COMP3000
Programming Languages

Assignment 3
Translation

Chaz Lambrechtsen

45426317

1 Table of Contents

2 Summary

This report will walk through the design and implementation in order to develop a Translator for FunLang. The translator will target a simple abstract machine called the SEC (Stack, Environment and Code) machine. This report will also include a description of the testing that has been carried out with details and explanations of different tests that have been utilised.

3 Design and Implementation

While designing and implementing this Translator I have utilised useful resources provided to us such as the Assignment 3 Spec [2] and Translator solutions from weeks 10 and 11 [3]. External resources such as Kiama documentation [1] have also been used in this implementation.

3.1 Value translation

Value instructions are translated with the passed value from FunLang.

For example:

```
case BoolExp (value) =>
    gen (IBool (value))
```

3.2 Variable translation

FunLang has IdnUse which can be translated to IVar

```
// Identifier Instruction
case IdnUse (idn) =>
    gen (IVar (idn))
```

3.3 Arithmetic translation

Arithmetic instructions are generally structured similarly as each Arithmetic instruction has an expression on left and right side.

Each instruction can be generated by translating left and right sides then generating ISub Instruction.

For example:

```
// Subtration
case MinusExp(l, r) =>
    genall (translateExpression (l))
    genall (translateExpression (r))
    gen (ISub ())
```

3.4 Equality and Less than translation

Equality instruction is like an arithmetic instruction in that it has an expression on either side. But the EqualExp should generate a IEqual instruction.

For example:

```
// Equality comparison
case EqualExp(l, r) =>
    genall (translateExpression (l))
    genall (translateExpression (r))
    gen (IEqual ())
```

Similarly, for LessExp there is an expression on either side and this should generate a ILess instruction

3.5 Branch translation

A branching expression utilises a conditional expression such as equal or less than in order to branch to one of two options (thenExp or elseExp). IBranch takes these two branching options as parameters.

For example:

```
// If conditional expression
case IfExp(cond, thenExp, elseExp) =>
    genall(translateExpression(cond))
    gen(IBranch (translateExpression(thenExp), translateExpression(elseExp)))
```

3.6 Closure translation

A block expression is more complicated to translate then previous expressions. Since a block expression contains definitions and expressions, we must match cases for both.

For example, with the block expression we match to definitions:

```
defns match {
    // When there is Val definition inside block expression
    case Val(IdnDef(i), v) +: a =>
        genMkClosure(i, BlockExp(a, exp))
        genall(translateExpression(v))
        gen(ICall())
    // Generate all other inside block expressions
    case _ =>
        genall(translateExpression(exp))
}
```

Expressions must be generated inside block expression and strictly only after evaluating identity definitions

3.7 Function translation

Function translation is also much more complicated because of the specific pattern of the functions that must be matched.

A FunGroup takes a Vector[Fun], Fun takes IdnDef and Lam as arguments [2]. Therefore a FunGroup is matched to this specific case and the block expression for the function is generated as well as the body, followed by ICall() instruction.

Since a Vector(FunGroup(Vector()),Val..) does not match to the initial case, then this will be matched to FunGroup(_).

For example, function translation within block expression:

```
// Translate function within block expression
case FunGroup (Fun(IdnDef(idn1), Lam(Arg(IdnDef(idn2),tipe), body)) +: funs) +: defns =>
    genMkClosure( idn1, BlockExp(Vector(FunGroup(funs)) ++ defns, exp))
    genMkClosure (idn2, body)
    gen (ICall())
// Generate block expression for function definition
case FunGroup(_) +: defns => genall(translateExpression(BlockExp(defns,exp)))
```

4 Testing

These tests confirm that my program is working correctly because they include edge cases, syntax error as well as testing for correct syntax and lexical expressions which produces the correct FunLang Tree corresponding to the assignment specification. The following tests are simple to start with to ensure the basic instructions in translation are working correctly.

4.1 Boolean value instructions

Test 1 – "true" should be "true"

True Boolean value should evaluate to true.

Test 2 – "false" should be "true"

False Boolean value should evaluate to false.

4.2 Variable instructions

Test 1 – "{ val a = 5 val b = 1 a }" should be 5

This block expression should evaluate to the value of a which is 5

4.3 Add instructions

Test 1 – "0 + 1 + 2 + 3" should be "6"

Since there is already a simple addition test, I have added a test for multiple addition expressions which should evaluate to the total sum.

