COMP332 - Assignment 3

Lintilla Translation and Execution

Sophie Kaelin (45198543)

Introduction

The aim of this assignment is to complete an unfinished translator for the Lintilla programming language, to enable the execution of logical operators, array methods and loops. It is assumed that all code received by the translator have passed both syntactic and semantic analysis phases. This report will review how these remaining translating methods are implemented through the designed translation schemed, as well as demonstrated the thoroughness of testing completed to ensure all translations are being executed correctly.

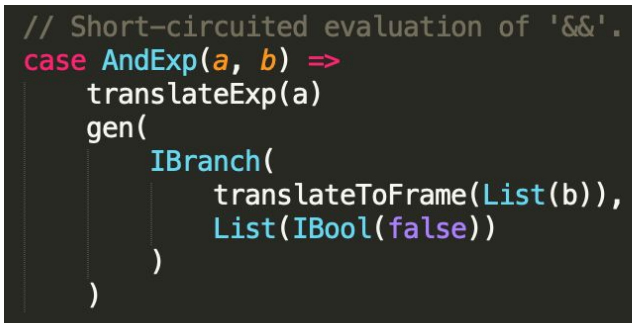
Design and Implementation

To achieve the goals of this assignment, various cases were added to the *translateExp* function which uses pattern matching to decide how a parameter expression gets translated and pushed onto the operation stack. Some cases were already included (such as *PlusExp, IfExp* and *BoolExp*) which were able to be used recursively to translate each element of a parent expression. Each newly added case match has been described below.

Logical Operators

And Operator

A sample translation scheme was provided in the Assignment specifications for the And Expression operator which made it fairly simple to implement. This case makes use of the provided IBranch method so that an expression can be evaluated based on whether the first provided value is true or false. If the left expression is false, then the operation will evaluate to false regardless of the value on the right side. If the left expression is true, the operation will evaluate to whatever value is on the right side. Using this knowledge, the left value was first pushed onto the stack, only to be popped off when the IBranch method is called to decide whether the expression will evaluate to the right value, or to false.

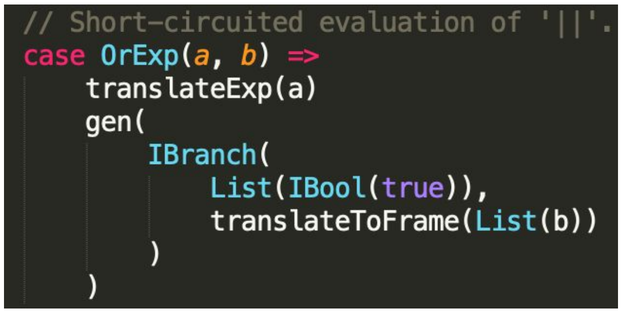


```
// Short-circuited evaluation of '&&'.
case AndExp(a, b) =>
    translateExp(a)
    gen(
        IBranch(
            translateToFrame(List(b)),
            List(IBool(false))
        )
    )
```

Or Operator

The Or Expression translator works similarly to the And Expression translator. If the left value is true then the expression will evaluate to true regardless of the right value. However, if the left value is false, then the expression will evaluate to the value on the right parameter. IBranch is utilised similarly to the and method.

// To evaluate **(b_1 || b_2)**
<SECD code translation of b_1>
IBranch(
    List(IBool(true)),
    <SECD code translation of b_2>
)



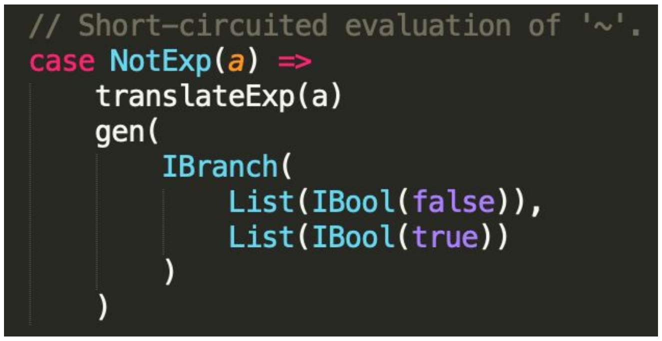
```
// Short-circuited evaluation of '||'.
case OrExp(a, b) =>
    translateExp(a)
    gen(
        IBranch(
            List(IBool(true)),
            translateToFrame(List(b))
        )
    )
```

Not Operator

Again using the IBranch method, we can set a true expression to push a false expression to the operation stack, and have a false expression push a true expression to the stack. This works similarly to the And expression and Or expression translators.

// To evaluate **(~b_1)**

```
<SECD code translation of b_1>
IBranch(
   List(IBool(false)),
   List(IBool(true))
)
```



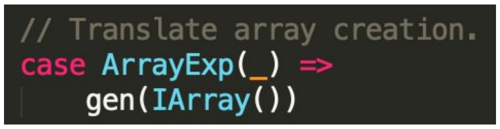
```
// Short-circuited evaluation of '~'.
case NotExp(a) =>
    translateExp(a)
    gen(
        IBranch(
            List(IBool(false)),
            List(IBool(true))
        )
    )
```

Array Operations

Array operations were quite simple to implement because there were already a lot of supporting execution methods which pushed and popped the values on the stack. For most of the array operation cases, I just had to ensure that the correct values were being pushed onto the stack in the correct order so that they could be used by these supporting methods.

