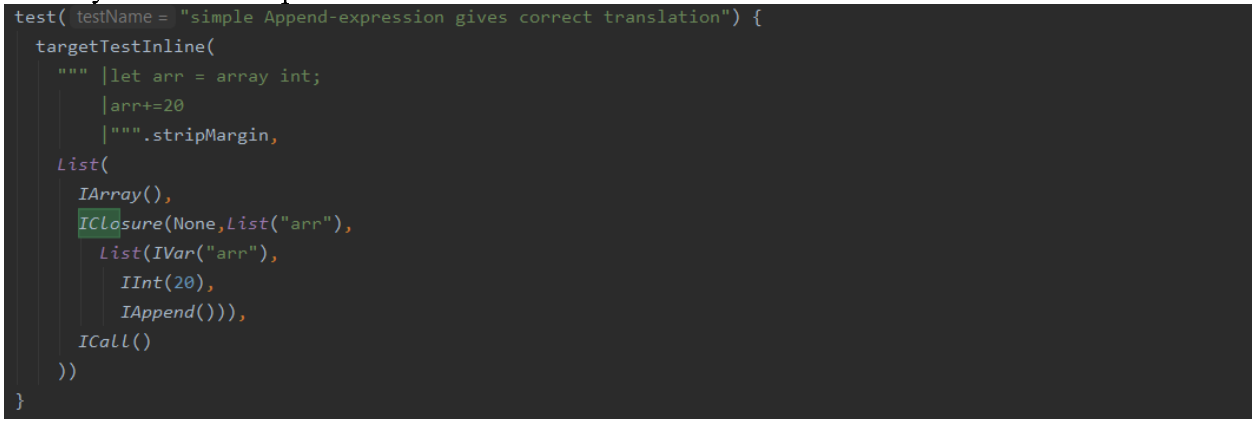
Test if correct tree is generated-

Array initialisation *array t:* Initialisation of the array translation scheme calls IArray()



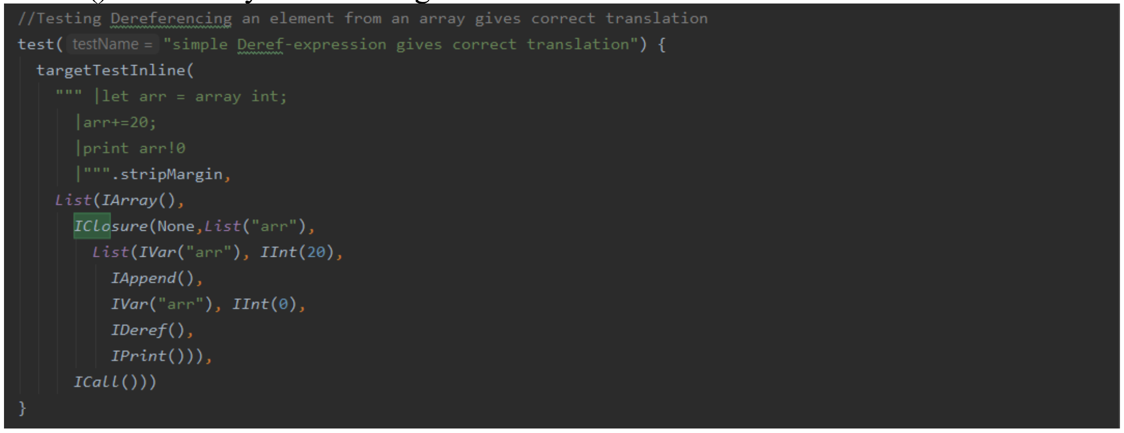
```
//Test if initialising an array gives the correct tree
test( testName = "simple array t Array-expression gives correct translation") {
  targetTestInline(
    """ |let arr = array int""".stripMargin,
    List(
      IArray(),
      IClosure(None,
        List("arr"),
        List()),
      ICall()
    ))
}
```

Appending of array *arr+=v*: Appending a value to array translation scheme calls IAppend() on an array arr and an expression



