COMP332 Programming Languages

Assignment 3 – Translation for the Lintilla Language

Name: George Hayward
Student Number: 43692486

Introduction:

This assignments objective was to complete a translator for the Lintilla programming language. We where provided with all other levels of a working compilation implementation such the Syntax and Semantic analysers etc, as well as a partially completed translator. Once the translator is completed Lintilla programs will be runnable.

The remaining parts of the translator which needed completion included: Logical operators (&&, ||, ~), Array operators (creation, dereference, append, length and assign), for loops operations, loop operations and break operations. This report will detail how each of these operations where implemented as well as how they where tested to be correct.

Logical Operators:

AND Operator Implementation:

The AND operator implementation was specified to be a short circuit implementation. This means that if the first Boolean value evaluated is false, there would be no need to evaluate the second value as the AND statement will be false regardless.

```
// Translate an 'AND' operator
case AndExp(l, r) =>
  translateExp(l) // Put left side of AND to top of stack
  gen(            // Generate instruction
    IBranch(      // Evaluate the truth value of the expression on top of the
stack.
      translateToFrame(List(r)), // LHS Expression = True: Then the AND operation
truth value is up to the right hand side.
      List(IBool(false)) // LHS Expression = False: Then the AND operation truth
value is automatically false without the need to inspect the RHS
    )
  )
```

The commenting of the above implementation sufficiently explains the operator.

AND Operator Tests:

Test Name	Test Information
1. True && True = True	Truth Table Test
2. True && False = False	Truth Table Test
3. False && False = False	Truth Table Test

4. True && True Conditional = True	Assert correct working for conditional expression.
5. Correct Sequence Output	Assert correct sequence tree output
6. Short Circuit Check RHS if LHS = True	Short Circuit Test
7. Short Circuit !Check RHS if LHS = False	Short Circuit Test

Justification of Test Range:

The tests 1-3 provide a truth table which indicates that the implementation gives the correct Boolean output for simple Boolean values such as true and false. Test 4 tries a Boolean expression instead of Boolean values and the correct output is achieved. Test 5 has the correct sequence output indicating that the implementation is generating the correct tree (this was taken from the forums!!). Test 6-7 provides a clear demonstration of the short circuit implementation.

OR Operator Implementation:

```
// Translate an 'OR' operator
case OrExp(l, r) =>
  translateExp(l) // Put left side of OR to top of stack
  gen(            // Generate instruction
    IBranch(      // Evaluate the truth value of the expression on top of the
stack.
      List(IBool(true)), // LHS Expression = True: Then the OR operation truth
value is automatically true without the need to inspect RHS.
      translateToFrame(List(r)) // LHS Expression = False: Then the OR operation
truth value is up to the right side.
    )
  )
```

As you can see the OR operator implementation works nearly exactly the same as the AND operator implementation. The only difference being the functional differences of AND / OR which can be seen in the use of IBranch().

The testing of the OR operator is nearly identical to the AND operator tests and therefore does not need to be discussed again.

NOT Operator Implementation:

The NOT operator has been implemented by using the IBranch() function to flip the truth value of the input expression. This is as simple as the required operator implementation can be.

```
IBranch(      // Evaluate the truth value of the expression on top of the stack.
  List(IBool(false)), // Expression = True: Flip value to false.
  List(IBool(true))   // Expression = False: Flip value to true.
```

NOT Operator Tests:

The NOT operator tests simply assert that when a truth value has the ~ operator before it, its value is reversed. This is done using Boolean values ‘true’, ‘false’ and expressions.

Arrays:

The implementation of all Array operators was quite simple. Considering the SECDTree functions have quite clear instructions as to what they require as inputs to operate. All of the array operators work by pushing expressions from the stack (besides array creation) in the right order and then generating instructions from the SECDTree class’s functions popping those expressions. I will therefore not go into the inner workings of each individual implementation.

Furthermore, the tests designed for each operator are quite brief and simply check whether or not the array operators are functioning correctly. E.g.

```
// ARRAY dereference tests
test("Array Dereference test: deref item at pos 1, should give int 2")  {
  execTestInline(
    """
      |let x = array int;
      |x += 1;
      |x += 2;
      |x += 3;
      |x += 4;
      |let y = x!1;
      |print y""".stripMargin,
    "2\n"
  )
}
```

This test clearly indicates that the dereference operator is getting the correct value from the array. There is not much need for extensive testing here as the range of possible tests is very small unless testing for semantic errors such as trying to append the wrong type to the array (which I believe does not need to be tested for).

Loops:

FOR loop Implementation

Could not get my loop section to work properly. Followed the template given in the assignment spec and altered elements so that it would execute similarly to the way previously completed (adding gen() etc.). The issue is with the getting of the Some(i) value of the input step[option]. I think that is why I am getting an error when trying to run for_array.lin. The step can't be negative integer value in my implementation. However, I don't know how else I'm supposed to extract the value from step[option] and plug it into the global variable step_value.

It might also be due to my implementation of the less than or greater than 0 for loop stopping mechanism but I can't be sure. Not really sure how to test these kinds of things in order to solve, in the end I'm just shooting in the dark now.