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 the 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`

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. 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

`for` *Id* `:=` *AExp* `upto` *AExp* `do` *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 anymore 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;
}
```

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, `java`, 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
```

in order to see which Java Byte Code has been generated for this program. You can compare this with the code generated for the Scala version of Hello Worlds.

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