Array Creation

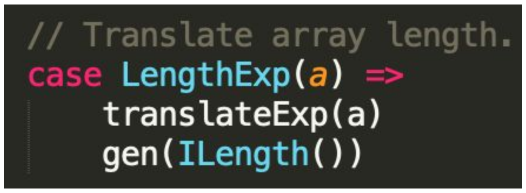
Since an array creation has no initial values, all that needed to be done was push a new array onto the stack. The type of array is irrelevant in this case.



```
// Translate array creation.
case ArrayExp(_) =>
    gen(IArray())
```

Array Length

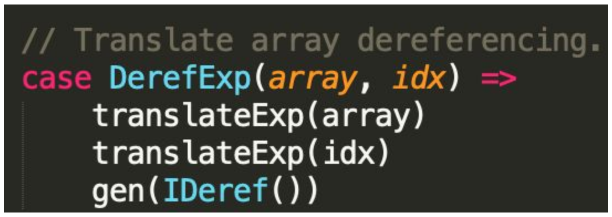
To find the array length, the parameter array must be pushed onto the stack, so that it can be popped off by the ILength() method to determine the value.



```
// Translate array length.
case LengthExp(a) =>
    translateExp(a)
    gen(ILength())
```

Array Dereferencing

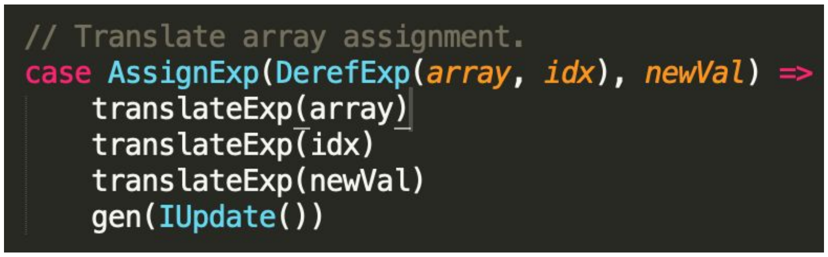
Firstly, the array parameter is pushed onto the stack, then the index parameter is pushed. After this, the IDeref() method is called to pop those two values back off the stack and push the value found at the provided index in the array onto the stack.



```
// Translate array dereferencing.
case DerefExp(array, idx) =>
    translateExp(array)
    translateExp(idx)
    gen(IDeref())
```

Array Assignment

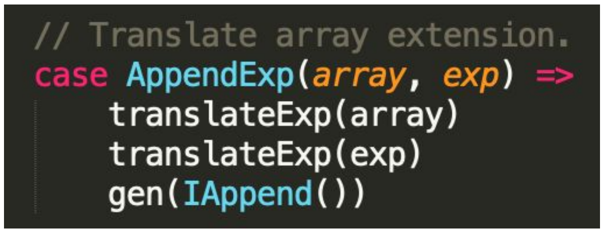
The ordering of items pushed onto the stack is important for this translator case to succeed. For the IUpdate() method to work correctly you must push the following parameters in this order: the original array, the index being altered, and then the new value to replace the old one. These values are then popped back off the stack by the IUpdate() method which executed the translation.



```
// Translate array assignment.
case AssignExp(DerefExp(array, idx), newVal) =>
    translateExp(array)
    translateExp(idx)
    translateExp(newVal)
    gen(IUpdate())
```

Array Extension

This method extends an array to have a new item on the end of it. The array is first pushed onto the stack, followed by the item to add. The IAppend() method is then called to perform the operation that updates the array to include the new item.



```
// Translate array extension.
case AppendExp(array, exp) =>
    translateExp(array)
    translateExp(exp)
    gen(IAppend())
```

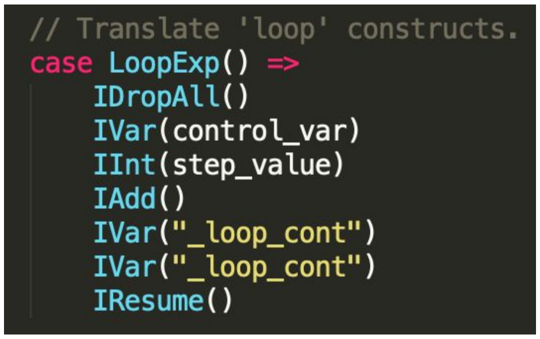
Loop Operations

For Loop Operation

This case was incomplete. There are missing instructions regarding the step_count and the alteration in instructions if the count was less than zero or greater than. Those elements have been commented out in the code to ensure there was no compilation errors.

