# COMP3000 – PROGRAMMING LANGUAGES

# ASSIGNMENT 3

## TRANSLATION

**TRIDEEP LAL DAS**

**45532125**

**6/10/2020**

# Introduction

The aim of this assignment is to build and create a simple functional programming language called FunLang. As with building a programming language, this assignment aims to develop the parsing or the translation section of the FunLang programming language.

The FunLang language (abbreviated to FL) is a small programming language written using the Kiama Library as a part of a learning exercise on designing and implementing a programming language. The Kiama library was used to implement code translation for FL programming language.

This report will contain a description of the design and implementation of the translator and an overview of the test cases and the sufficiency of the test cases.

# Design and Implementation

## SEC Machine

The SEC machine, Stack, Environment, Code machine is a stack-based machine is where computations are carried out by pushing and popping data onto and off a stack. The operations performed on elements in the stack are dictated by low-level SEC instructions.

The best way to think about this implementation is in terms of High/Low Level Assembly where for example, we have a value a = 5, a + 5, we would call

```
lvalue a

push 5

assign
```

Then to evaluate a + 5 we would get

```
rvalue a

add
```

The instructions which comprise the program occur in nested lists together with other constructs, example IClosure takes in variable names and lists of instructions as it's arguments, IMult being the operation for multiplication, and the list goes on. The list of available instructions can be found in the assignment specification document.

## Translator

The translation of the code can be seen in the Translator Scala file and briefly going through how it works, for a single expressions such as Integers, Booleans and a Var, it will take in an argument, and evaluate it. The appropriate instructions are called so for Boolean, IBool is called, Int would be IInt and Var would be IVar.

For Arithmetic and Logical Operators Plus, Minus, Slash, Star, App, Equal and Less Expressions, they accept two arguments, with the left side then the right side arguments being generated and evaluated and ultimately having the appropriate instruction called, example Plus calls IAdd, Equal calls IEqual and so on according to the assignment specifications for instructions.

The If Expression takes in three arguments, a condition, a then Expression and an else Expression, and acts such that if an if-statement is found, we get a condition, which is then branch into executing either the then expression or the else expression. This works exactly like an if-statement would in any programming language. The standard template for if statements is if-then-else, so if-(if-then-else)-else which can become if-(if-(if-then-else)-else)-else.

As for the Block Expression, it takes in two arguments, a defines argument and an expression argument. If the parser does find a 'def' or a 'val' call, depending on whether a value or a function was found, it will be translated accordingly. Depending on which is called, the left and right side would be evaluated and finally an ICall will be generated.

If there is nothing to be found, or any other inputs found not stated here, the translator will either return a 0 or an error.

# Testing

## Overview

To give an overview of the test cases, what tests were performed and why they were performed, the tests performed can be divided into Associativity and Precedence tests, including multi-operator tests, tests for constructs such as If-statements and definitions and finally block tests which test tests assignment of values, creating functions, having an if-statement function and mixing function functionality. There is a total of 87 test cases to properly test each test case.

## Test Cases

As for the test cases, all associativity and precedence testing can be seen from line 47-238 where they. They including testing for simple arithmetic operations and logical operations to tests for associativity and precedence by doing tests like 1/2 == 2/1 to be false and 1/2 < 2/1 to be true and having bigger operations such as 10+20/50*20 < 0 to be false and so on. We do not have tests for having multiple Equals and Less expressions as we already know that they do not exists in FL programming Language (it also returns an error which we test for).

For testing if-statements, the tests cases can be found from Line 248-324 whereby we test if-statements for if the condition is true, it returns the value in the then statement and if the if statement is false, we return the else statement. We then test for if we can have all arithmetic and logical operators in the if-statement condition and evaluate them by tests such as if(5<3) then 20 else 50(this will give us false and return 50) to tests with nested if-statements to see if that can be parsed correctly.

Finally, the last set of tests which are the block tests from line 327-388 which test for all block functionality. This includes block tests for defining the value of x, functions that can evaluate associativity and precedence tests, functions that can evaluate functions, functions that can evaluate if statements and finally functions that can evaluate functions and if-statement(much like the popular meme of apple pen and pineapple pen, there is a similar amount of tests done for function+function+if-statement combination).