Coursework 3

This coursework is worth 4% and is due on 13 December at 16:00. You are asked to implement a compiler for the WHILE language that targets the assembler language provided by Jasmin. This assembler is available from

```
http://jasmin.sourceforge.net
```

There is a user guide for Jasmin

```
http://jasmin.sourceforge.net/guide.html
```

and also a description of some of the instructions that the JVM understands

```
http://jasmin.sourceforge.net/instructions.html
```

If you generated a correct assembler file for Jasmin, for example `loops.j`, you can use

```
java -jar jasmin-2.4/jasmin.jar loops.j
```

in order to translate it to Java byte code. The resulting class file can be run with

```
java loops
```

where you might need to give the correct path to the class file.

You need to submit a document containing the answers for the two questions below. You can do the implementation in any programming language you like, but you need to submit the source code with which you answered the questions. Otherwise the submission will not be counted. However, the coursework will *only* be judged according to the answers. You can submit your answers in a txt-file or as pdf.

Question 1 (marked with 2%)

You need to lex and parse WHILE programs and submit the assembler instructions for the Fibonacci program and for the program you submitted in Coursework 2 in Question 3. The latter should be so modified that a user can input the upper bound on the console (in the original question it was fixed to 100).

Question 2 (marked with 2%)

Extend the syntax of you language so that it contains also `for`-loops, like

$$\texttt{for}\ \ \mathit{Id}\ \texttt{:=}\ \mathit{AExp}\ \ \texttt{upto}\ \ \mathit{AExp}\ \ \texttt{do}\ \mathit{Block}$$

The intended meaning is to first assign the variable *Id* the value of the first arithmetic expression, then go through the loop, at the end increase the value of the variable by 1, and finally test wether the value is not less or equal to the value of the second arithmetic expression. For example the following instance of a `for`-loop is supposed to print out the numbers `2`, `3`, `4`.

```
for i := 2 upto 4 do {
    write i
}
```

There are two ways how this can be implemented: one is to adapt the code generation part of the compiler and generate specific code for `for`-loops; the other is to translate the abstract syntax tree of `for`-loops into an abstract syntax tree using existing language constructs. For example the loop above could be translated to the following `while`-loop:

```
i := 2;
while (i <= 4) do {
    write i;
    i := i + 1;
}
```

In this question you are supposed to give the assembler instructions for the program

```
for i := 1 upto 10000 do {
  for i := 1 upto 10000 do {
  skip
  }
}
```

Further Information

The Java infrastructure unfortunately does not contain an assembler out-of-the-box (therefore you need to download the additional package Jasmin—see above). But it does contain a disassembler, called `javap`. A dissembler does the "opposite" of an assembler: it generates readable assembler code from Java byte code. Have a look at the following example: Compile using the usual Java compiler the simple Hello World program below:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

You can use the command

```
javap -v HelloWorld
```

to see the assembler instructions of the Java byte code that has been generated for this program. You can compare this with the code generated for the Scala version of Hello World.

```
object HelloWorld {
   def main(args: Array[String]) {
      println("Hello World!")
  }
}
```

Library Functions

You need to generate code for the commands `write` and `read`. This will require the addition of some "library" functions to your generated code. The first command even needs two versions, because you might want to write out an integer or a string. The Java byte code will need two separate functions for this. For writing out an integer, you can use the assembler code

```
.method public static write(I)V
    .limit locals 5
    .limit stack 5
    iload 0
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V
    return
.end method
```

This function will invoke Java's `println` function for integers. Then if you need to generate code for `write x` where `x` is an integer variable, you can generate

```
iload n
invokestatic XXX/XXX/write(I)V
```

where `n` is the index where the value of the variable `x` is stored. The `XXX/XXX` needs to be replaced with the class name which you use to generate the code (for example `fib/fib` in case of the Fibonacci numbers).

Writing out a string is similar. The corresponding library function uses strings instead of integers:

```
.method public static writes(Ljava/lang/String;)V
   .limit stack 2
   .limit locals 2
   getstatic java/lang/System/out Ljava/io/PrintStream;
   aload 0
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
   return
.end method
```

The code that needs to be generated for `write "some_string"` commands is

```
ldc "some_string"
invokestatic XXX/XXX/writes(Ljava/lang/String;)V
```

Again you need to adjust the `XXX/XXX` part in each call.

The code for `read` is more complicated. The reason is that inputting a string will need to be transformed into an integer. The code in Figure 1 does this. It can be called with

```
invokestatic XXX/XXX/read()I
istore n
```

where `n` is the index of the variable that requires an input.

```
.method public static read()I
      .limit locals 10
      .limit stack 10

      ldc 0
      istore 1  ; this will hold our final integer
Label1:
      getstatic java/lang/System/in Ljava/io/InputStream;
      invokevirtual java/io/InputStream/read()I
      istore 2
      iload 2
      ldc 10   ; the newline delimiter
      isub
      ifeq Label2
      iload 2
      ldc 32   ; the space delimiter
      isub
      ifeq Label2

      iload 2
      ldc 48   ; we have our digit in ASCII, have to subtract it from 48
      isub
      ldc 10
      iload 1
      imul
      iadd
      istore 1
      goto Label1
Label2:
      ;when we come here we have our integer computed in Local Variable 1
      iload 1
      ireturn
.end method
```

Figure 1: Assembler code for reading an integer from the console.