

# 6CCS3CFL-CW4

Q1

For this question

- I added some code in the compiler header for JVM section. The code that was added was provided as part of the CW4 specification.
- I added read and write functions in this header section.
- I then added **read**, **writeVar** and **writeStr** in **compile\_stmt** for the compile to recognise these parsed tokens.

In my implementation I do not need to add quotes back in the compiler phase. Hence my compile function works without any string modification

Since input was not being recognised by my Windows PC by running the compile\_and\_run function, I edited the function to the following:

```
def compile_and_run(bl: Block, class_name: String) : Unit = {  
  println(s"Start of compilation")  
  compile_to_file(bl, class_name)  
  println(s"generated $class_name.j file")  
  os.proc("java", "-jar", "jasmin.jar", s"$class_name.j").call()  
  println(s"generated $class_name.class file ")  
  //println(os.proc("java", s"${class_name}/${class_name}").call().out.text())  
  // os.proc("java", s"${class_name}/${class_name}").call(stdout = os.Inherit)  
  println("")  
  println(s"You may now manually run the file.")  
}
```

The function

- Lexes, parses
- Generates J file
- Generates Class file

After that the code has to be manually executed by the following command

*java < class >/< class >*

The two required J files can be found under the root directory by the following name:

1. Fibonacci – fib.j
2. Factorial – fact.j

Q2

I added the for loop by extending the compile\_stmt, and adding support for For loops in Stmt parser.

Within the Stmt parser

- Added support for For

```
(T_KMD("while") ~ BExp ~ T_KMD("do") ~ Block).map[Stmt] { case _ ~ y ~ _ ~ w => While(y, w) } ||
(T_KMD("for") ~ Stmt ~ T_KMD("upto") ~ AExp ~ T_KMD("do") ~ Block).map[Stmt] { case _ ~ s ~ _ ~ a ~ _ ~ b1 => For(s, a, b1): Stmt }
```

Within compile\_stmt

- Added support for For
- Used pattern matching to match i and a
- Changed the underlying mechanism to that of a While Loop as directed in the Coursework Specification

```
}
case For(st, ar, b1) => {
  // Here I and a equal to the values passed into st
  // ST here will be of the form Assign(x,y)
  // Hence i = x, and a = y
  val Assign(i, a) = st
  compile_block(
    List(
      st,
      While(
        Bop("<=", Var(i), ar),
        b1 ++ List(Assign(i, Aop("+", Var(i), Num(1))))
      )
    ),
    env
  )
}
```

Q3

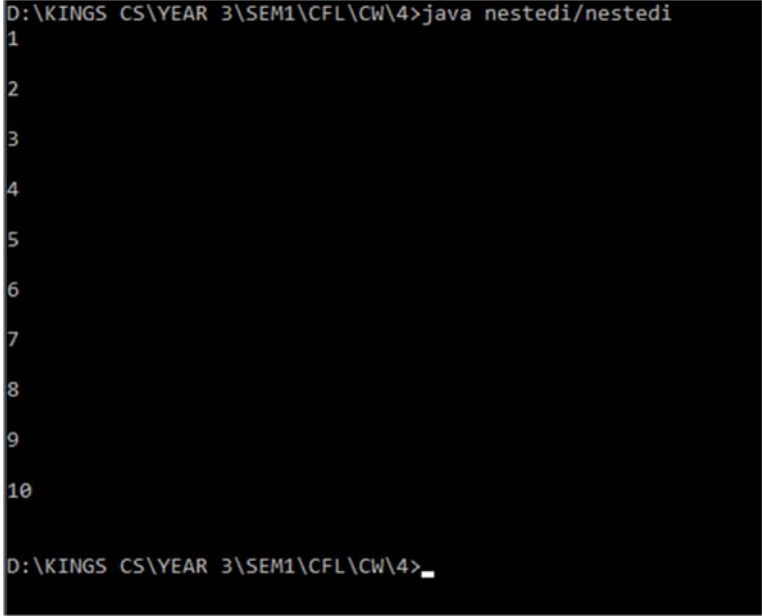
In my language, running the following command describes a situation more commonly referred to as “Lack of Scope”.

```
for i := 1 upto 10 do {  
    for i := 1 upto 10 do {  
        write i  
    }  
}
```

The assembler instructions can be found in [nested.i](#)

Currently since the variables are not scoped, the value of i is sort of treated as a global.

- We start with the outer loop where the value is set to be 1.
- Then we start with the inner loop where the value is again 1
- We iterate through the inner loop until the condition  $i \leq 10$  is not met
- When we exit the inner loop, we compare this new i with the condition which will also not be satisfied.
- Hence the outer loop will also exit
- Therefore, the output will look like the following



```
D:\KINGS CS\YEAR 3\SEM1\CFL\CW\4>java nested/nested.i  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
  
D:\KINGS CS\YEAR 3\SEM1\CFL\CW\4>_
```