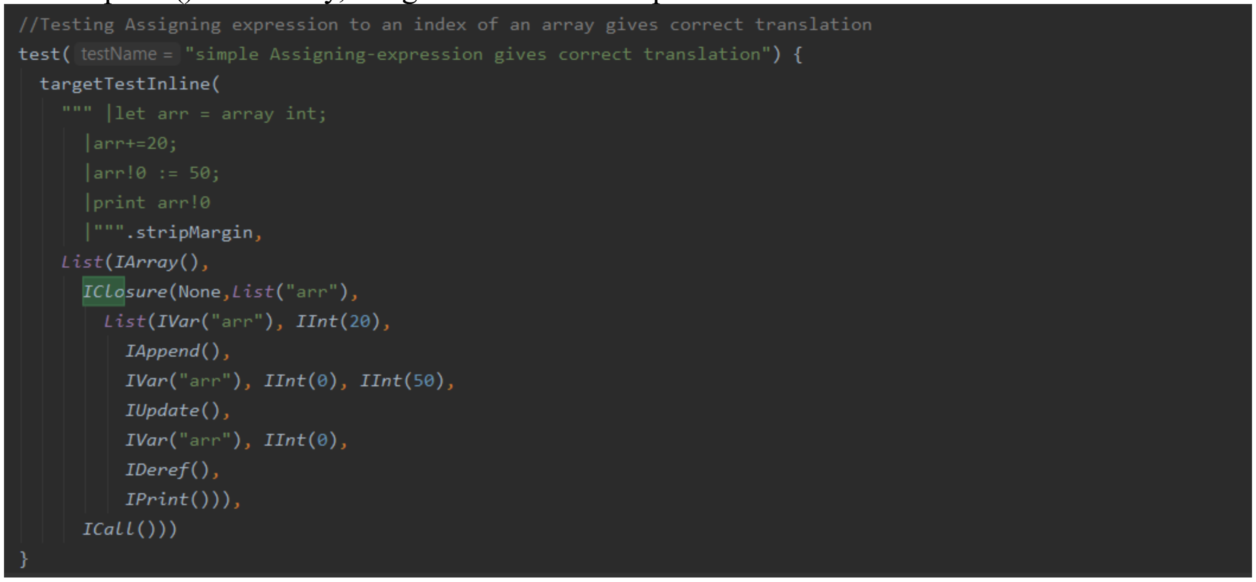
```
test( testName = "simple Append-expression gives correct translation") {
  targetTestInline(
    """ |let arr = array int;
        |arr+=20
        |""".stripMargin,
    List(
      IArray(),
      IClosure(None,List("arr"),
        List(IVar("arr"),
          IInt(20),
          IAppend())),
      ICall()
    ))
}
```

Dereferencing an array at an index *arr!i* :Dereferencing an array translation scheme calls IDeref() on an array arr and integer index



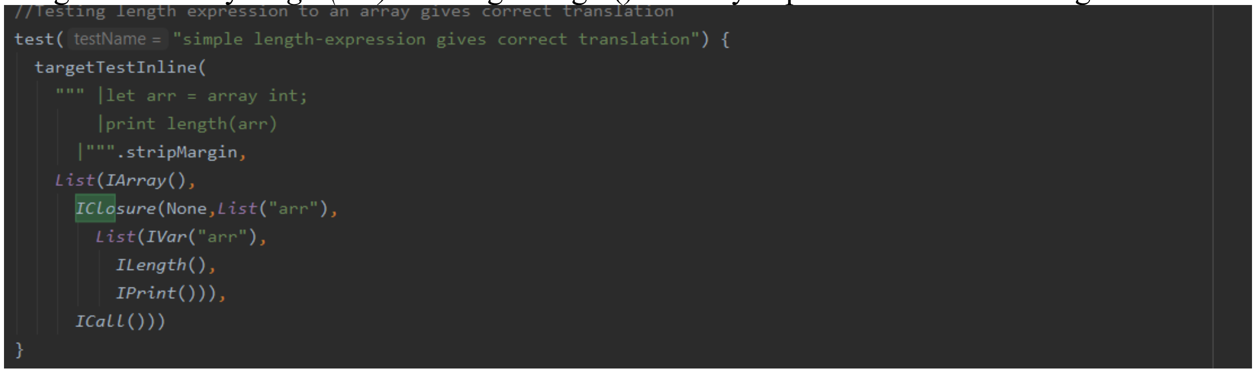
```
//Testing Dereferencing an element from an array gives correct translation
test( testName = "simple Deref-expression gives correct translation") {
  targetTestInline(
    """ |let arr = array int;
      |arr+=20;
      |print arr!0
      |""".stripMargin,
    List(IArray(),
      IClosure(None,List("arr"),
        List(IVar("arr"), IInt(20),
          IAppend(),
          IVar("arr"), IInt(0),
          IDeref(),
          IPrint())),
      ICall()))
}
```

Assigning an expression to an index *arr!i ≔ exp* :Assigining an expression to an array at index i calls IUpdate() to an array, integer index and an expression