4.4 Minus instructions

Test 1 – "10 – 8" should be "2"

A simple subtraction test to ensure the correct output is given.

Test 2 – "10 – 2 - 2" should be "6"

Multiple subtraction expressions to ensure all values are subtracted and therefore the correct output is given.

4.5 Divide instructions

Test 1 – "50 / 2" should be "25"

A simple division test to ensure the correct output is given.

Test 2 – “5 / 5” should be “1”

Dividing an integer by the same integer should return 1.

Test 3 – “5 / 10” should be “0”

Dividing a number by a larger number should result in 0 as the decimal is rounded down.

Test 4 – “20 / 2 / 2” should be “5”

Multiple division expressions should return the correct answer.

4.6 Multiply instructions

Test 1 – “20 * 5” should be “100”

A simple multiplication test to ensure the correct output is given

Test 2 – “5 * 20” should be “100”

Multiplying the same numbers in a different order should return the same result.

Test 3 – “6 * 7 * 2” should be “84”

Ensure multiplying multiple integers returns the correct answer

4.7 Equality instructions

Test 1 – “1 == 2” should be “false”

Ensure a false equality expression evaluates to false

Test 2 – “3 == 3” should be “true”

Ensure a true equality expression evaluates to true

4.8 Less than instructions

Test 1 – “3 < 3” should be “false”

Ensure a Less than condition evaluates false when left side is equal to right side

Test 2 – “4 < 3” should be “false”

Ensure a Less than condition evaluates false when left side is more than right side

Test 3 – “2 < 3” should be “true”

Ensure a Less than condition evaluates true when left side is less than right side

4.9 Branch instruction

Test 1 – “if (true) then 1 else 2” should be “1”

This test ensures that a conditional if statement chooses the correct branching expression when condition is true.

Test 2 – “if (false) then 1 else 2” should be “2”

This test ensures that a conditional if statement chooses the correct branching expression when condition is false.

Test 3 – "if (5 == 4) then 1 * 2 else 2 + 3" should be "5"

This test ensures that a conditional if statement can be evaluated with an equality instruction and use arithmetic instructions within branching expressions and still evaluate the correct result.

4.10 Closure instruction

Test 1 – "{ val a = 1 a + 1 }" should be "2"

This test ensures that a simple block expression starting with a val definition and a simple addition expression will evaluate the correct result

Test 2 – "{ val a = 5 val b = a + 1 a * b }" should be "30"

Multiple val definitions can be defined with expression in definition and multiple arithmetic expressions to evaluate the correct answer.

4.11 Function tests

Test 1 – { val x = 15 def double (a : Int) = a * 2 double (x) }" should be "30"

This test creates a function named double, which will double any integer given as argument and return the result.

4.12 Other example tests

The following example tests are provided with the assignment. These examples are tested to ensure my translation behaves correctly with input that is guaranteed to be correct.

Test 1 – "{ 2 + 3 * 4 }" should be "14"

Simple.fun example from assignment 3 file [2]. A basic block expression evaluates the correct result

Test 2 – "{ val a = 5 val b = a + 1 a * b }" should be "30"

Block.fun example from assignment 3 file [2]. Ensures block expression translate correctly.

Test 3 – "{ val x = 100 def inc (a : Int) = a + 1 inc (x) }" should be "101"

Function.fun example from assignment 3 file [2]. Ensures function definition translate correctly.

Test 4 – "{ val a = 3 a + 1 } * { val b = 10 b - 1 }" should be "36"

Nesting.fun example from assignment 3 file [2]. Ensures block expressions translate correctly when combined with arithmetic expression

5 References

[1] "inkytonik/kiama", GitHub, 2020. [Online]. Available: https://github.com/inkytonik/kiama/blob/master/wiki/ParserCombs.md. [Accessed: 29- Sep- 2020].

[2] "iLearn Assignment 3 Spec", Ilearn.mq.edu.au, 2020. [Online]. Available: https://ilearn.mq.edu.au/mod/page/view.php?id=6035934. [Accessed: 29- Sep- 2020].

[3] "Week 10 & 11 Translator.scala", ilearn.mq.edu.au, 2020. [Online]. Available: https://ilearn.mq.edu.au/pluginfile.php/6430554/mod_resource/content/4/Translator.scala. [Accessed: 29- Sep- 2020].