Lintilla Translation and Execution

10th November 2019

Mohamed Maatouk (45216584)

MACQUARIE
University

School of Engineering
Macquarie University

1. Introduction

This assignment provides one with a skeleton project of a Lintilla programming language. This task involves extending the translator to allow the compiler to generate code for logical operators, array operations and for loops.

This document explains my thought process as I developed my solution to the assignment and also how I ensured that my program's functionality was reliable due to a series of clever test cases.

2. Design and Implementation

This section is split into three sections with each section focusing on the following:

- Logical operators (and, or, not)
- Array Operations
- For, Loop and Break Operations

2.1 Logical Operators

2.1.1 And Expressions

```
1 <SECD code translation of b_1>
2 IBranch(
3     <SECD code translation of b_2>,
4     List(IBool(false))
5 )
```

Figure 1

The code in figure 1 was provided by the project manager in order to give one a head start on the task. The translation for b_1 is simply *`translateExp(b_1)`*, this puts the value into the operand stack.

`IBranch()` pops a Boolean value from the top of the stack (in this case b_1) and evaluates line 3 if b_1 is true, otherwise evaluates line 4 from figure 1.

The translation for b_2 cannot be the same as b_1 since *`IBranch()`* only accepts *`Frame`* as a parameter. So instead we can convert b_2 to a frame using the *`translateToFrame()`* method. However *`translateToFrame()`* only accepts a List(Expression) as a parameter so we can either wrap b_2 with a *`Block()`* or put b_2 into a *`List()`*.

It is better to use *List()* as expressions such as the following won't evaluated unless put in brackets when using Block() as shown in figure 2 below.

```
print false && true // Won't be evaluated using Block(b_2)

print {false} && {true} // Will be evaluated using Block(b_2)
```

Figure 2

2.1.2 Or Expressions

The or expression is very similar to the and expression above but instead the *IBranch()* parameters are swapped and the Boolean has changed to true as you can see in the figure 3 below.

```
<SECD code translation of b_1>
IBranch(
    List(IBool(true))
    <SECD code translation of b_2>
)
```

Figure 3

We simple do the same for or expression as we did for and in translating b_1 and b_2.

2.1.3 Not Expressions

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

Figure 4

Very simple, just translate b_1 using *translateExp()* and return opposite Boolean values when b_1 is evaluated using *IBranch()*.

2.2 Array Operations

There are five array operations which are used which were quite simple to implement since they didn't require one to implement array instructions but instead call these instructions, these operations are described in detail below:

- Array Initialisation
    - Calls the *IArray()* instruction which creations a new empty array and pushes it onto the operand stack.

    - The parameter tipe is ignored because the Lintilla compiler does such a good job of type checking that we don't need to carry the types of things from compile to run time.

- Array Dereferencing
    - *DerefExp()* has two parameters, array and index, which we will push onto the operand stack using *translateExp()*
    - This is then used by *IDeref()* which pops an integer index and array from the stack and pushes the value found in the array at the given index back onto the operand stack.

- Array Assignment
    - In order to assign a value to and index in an array we need the array, index and the value we want to assign to that index in the array. However since *AssignExp()* only accepts two parameters we simply use *DerefExp()* as the left assignment since it is the index we want to assign the value to.
    - We then push these values onto the operand stack using *translateExp()*
    - Finally we us *IUpdate()* which pops these three values from the operand stack to update that index of the array with our specified value.

- Array Extension
    - *AppendExp()* has two parameters, array and index, which we will push onto the operand using *translateExp()*
    - This is then used by *IAppend()* which pops an integer and array from the stack and extends the array by adding the given value to the end of the array.

- Array Length
    - *LengthExp()* accepts arrays as its parameter, this is pushed onto the operand stack.
    - *ILength()* pops the array from the operand stack and pushes its length onto the stack in its place.

2.3 For, Loop, and Break Operations

2.3.1 For Operation

For this part of the task we were given some code to complete as shown in figure 5 below.

```
<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_cont")
            )
        ),
        ICallCC(),
        IClosure(
            None,
            List(control_var, "_loop_cont"),
            List(
                <SECD code to test for loop termination (see below)>,
                IBranch(
                    List(
                        IVar("_break_cont"),
                        IResume()
                    ),
                    List()
                ),
                <SECD code translation of body>,
                IVar(control_var),
                IInt(step_value),
                IAdd(),
                IVar("_loop_cont"),
                IVar("_loop_cont"),
                IResume()
            )
        ),
        ICall()
    )
),
ICallCC()
```

Figure 5

My task was to complete the translations for line 1, 2, 20, and 28 in figure 5 as well as deal with the optional step value parameter in *ForExp()*.

- For the translation of lines 1 and 2, we simply use *translateExp()* to push these values into the operand stack.
- To complete the translation for line 20 we first need to deal with the optional step value, we do this by creating a variable above the code in figure 5 to store the step_value. We then use the Option class *getOrElse()* method to obtain the value from step_value parameter. Inside the parameter we put IntExp(1) since the default value for step is 1 when not specified. Finally we want to convert the option from an expression to an integer to be used in our translation using the *evalIntConst()* method.
- Now since we have a step_value to work with we can write the translation to check for loop termination. We can simply write and if statement like in the figure 6 below.

```
1 if(step_value > 0)
2   IVar("_to")
3 else
4   IVar(control_var),
5
6 if(step_value > 0)
7   IVar(control_var)
8 else
9   IVar("_to")
```

Figure 6

- For the translation of the body we simply want to add it to our list of instructions by appending it to the list similar to the format below.

```
1 List( ... ) ++ translateToFrame(body.stmts) ++ List( ... )
```

Figure 7

2.3.2. Loop Operation

This task is provided by the project manager and is described in the figure below.

```
1 IDropAll()
2 IVar(control_var)
3 IInt(step_value)
4 IAdd()
5 IVar("_loop_cont")
6 IVar("_loop_cont")
7 IResume()
```

Figure 8

To complete this task we need control_var and step_value from ForExp, we can do this by creating a global variable within the Translator class and update them when ForExp is being used as seen in the figure below:

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

// Update control_var and step_value for the loop we are currently translating
control_var = id
step_value = evalIntConst(step.getOrElse(IntExp(1)))
```

Figure 9

2.3.3 Break Operation

This task is provided by the project manager and is described in the figure below.

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

Figure 10

Testing

In order to ensure my code was correct the following tests were performed:

- Short Circuit evaluation for '&&' and "||" expressions
- Truth table evaluation for '&&' and "||" expressions
- '~' expression evaluation for true and false
- Array initialization
- Adding items to an array
- Dereferencing array values
- Assigning values to an index in an array
- For loop step break and loop evalations