Loop Operation

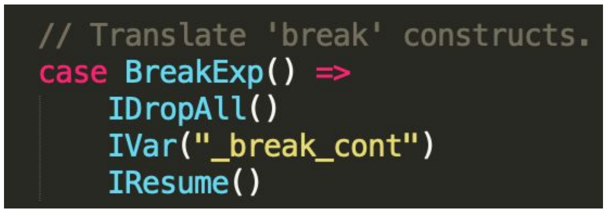
The loop operation translator was created as per the schema provided. The stack is first emptied before anything else is done. The control variable and step value are then pushed onto the stack, and popped off when IAdd() is called to compute what the control variable should be. The loop continuation is then pushed onto the stack twice so that when IResume() is called, the value isn't lost. The resume call moves the operation back to the top of the loop



```
// Translate 'loop' constructs.
case LoopExp() =>
    IDropAll()
    IVar(control_var)
    IInt(step_value)
    IAdd()
    IVar("_loop_cont")
    IVar("_loop_cont")
    IResume()
```

Break Operation

Similar to the Loop Operator, firstly the stack is emptied before the break continuation variable is pushed onto the stack and then a call to IResume() is made which will translate to a jump to the end of a loop, performing a “break” in the loop



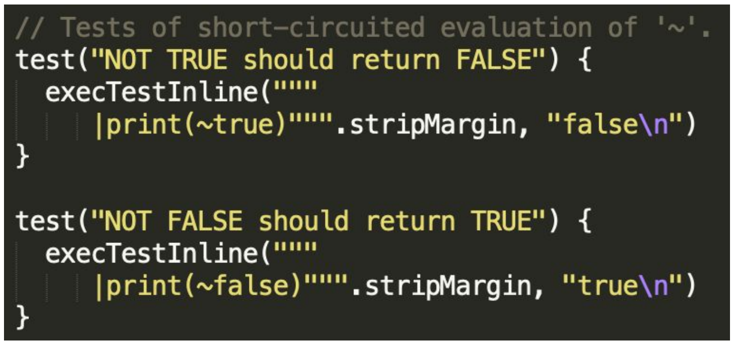
```
// Translate 'break' constructs.
case BreakExp() =>
    IDropAll()
    IVar("_break_cont")
    IResume()
```

Testing

Testing Logical Operators

Two types of testing was undertaken for each logical operator. Firstly to check if the output is correct, and secondly to check if the correct translation process is being taken given different outputs.

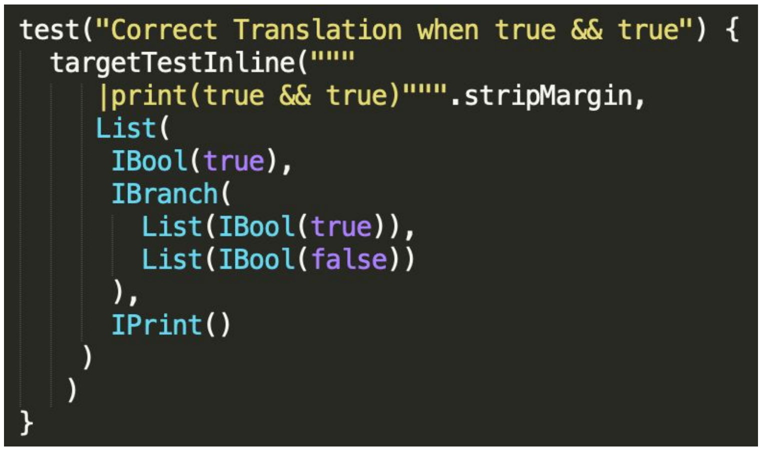
For each of the and, or and not operators, I tested every possible combination of two boolean expressions to ensure the translation was executing as expected. An example of this can be seen in my collection of tests for the not operation (which could either have a “true” or a “false” parameter.



```
// Tests of short-circuited evaluation of '~'.
test("NOT TRUE should return FALSE") {
  execTestInline("""
      |print(~true)""".stripMargin, "false\n")
}

test("NOT FALSE should return TRUE") {
  execTestInline("""
      |print(~false)""".stripMargin, "true\n")
}
```

For each of them, I also performed a target test which ensured that the correct translation process was occuring. Performing this test as well as the output test was important to show that the correct process was occuring. For both the “and” and “or”, I chose to test the combination of booleans that provided a unique result to the other combinations (i.e. true && true, false || false) An example of this can be seen for my “and” test.



```
test("Correct Translation when true && true") {
  targetTestInline("""
      |print(true && true)""".stripMargin,
      List(
       IBool(true),
       IBranch(
         List(IBool(true)),
         List(IBool(false))
       ),
       IPrint()
     )
    )
}
```

Testing Array Operations

Testing was undertaken to determine if an array was pushed to the stack and to check if it’s length was correct. The thoroughness of testing for this category is poor, as a consequence of time constraints. If time had permitted, testing would have been undertaken for all array operator translators which different variations (i.e. testing on empty arrays as well as full arrays)

Testing For Loop Operations

Since the Loop translator was incomplete, this element could not be tested.