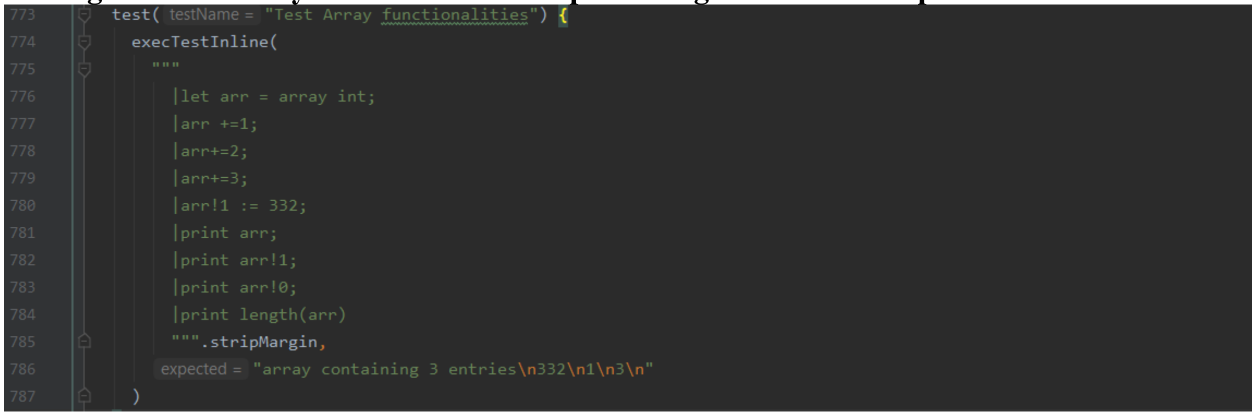
```
//Testing Assigning expression to an index of an array gives correct translation
test( testName = "simple Assigning-expression gives correct translation") {
  targetTestInline(
    """ |let arr = array int;
      |arr+=20;
      |arr!0 := 50;
      |print arr!0
      |""".stripMargin,
    List(IArray(),
      IClosure(None,List("arr"),
        List(IVar("arr"), IInt(20),
          IAppend(),
          IVar("arr"), IInt(0), IInt(50),
          IUpdate(),
          IVar("arr"), IInt(0),
          IDeref(),
          IPrint())),
      ICall()))
}
```

Length of an array *length(arr)*: calling ILength() on array expression returns its length



```
//Testing length expression to an array gives correct translation
test( testName = "simple length-expression gives correct translation") {
  targetTestInline(
    """ |let arr = array int;
       |print length(arr)
     |""".stripMargin,
    List(IArray(),
      IClosure(None,List("arr"),
        List(IVar("arr"),
          ILength(),
          IPrint())),
      ICall()))
}
```

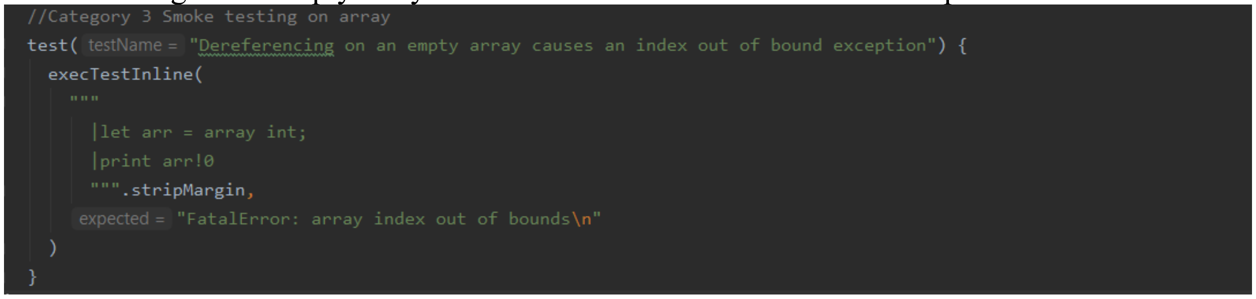
Testing to check array functionalities are producing the correct output



```
773      test( testName = "Test Array functionalities") {
774        execTestInline(
775          """
776            |let arr = array int;
777            |arr +=1;
778            |arr+=2;
779            |arr+=3;
780            |arr!1 := 332;
781            |print arr;
782            |print arr!1;
783            |print arr!0;
784            |print length(arr)
785            """.stripMargin,
786          expected = "array containing 3 entries\n332\n1\n3\n"
787        )
```

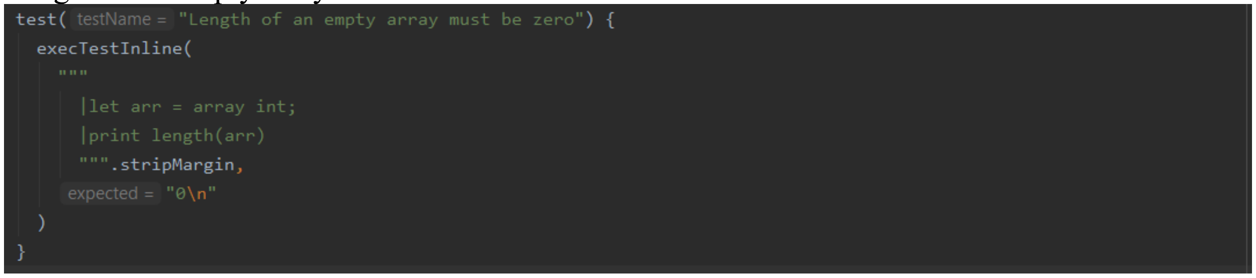
The snippet above checks array intiaialisation at line 776, appending to an array at line 777, updating the array at index 1 at line 780 and 782, dereferencing array at line 782 and 783 and finally, the length at line 784.

Smoke testing on array operators
In this category, tests were made to reveal failures on array operators
Dereferencing on an empty array shall assert an index out of bounds exception



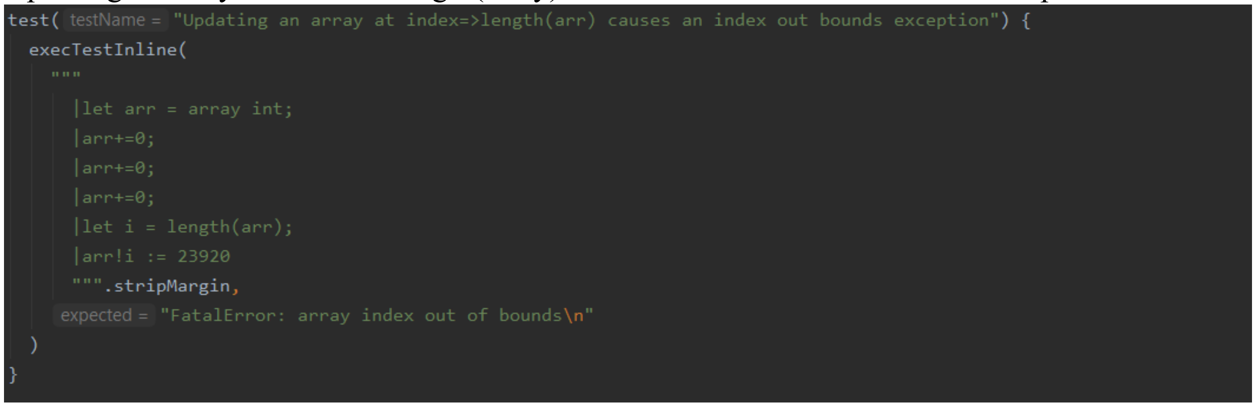
```
//Category 3 Smoke testing on array
test( testName = "Dereferencing on an empty array causes an index out of bound exception") {
  execTestInline(
    """
      |let arr = array int;
      |print arr!0
      """.stripMargin,
    expected = "FatalError: array index out of bounds\n"
  )
}
```

Length of an empty array must be 0



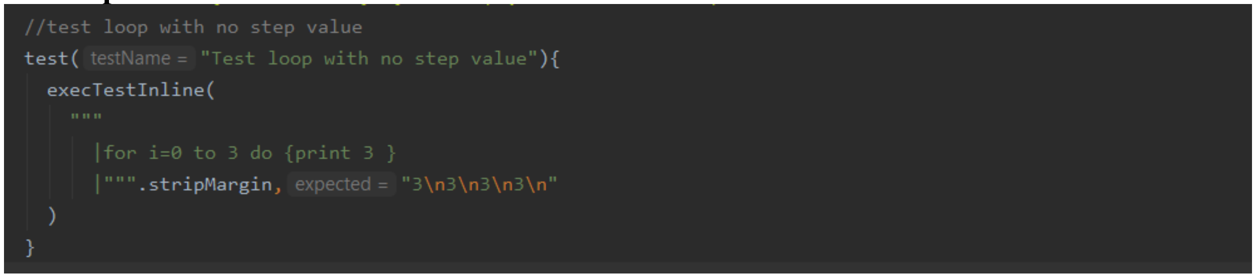
```
test( testName = "Length of an empty array must be zero") {
  execTestInline(
    """
      |let arr = array int;
      |print length(arr)
      """.stripMargin,
    expected = "0\n"
  )
}
```

Updating an array at index ⩾ length(array) must assert index out of bounds exception



```
test( testName = "Updating an array at index=>length(arr) causes an index out bounds exception") {
  execTestInline(
    """
      |let arr = array int;
      |arr+=0;
      |arr+=0;
      |arr+=0;
      |let i = length(arr);
      |arr!i := 23920
      """.stripMargin,
    expected = "FatalError: array index out of bounds\n"
  )
}
```

Array operations
For loop



```
//test loop with no step value
test( testName = "Test loop with no step value"){
  execTestInline(
    """
      |for i=0 to 3 do {print 3 }
      |""".stripMargin, expected = "3\n3\n3\n3\n"
  )
}
```

Aaannnd im out of time